AAD: Breaking the Primal Barrier

Dmitri Goloubentsev and Evgeny Lakshtanov present a new, speedier approach for AAD.

In this article we present a new approach for automatic adjoint differentiation (AAD) with a special focus on computations where derivatives $\frac{\partial F(X)}{\partial X}$ are required for multiple instances of vectors X. In practice, the presented approach is able to calculate all the differentials faster than the primal (original) C++ program for F. Major application areas are:

- Gradient methods for optimization problems, including global model calibration, speech recognition, deblurring of images and machine learning in general.
- Derivatives of mathematical expectation.
- Pathwise sensitivities of stochastic differential equations.

Code transformation vs. operator overloading

Currently, two main approaches are used for the AAD tools:

- *Code transformation* (CT). Analyses the computer program which implements function F to produce a code of the adjoint differentiation (AD) method.
- *Operator overloading* (OO). All mathematical operations are overloaded in such a way that the information about a computational graph of F is saved in the data structure called Tape.[1] Tape is used afterwards to process the backward pass of the AD method.

There are rather succesful CT AAD tools, however, they limit the available language features and make the build system more complex. The difficulty in building such a tool is further reflected in the fact that there is currently no CT AAD available for C++. The OO approach usually demonstrates weaker speed performance due to a runtime overhead in each iteration. Let us enter into more details at this point.

Consider a schematic processing of multiple samples X_i using a standard OO AAD library:

```
Loop  i =0.. N –1
  BeginTapeRecording ( ) ;
  Y [ i ]  =  F ( X [ i ] ) ;
  StopTapeRecording ( ) ;
  dX [ i ]  =  Reverse ( dY [ i ] ,  tape ) ;
Next  i ;
```

Each overloaded operator collects information on valuations, after which the second backwards pass updates the adjoint variables. This approach comes with a number of disadvantages:

- OO or/and tape interpretation runtime overhead for each iteration.
- Multithreading is only possible if the original function $F(X[i])$ is thread-safe.
- Relying on the compiler for CPU vectorization of a scalar primal function means that vectorization is used very sparsely.
- $F()$ may perform unnecessary operations that don't depend on X, and yet are executed at each iteration (e.g. mathematical operations, virtual function calls, dictionary lookups, etc.).
- Additional memory is required to store the Tape data structure. Its size is proportional to the number of operations of the primal function F, which may be prohibitive in some cases.

OO × CT

Our innovative idea is to cross[2] both approaches, namely to use the overloaded operators to autogenerate an AD version of the primal function at runtime. The created AD functions can be used for different X_i instead of performing the classic OO AAD approach on each $F(X_i)$.

During the first run of the original function, every overloaded function – or operator – will generate instructions for a forward and reverse AD pass. For instance, consider the function $f(a, b, c) = a * b + c$. The first column of Table 1 lists the consecutive calls of atomic valuations during the execution of the function $f(a, b, c)$.

Table 1: Construction of Forward and Reverse AAD functions

Valuation	→ Forward()	→Reverse()
Initialization	`v0 = a; v1 = b;`	
Initialization	`v2 = c;`	
operator *	`v3 = v0 * v1;`	`d1 += d3 * v0;`
		`d0 += d3 * v1;`
operator +	`v4 = v3 + v2;`	`d2 += d4;`
		`d3 += d4;`
Initialization	`f = v4;`	
Initialization		`d0 = 0; d1 = 0;`
Initialization		`d2 = 0; d3 = 0;`

Concatenated entries of the second column of Table 1 form the body of the forward pass, while the body of the reverse pass is formed by the entries in the third column in reverse order:

```
void  AD_Function
 ( double  a , b , c ,& f ,& d0 ,& d1 ,& d2 , d4 )  {
double  v3 , d3 ( 0 ) ;
v0 = a ;  v1 = b ;  v2 = c ;
v3 = v0 * v1 ;
v4 = v3 + v2 ;
f = v4 ;
//  Reverse
d2  +=  d4 ;
d3  +=  d4 ;
d0  +=  d3  *  v1 ;
d1  +=  d3  *  v0 ;
}
```

Now all differentials of the function f and its value at the point (a, b, c) can be computed by calling

```
double  d0 (0) , d1 (0) , d2 (0) , d4 (1) , f ;
AD_function ( a , b , c , f , d0 , d1 , d2 , d4 )
```

Within the proposed framework, the OO is used for only one sample X, after which the resulting program can process multiple instances of input data. Let us list the immediate benefits of this approach:

- Unlike the classic OO AAD, the AD function does not change from one iteration to the next. Hence, there is no OO or tape interpretation runtime overhead per X_i sample.
- The AD function is completely segregated from the user program. All user data are encapsulated within the AD function, and its memory state is limited to vectors v and d. Thus, multiple samples of X can be processed safely in the multithreaded mode.
- The AD function can be generated to consistently utilize native CPU vectorization to process 4(8)-double chunks of user data (AVX2\AVX512 speed-up ×4–×8).
- Highly efficient (i.e. operations that don't depend on X are not included in the constructed AD function).
- Although additional memory is also required to store the AD function, its code remains static and can be shared between CPU cores.

Combining all the stated benefits, one can achieve a fantastic performance of 0.4 at one core, compared to the original program implemented using a standard *double* arithmetic. For anybody who wishes to play around with this, we have prepared a "prototype" C++ implementation available at www.matlogica.com free of charge. This prototype implementation is simplified and cannot work effectively with large computations. In addition to a two-pass compilation, the relative compilation time and volume of memory involved are unacceptable for practical use.[3] Part of the reason for this is that the size of the AAD functions is proportional to the tape – that is, to the linearized (unfolded) version of the primal algorithm. Below we discuss how we can address these drawbacks.

A prototype C++ implementation

The code of the prototype library is short and self-explanatory. Nevertheless, we found it reasonable to provide some comments in case it needs further clarification. According to the canonical AD, all variables can be distinguished by their roles as inputs, intermediates, or outputs. The header file `NaiveAADLib.h` defines the active class *dagdouble*, which transforms each double variable into a node of a future calculation directed acyclic graph (DAG):

```
class  dagdouble  {
public :
  dagdouble ()  {}
  dagdouble ( const  double &  val ,
     bool  isInput  =  f a l s e )
   :  val ( val )  {
   indx  =  getNextVarCounter ( ) ;
   i f  ( ! isInput )  {
    aadAssignConst ( indx ,  val ) ;
   }  else  {
    inputIndex . insert ( indx ) ;
   }
  }
  dagdouble ( const  dagdouble &  other )  :
     val ( other . val )  {
   indx  =  getNextVarCounter ( ) ;
   aadAssign ( indx ,  other . indx ) ;
  }
  dagdouble &  operator  =
    ( const  dagdouble &  other )  {
   val  =  other . val ;
   indx  =  getNextVarCounter ( ) ;
   aadAssign ( indx ,  other . indx ) ;
   return  * this ;
  }
  void  markAsOutput ()  {
   outputIndex . insert ( indx ) ;
  }
  double  val ;
  int  indx ;
};
```

As an example, we supplied an overloaded version of the multiplication operator. Here, we provide only the scalar version of the code. For the vector version, the reader is encouraged to consult the prototype library source code.

```
dagdouble  operator  *( const  dagdouble &  a ,
       const  dagdouble &  b )  {
  dagdouble  res ;
  res . val  =  a . val  *  b . val ;
  res . indx  =  getNextVarCounter ( ) ;
  aadMult ( res . indx ,  a . indx ,  b . indx ) ;
```

```
  return res;
}
void aadMult(int res_indx, int a_indx,
     int b_indx) {
  stringstream fstr, rstr;
  if (codeVersion == ScalarCode) {
   fstr << "v" << res_indx << "=" << "v"
    << a_indx << "*v" << b_indx << ";";
   rstr << "d" << a_indx << "+=" << "d"
    << res_indx << "*v" << b_indx << ";"
    << "d" << b_indx << "+=" << "d"
    << res_indx << "*v" << a_indx << ";";
  }
  aad_func_fwd.push_back(fstr.str());
  aad_func_rev.push_back(rstr.str());
}
```

It creates a new variable (`getNextVarCounter()`) and writes the corresponding code of the forward and reverse passes (applying `aadMult()`).
The distribution contains a basic example `main.cpp` where the library's functioning is carried out by defining a preprocessor variable `#define USE_GENRATED_AAD_FUNCTION`. Depending on its value, the user program either generates a file which contains a newly created AD function or uses the earlier generated one.

The included synthetic benchmark test (`example.cpp`) produces the results shown in Table 2.

Table 2: The prototype library benchmarks using clang++

	Absolute time (ms)	Relative factor
Primal	83	1
AVX2 Adjoint	64	0.73
AVX512 Adjoint	39	0.46

All of the aforementioned disadvantages of the prototype approach can be addressed by generating a binary code directly from OO. This idea was implemented in the AAD-Compiler by MathLogic Ltd.

Just-in-time AAD-Compiler

The just-in-time AAD-Compiler (patent pending) is a professional version of the prototype library. It represents a completely enabled OO AAD library and offers the following features:

- Optimized machine binary code generation optimized for both runtime performance and quick code generation.
- Streaming compilation.
- Incremental checkpointing.
- Proprietary AD code-folding compression.
- Support for multiple platforms and C++ compilers.
- Support for AVX2 and AVX512 vectorization.
- Enables multithreaded valuations even when underlying user program isn't multithread safe.

The AAD-Compiler has been tested and documented extensively. Licensing terms distribution can be found at www.matlogica.com.

AAD-Compiler speed-benchmark results

The benchmark results for the AAD-Compiler are based on tests of different nature, including random synthetic tests, previously published tests such as LW, Toon, or GMM, as well as standard financial models like the linear mixed model (LMM) or stochastic volatility calibration [1, 2, 4].

The first test we consider is based on the open-source benchmark from [2]. The authors test various AAD tools for the Gaussian mixed model (GMM) with 2.5M iterations and get the results (absolute time of adjoints in seconds) as shown in Table 3.

Table 3: Absolute time of adjoints for GMM model from [2]

#Variables		5.36e+4	4.29e+5
Manual		3.89e+2	6.16e+3
Finite difference			
Adept	C++	4.09e+3	3.99e+4
ADOLC	C++	1.04e+4	
Ceres	C++		
Tapenade	C	1.32e+3	1.59e+4
DiffSharp	F#		
MuPAD	Matlab	•	•
Julia-F	Julia		
Julia-F (vect)	Julia	•	•
Autograd	Python		
Theano	Python	•	•
Theano (vect)	Python	•	•

Note: The bullet symbolizes that a tool crashed and no entry means that a tool did not finish in the time limit. Only tools that could compute at least one problem instance are shown.

We execute the Adept and AAD-Compiler versions of the GMM with 429,000 arguments and 2.5M iterations and get the absolute time of adjoints (in seconds) as shown in Table 4. The AAD-Compiler has an optionally activated code compressor, so both cases are presented.

Table 4: GMM benchmark using AADC and Adept

Tool	Compressor	AVX2	AVX512
Adept	—	2.89e+4	N/A
AADC	On	1.3e+3	1.17e+3
AADC	Off	2.86e+3	2.86e+3

Note that the huge number of differentials required (0.5M) makes the GMM a memory-bound problem and memory bandwidth is almost maxed out at AVX2.

For smaller-sized problems, performance scales well with the AVX vector size.

AAD-Compiler relative time of adjoints for real-world models

Table 5 demonstrates the relative time of the full gradient to the execution time of a primal algorithm.

Table 5: Relative time of adjoint execution to primal

Model	AVX2	AVX512
Toon	0.25	0.2
Heston	0.37	0.22
LMM	0.35	0.21

The Toon benchmark is assumed to run over multiple input data.
The behavior of the relative performance time based on number of CPU cores can be found in Table 6.

Table 6: Relative time of LMM adjoints using multi-thread mode

LMM	1 Core	4 Cores	8 Cores
AVX2	0.35	0.095	0.05
AVX512	0.21	0.06	0.035

AAD-Compiler relative compilation time for various real-world and toy models

From Table 7, one can conclude that the compilation time is equivalent to around 400 executions of the primal algorithm.

Table 7: One-time cost of adjoint functions construction

Model	Relative compilation time
Heston	970
LMM	650
GMM	700
Toon	240

AAD-Compiler memory-benchmark results

For memory benchmarks we also used the conventional Lax and Wendroff (LW)/ Toon tests from [1], as shown in Table 8.

Table 8: Memory use for various public benchmarks

Tool	Compressor	LW	Toon	GMM
Adept		3MB	24.5MB	23MB
AADC	ON	42kB	11.5MB	7MB
AADC	OFF	16MB	59.5MB	15MB

AADC's values are normalized to a single sample. To obtain the real memory consumption, one needs to multiply the values by an AVX vector length.

Forward function

The forward function is useful on its own and constitutes a vectorized replication of the primal algorithm. Similar to the complete AD function it is multithread safe, even if the primal algorithm is not. This replication proves extremely useful if one wishes to use finite differences, or just accelerate any complex computations. Given the current trend in using multiple CPU cores to accelerate computations, this feature is useful in its own right.

Acknowledgments

Evgeny Lakshtanov is partially supported by Portuguese funds through the Center for Research and Development in Mathematics and Applications and the Portuguese Foundation for Science and Technology, within project UID/ MAT/0416/2019.

Dmitri Goloubentsev has 15 years of combined experience in model development working on C++ quant libraries. He worked as a Senior Quant Analyst in Interest rate derivatives and played a leading role in delivering XVA solution at a major Canadian bank. Prior to focusing on AAD, he was responsible for construction of SIMM/MVA model. Dmitri earned his degree in Maths and Applied Maths from the Moscow State University.

Evgeny Lakshtanov completed his PhD in Mathematical Physics in 2004 at the Moscow State University. His interests cover a wide range of applications including Statistical Physics, Game Theory and Inverse Problems for PDEs. He currently serves as a Principal Investigator at the University of Aveiro, Portugal. His expertise in gradual perception of importance of the AAD for inverse problems led him to become an external research consultant at MathLogics LTD.

ENDNOTES

1. Its choice depends on a concrete AAD tool.
2. Not to be confused with AAD of the mixed type, where the tool does an analysis first and then takes the decision on what approach should be used in each particular situation.
3. See the relative compilation time benchmarks for prototype and professional AAD compilers.

REFERENCES

[1] Hogan, R. J. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. ACM *Transactions on Mathematical Software* (TOMS) 40(4), 26.
[2] Srajer, F., Kukelova, Z., and Fitzgibbon, A. 2018. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software* 33(4–6), 889–906.
[3] AAD-Compiler prototype library. https://github.com/matlogica/aadc-prototype.
[4] Goloubentcev, D. and Lakshtanov, E. 2019. Remarks on stochastic automatic adjoint differentiation and financial models calibration. arXiv preprint 1901.04200.
[5] AAD-Compiler by MathLogic Ltd. www.matlogica.com.