

HPC AAD-Compiler C++ Software Library

User Guide

MatLogica LTD, London, UK

January 31, 2022

Abstract

This document is a user guide for the AADC library. We extrapolate on the examples in the test directory presenting the programming techniques required to work with the library. We also discuss how to use Eigen, integration strategies and troubleshooting, and explain a few advanced methods.

Contents

Copyright and license for AADC software	3
1 Overview	3
1.1 What is an HPC AAD-Compiler?	3
1.2 Main features	4
2 Installing AADC	4
2.1 Installing on Linux	5
2.2 Installing on Windows	5
3 Using AADC for automatic differentiation	5
3.1 Example 1. Hello, World!	6
3.1.1 Recording and the forward kernel	6
3.1.2 Reverse kernel	9
3.2 Example 2. Black-Scholes	10
3.3 Example 3. Monte Carlo	12
3.4 Example 4. Jacobian	14
3.5 Example 5. Valuation involving interpolation (using <i>ibool</i> and <i>iint</i>) . . .	15
3.6 Example 6. Incremental Checkpointing	16
3.7 Example 7. Multithread for a simple MonteCarlo	17
3.8 Example 8. Interfacing AADC to a minimization library	20
4 Using Eigen during the AADC-recording (introduces Examples 15 and 16)	22

5	Integration strategies	23
5.1	Redefining <i>double</i> as <i>idouble</i> (based on Example 11)	23
5.1.1	Automatically locating stochastic <i>ibool</i> -to- <i>bool</i> conversions	24
5.2	Objects in mixed (native/AAD) mode (based on Example 9)	25
5.2.1	Segregating active and native code using namespaces	25
5.2.2	Handling virtual functions	27
6	Troubleshooting (based on Example 12)	28
6.1	AADC Debug Tool	28
6.2	Localization of Not-a-Number (NaN) and other discrepancies	30
7	Advanced Methods	31
7.1	Example 13: Using AADC for external functions	31
7.2	Example 14: AAD for Mean Square Error of Expectations	33
8	Functions of the AADC library	36
8.1	<i>idouble</i> , <i>ibool</i> , <i>iint</i> methods	36
8.2	Auxiliary vector functions	36
8.3	<i>AADCFunctions</i> class	37
8.3.1	Statistics	37
8.4	<i>AADCWorkSpace</i> class	38
8.5	Vector generalizations	38
8.6	Mathematical operations with <i>idouble</i> , <i>ibool</i> , <i>iint</i> , and arrays	39
8.6.1	Comparisons	39
8.6.2	min/max operations	39
8.6.3	Conditional operations	40
8.6.4	Casting	40
8.6.5	Arrays	40
8.7	Global pragmas and internal namespaces	40
8.8	Checkpointing and executing program segments	41
8.9	Complex numbers	41
A	Appendix A - List of Examples	42
A.1	Example 1. Hello, World!	42
A.2	Example 2. Black-Scholes	43
A.3	Example 3. Monte Carlo	46
A.4	Example 4. Jacobian	47
A.5	Example 5. Valuation involving interpolation (using <i>ibool</i> and <i>iint</i>)	48
A.6	Example 6. Incremental Checkpointing	51
A.7	Example 7. Multithread for a simple MonteCarlo	52
A.8	Example 8. Interfacing AADC to a minimization library	55
A.9	Example 9. Reusing non-AAD objects. Global <i>idouble</i> Casting	56
A.10	Example 10. European Option Monte-Carlo Pricing	59
A.11	Example 11. Option Pricing without templates	63
A.12	Example 12. Option Pricing with Debug	68
A.13	Example 13. Using AADC for external functions	72
A.14	Example 14. AAD for Mean Square Error of Expectations	74
A.15	Example 15. Recording the Eigen routines using AADC	77
A.16	Example 16. Using Eigen with complex <i>idouble</i>	78

Copyright and license for AADC software

Authors: Dmitri Goloubentsev, Evgeny Lakshtanov (dmitri, lakshtanov at matlogica LTD, UK)

Copyright ©2019 MathLogic LTD, London, UK.

Version: 1.0

See the LICENSE document included with the package.

1 Overview

We start by explaining the basics of the technology and give an outline of the document. We then list the main features of the AADC compiler.

1.1 What is an HPC AAD-Compiler?

AADC is a novel approach to performing adjoint differentiation for C++ programs. It combines the integration ease of Operator Overloading with the efficiency of Code Transformation. The library extracts mathematical expressions from the user code and builds highly optimized versions of the *forward* (replicating user valuations) and *reverse*¹ (F/R) kernels.

The AADC compiler interface uses an operator overloading approach to efficiently perform on-the-fly compilation of user valuations into CPU binary instructions. Unlike the normal C++ compiler, AADC works at runtime and builds CPU binary code for the forward and reverse (adjoint method) computations. The original code only runs once but the obtained binary code can be executed multiple times to efficiently solve specific user problems. The approach is particularly efficient when repeated calculations are required (e.g. in Monte Carlo simulations or minimization problems).

The library interface of the compiling (recording) stage is similar to other Operator Overloading AAD tools like Adept, ADOL-C, and DCO/C++. Unlike those AAD tools, AADC does not build AAD tape but compiles native binary² F/R kernels.

After the compilation is complete, the produced kernels are used by the library client to get gradients or just accelerate multiple parallel computations of the original algorithm. Due to the novelty of this approach, the required programming techniques are somewhat different from those required in prior tools for automatic differentiation.

This user guide describes how to use the AADC software library for your code, with examples extrapolating on the test directory of the AADC software package. We start by briefly introducing the main features of AADC, then Section 2 explains how to install AADC on Linux or Windows. Section 3 introduces a minimal set of techniques required to solve typical problems. Section 4 explains how to use Eigen during the AADC recording. Sections 5 and 6 concern integration and troubleshooting respectively. Section 7 deals with more advanced topics (external functions with no source code and computing the mean square error of expectations). Section 8 describes the functions in the AADC library and Appendix A gives a formal specification for all the examples (also present in the test directory).

¹Also called *adjoint* or *backward*, see https://en.wikipedia.org/wiki/Automatic_differentiation#Reverse_accumulation.

²Currently supports AVX2 and AVX512.

1.2 Main features

We briefly introduce the main features of the AADC-compiler which distinguish it from the previous approaches:

1. *On-the-fly Adjoint Differentiation Streaming Compiler*. Patent pending. The implementation is focused on getting high speed and low memory requirements for both compilation and the resulting kernels. The compiler works in a streaming mode and converts the stream of user valuations into two binary streams for F/R algorithms. The following table explains the memory requirements:

	Random memory	Sequential memory
Compiling stage	#Active Variables	(Size of the algorithm)
Execution stage	#Active Variables	$C \cdot$ (Size of the algorithm)

Here we have $C = 1$ when the Code Compressor is off. Using innovative compression, memory requirements can be drastically lowered. C is around 0.1 in real-world models like LMM or Stochastic Volatility Calibration.

2. *Reusing non-AAD objects in the AAD recording step*. AADC provides a much simpler way to integrate the active type into the existing codebase. The active type occupies the same memory as its non-active analog. It just allows casting a pointer from a non-AAD object to an AAD-enabled class during the user's valuations. See Section 5.2.
3. *Automatic vectorization*. AADC builds the code taking advantage of the AVX2/AVX512 technology without changing the user code.
4. *Free multithread valuations*. Compiled F/R kernels segregate the execution context and the hardwired static user data. So they are safe to execute in parallel even if the original code is not.
5. *Easier troubleshooting*. AAD-Compiler does not use C++ meta-programming, so it is easier to troubleshoot problems and compilation time is comparable to that of the original program.

The efficient binary code, SIMD³ vectorization, and multithreading combined can lead to outstanding x50-x1000 performance improvement compared to other single-threaded AAD tools.

2 Installing AADC

The AADC library should be compatible with any C++11 compiler. We have tested Intel, MSVC, GCC, and CLang compilers and validated they do work. The library comes with a battery of unit tests and all the tests completed successfully on all compilers listed above.

AADC is supported on modern versions of Windows and Linux operating systems, 64-bit only. We tested the library on Ubuntu 17.10 and Windows 10. AADC can also be supported in other versions of compilers and operating systems on request.

Hardware requirements: both Intel and AMD CPUs are supported. AVX2 instruction set is a minimum requirement. For best performance, we recommend modern processors with AVX512 instruction set support.

³Single Instruction Multiple Data.

2.1 Installing on Linux

First, install the latest CMake from the source:

```
download latest from https://cmake.org/download :
wget -v https://github.com/Kitware/CMake/releases/download/v3.14.3/cmake-3.14.3.tar.gz
tar xf cmake-3.14.3.tar.gz
cd cmake-3.14.3
./bootstrap
make
# uninstall current version installed by apt-get
sudo apt-get remove cmake cmake-data
sudo make install
```

Then build and install the AADC library from source:

```
source /opt/intel/bin/iccvars.sh intel64
mkdir build
cd build
export CC=/opt/intel/bin/icc
export CXX=/opt/intel/bin/icpc
cmake ..
make
../unit-test
../build/unit-test/runUnitTests
sudo make install
```

After running the “make install” command, the library and header files will be installed in “home/lib” and “home/include” folders respectively.

2.2 Installing on Windows

On Windows, tbc.

3 Using AADC for automatic differentiation

AAD has become a widespread tool in applications due to the following property:

Suppose one has an algorithm for a function

$$f : \mathbb{R}_x^n \rightarrow \mathbb{R}_y^m,$$

or coordinate-wise:

$$\begin{aligned} y_1 &= f_1(x_1, x_2, \dots, x_n), \\ y_2 &= f_2(x_1, x_2, \dots, x_n), \\ &\vdots \\ y_m &= f_m(x_1, x_2, \dots, x_n). \end{aligned}$$

Adjoint differentiation maps each vector $(\lambda_1, \dots, \lambda_m) \in \mathbb{R}^m$ to the following set of linear combinations of $\frac{\partial f_i}{\partial x_j}$:

$$\left\{ \sum_{i=1}^m \lambda_i \frac{\partial f_i}{\partial x_j}, \quad j = 1, \dots, n \right\} \quad (1)$$

Then the computational cost of this set does not depend on n (the number of inputs). It exceeds the computational cost of f only by a constant factor that is usually between 2 and 10 (depending on the specific AAD tool).

The present version of the AAD-Compiler has been tested for LMM and Stochastic volatilities calibration. Using SIMD optimization we achieve a groundbreaking speedup by a factor of 0.4 compared to the standard double. In both cases, we run single-threaded simulations and achieve the speedup due to the efficiency of the AAD binary code and AVX512 technology.

AADC can automatically differentiate the following operators and functions, see Section 8 for details:

- Standard binary mathematical operations $+$, $-$, \times and $/$.
- Assigning versions of the above operators.
- Mathematical functions `sqrt`, `exp`, `abs`, `log`, `pow`, etc.
- Logical functions `if`, `lower_bound`, `max`, `min`.
- Comparison operators `==`, `<`, `>`, `!=`, `<=`, `>=`.

Variables that appear in expressions that are to be differentiated must have a special “active” type that we call *idouble* in the AADC. Besides, integer or Boolean variables should be substituted by their “active” analogs *iint* and *ibool* if their values are not constant.

If you have used Adept, ADOL-C, CppAD, or Sacado, you are already familiar with applying an operator-overloading automatic differentiation package to your code. Unlike other tools where the user needs to recreate all objects with active type, AADC supports the differentiation of objects already in the memory. This makes it easy to fully or partially integrate AAD into a large codebase.

We will now walk the reader through a series of examples gradually increasing the complexity. We hope that this approach will give a good overview of what AADC is capable of. We give a formal specification in appendix A.

3.1 Example 1. Hello, World!

We record a function and run the forward and reverse AADC kernels.

3.1.1 Recording and the forward kernel

We record a simple function

$$f = \exp\left(\frac{x}{y} + z\right)$$

and check that the forward AADC kernel gives the same result as the original function. See the code below followed by a detailed explanation. In this example only we will give the full code in the body of the text (can also be found in A.1).

```

#include <iostream>
#include <aadc/aadc.h>
#include "examples.h"

void example_simple()
{
    idouble x(0.5),y(1.1),z,f;

    // size of the active type is same as that of the native type
    assert(sizeof(double) == sizeof(idouble));

    // We can use idouble as a replacement for native double
    z = x + sin(y);

    // We even print it to stdout, etc. See it in the
    // TERMINAL below
    std::cout << "z:" << z << std::endl;

    ////////////////////////////////////
    // Record Function
    ////////////////////////////////////

    typedef __m256d mmType;
    aadc::AADCFUNCTIONS<mmType> aad_funcs;

    aad_funcs.startRecording();
    aadc::AADCArgument x_arg(x.markAsInput());
    aadc::AADCArgument y_arg(y.markAsInput());
    aadc::AADCArgument z_arg(z.markAsInput());
    // Here the calculations are simple, but in practice this
    // call can travel through a lot of calls to the
    // object-oriented code.
    f=std::exp(x/y+z);
    aadc::AADCResult f_res(f.markAsOutput());
    aad_funcs.stopRecording();

    std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
createWorkspace());

    ////////////////////////////////////
    // Execute the function
    ////////////////////////////////////

    // set initial values for variables x, y & z
    ws->setVal(x_arg, _mm256_set_pd(1.0, 2.0, 3.0, 4.0));
    ws->setVal(y_arg, 2.0);
    ws->setVal(z_arg, _mm256_set_pd(0.1, 0.2, 0.3, 0.4));

    // execute the forward function
    aad_funcs.forward(*ws);

    // check the results for the forward function, these
    // should match the original f=exp(x/y+z)

```

```

for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi) {
    std::cout
        << "avx[" << avxi << "]" "
        << " f(" << ws->valp(x_arg)[avxi]
        << ", " << ws->valp(y_arg)[avxi]
        << ", " << ws->valp(z_arg)[avxi]
        << ")=" << ws->valp(f_res)[avxi]
        << std::endl;
}

```

We now explain the steps in detail:

- `typedef __m256d mmType;`
`aadc::AADCFunctions<mmType> aad_funcs;`

Creating an instance of the `aadc::AADCFunctions` class initializes the AADC library. When the *recording* finishes, the `aad_funcs` object will contain compiled forward and reverse (F/R) kernels. *mmType* is a notational convention for the variable type to be used by the compiled F/R kernels. It corresponds to `__m256d` in the case of AVX2 and to `__m512d` in the case of AVX512.

- `aad_funcs.startRecording();`
`AADCArgument x_arg(x.markAsInput());`
`AADCArgument y_arg(y.markAsInput());`
`AADCArgument z_arg(z.markAsInput());`

Compilation of F/R kernels takes place during the execution of the code between `StartRecording` and `StopRecording` methods. The method `markAsInput`⁴ selects specific variables to be the F/R kernel input (the values can change for different calls to the generated kernels). This method returns an index (of type `AADCArgument`) which can then be used to reach the corresponding variable (or its adjoint) in the F/R kernels.

- `f=exp(x/y+z);`
`AADCResult f_res(f.markAsOutput());`
`aad_funcs.stopRecording();`

All variables to be differentiated should be marked as *outputs* immediately before calling `StopRecording`.

In the example the only requirement on the values of x, y, z is that the valuations compile correctly, that is y cannot be zero. Some other important points will be addressed further in this section.

Note that AADC is a streaming compiler. As such, it saves the binary code for F/R kernels in the sequential memory (that can be very large). The size of the compiler state is proportional to the number of active variables at any time during the recording phase.

⁴or others (but only during the recording stage), see 8.1.

- `std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
createWorkspace());`

This instruction creates the execution context *ws*. There may be multiple execution contexts for the same *aad_funcs* (for example, to accomodate parallel execution - see Section 3.7). In fact, the workspace is just a tape of *mmType* variables and the size is the unique information it gets from the object of *AADCFunctions* type.

- `ws->setVal(x_arg, _mm256_set_pd(1.0, 2.0, 3.0, 4.0));
ws->setVal(y_arg, 2.0);
ws->setVal(z_arg, _mm256_set_pd(0.1, 0.2, 0.3, 0.4));`

We must set all *input* variables before calling the compiled forward kernel. For any *idouble* variable the compiler meets during the recording there is a corresponding *mmType* variable in the execution context. The variable is accessed via its index recorded before. In some cases it can be beneficial to cache variable indices to speed up the workspace access.

The size of the *idouble* type is the same as that of *double*. But variables in the execution context are of *mmType* so they can be considered as vectors of *double* type variables. Therefore, the produced AADC kernels are always vectorized by design and we can always use SIMD⁵ to compute the values of the function at multiple points in one pass.

Note that in the second line we invoke the command `ws->setVal` with the scalar value 2.0, this achieves the same thing as

```
ws->setVal(x_arg, mmSetConst<mmType>(2))
```

where `mmSetConst<mmType>(2)` creates a constant vector with all elements equal to 2.0.

- `aad_funcs.forward(*ws);`

This instruction calls the compiled forward kernel preallocating the execution context *ws*. The *mmType* vector `ws->val(f_res)` contains the final value of *f*.

3.1.2 Reverse kernel

We can now call the (automatically generated) reverse kernel to compute the partial derivatives (gradient) of *f* at the points

$$(x, y, z) = (1, 2, 0.1), (2, 2, 0.2), (3, 2, 0.3), (4, 2, 0.4)$$

(that we set using `setVal` and calculated the forward function at).

```
// prepare for the reverse pass, set f() equal to 1
ws->setDiff(f_res, 1.0);

// execute the reverse pass
aad_funcs.reverse(*ws);
```

⁵Single Instruction Multiple Data.

```

// now we have all requested derivatives and can print them out
// (below we print only the first element of the __m256d
// type)
for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi
) {
    std::cout
        << "avx[" << avxi << "]" "
        << " df/dx=" << ws->diffp(x_arg)[avxi] << "," << "df
/dy=" << ws->diffp(y_arg)[avxi] << "," << "df/dz=" << ws->
diffp(z_arg)[avxi]
        << std::endl;
    }
    std::cout << "Example 1 is done" << std::endl;
}

```

- `ws->setDiff(f_res, 1.0);`

For each variable with index *Ind* from the execution context there is a corresponding adjoint variable `ws->diff(Ind)`. Before starting the backward propagation, one should initialize adjoint variables that have value 0 by default.

- `aad_funcs.reverse(*ws);`

This instruction executes the backward AD algorithm. The *mmType* vector `ws->diff(x_ind)` contains the value of the partial derivative $\frac{\partial f}{\partial x}$ at the point (similarly for *y* and *z*).

The result of the execution is:

```

f(4,2,0.4)=11.0232. df/dx=5.51159; df/dy=-11.0232; df/dz
=11.0232
f(3,2,0.3)=6.04965. df/dx=3.02482; df/dy=-4.53724; df/dz
=6.04965
f(2,2,0.2)=3.32012. df/dx=1.66006; df/dy=-1.66006; df/dz
=3.32012
f(1,2,0.1)=1.82212. df/dx=0.911059; df/dy=-0.45553; df/dz
=1.82212

```

Note that the results are printed in the order opposite to the values we set using `setVal`.

3.2 Example 2. Black-Scholes

As a slightly more complicated example, we record the Black-Scholes formula giving the value of the European option. We then calculate the derivatives and compare them with the Finite Differences derivatives. The full code is given in [A.2](#). In the next section, we will record a Monte-Carlo simulation to estimate the same quantity and its derivatives, and compare the results.

Let $S(t)$ denote the price of the stock at time t and let K be the strike price. Then the payoff to the option holder at time T is equal to

$$(S(T) - K)^+ = \max\{0, S(T) - K\}.$$

To get the present value of the payoff we multiply by a discount factor e^{-rT} , where r is a continuously compounded interest rate. We denote the expected present value by

$$f(S(0), K, r, \sigma, T) = \mathbb{E}[e^{-rT}(S(T) - K)^+] \quad (2)$$

where \mathbb{E} stands for the expectation and the distribution of $S(T)$ is specified by the Black-Scholes model stochastic differential equation

$$\frac{dS(t)}{S(t)} = rdt + \sigma dW(t)$$

with W a standard Brownian motion and σ a volatility parameter. Let Φ denote the standard normal cumulative distribution function. The Black-Scholes formula for a call option⁶ then says that

$$f(S(0), K, r, \sigma, T) = S(0)\Phi\left(\frac{\log\left(\frac{S}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}\right) - e^{-rT}K\Phi\left(\frac{\log\left(\frac{S}{K}\right) + \left(r - \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}\right)$$

The code is pretty similar to that in Section 3.1. To be brief, we only print out the computed derivatives with respect to $S(0), K, r$ and calculate the Finite Differences derivatives with respect to $S(0)$ using the following function (here `BlackScholes(S0, K, r, vol, T)` stands for $f(S(0), K, r, \sigma, T)$):

```
//Computes the Finite Differences derivative
// with respect to the first variable S0
template<typename mdouble>
mdouble BlackScholesFD(
    mdouble S0, mdouble K, mdouble r, mdouble vol, mdouble T,
    mdouble h
) {
    return (BlackScholes(S0 + h, K, r, vol, T) - BlackScholes(S0,
        K, r, vol, T))/h;
}
```

Setting $h = 10^{-6}$ we indeed get the same derivatives. The result of the execution is:

```
BS(100,110,0.1,0.25,1)=10.1601. dBS/dS0=0.557155; dBS/dK
=-0.41414; dBS/dr=45.5554
BS(100,110,0.01,0.25,1)=6.53345. dBS/dS0=0.4144; dBS/dK
=-0.317332; dBS/dr=34.9066
BS(100,110,0.1,0.5,1)=19.9299. dBS/dS0=0.602329; dBS/dK
=-0.366391; dBS/dr=40.303
BS(100,110,0.1,0.5,2)=18.3297. dBS/dS0=0.626731; dBS/dK
=-0.403122; dBS/dr=83.2899
dBS/dS0(100,110,0.1,0.25,1)=0.557155
dBS/dS0(100,110,0.01,0.25,1)=0.4144
dBS/dS0(100,110,0.1,0.5,1)=0.602329
dBS/dS0(100,110,0.1,0.5,2)=0.626731
```

⁶See e.g. Paul Glasserman, *Monte Carlo Methods in Financial Engineering*, Section 1.1.2.

3.3 Example 3. Monte Carlo

We demonstrate how AADC can be applied to a simple Monte Carlo simulation. Let $f(S(0), K, r, \sigma, T)$ be given by (2) as in the previous example. To estimate this integral it is enough to do the following⁷:

- Generate independent normal variables Z_i , $i = 1, \dots, n$.
- Set $S_i(T) = S(0) \exp\left(\left(r + \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}Z_i\right)$.
- Set $C_i = e^{-rT}(S_i(T) - K)^+$.
- The integral is estimated by $\frac{C_1 + \dots + C_n}{n}$.

The full code for this example is given in A.3. We skip the kernel declaration and the call of the AADC compiler and start with the introduction of the *idouble* variables.

```
aadc::AADCArument random_sample_arg(random_sample.
markAsInputNoDiff());
aadc::AADCArument S0_arg(S0.markAsDiff());
aadc::AADCArument K_arg(K.markAsDiff());
aadc::AADCArument r_arg(r.markAsDiff());
aadc::AADCArument vol_arg(vol.markAsDiff());
aadc::AADCArument T_arg(T.markAsDiff());
```

In the previous examples we used the method `markAsInput()`. The user should apply this method to an *idouble* variable v when it is an argument of the forward function AND the computation of the derivative with respect to v is required. Otherwise, the following functions can be used (also see Section 8.1):

`markAsDiff()` explains to AADC that the value of v is fixed, and AADC will use this value everywhere in the object code for the F/R kernels.

`markAsInputNoDiff()` explains to AADC that we are not interested in the derivative with respect to v , so some arithmetic operations can be omitted.

The following records the mathematical operations corresponding to the instructions above:

```
idouble S_T = S0 * std::exp((r - vol*vol / 2) * T +
vol * std::sqrt(T) * random_sample);
idouble price = std::exp(-r*T) * std::max(S_T-K, 0.);

aadc::AADCResult price_res(price.markAsOutput());
aad_funcs.stopRecording();
```

The code below adjusts the number of Monte Carlo simulations depending on the size of the *mmType* vector.

```
int num_sim=1000000;
int count = aadc::mmSize<mmType>();
int num_sim_avx=(int)floor(num_sim/count);
```

The variable `integral` (`integral_S0`, `integral_K`, `integral_r`) will contain integral sums for $f(S(0), K, r, \sigma, T)$ (the derivatives with respect to $S(0), K, r$).

⁷Again, see Paul Glasserman, *Monte Carlo Methods in Financial Engineering*, Section 1.1.2.

```

mmType integral_S0(mmSetConst<mmType>(0)), integral_K(
mmSetConst<mmType>(0)),
integral_r(mmSetConst<mmType>(0)), integral(mmSetConst<mmType>
>(0));

```

We then initialize the loop with `num_sim_avx` iterations. Note that the code does not depend on the iterator `i`.

```

for (int i=0; i<num_sim_avx; ++i) {

    double* _x = (ws->valp(random_sample_arg));
    for (int ci=0; ci<count; ++ci) {
        _x[ci]= norm_distrib(gen);
    }

    aad_funcs.forward(*ws);
}

```

In the internal loop, we generate normally distributed numbers and assign to the corresponding entries of the `mmType` vector `ws->valp(random_sample_arg)`.

In the following code the variable adjoint to f is initialized as 1 for each entry of the vector. Note that after the backward pass, the vector `ws->diff(S0_arg)` (similarly for K and r) will contain the entries corresponding to those of `ws->valp(random_sample_arg)`.

```

ws->resetDiff();
ws->setDiff(price_res, 1.0);

aad_funcs.reverse(*ws);
integral_S0=mmAdd(ws->diff(S0_arg), integral_S0);
integral_K=mmAdd(ws->diff(K_arg), integral_K);
integral_r=mmAdd(ws->diff(r_arg), integral_r);

```

The instruction `integral_S0=mmAdd(ws->diff(S0_arg), integral_S0);` calls the intrinsic processor realizing coordinate-wise addition (see Section 8.2). It can be replaced by a slower loop:

```

for (int ci=0; ci<count; ++ci) {
    partialSum+=ws->diffp(S0_arg)[ci];
}

```

Finally, we print out the results:

```

std::cout << "f(100,110,0.1,0.25,1)=" << aadc::mmSum(
integral) / num_sim <<
". df/dS0=" << aadc::mmSum(integral_S0) / num_sim << "; df/
dK=" <<
aadc::mmSum(integral_K) / num_sim << "; df/dr=" <<
aadc::mmSum(integral_r) / num_sim << std::endl;

```

Note that we are using `mmSum()` which is another intrinsic function returning the sum of the components (see Section 8.2). Also we normalize the integral sums with factor the full number of iterations `num_sim` (since for each value of the external iterator `i` there are `count` internal summations). The result of the execution is:

```
f(100,110,0.1,0.25,1)=10.1724. df/dS0=0.55731; df/dK
=-0.414169; df/dr=45.5586
```

Note that it is very close to the first line (one with the same inputs) of the result obtained by recording the Black-Scholes formula in the previous example.

3.4 Example 4. Jacobian

We give an example of how to compute the Jacobian (matrix of partial derivatives of coordinate-wise functions) of a function

$$f : \mathbb{R}^3 \rightarrow \mathbb{R}^2$$

at two points in one pass. Unlike the previous examples, we will now have two outputs. Let f be given by

$$\begin{cases} f_1 = -3xy + \frac{16x^2}{z} - \frac{xy}{(1+xy)} \\ f_2 = x^2y + f_1 \end{cases}$$

We will calculate the full Jacobian at the points $(x, y, z) = (1, 2, 4), (1, 3, 5)$. From (1) one can get the full Jacobian by consecutively applying the reverse kernel with a choice of the unique non-zero adjoint variable \bar{f}_i (in this example $i = 1$ or $i = 2$) and setting all the other \bar{f}_j , $j \neq i$ to zero. See the full code for this example in Section A.4, we will comment in notable places only.

- ```
ws->setVal(x_arg, mmSetConst<mmType>(1));
ws->setVal(y_arg, _mm256_set_pd(2.0, 2.0, 3.0, 3.0));
ws->setVal(z_arg, _mm256_set_pd(4.0, 4.0, 5.0, 5.0));
```

Both points  $(1, 2, 4)$  and  $(1, 3, 5)$  have their first coordinates equal to 1 which corresponds to the instruction in the first line<sup>8</sup>. The 1st and 2nd component of the *mmType* work space variables will serve the computation of  $f_1$ ,  $f_2$  and their derivatives at the point  $(x, y, z) = (1, 2, 4)$  while the 3rd and 4th refer to the point  $(x, y, z) = (1, 3, 5)$ .

- ```
ws->setDiff(f1_res, _mm256_set_pd(1.0, 0.0, 1.0, 0.0));
ws->setDiff(f2_res, _mm256_set_pd(0.0, 1.0, 0.0, 1.0));
```

If the forward kernel modifies any input variables, then one should set them again before each call of the reverse kernel. In the first line we set $\bar{f}_1 = 1$ and $\bar{f}_2 = 0$ and in the second line $\bar{f}_1 = 0$ and $\bar{f}_2 = 1$.

- ```
for (int i=0; i<4; ++i) {
 int dfi = (i < 2 ? i+1 : i-1);
 std::cout << " f1(" <<ws->valp(x_arg)[0]
 << ", " << ws->valp(y_arg)[i]
 << ", " << ws->valp(z_arg)[i]
 << ")=" << ws->valp(f1_res)[i]
 << " f2(" <<ws->valp(x_arg)[0]
 << ", " << ws->valp(y_arg)[i]
```

---

<sup>8</sup>Note that we could have just written `ws->setVal(x_arg, 1.0)`. instead as in Sections 3.1, 3.2.

### 3.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

```

 << ", " << ws->valp(z_arg)[i]
 << ")=" << ws->valp(f2_res)[i]
 << ". df" << dfi << "/dx=" << ws->diffp(x_arg)[i]
]
 << "; df" << dfi << "/dy=" << ws->diffp(y_arg)[i]
]
 << "; df" << dfi << "/dz=" << ws->diffp(z_arg)[i]
]
 << std::endl;
}

```

We print out the derivatives. Note that for each execution only one of the adjoint variables is nonzero.

The result of the execution is:

```

f1(1,3,5)=-6.55 f2(1,3,5)=-3.55. df1/dx=3.2125; df1/dy=-2.0625;
df1/dz=-0.64
f1(1,3,5)=-6.55 f2(1,3,5)=-3.55. df2/dx=-2.7875; df2/dy=-3.0625;
df2/dz=-0.64
f1(1,2,4)=-2.66667 f2(1,2,4)=-0.666667. df1/dx=5.77778; df1/dy
=-2.11111; df1/dz=-1
f1(1,2,4)=-2.66667 f2(1,2,4)=-0.666667. df2/dx=1.77778; df2/dy
=-3.11111; df2/dz=-1

```

### 3.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

In this section we provide a full working example with *ibool* and *iint* objects involved. We use the mathematical method of *Interpolation* for this purpose. We calculate

$$F(v_1, v_2, v_3) = \mathbb{E}f(\xi), \quad \text{where } \xi \sim N(0, 1)$$

is the normal random variable with mean 0 and variance 1 and  $f$  is a piecewise constant function

$$f = \begin{cases} v_3, & 3 < x < \infty \\ v_2, & 1 < x < 3 \\ v_1, & x < 1 \end{cases}$$

We introduce a class *PiecewiseCurve* to model general piecewise constant functions. It is initialized with the set of nodes.

```

#include <ibool.h>
#include <iint.h>

template<typename mdouble>
class PiecewiseCurve {
public:
 PiecewiseCurve(const std::vector<double>& nodes)
 : m_s(nodes)
 {}
}

```

```

void init(const std::vector<mdouble>& vals) {
 m_vals.resize(m_s.size());
 for (int i=0; i<m_vals.size(); i++) {
 m_vals[i]=vals[i];
 }
}

mdouble interpolatedValue (const mdouble& s) {
 auto i_s = lower_bound(m_s, s);
 return get(m_vals,i_s);
}

std::vector<mdouble> m_vals;
std::vector<double> m_s;
};

```

We are interested in derivatives with respect to  $v_i$ . Therefore these variables are of *idouble* type and the method `interpolatedValue` should use an overloaded version of the `lower_bound`. It returns the value of *int* type. One has to use the function `get(m_vals,i_s)`<sup>9</sup> to get the element `i_s` from the array `m_vals`. It is important to note that both `get()` and `lower_bound()` also have overloading for native types. So the templated code can be used in AAD and non-AAD cases. The full code for this example is given in Section A.5.

### 3.6 Example 6. Incremental Checkpointing

The natural realization of the AD algorithm assumes computation of intermediate differentials during the forward pass and their further use during the reverse pass. Strictly adhering to this plan can lead to high requirements for tape storage. A checkpoint is a full or partial record of the program state at a particular point of execution  $t_0$  which allows recovering later differentials starting the calculations from  $t_0$ . Since the 80s<sup>10</sup> checkpointing has been a subject of interest for researchers who were looking for a compromise between time and memory.

The AAD-Compiler uses a proprietary checkpointing design. Users can mark checkpoints using the command `idouble::CheckPoint()`; Reasonable places for checkpoints correspond to local minimums of the currently active variables. Practically, a well-designed program uses checkpointing to separate large and almost independent parts of the code.

To show the effectiveness of the AADC checkpointing tool, we consider the following code. The full example is given in Section A.6.

```

idouble x(0),y(1),z,f;
int x_ind(x.markAsInput()), y_ind(y.markAsInput());

aad_funcs.startRecording();
for (int i=0; i<=100; i++) {
 if (i%10 == 0) idouble::CheckPoint();
}

```

<sup>9</sup>An analogue of `m_vals[i_s]`.

<sup>10</sup>e.g. Volin, Y. M., Ostrovskii, G. M, *Automatic computation of derivatives with the use of the multilevel differentiating technique-1*, Algorithmic basis, Computers & mathematics with applications, (1985) (11), 1099–1114.



### 3.7 Example 7. Multithread for a simple MonteCarlo

```

for (int j=1; j<100; j++) {
z=exp(log(x*y)/sqrt(2));
y=exp(log(x/y)/sqrt(2));
x=z;
}
}
f=x;
int f_ind(f.markAsOutput());
aad_funcs.stopRecording();

std::shared_ptr<aadc::ADCWorkspace<mmType>> ws(aad_funcs.
createWorkspace());

cout << "*****" << std::endl;
cout << "Code size forward : " << aad_funcs.getCodeSizeFwd() <<
"\n";
cout << "Code size reverse : " << aad_funcs.getCodeSizeRev() <<
"\n";
cout << "Work array size : " << aad_funcs.getWorkArraySize() <<
"\n";
cout << "Stack size : " << aad_funcs.getStackSize() <<
"\n";
cout << "Const data size : " << aad_funcs.getConstDataSize() <<
"\n";
cout << "Checkpoint size : " << aad_funcs.getNumCheckPointVars
() << "\n";
cout << "*****" << std::endl;

```

Here is the difference between the runtime stack sizes with and without checkpointing:

| With Checkpointing          | Without Checkpointing       |
|-----------------------------|-----------------------------|
| *****                       | *****                       |
| Code size forward : 600928  | Code size forward : 597584  |
| Code size reverse : 1083661 | Code size reverse : 1080325 |
| Work array size : 16        | Work array size : 16        |
| Stack size : 5940           | Stack size : 59994          |
| Const data size : 235       | Const data size : 235       |
| Checkpoint size : 11        | Checkpoint size : 0         |
| *****                       | *****                       |
| f(3,2)=3.55006              | f(3,2)=3.55006              |
| df/dx=0.836758              | df/dx=0.836758              |
| df/dy=1.25514               | df/dy=1.25514               |

### 3.7 Example 7. Multithread for a simple MonteCarlo

Here we do a multithread version of the Monte Carlo simulation (see Section 3.3) for a simple function  $f(x)$ . The full code is given in Section A.7 and a smiliar example computing European Option pricing can be found in Section A.10. We will calculate

### 3.7 Example 7. Multithread for a simple MonteCarlo

$F(a=2)$ ,  $F'(a=2)$ , where  $F(a) = \int_0^1 f(x)dx$ , and  $f(x)$  is given by

$$f(x) = \begin{cases} 0, & x \leq 0.00001 \\ \frac{e^{ax}}{\sqrt{x}} - 1, & x > 0.00001 \end{cases}$$

The main difference when using multithreading is that the user should create various independent execution contexts. Namely, the user should first realize *recording* creating an object of the *AADCFunctions* class and then create *WorkSpace* objects inside each thread.

The number of iterations inside each thread should reduce due to simultaneously using the SIMD and AVX technologies. For example, for an AVX2 processor with 4 cores the number of iterations in each thread is 16 times smaller than the initial number of simulations:

```
int num_sim=4000; // Divisible by (count*treads)
int avx_count = aadc::mmSize<mmType>();
int num_treads=4;
int num_sim_avx=num_sim/(avx_count*num_treads);

template<typename mdouble>
mdouble funMath(const mdouble& x, mdouble& a) {
 return iIf (x<0.00001, 0, std::exp(a*x)/std::sqrt(x));
}
```

Note that the definition of the *funMath* function contains the operator *iIf* which should always be substituted instead of the usual conditional operator *if* when its condition depends on any variable marked as *Input*, see Section 8.6.3.

Each thread should have access to the following objects:

- The compiled F/R kernels.
- Indices of the *idouble* variables.
- References to the variables, the partial sums of which are recorded.

To minimize the number of objects for transmission we define the function to be executed in each thread via a lambda-expression inside the body of the main function:

```
void example_multi_thread()
{
 idouble aad_x, aad_a(2), aad_f;

 double d_integral(0), integral(0);

 aadc::AADCFunctions<mmType> aad_funcs;

 aad_funcs.startRecording();
 aadc::AADCArument x_index = aad_x.markAsInputNoDiff();
 aadc::AADCArument a_index = aad_a.markAsDiff();

 aad_f=funMath(aad_x,aad_a);

 aadc::AADCArument f_index = aad_f.markAsOutput();
```

### 3.7 Example 7. Multithread for a simple MonteCarlo

```

 aad_funcs.stopRecording();

 std::vector<double> integral_c(num_treads), d_integral_c(
num_treads);

 auto threadWorker = [&] (
 double &integral,
 double &d_integral,
 const int i
) {
 std::mt19937_64 gen;
 std::uniform_real_distribution<> uni_distrib(0.0,
1.0);
 gen.seed(i*17+31);

 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(
aad_funcs.createWorkspace());

 mmType integral_mm(mmSetConst<mmType>(0)),
d_integral_mm(mmSetConst<mmType>(0));

 for (int i=0; i<num_sim_avx; ++i) {

 mmType rand_mm;
 double* _rand_mm = toDblPtr(rand_mm);
 for (int ci=0; ci<avx_count; ++ci) {
 _rand_mm[ci]= uni_distrib(gen);
 }
 ws->setVal(x_index, rand_mm);

 aad_funcs.forward(*ws);
 integral_mm = aadc::mmAdd(ws->val(f_index),
integral_mm);

 ws->resetDiff();
 ws->setDiff(f_index, 1.0);

 aad_funcs.reverse(*ws);
 d_integral_mm = aadc::mmAdd(ws->diff(a_index),
d_integral_mm);
 }

 d_integral = mmSum(d_integral_mm);
 integral = mmSum(integral_mm);
 };

```

First, the *threadWorker* defines a random generator's seed as a function of the thread's number. It then creates a *Workspace* for it that is separated from the other *WorkSpaces*.

There is an important detail to note in the final part of the code above. Each thread keeps intermediate partial sums in the corresponding cells of arrays *integral\_mm* and *d\_integral\_mm*. These values will be summed up only when all threads complete their work. Each thread calls its independent copy to compute the partial sums:

```
std::vector<std::unique_ptr<thread>> threads;
```

### 3.8 Example 8. Interfacing AADC to a minimization library

```

for(int i=0; i< num_treads; i++) {
 threads.push_back(
 std::unique_ptr<thread>(
 new thread(
 threadWorker
 , std::ref(integral_c[i])
 , std::ref(d_integral_c[i])
 , i
)
)
);
}
for(auto&& t: threads) t->join();

for(int i=0; i< num_treads; i++) {
 integral+=integral_c[i];
 d_integral+=d_integral_c[i];
}

integral/=num_sim;
d_integral/=num_sim;

std::cout << "F(2)=" << integral << std::endl;
std::cout << "F'(2)=" << d_integral << std::endl;
}

```

The result of the execution is:

```

F(2)=4.70829
F'(2)=2.47285

```

### 3.8 Example 8. Interfacing AADC to a minimization library

Suppose we want to find the vector  $v$  minimizing the function  $f(v)$  that is given by some large algorithm coded using the AADC library. In this section, we illustrate how it can be interfaced to a third-party minimization algorithm with a C++ interface, we will use LBFGS++ for this example. We use a simple parabolic function

$$f(x, y) = (x - 1)^2 + (y - 2)^2,$$

for the demonstration with the initial guess the origin  $(x, y) = (0, 0)$ . The full code is given in Section A.8.

The external library requires a function computing the algorithm and the gradient of the function. So the compilation and execution of the forward and reverse kernels should be realized separately. For this purpose we declare *AADCFunctions* and *AADCWorkspace* to be global objects (it is more natural to realize them in a *class*, see a more detailed example of minimization in Section 7.2):

```

using Eigen::VectorXd;
using namespace LBFGSpp;

```

### 3.8 Example 8. Interfacing AADC to a minimization library

```
using namespace aadc;

typedef __m256d mmType;
std::shared_ptr<aadc::AADCFunctions<mmType> > aad_funcs;
std::shared_ptr<aadc::AADCWorkspace<mmType> > ws;
aadc::AADCArument x_ind,y_ind;
aadc::AADCResult f_ind;

void record () {
 aad_funcs = std::make_shared<aadc::AADCFunctions<mmType>
>();

 idouble x,y,f;

 aad_funcs->startRecording();
 x_ind=x.markAsInput();
 y_ind=y.markAsInput();

 f=(x-1)*(x-1)+(y-2)*(y-2);

 f_ind=f.markAsOutput();
 aad_funcs->stopRecording();

 ws = aad_funcs->createWorkspace();
}
```

Variables `x_ind`, `y_ind`, `f_ind` keep all *idouble* indices to be used as references to the workspace variables and avoid using the *varIndex()* method.

The following code defines a function that gives both the value and the full gradient (see the first part of the LBFGSpp demonstration example <http://yixuan.cos.name/LBFGSpp/doc/index.html>):

```
double calc_grad(const VectorXd& x, VectorXd& grad) {

 ws->setVal(x_ind,x[0]);
 ws->setVal(y_ind,x[1]);

 (*aad_funcs).forward(*ws);

 ws->resetDiff();
 ws->setDiff(f_ind, 1.0);

 (*aad_funcs).reverse(*ws);

 double *_dx = (ws->diffp(x_ind));
 double *_dy = (ws->diffp(y_ind));
 double *_f = (ws->valp(f_ind));

 grad[0] = _dx[0];
 grad[1] = _dy[0];
 return _f[0];
}
```

```
}
```

Now we just need to set up parameters, create a solver object, provide an initial guess, and run the minimization function (see <http://yixuan.cos.name/LBFGSpp/doc/index.html> again):

```
void example_minimization()
{
 // Set up parameters
 LBFGSParam<double> param;
 param.epsilon = 1e-6;
 param.max_iterations = 100;
 // Create the solver object

 LBFGSSolver<double> solver(param);
 record();

 // Initial guess
 VectorXd x = VectorXd::Zero(2);
 // x will be overwritten to the best point found
 double fx;
 int niter = solver.minimize(calc_grad, x, fx);

 std::cout << niter << " iterations" << std::endl;
 std::cout << "x = \n" << x.transpose() << std::endl;
 std::cout << "f(x) = " << fx << std::endl;
}
```

The result of the execution is

```
2 iterations
x =
1 2 0
f(x) = 2.46519e-31
```

Indeed the corresponding vector minimizes the function. Note that CPU vectorization isn't utilized here but it comes in handy in more complex examples, see Section 7.2.

## 4 Using Eigen during the AADC-recording (introduces Examples 15 and 16)

To correctly use the Eigen library one should include `<aadc/aadc_eigen.h>` that defines `Eigen::NumTraits<idouble>`. The most important thing to ensure is that Eigen knows that the copy-constructor is obliged to call the `idouble`-constructor. After that most of the Eigen functionality like matrix multiplications FFT will work correctly. Some Eigen routines like matrix-inversion involve algorithm branching and those can be treated either using the pragma `AADC_ALLOW_TO_PASSIVE_BOOL` as explained in the next section or using the external functions interface.

We have made Examples A.15 and A.16 to illustrate how to use Eigen with AADC. Example A.15 records the operation `(A+B).norm()` where  $A, B$  are of type `Eigen::Matrix<idouble, Dynamic, Dynamic>` and then performs the operation and computes

adjoints using the forward and reverse AADC kernels. Example A.16 uses the `unsupported/Eigen/FFT` module to compute the Fast Fourier Transform and its derivatives. In particular, it demonstrates the use of `std::complex<idouble>`.

## 5 Integration strategies

We consider various aspects of using AAD with Operator Overloading in practice. The goal is to make the program support both active and native versions of the code with minimal C++ source code changes. There are many ways one can try to achieve that and in this section we suggest those that we have extensively tested already. One approach is using *Function Templates* which have appeared in Section 3 several times<sup>11</sup>. This approach is relatively easy to implement but has several disadvantages. Templates may require extensive code changes and can significantly affect the compilation time.

In Section 5.1 we discuss a different approach which is more practical for real-life projects. Namely, we redefine *double* as *idouble*. Using this approach AADC has been successfully integrated into large opensource projects like QuantLib<sup>12</sup> and ORE<sup>13</sup>.

In Section 5.2 we show how to reuse (during run-time) the original *double* objects in the AADC-enabled code.

### 5.1 Redefining *double* as *idouble* (based on Example 11)

The *idouble* variables occupy the same memory as their *double* counterparts. Therefore, the only difference between running the original program and the program with *double* substituted by *idouble* is that we need to check that at each mathematical operation the recording is on. That slows down the execution of the whole project by up to a factor of 2. But the recording happens only once and the recorded kernels are executed many times, so that is very reasonable.

All the required code modifications concern branchings only and we demonstrate how to proceed using the example `OptionPricing.cpp` of Section A.10. It is a synthetic example that prices options of three types: European, Asian, and Digital. The full code of the required modifications is given in Section A.11.

As we explained earlier in Section 3.7, the operator *if* is not defined for the active boolean type, and it should be substituted by the operator *if*. In fact, the substitution is needed only when the condition is *stochastic*, that is when it depends non-trivially on the variables marked as *input*. AADC gives the option of implicitly casting *ibool* to *bool* as follows.

Define the pragma `AADC_ALLOW_TO_PASSIVE_BOOL` before including `aadc.h`:

```
#define AADC_ALLOW_TO_PASSIVE_BOOL
#include <aadc/aadc.h>
```

Using `min` / `max` like in the following

---

<sup>11</sup>Also see our XVA benchmark at <https://matlogica.com/xva-benchmark-aug-2020/> and the link to GitHub in there.

<sup>12</sup><https://www.quantlib.org/>.

<sup>13</sup><https://www.opensourcerisk.org/>.

```
if (payoff_type == optionTypes::European) one_path_price = max
(asset-strike, vtype(0.));
```

does not require any changes to work correctly because the overloading of *max* and *min* uses *iIf*.

Similarly, the following code (that checks if the simulation date should be taken into account when averaging the Asian payoff) will work correctly.

```
if (payoff_type == optionTypes::Asian && std::count(
 averaging_base_times.begin(), averaging_base_times.end(),
 simulation_times[t_i+1])) {
 one_path_price += asset;
}
```

This works because *times*, though an active type, does not depend on the marked variables *rate*, *asset* and *vol* in this example. Now consider the line:

```
if (strike>asset && payoff_type == optionTypes::Digital)
 one_path_price = 1;
```

where the condition *strike>asset* is stochastic. The compilation terminates without errors, but executing gives a wrong result since the value of *strike>asset* becomes constant for all runs of MLKernels. It should be changed to:

```
if (payoff_type == optionTypes::Digital) one_path_price = iIf (
 asset>strike), 1 , 0);
```

### 5.1.1 Automatically locating stochastic *ibool-to-bool* conversions

During the execution, the AADC library will detect all instances of

stochastic *ibool*  $\rightarrow$  *bool*

conversions and the number of such conversions can be obtained using the method `printPassiveExtractLocations` as follows:

```
aad_funcs.printPassiveExtractLocations(std::cout, "
OptionPricingWithoutTemplates");
```

If the obtained number is zero, the MLKernel will work as expected. But if the number of conversions is positive, one should recompile the code in the debug regime using the following option of `AADCFunctions`:

```
aadc::AADCFunctions<mmType> aad_funcs({
 {AADC_BreakOnActiveBoolConversion, 1}
});
```

This option forces the debugging process to stop each time when an *ibool-to-bool* conversion is encountered. Recall<sup>14</sup> that *builtin operators* `&&` and `||` perform short-circuit evaluation (do not evaluate the second operand if the result is known after evaluating the first), but overloaded operators behave like regular function calls and always evaluate both operands. So calls of AND and OR with stochastic arguments should get

<sup>14</sup>[https://en.cppreference.com/w/cpp/language/operator\\_logical](https://en.cppreference.com/w/cpp/language/operator_logical)



## 5.2 Objects in mixed (native/AAD) mode (based on Example 9)

a special attention from the user. For this reason, there is no *ibool* overloading in the AADC namespace for these operators, that is

```
one_path_price = iIf (asset>strike && payoff_type ==
optionTypes::Digital, 1 ,0);
```

will cause a compilation error. Instead, we suggest analysing such a call and using the `aadcBoolOps` namespace:

```
{
 using namespace aadcBoolOps;
 one_path_price = iIf (asset>strike && payoff_type ==
optionTypes::Digital, 1 ,0);
}
```

or the smoothed analog if derivatives are required:

```
if (payoff_type == optionTypes::Digital) one_path_price = (1+
tanh((asset-strike)*1e+1))/2;

if (payoff_type == optionTypes::European) one_path_price =
max(asset-strike,vtype(0.));
if (payoff_type == optionTypes::Asian) one_path_price /=
simulation_times.size();
return one_path_price;
```

## 5.2 Objects in mixed (native/AAD) mode (based on Example 9)

In this section, we demonstrate one very interesting feature of the AADC library: the ability to mix AAD and non-AAD modes. This is very useful in real-world systems where there are numerous objects in the memory and AAD is needed only at certain points in the execution. Unlike other AAD libraries that require recreating all AAD-compatible objects every time AAD is required, AADC supports changing the mode from native to AAD (and vice versa) for objects already in memory.

That is done via “casting” and is possible by the design of the *idouble* class in the AADC library. We’ve put in special effort to make the active *idouble* type contain exactly one *double* data member. Note that casting to the active type is not enabled in any of the currently existing Operator Overloading AAD tools. That is because normally active type objects contain IDs to track valuation dependencies in addition to the native *double* data member.

We explain how to mix AAD and non-AAD modes. The full code is given in [Section A.9](#).

### 5.2.1 Segregating active and native code using namespaces

We define a `MyAnalytics` class in the corresponding `.cpp/.h` file. Similarly to the function template approach the definition of `MyAnalytics` uses a special parameter type *mdouble* that should be substituted by *idouble* to enable AAD or by *double* to use the native mode.

`MyAnalytics.h` can be defined using the following scheme:

## 5.2 Objects in mixed (native/AAD) mode (based on Example 9)

```
#ifndef _MY_ANALYTICS_H
#define _MY_ANALYTICS_H

#include "idouble.h"

namespace MyApp {

 class MyAnalytics {
 // Declaring class members (uses mdouble when necessary).
 }
};

#ifndef AADC_PASS
#define AADC_PASS
#undef _MY_ANALYTICS_H
namespace AAD {
 #include "MyAnalytics.h"
};
#undef AADC_PASS
#endif

#endif // _MY_ANALYTICS_H
```

We perform two passes of the `MyAnalytics.h` file. The *mdouble* type is a typedef to *double* in the global namespace and `AAD::mdouble` is defined as *idouble* in the AAD namespace. This results in two versions of the `MyAnalytics` class. One is the native mode in the `MyApp` namespace and the other is the active mode in the `AAD::MyApp` namespace. The corresponding `.cpp` file follows a similar structure:

```
#include "MyAnalytics.h"

namespace UserNameSpace {

 // Complete definitions for class members (uses mdouble when
 // necessary).

};

#ifndef AADC_PASS
#define AADC_PASS
namespace AAD {
 #include "MyAnalytics.cpp"
};
#undef AADC_PASS
#endif
```

Then the non-AAD native class is referred to as `MyApp::MyAnalytics` and the AAD active class is referred to as `AAD::MyApp::MyAnalytics`. Next, we can either create a new object of the corresponding active type or simply change the type of the existing object.

```
AAD::MyApp::MyAnalytics& aad_my_analytics (*(AAD::MyApp::
MyAnalytics*)&my_analytics));
```

### 5.2.2 Handling virtual functions

The previous section demonstrates the mixing mode capability since the `aad_my_analytics` object is the result of casting `my_analytics`. This approach works correctly for all objects referenced within the `MyAnalytics` class since after the initial casting all objects (existing or intermediate) are referenced from the AAD namespace.

However, it does not work for dynamic classes (i.e. classes with virtual functions). The reason is that the addresses for the virtual functions are stored as part of the objects themselves at the object construction step. This means that calling a virtual method on an object created in the native mode at the AAD recording stage will actually call the native method instead of the AAD counterpart. This results in losing track of adjoints or even a program crash<sup>15</sup>.

The solution is to patch the virtual function tables at runtime to enable switching the objects between modes. AADC provides a simple user interface for this process that we overview now. We give a formal recipe and then show how to apply it in the case when `MyAnalytics` contains a virtual method. See Section A.9 for the full code of this slight modification of the previous example.

- Add inheritance from the public virtual `aadc::VReg` class. This can be done using the `AADC_VREG_BASE` macro.
- Add `AADC_VREG_OBJ_ADD` to the object constructor. It will aid linking of AAD and non-AAD types and only consumes CPU cycles once per type (not per object created).
- Add the virtual table switch call `AADC_VREG_SWITCH`. This macro adds a virtual function to perform the AAD mode switch.

Here is the comparison of the two portions of the code.

| User code                                                                                                                                                                                    | Modified user code                                                                                                                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>template&lt;typename mdouble&gt; class MyAnalytics { public:      virtual mdouble val         (const mdouble&amp; a) {         return (a+1)/a;     };      mdouble var1, var2; };</pre> | <pre>template&lt;typename mdouble&gt; class MyAnalytics : AADC_VREG_BASE { public:     MyAnalytics() { AADC_VREG_OBJ_ADD }     AADC_VREG_SWITCH;      virtual mdouble val(const mdouble&amp; a) {         return (a+1)/a;     };      mdouble var1, var2; };</pre> |

Now, establish a link between the two versions of the `MyAnalytics` class calling  
`aadc::VReg::AddLink<MyAnalytics<idouble>, MyAnalytics<double>>();`

---

<sup>15</sup>If the reader adds the keyword *virtual* before the declaration of any method in `MyAnalytics.h`, it may lead to an incorrect result.

The code can then assume two modes: normal or AAD. In the normal mode the user can call the original *double* methods and the AAD mode uses the *idouble* counterparts. The mode switch can be called any number of times with the command `aadc::VReg::SwitchAll();`. Note that for now AADC only supports the default constructible classes.

## 6 Troubleshooting (based on Example 12)

One can localize any unexpected behavior of the forward and reverse AADC kernel outputs using the internal debugger. Practice shows that in most situations it is enough to check that:

- The number of idouble-to-double conversions is zero (see Section 5.1.1).
- Values of the *outputs* are set in the Workspace before the `.reverse()` call. Very often users fail to set the values of ALL (including intermediate) variables marked as outputs.

The general procedure to locate the source of non-expected behavior is as follows.

- Check that the forward kernel gives the same result as the original program.
- In case of repetitive calls of the kernel, identify a call (e.g. the number of Monte-Carlo paths) for which the wrong result is achieved.
- Using the AADC-Debug tool, localize the problem by doing one of the following:
  - Wrap the variables with `AADC_PRINT` by hand.
  - Use the automatic tools to run the commands `export AADC_INSTRUMENT_CODE=1` and `export AADC_DEBUG_STOP_RECORDING_AT=X`.

We will learn by example debugging the `OptionPricing.cpp` of Section A.10. The full code for the modifications discussed below can be found in Section A.12.

### 6.1 AADC Debug Tool

Debugging the AADC kernels has two main parts:

- Running the technical AADC tools and interpreting their output.
- Locating the source of the wrong result.

In order to enable the AADC debugger:

- Add `#include <aadc/aadc_debug.h>`
- Choose the point in the code and the variable you want to scrutinize. Wrap this variable<sup>16</sup> using the method `AADC_PRINT`, as shown below:

```
idouble payoff = onePathPricing(
 aad_asset, strike, t, rate_obj, vol_obj,
 aad_random_samples
);
payoff=AADC_PRINT(payoff);
aadc::AADCRresult payoff_arg(payoff.markAsOutput());
```

<sup>16</sup>`AADC_PRINT` can also be used to wrap arbitrarily many variables.

```
aad_funcs.stopRecording()
```

- At the execution stage add a call of the debugging method `debugTestAdjointsUsingFD` after the calls of `.forward()` and `.reverse()` (if applicable):

```
if (mc_i==0) debugTestAdjointsUsingFD(std::cout, *ws,
 aad_funcs, payoff_arg, true, true);
```

If one omits the `if (mc_i==0)` above, then the report for each MC path will be printed out and that is generally not useful. For the same reason, in a multithread environment, it is prudent to set the number of threads equal to one.

We declare the method `debugTestAdjointsUsingFD` as follows:

```
void debugTestAdjointsUsingFD(
 stream& logs
 , aadc::AADCWorkspace<mmType>& ws
 , const aadc::AADCFunctions<mmType>& aadc_func
 , aadc::AADCResult result
 , bool check_fwd_vs_rec = false
 , bool print_all = false
)
```

The penultimate argument `check_fwd_vs_rec` is required if one needs to print out the results of the recording stage. If the last argument `print_all` is set to false, the report appears only for the variables whose AADC adjoint values do not coincide with their Finite Differences (F.D.) adjoints.

The result of the execution is:

```
/aadc/build$./examples/Manual/manual 12
Total debug prints : 2
AADCDebug::0000000000::Adj[0]:FD_mid 1.000000000000034 [rv,adj]
AADCDebug::0000000000::Adj[0]:FD_left 1.000000000000165 [rv,adj]
AADCDebug::0000000000::Adj[0]:FD_right 1.000000000000165 [rv,adj]
AADCDebug::0000000000::Adj[0]:aadc_adj 1 [rv,adj]
AADCDebug::0000000000::Val[0]:aadc_val 7.66426894809624 [rv,adj]
AADCDebug::0000000000::Val[0]:rec_val 7.66426894809624 [rv,adj]
{payoff} ./aadc/examples/Manual/optionPricingDebug.cpp:128
```

We now explain how to read this report:

- 00000000 - variable index attributed during the recording stage.
- payoff - the name of the wrapped variable.
- ::Adj[i] - the  $i$ -th component of the adjoint variable AVX vector.
- ::Val[i] - the  $i$ -th component of the forward variable AVX vector.
- :rec\_val - the value at the recording stage.
- :aadc\_val - the value computed by the AADC forward kernel.
- :aadc\_adj - the value computed by the AADC reverse kernel.
- [rv] - appears only if the value depends on variables marked as inputs.
- [adj] - appears only if the computation of an adjoint is required for this variable, i.e. the variable depends on variables marked as Diff.

F.D derivatives are computed using forward kernels and 6-th order polynomials<sup>17</sup> as functions of a small translation  $h$ . The default value  $h = 1e - 5$  can be modified in the `aadc_debug.h` if necessary.

## 6.2 Localization of Not-a-Number (NaN) and other discrepancies

There are several typical reasons why NaNs can appear as outputs of the forward or the reverse kernel. NaN in the forward values usually appear because the result of any arithmetic operation with NaNs is also NaN. For example, the code `x=iIf (v==0,x , x/v)` corresponds to the following SIMD-consistent algebraic form

$$x = x * (v == 0) + x/v * !(v == 0)$$

If  $v == 0$ , then the result of this instruction is NaN since  $\text{NaN} * 0 = \text{NaN}$ . NaN values can also appear in adjoints. For example, that can happen due to multiplication by  $\sqrt{0}$ . Such situations can be checked automatically with the following procedure.

Execute `export AADC_INSTRUMENT_CODE=1` in the terminal<sup>18</sup>. This flag means that the result of each mathematical operation is automatically wrapped with `AADC_PRINT`. From now on we recommend using the flag `print_all = false`

```
if (mc_i==0) debugTestAdjointsUsingFD(std::cout, *ws, aad_funcs,
 payoff_arg, true, false);
```

to print only the variables with observed discrepancies.

In particular, when the result of each mathematical operation is automatically wrapped with `AADC_PRINT` (that is `AADC_INSTRUMENT_CODE=1`), the C++-debugger (e.g. `gdb` or `lldb`) will stop at each code point causing a NaN value at the recording stage. Consider the following example:

```
120 idouble payoff=aad_rate;
121 payoff=payoff/0;
122 payoff=onePathPricing_debug(aad_asset,
 strike, t, rate_obj, vol_obj,
 aad_random_samples);
```

This modification doesn't influence the forward result since the variable is reassigned in the next line. But it influences the adjoint variables due to the mentioned property that  $\text{NaN} * 0 = \text{NaN}$ . AADC will stop the debugger each time it meets a NaN value during the recording.

The recording stage allows catching only a priori non-correct behaviors like NaN values. Imagine that output values of the original program are finite but differ from the forward kernel outputs, or there are discrepancies between Finite Differences and AADC adjoints. The latter can be easily found by adding `payoff+=payoff*sqrt(0.0000000001*payoff);` to the code. Note that this piece of code has a minor influence on the forward values:

```
payoff=onePathPricing_debug(aad_asset, strike, t,
 rate_obj, vol_obj, aad_random_samples);
```

<sup>17</sup>[https://en.wikipedia.org/wiki/Finite\\_difference\\_coefficient](https://en.wikipedia.org/wiki/Finite_difference_coefficient)

<sup>18</sup>Equivalently, recompile with the option `{aacd::AADC_InstrumentCode, 0}`.

```
payoff+=sqrt(0.0000000001*payoff);
```

Now the output will contain the following report:

```
Total debug prints : 78
AADCDebug::0000000073::Adj[0]:FD_mid -nan [rv,adj]
AADCDebug::0000000073::Adj[0]:FD_left -nan [rv,adj]
AADCDebug::0000000073::Adj[0]:FD_right 31.7885877303599 [rv,adj]
AADCDebug::0000000073::Adj[0]:aadc_adj 18060.7023890307 [rv,adj]
AADCDebug::0000000073::Val[0]:aadc_val 7.66426894809624e-10 [rv,
adj]
AADCDebug::0000000073::Val[0]:rec_val 7.66426894809624e-10 [rv,
adj]
```

Executing `export AADC_DEBUG_STOP_RECORDING_AT=73` in the terminal<sup>19</sup> will set the debugger breakpoint to the location of the variable with index 73.

## 7 Advanced Methods

We will now discuss a few examples of more advanced situations that arise when using AADC.

### 7.1 Example 13: Using AADC for external functions

Practitioners often face a situation when there is no source code available for some functions, that is their functionality is implemented by a third party “black box” library. In this case, the user can wrap the required external function and use any method to compute its local gradients (for instance, Finite Differences). This approach is also useful in special cases (such as matrix products) where it is more efficient to manually code the local gradients. In this section, we show how to create an active version of an external function that works in both modes.

We consider an example and discuss how it can be adapted to other situations. The full code can be found in Section A.13.

Let’s assume that `myfunc` is an external function.

```
inline double myfunc(const double& x, const double& y) {
 ...
}
```

We show how to integrate this function into the AADC library. The overloaded version of `myfunc` should take active types as inputs.

```
inline idouble myfunc(const idouble& x, const idouble& y) {
 idouble res;

 aadc::addConstStateExtFunction(std::make_shared<
 MyFuncAADWrapper>(res, x, y));

 return res;
}
```

---

<sup>19</sup>Equivalently, recompiling with the option `{aacd::AADC_DebugStopRecordingAt, 73}`.

Here we create an `MyFuncAADWrapper` for each external function. This class plays a role of an external function holder and implements the methods for the `forward()` and the `reverse()` execution.

```
class MyFuncAADWrapper : public aadc::ConstStateExtFunc {
public:
 MyFuncAADWrapper(idouble& res, const idouble& x, const idouble&
 y)
 : xi(x), yi(y)
 , resi(res)
 {
 // This code is executed during the recording stage.
 res.val = myfunc(x.val,y.val);
 }

 template<typename mmType>
 void forward(mmType* v) const {
 // This code is executed during the forward pass.
 // Note that we should use a mutex lock here if myfunc is
 not multithread safe.
 for (int avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi)
 toDblPtr(v[resi])[avxi] = myfunc(toDblPtr(v[xi])[avxi],
 toDblPtr(v[yi])[avxi]);
 }

 template<class mmType>
 void reverse(const mmType* v, mmType* d) const {
 // code to calculate the local gradient and update the adjoint
 variables.
 };

private:
 ExtVarIndex xi, yi;
 ExtVarIndex resi;
};
```

Data members of this class are the workspace indices for each of the `myfunc` input and output variables.

Recall that the compiled forward and reverse kernels support SIMD. In the presence of an external function, the user code should run it for each element of the SIMD vector.

The `reverse` method can be implemented using Finite Differences or the closed form (if available):

```
template<class mmType>
void reverse(const mmType* v, mmType* d) const {
 double h(0.00001);
 for (int avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi) {
 double d1 = (myfunc(toDblPtr(v[xi])[avxi] + h, toDblPtr(
 v[yi])[avxi]) - myfunc(toDblPtr(v[xi])[avxi] - h, toDblPtr(v[
 yi])[avxi])) / (2 * h);
 double d2 = (myfunc(toDblPtr(v[xi])[avxi], toDblPtr(v[
 yi])[avxi] + h) - myfunc(toDblPtr(v[xi])[avxi], toDblPtr(v[
 yi])[avxi] - h)) / (2 * h);
 toDblPtr(d[xi])[avxi] += toDblPtr(d[resi])[avxi] * d1;
 toDblPtr(d[yi])[avxi] += toDblPtr(d[resi])[avxi] * d2;
 }
```



```
 }
};
```

The method also computes increments to the adjoint input variables.

## 7.2 Example 14. AAD for Mean Square Error of Expectations

In this example, we apply the LBFGS++ library to minimizing a function of the form

$$G(a, b) = \left( \int_0^1 f_1(x) dx - c_1 \right)^2 + \left( \int_0^1 f_2(x) dx - c_2 \right)^2.$$

To give a concrete example we let  $c_1 = 3, c_2 = 5$  and

$$\begin{cases} f_0 = ax + bx^2 \\ f_1 = af_0 \\ f_2 = e^{f_0} \end{cases}$$

Our initial guess is in the origin.

In 2017 C.Fries considered<sup>20</sup> the problem of computing expectations when some of the intermediate operations are expectations themselves. In this situation combining the AAD and multithreading approaches is not trivial. Fries suggests using the term *Stochastic AAD* for these situations and we will follow this terminology even outside the probabilistic context. Note that our paper on applying AADC to calibrate stochastic volatilities also uses Stochastic AAD<sup>21</sup>.

In the context of our example, using AADC to apply Stochastic AAD consists of the following steps:

1. Record the algorithm calculating  $f_1$  and  $f_2$  for random  $a, b, x$ .
2. Calculate the integrals  $I_i = \int_0^1 f_i(x) dx - c_i$ ,  $i = 1, 2$ .
3. For a random  $x$  calculate the sum

$$I_1 \nabla f_1(x) + I_2 \nabla f_2(x)$$

by executing the forward function, initializing the adjoint variables by

$$\overline{f_i} = 0.5 I_i, \quad i = 1, 2$$

and executing the reverse function, see (1).

We'll now look at the corresponding code step by step. The full code is given in [A.14](#). First, we create the class

---

<sup>20</sup>Christian P. Fries, *Stochastic Automatic Differentiation: Automatic Differentiation for Monte-Carlo Simulations*, SSRN (2017) DOI: 10.2139/SSRN.2995695.

<sup>21</sup>D. Goloubentsev, E. Lakshtanov, *Remarks on stochastic automatic adjoint differentiation and financial models calibration*.

## 7.2 Example 14. AAD for Mean Square Error of Expectations

```

class Integral {
 typedef __m256d mmType;
public:
 Integral () {
 idouble a,b,x,f0,f1,f2;

 v_index.resize(3);
 res_index.resize(2);

 aad_funcs = std::make_shared<aadc::AADCFunctions<
mmType>> >();

 aad_funcs->startRecording();
 v_index[0]=x.markAsInput();
 v_index[1]=a.markAsInput();
 v_index[2]=b.markAsInput();

 f0=a*x+b*x*x;
 f1=a*f0;
 f2=std::exp(f0);

 res_index[0]=f1.markAsOutput();
 res_index[1]=f2.markAsOutput();

 aad_funcs->stopRecording();

 ws = aad_funcs->createWorkspace();
 }

```

We keep all the *idouble* variables' indices in the vector `v_index`. The first step (the recording) is over. We now realize the second step, computing the integrals  $I_1$  and  $I_2$ .

```

double operator()(const VectorXd& x, VectorXd& grad)
{
 std::mt19937_64 gen;
 std::uniform_real_distribution<> uni_distrib(0.0, 1.0);

 mmType integral1(mmSetConst<mmType>(0)),
 integral2(mmSetConst<mmType>(0)),
 D_integral1(mmSetConst<mmType>(0)),
 D_integral2(mmSetConst<mmType>(0));

 for (int i=0; i<num_sims_count; ++i) {

 ws->setVal(v_index[1],mmSetConst<mmType>(x[0]));
 ws->setVal(v_index[2],mmSetConst<mmType>(x[1]));

 double *_x = (ws->valp(v_index[0]));
 for (int ci=0; ci<count; ++ci) {
 *_x[ci]= uni_distrib(gen);
 }

 (*aad_funcs).forward(*ws);
 }

```

## 7.2 Example 14. AAD for Mean Square Error of Expectations

```

 integral1=mmAdd(ws->val(res_index[0]), integral1);
 integral2=mmAdd(ws->val(res_index[1]), integral2);
 }

 double _integral1(mmSum(integral1));
 double _integral2(mmSum(integral2));

```

Now the following code realizes the third step computing  $I_1 \nabla f_1(x) + I_2 \nabla f_2(x)$  and calculating the corresponding integral sum:

```

 for (int i=0; i<num_sims_count; ++i) {

 ws->setVal(v_index[1], mmSetConst<mmType>(x[0]));
 ws->setVal(v_index[2], mmSetConst<mmType>(x[1]));

 double *_x = (ws->valp(v_index[0]));
 for (int ci=0; ci<count; ++ci) {
 _x[ci]= uni_distrib(gen);
 }

 (*aad_funcs).forward(*ws);
 ws->resetDiff();

 ws->setDiff(res_index[0], _integral1-3);
 ws->setDiff(res_index[1], _integral2-5);

 (*aad_funcs).reverse(*ws);

 D_integral1=mmAdd(ws->diff(v_index[1]), D_integral1);
 D_integral2=mmAdd(ws->diff(v_index[2]), D_integral2);
 }

 grad[0] = mmSum(D_integral1)/num_sims;
 grad[1] = mmSum(D_integral2)/num_sims;

 return 0.5*((_integral1-3)*(_integral1-3)+ (_integral2-5)*(_integral2-5));
}

```

The call of the LBFGS++ library is realized similarly to what we did in Section 3.8.

```

//Then we just need to set up parameters, create a solver
object, provide an initial guess, and run the minimization
function.
void example_expectations_mse()
{
 // Set up parameters
 LBFGSPParam<double> param;
 param.epsilon = 1e-6;
 param.max_iterations = 100;
 // Create a solver and function object

 LBFGSSolver<double> solver(param);
 Integral fun;
 // Initial guess

```

```

VectorXd x = VectorXd::Zero(2);
// x to be overwritten to the best point found
double fx;
int niter = solver.minimize(fun, x, fx);

std::cout << niter << " iterations" << std::endl;
std::cout << "x = \n" << x.transpose() << std::endl;
std::cout << "f(x) = " << fx << std::endl;
}

```

The result of the execution is:

```

100 iterations
x = 5.7696 -6.67508
f(x) = 4.25427

```

## 8 Functions of the AADC library

We give a specification for functions of the AADC library.

### 8.1 *idouble, ibool, iint* methods

- `AADCResult markAsOutput()`

All variables subject to differentiation should be marked as *outputs*.

- `AADCArgument markAsInput()`

The user should apply this method to an *idouble* variable  $v$  when it is an argument of the forward function AND the computation of the derivative with respect to  $v$  is required. If only one of these is true, then the user can use the following methods instead:

- `AADCArgument markAsDiff()`

When the derivative with respect to  $v$  is required but its value is fixed.

- `AADCArgument markAsInputNoDiff()`

When the value of  $v$  varies but the derivative with respect to  $v$  is not required.

- `AADCArgument markAsScalarInput()`

When the value of  $v$  varies but is the same for all AVX-components. Note that the corresponding adjoint is scaled with the factor `mmSize<mmType>()`.

### 8.2 Auxiliary vector functions

*mmType* is the notational convention for the variable type in a `WorkSpace`. Its size depends on the architecture of the processor:

AVX2: `typedef __m256d mmType;`

AVX512: `typedef __m512d mmType;`

**Note:** Some C++ compilers do not support the correct AVX memory alignment<sup>22</sup>, so we recommend to avoid using `std::vector<mmType>` and use `aadc::mmVector<mmType>` instead. Examples can be found in A.11.

<sup>22</sup>[https://en.wikipedia.org/wiki/Data\\_structure\\_alignment](https://en.wikipedia.org/wiki/Data_structure_alignment)

The following methods are available for operations with *mmType* vectors. These operations use the processor's intrinsic structure and work faster than an alternative set of coordinate-wise operations:

- `inline int mmSize()` Returns the number of doubles in *mmType*.
- `mmType mmSetConst(const double& v)` Forms an *mmType* vector with equal entries *v*. An example can be found in Section 3.1.
- `double mmSum(const mmType& v)`  
Returns the sum of AVX-components. See examples in Sections 3.3 and 3.7.
- `mmType mmAdd(const mmType& a, const mmType& b)`  
Performs coordinate-wise addition. An example can be found in Section 3.3. The following functions are defined similarly:
- `mmType mmSub(const mmType& a, const mmType& b)`
- `mmType mmMul(const mmType& a, const mmType& b)`
- `mmType mmDiv(const mmType& a, const mmType& b)`
- `mmType mmMax(const mmType& a, const mmType& b)`
- `mmType mmFabs(const mmType& a)`
- `mmType mmIndicatorFunction(const mmType& a)`

### 8.3 *AADCFunctions* class

An object of *AADCFunctions* class contains compiled forward and reverse kernels.

- `std::shared_ptr<AADCWorkspace<mmType> > createWorkspace()`  
Creates a Workspace. There can be several WorkSpaces for the same *AADCFunctions* object.
- `void forward(AADCWorkspace<mmType>& workspace)`  
Executes the forward function.
- `void reverse(AADCWorkspace<mmType>& workspace)`  
Executes the reverse function.

See Sections 3.1 and 3.8 for examples.

#### 8.3.1 Statistics

- `void printPassiveExtractLocations(ostream& ostr, const std::string& func_name)`  
);  
Provides the number of stochastic variable idouble-to-double conversions during the recording stage. See Section 5.1.
- `void outStats(ostream& ostr, const std::string& func_name);`  
Prints out the whole statistics. Particular characteristics can be obtained using the following instructions:
- `getCodeSizeFwd()`  
Gives the size of the forward function (in bytes).

- `getCodeSizeRev()`  
Gives the size of the reverse function.
- `getWorkArraySize()`  
Gives the size of the Work Array, that is the total memory required for MLKernels execution.
- `getStackSize()`  
Gives the size of the stack, that is the memory required at the forward stage to be reused during the backpropagation.
- `getConstDataSize()`  
Gives the size of the Const Data.

An example can be found in Section 3.6.

## 8.4 *AADCWorkspace* class

An object of *AADCWorkspace* class contains *mmType* variables to be used by the compiled forward and reverse function. There is a one-to-one correspondence between its elements and input data of the original function.

- `double& val(const AADCArgument& var_idx)`  
`double& val(const AADCResult& var_idx)`  
`mmType& val(const AADCArgument& var_idx)`  
`mmType& val(const AADCResult& var_idx)`  
Give the vector value of the variable with reference `var_idx`.
- `mmType& diff(const AADCArgument& var_idx)`  
Gives the vector value of the adjoint variable with reference `var_idx`.
- `AADCWorkspace& setVal(const AADCArgument& var_idx, const mmType& val)`  
Attributes the vector value of the variable with reference `var_idx`. Note that we can also invoke `setVal` with a scalar value  $a \in \mathbb{R}$ , this achieves the same result as  
`setVal(x_arg, mmSetConst<mmType>(a))`

where `mmSetConst<mmType>(a)` creates a constant vector with value  $a$ . See examples of use in Sections 3.1, 3.4.

- `void resetDiff()`  
The call of this method sets all the adjoint variables in the workspace to zero. It is usually required between iterated executions of the reverse function. See an example in Section 3.8.

## 8.5 Vector generalizations

Here we collect the instructions useful when input variables form a vector. They can be used if one does `#include <aadc/aadc_matrix.h>`. Some examples of using these operations can be found in A.11.

For the sake of simplicity, AADC uses

```
typedef std::vector<aadc::AADCArument> VectorArg;
typedef std::vector<aadc::AADCResult> VectorRes;
typedef std::vector<aadc::AADCSArument> ScalarVectorArg;
```

We have the following generalizations of the scalar functions:

- `void setAVXVector(aadc::AADCWorkspace<mmType>& ws, const VectorArg& vi, const mmVector<mmType>& v)`
- `setScalarVectorInput(aadc::AADCWorkspace<mmType>& ws, const ScalarVectorArg & vi, const std::vector<double>& v)`
- `void markVectorAsInput(VectorArg& vi, const std::vector<idouble>& v, const bool Diff)`
- `void markVectorAsOutput(VectorRes& mi, const std::vector<idouble>& m)`
- `void markScalarVectorAsInput(ScalarVectorArg& vi, std::vector<idouble>& v)`

## 8.6 Mathematical operations with *idouble*, *ibool*, *iint*, and arrays

### 8.6.1 Comparisons

The following comparison operators are available:

```
namespace aadcBoolOps {
 ibool operator && (const ibool& a, const ibool& b)
 ibool operator || (const ibool& a, const ibool& b)
}
ibool operator!(const ibool& a)
ibool operator < (const idouble& a, const idouble& b)
ibool operator < (const iint& a, const iint& b)
ibool operator <= (const idouble& a, const idouble& b)
ibool operator != (const idouble& a, const idouble& b)
ibool operator == (const idouble& a, const idouble& b)
ibool operator == (const iint& a, const iint& b)
ibool operator >= (const idouble& a, const idouble& b)
ibool operator > (const idouble& a, const idouble& b)
ibool operator > (const iint& a, const iint& b)
```

### 8.6.2 min/max operations

The syntax for min/max operations is standard:

```
namespace std {
 idouble max(const idouble& a, const idouble& b)
 idouble min(const idouble& a, const idouble& b)
}
iint max(const iint& a, const iint& b)
iint min(const iint& a, const iint& b)
```

```
iint max(const iint& a, const iint& b)
iint min(const iint& a, const iint& b)
```

### 8.6.3 Conditional operations

The *if* operator can not be overloaded so an AADC user has to use the conditional operator *iIf* in all cases when the arguments depend on the Workspace variables. A full working example can be found in Section 3.7.

```
iint iIf(const ibool& cond, const iint& a, const iint& b)
double iIf(const bool& cond, const double& a, const double& b)
idouble iIf(const ibool& cond, const idouble& a, const idouble& b
)
```

### 8.6.4 Casting

```
iint toInt(const idouble& x)
int toInt(const double& x)
```

For suitable values transforms the *idouble* type to *iint* type.

### 8.6.5 Arrays

- `template<class A> idouble get(const A& arr, const iint& i)`  
`template<class A> double get(const A& arr, const int& i)`

Returns the *i*-th element of the array *A*. See an example in Section 3.5. To avoid a possible conflict the method `get` should be inserted into the namespace `aadcArrayOps`.

- `template<class A> int lower_bound(const A& arr, const double & x)`  
`template<class A> iint lower_bound(const A& arr, const idouble& x)`

The functionality is the same as that of the standard `std::lower_bound`. See an example in Section 3.5.

## 8.7 Global pragmas and internal namespaces

- `#define AADC_ALLOW_TO_PASSIVE_BOOL.`  
 Allows automatically casting *ibool* to *bool*. See Section 5.1 for details.
- `using namespace aadcBoolOps.`  
 Required for binary logic operations like `&&` and `||`. See Section 5.1 for details.
- `using namespace aadcArrayOps`  
 An example of use can be found in 3.5.



## 8.8 Checkpointing and executing program segments

- `idouble::CheckPoint()` Sets the checkpoint. To run the forward (reverse) kernel from (to) this point, use the additional arguments in the `forward()` and `reverse()` methods:
- `void forward( AADCWorkspace<mmType>& workspace, int first_cp = 0, int last_cp = -1)`
- `void reverse(AADCWorkspace<mmType>& workspace, int first_cp = 0, int last_cp = -1)`

Each checkpoint divides the program into segments. The first segment has index 0 and the last one can be referenced with index  $-1$ . These two methods run the portion of the forward (reverse) kernel that corresponds to the segments from `first_cp` to `last_cp`.

## 8.9 Complex numbers

The class `std::complex<idouble>` is available after doing `#include <aadc/icomplex.h>`. The interface is similar to the standard `std::complex<double>`. An example of use can be found in [A.16](#).

## A Appendix A - List of Examples

We list all the examples from the test directory of the AADC software package. Details and explanations can be found in Section 3 (Examples 1-7), Section 4 (Examples 14 and 15), Section 5 (Examples 8 and 10), Section 6 (Example 11) and Section 7 (Examples 12 and 13).

### A.1 Example 1. Hello, World!

```

1 #include <iostream>
2 #include <aadc/aadc.h>

4 void exampleHelloWorld()
5 {
6 idouble x(0.5), y(1.1), z, f;

8 // size of the active type is same as that of the native type
9 assert(sizeof(double) == sizeof(idouble));

10 // We can use idouble as a replacement for native double

12 z = x + sin(y);

14 // We even print it to stdout, etc. See it in the
15 // TERMINAL below

16 std::cout << "z:" << z << std::endl;

18 //////////////////////////////////
19 // Record Function
20 //////////////////////////////////

22 typedef __m256d mmType;
23 aadc::AADCFunctions<mmType> aad_funcs;

24
25 aad_funcs.startRecording();
26 aadc::AADCArument x_arg(x.markAsInput());
27 aadc::AADCArument y_arg(y.markAsInput());
28 aadc::AADCArument z_arg(z.markAsInput());
29 // Here the calculations are simple, but in practice this
30 // call can travel through a lot of calls to the
31 // object-oriented code.
32 f=std::exp(x/y+z);
33 aadc::AADCArument f_arg(f.markAsOutput());
34 aad_funcs.stopRecording();

36 std::shared_ptr<adc::ADCWorkspace<mmType>> ws(aad_funcs.
37 createWorkspace());

38 //////////////////////////////////
39 // Execute the function
40 //////////////////////////////////

42 // set initial values for variables x, y & z
43 ws->setVal(x_arg, __m256_set_pd(1.0, 2.0, 3.0, 4.0));

```

```

44 ws->setVal(y_arg, 2.0);
45 ws->setVal(z_arg, _mm256_set_pd(0.1, 0.2, 0.3, 0.4));
46
47 // execute the forward function
48 aad_funcs.forward(*ws);
49
50 // check the results for the forward function, these
51 // should match the original f=exp(x/y+z)
52 for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi
53) {
54 std::cout
55 << "avx[" << avxi << "]" "
56 << " f(" << ws->valp(x_arg)[avxi]
57 << ", " << ws->valp(y_arg)[avxi]
58 << ", " << ws->valp(z_arg)[avxi]
59 << ")=" << ws->valp(f_res)[avxi]
60 << std::endl;
61 }
62
63 // prepare for the reverse pass, set f() equal to 1
64 ws->setDiff(f_res, 1.0);
65
66 // execute the reverse pass
67 aad_funcs.reverse(*ws);
68
69 // now we have all requested derivatives and can print them out
70 // (below we print only the first element of the __m256d
71 // type)
72 for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi
73) {
74 std::cout
75 << "avx[" << avxi << "]" "
76 << " df/dx=" << ws->diffp(x_arg)[avxi] << ", " << "df
77 /dy=" << ws->diffp(y_arg)[avxi] << ", " << "df/dz=" << ws->
78 diffp(z_arg)[avxi]
79 << std::endl;
80 }
81 std::cout << "Example 1 is done" << std::endl;
82 }
83
84 int main() {
85
86 exampleHelloWorld();
87
88 return 0;
89 }

```

Ex1HelloWorld.cpp

## A.2 Example 2. Black-Scholes

```

#include <iostream>
2 #include <aadc/aadc.h>

```

```

4 using namespace aadc;

6 // The Black-Scholes formula for the present value of a European
 option payoff with:
 // S0 - price at time t=0
8 // K - strike price
 // r - interest rate
10 // vol - volatility
 // T - call time
12
13 template<typename mdouble>
14 mdouble BlackScholes(
 mdouble S0, mdouble K, mdouble r, mdouble vol, mdouble T
15) {
 return S0*std::cdf_normal((std::log(S0/K) + (r + 0.5 * (std::
 pow(vol, 2))))/
16 (vol*std::sqrt(T)))-std::exp(-r*T)*K*std::cdf_normal((std::
 log(S0/K)+
 (r-0.5*(std::pow(vol, 2))))/(vol*std::sqrt(T)));
20 }

22 //Computes the Finite Differences derivative
 // with respect to the first variable S0
24 template<typename mdouble>
 mdouble BlackScholesFD(
25 mdouble S0, mdouble K, mdouble r, mdouble vol, mdouble T,
 mdouble h
26) {
27 return (BlackScholes(S0 + h, K, r, vol, T) - BlackScholes(S0,
 K, r, vol, T))/h;
28 }
30

32 void exampleBlackScholes()
 {
33 typedef __m256d mmType;
 aadc::AADCFUNCTIONS<mmType> aad_funcs;
34
35 idouble S0,K,r,vol,T,BS;
36
37 aad_funcs.startRecording();
38
39 aadc::AADCArgument S0_arg(S0.markAsInput());
40 aadc::AADCArgument K_arg(K.markAsInput());
41 aadc::AADCArgument r_arg(r.markAsInput());
42 aadc::AADCArgument vol_arg(vol.markAsInput());
43 aadc::AADCArgument T_arg(T.markAsInput());
44
45 BS=BlackScholes(S0,K,r,vol,T);
46
47 aadc::AADCResult BS_res(BS.markAsOutput());
48
49 aad_funcs.stopRecording();
50
51
52

```

```

std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(aad_funcs.
createWorkspace());
54
ws->setVal(S0_arg, 100.0);
56 ws->setVal(K_arg, 110.0);
ws->setVal(r_arg, _mm256_set_pd(0.1, 0.1, 0.01, 0.1));
58 ws->setVal(vol_arg, _mm256_set_pd(0.5, 0.5, 0.25, 0.25));
ws->setVal(T_arg, _mm256_set_pd(2.0, 1.0, 1.0, 1.0));
60
aad_funcs.forward(*ws);
62
ws->setDiff(BS_res, 1.0);
64
aad_funcs.reverse(*ws);
66 // Print out the values of the function and derivatives with
respect to S0, K, r.
for (int i=0; i<4; ++i) {
68 std::cout << " BS(" <<ws->valp(S0_arg)[0]
 << "," << ws->valp(K_arg)[i]
70 << "," << ws->valp(r_arg)[i]
 << "," << ws->valp(vol_arg)[i]
72 << "," << ws->valp(T_arg)[i]
 << ")= " << ws->valp(BS_res)[i]
74 << ". dBS/dS0=" << ws->diffp(S0_arg)[i]
 << "; dBS/dK=" << ws->diffp(K_arg)[i]
76 << "; dBS/dr=" << ws->diffp(r_arg)[i]
 << std::endl;
78 }
// Print out the Finite Differences derivatives
80 // with respect to the first variable S0 and with h=0.000001
std::cout << "dBS/dS0(100,110,0.1,0.25,1)=" <<
82 BlackScholesFD(100.0,110.0,0.1,0.25,1.0, 0.000001) << std::
endl;
std::cout << "dBS/dS0(100,110,0.01,0.25,1)=" <<
84 BlackScholesFD(100.0,110.0,0.01,0.25,1.0, 0.000001) << std::
endl;
std::cout << "dBS/dS0(100,110,0.1,0.5,1)=" <<
86 BlackScholesFD(100.0,110.0,0.1,0.5,1.0, 0.000001) << std::
endl;
std::cout << "dBS/dS0(100,110,0.1,0.5,2)=" <<
88 BlackScholesFD(100.0,110.0,0.1,0.5,2.0, 0.000001) << std::
endl;
90 }

92 int main() {
94 exampleBlackScholes();
96 return 0;
}

```

Ex2BlackScholes.cpp

### A.3 Example 3. Monte Carlo

```

1 #include <iostream>
 #include <aadc/ibool.h>
3 #include <aadc/aadc.h>
 #include <random>
5
 using namespace aadc;
7
 void exampleMonteCarlo()
9 {
 typedef __m256d mmType;
11
 std::mt19937_64 gen;
13 std::normal_distribution<> norm_distrib(0.0, 1.0);
 idouble S0=100, K=110, r=0.1, vol=0.25, T=1, random_sample;
15
 int num_sim=1000000;
17 int count = aadc::mmSize<mmType>();
 int num_sim_avx=(int)floor(num_sim/count);
19
 aadc::AADCFunctions<mmType> aad_funcs;
21
 aad_funcs.startRecording();
23
 aadc::AADCArument random_sample_arg(random_sample.
markAsInputNoDiff());
25 aadc::AADCArument S0_arg(S0.markAsDiff());
 aadc::AADCArument K_arg(K.markAsDiff());
27 aadc::AADCArument r_arg(r.markAsDiff());
 aadc::AADCArument vol_arg(vol.markAsDiff());
29 aadc::AADCArument T_arg(T.markAsDiff());
31
 idouble S_T = S0 * std::exp((r - vol*vol / 2) * T +
33 vol * std::sqrt(T) * random_sample);
 idouble price = std::exp(-r*T) * std::max(S_T-K, 0.);
35
 aadc::AADCResult price_res(price.markAsOutput());
37 aad_funcs.stopRecording();
39
 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(aad_funcs.
createWorkspace());
41
 mmType integral_S0(mmSetConst<mmType>(0)), integral_K(
mmSetConst<mmType>(0)),
 integral_r(mmSetConst<mmType>(0)), integral(mmSetConst<mmType>
>(0));
43
 for (int i=0; i<num_sim_avx; ++i) {
45
 double* _x = (ws->valp(random_sample_arg));
47 for (int ci=0; ci<count; ++ci) {
 _x[ci]= norm_distrib(gen);

```

```

49 }

51 aad_funcs.forward(*ws);
 integral=aadc::mmAdd(ws->val(price_res), integral);

53

55 ws->resetDiff();
 ws->setDiff(price_res, 1.0);

57 aad_funcs.reverse(*ws);
 integral_S0=mmAdd(ws->diff(S0_arg), integral_S0);
59 integral_K=mmAdd(ws->diff(K_arg), integral_K);
 integral_r=mmAdd(ws->diff(r_arg), integral_r);
61 }

63 std::cout << "f(100,110,0.1,0.25,1)=" << aadc::mmSum(
integral) / num_sim <<
 ". df/dS0=" << aadc::mmSum(integral_S0) / num_sim << "; df/
dK=" <<
65 aadc::mmSum(integral_K) / num_sim << "; df/dr=" <<
 aadc::mmSum(integral_r) / num_sim << std::endl;
67 }

69 int main() {

71 exampleMonteCarlo();

73 return 0;
}

```

Ex3MonteCarlo.cpp

## A.4 Example 4. Jacobian

```

#include <iostream>
2 #include <aadc/aadc.h>

4 using namespace aadc;

6 void exampleJacobianFull()
{
8 typedef __m256d mmType;
 aadc::AADCFunctions<mmType> aad_funcs;

10

12 idouble x,y,z,f1,f2;

14 aad_funcs.startRecording();

16 aadc::AADCArument x_arg(x.markAsInput());
 aadc::AADCArument y_arg(y.markAsInput());
 aadc::AADCArument z_arg(z.markAsInput());

18

20 f1=-3*x*y+16*x*x/z-x*y/(1+x*y);
 f2=x*x*y+f1;

22 aadc::AADCResult f1_res(f1.markAsOutput());

```

## A.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

```

aadc::AADCResult f2_res(f2.markAsOutput());
24
 aad_funcs.stopRecording();
26
 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(aad_funcs.
createWorkspace());
28
 ws->setVal(x_arg, mmSetConst<mmType>(1));
30
 ws->setVal(y_arg, _mm256_set_pd(2.0, 2.0, 3.0, 3.0));
 ws->setVal(z_arg, _mm256_set_pd(4.0, 4.0, 5.0, 5.0));
32
 aad_funcs.forward(*ws);
34
 ws->setDiff(f1_res, _mm256_set_pd(1.0, 0.0, 1.0, 0.0));
36
 ws->setDiff(f2_res, _mm256_set_pd(0.0, 1.0, 0.0, 1.0));

38
 aad_funcs.reverse(*ws);

40
 for (int i=0; i<4; ++i) {
 int dfi = (i < 2 ? i+1 : i-1);
42
 std::cout << " f1(" <<ws->valp(x_arg)[0]
 << "," << ws->valp(y_arg)[i]
44
 << "," << ws->valp(z_arg)[i]
 << ")= " << ws->valp(f1_res)[i]
46
 << " f2(" <<ws->valp(x_arg)[0]
 << "," << ws->valp(y_arg)[i]
48
 << "," << ws->valp(z_arg)[i]
 << ")= " << ws->valp(f2_res)[i]
50
 << ". df" << dfi << "/dx=" << ws->diffp(x_arg)[i]
 << "; df" << dfi << "/dy=" << ws->diffp(y_arg)[i]
52
 << "; df" << dfi << "/dz=" << ws->diffp(z_arg)[i]
 << std::endl;
54
 }
}
56
int main() {
58
 exampleJacobianFull();
60
 return 0;
62 }

```

Ex4Jacobian.cpp

## A.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

```

#include <iostream>
2 #include <random>
#include <aadc/idouable.h>
4 #include <aadc/aadc.h>
#include <aadc/ibool.h>
6 #include <aadc/iint.h>

```



## A.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

```

8 using namespace aadc;
 using namespace aadcArrayOps;
10
 template<typename mdouble>
12 class PiecewiseCurve {
 public:
14 PiecewiseCurve(const std::vector<double>& nodes)
 : m_s(nodes)
16 {}

18 void init(const std::vector<double>& vals) {
 m_vals.resize(m_s.size()+1);
20 for (int i=0; i<m_vals.size(); i++) {
 m_vals[i]=vals[i];
22 }
 }

24 mdouble interpolatedValue (const mdouble& s) {
26 auto i_s = lower_bound(m_s, s);
 return get(m_vals,i_s);
28 }

30 std::vector<mdouble> m_vals;
 std::vector<double> m_s;
32 };

34
 template<typename mdouble>
36 mdouble funMath(
 mdouble& x, PiecewiseCurve<mdouble>& m_curve
38) {
 return m_curve.interpolatedValue(x);
40 }

42
 void exampleInterpolation()
44 {
 typedef __m256d mmType;
46
 int num_sims = 50000;
48 int count = aadc::mmSize<mmType>();
 int num_sims_count = num_sims/count;
50
 std::mt19937_64 gen;
52 std::normal_distribution<> normal_distrib(0, 1);

54 std::vector<double> curve_nodes({1,3});
 std::vector<double> curve_vals({0,0,0});
56 std::vector<double> curve_vals2({3,5,8});

58 aadc::AADCFUNCTIONS<mmType> aad_funcs;

60 PiecewiseCurve<idouble> aad_curve(curve_nodes);
 aad_curve.init(curve_vals2);

```

## A.5 Example 5. Valuation involving interpolation (using *ibool* and *iint*)

```

62 idouble x,f;
64
66 aad_funcs.startRecording();
68 aadc::AADCArument x_arg(x.markAsInput());
69 std::vector<adc::AADCArument> curve_val_args(aad_curve.
70 m_vals.size());
71 for (int i=0; i<aad_curve.m_vals.size(); ++i) {
72 curve_val_args[i] = aad_curve.m_vals[i].markAsDiff();
73 }
74
75 f=funMath(x,aad_curve);
76
77 aadc::AADCResult f_res(f.markAsOutput());
78 aad_funcs.stopRecording();
79
80 std::shared_ptr<adc::AADCWorkspace<mmType>> ws(aad_funcs.
81 createWorkspace());
82
83 mmVector<mmType> D_integral(aad_curve.m_vals.size());
84 std::fill(D_integral.begin(),D_integral.end(),mmSetConst<
85 mmType>(0));
86 double integral(0);
87
88 for (int i=0; i<aad_curve.m_vals.size(); ++i) {
89 ws->setVal(curve_val_args[i], mmSetConst<mmType>(
90 curve_vals2[i]));
91 }
92
93 for (int i=0; i<num_sims_count; ++i) {
94
95 for (int ci=0; ci<count; ++ci) {
96 auto _x = (ws->valp(x_arg));
97 _x[ci] = normal_distrib(gen);
98 }
99
100 aad_funcs.forward(*ws);
101
102 integral+= aadc::mmSum(ws->val(f_res));
103
104 ws->resetDiff();
105 ws->setDiff(f_res, 1.0);
106 aad_funcs.reverse(*ws);
107
108 for (int i=0; i<aad_curve.m_vals.size(); ++i) {
109 D_integral[i]=adc::mmAdd(ws->diff(curve_val_args[i])
110 , D_integral[i]);
111 }
112 }
113
114 integral/=num_sims;
115 std::cout << "f=" << integral << std::endl;
116

```

```

 for (int i=0; i<aad_curve.m_vals.size(); ++i) {
112 std::cout << "df/v"<< i << " = " << aadc::mmSum(
 D_integral[i]) / num_sims << std::endl;
 }
114 std::cout << "-----" << std::endl;
116 }

118 int main() {

120 exampleInterpolation();

122 return 0;
 }

```

Ex5Interpolation.cpp

## A.6 Example 6. Incremental Checkpointing

```

1 #include <iostream>
 #include <aadc/aadc.h>
3
 using namespace aadc;
5
 bool checkPoints = true;
7
 void exampleCheckpoint()
9 {
 typedef __m256d mmType;
11 aadc::AADCFunctions<mmType> aad_funcs;

13 idouble x(0),y(1),z,f;

15
 aad_funcs.startRecording();
17 auto x_ind(x.markAsInput()), y_ind(y.markAsInput());
 for (int i=0; i<=100; i++) {
19 if (i%10 == 0) idouble::CheckPoint();
 for (int j=1; j<100; j++) {
21 z=std::exp(std::log(x*y)/std::sqrt(2));
 y=std::exp(std::log(x/y)/std::sqrt(2));
23 x=z;
 }
25 }
 f=x;
27 auto f_ind(f.markAsOutput());
 aad_funcs.stopRecording();

29
 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
createWorkspace());

31
 std::cout << "*****" << std
::endl;
33 std::cout << "Code size forward : " << aad_funcs.
getCodeSizeFwd() << std::endl;

```

```

 std::cout << "Code size reverse : " << aad_funcs.
getCodeSizeRev() << std::endl;
35 std::cout << "Work array size : " << aad_funcs.
getWorkArraySize() << std::endl;
 std::cout << "Stack size : " << aad_funcs.getStackSize
() << std::endl;
37 std::cout << "Const data size : " << aad_funcs.
getConstDataSize() << std::endl;
 std::cout << "Checkpoint size : " << aad_funcs.
getNumCheckPoints() << std::endl;
39 std::cout << "*****" << std::
endl;

41 double x_init(3),y_init(2);

43 ws->setVal(x_ind,mmSetConst<mmType>(x_init));
 ws->setVal(y_ind,mmSetConst<mmType>(y_init));
45
 aad_funcs.forward(*ws);
47
 double *_f = (ws->valp(f_ind));
49 std::cout
 << " f(" << x_init
51 << "," << y_init
 << ")=" << _f[0]
53 << std::endl;

55
 ws->setDiff(f_ind, 1.0);
57 aad_funcs.reverse(*ws);

59 double *_dx = (ws->diffp(x_ind));
 double *_dy = (ws->diffp(y_ind));
61 std::cout
 << "df/dx=" << _dx[0] << std::endl
63 << "df/dy=" << _dy[0] << std::endl
 << std::endl;
65 }

67 int main() {

69 exampleCheckpoint();

71 return 0;
}

```

Ex6Checkpointing.cpp

## A.7 Example 7. Multithread for a simple MonteCarlo

```

#include <iostream>
2 #include <random>
#include <thread>
4 #include <aadc/ibool.h>
#include <aadc/aadc.h>

```

```

6 #include <aadc/aadc_tools.h>

8 using namespace std;
 using namespace aadc;
10
 typedef __m256d mmType;
12
 int num_sim=4000; // Divisible by (count*treads)
14 int avx_count = aadc::mmSize<mmType>();
 int num_treads=4;
16 int num_sim_avx=num_sim/(avx_count*num_treads);

18
 template<typename mdouble>
20 mdouble funMath(const mdouble& x, const mdouble& a) {
 return iIf (x<0.00001, 0, std::exp(a*x)/std::sqrt(x));
22 }

24 void exampleMultithread()
 {
26 idouble aad_x, aad_a(2), aad_f;

28 double d_integral(0), integral(0);

30 aadc::AADCFunctions<mmType> aad_funcs;

32 aad_funcs.startRecording();
 aadc::AADCArument x_index = aad_x.markAsInputNoDiff();
34 aadc::AADCArument a_index = aad_a.markAsDiff();

36 aad_f=funMath(aad_x,aad_a);

38 aadc::AADCResult f_index = aad_f.markAsOutput();
 aad_funcs.stopRecording();
40
 std::vector<double> integral_c(num_treads), d_integral_c(
num_treads);
42
 auto threadWorker = [&] (
44 double &integral,
 double &d_integral,
46 const int i
) {
48 std::mt19937_64 gen;
 std::uniform_real_distribution<> uni_distrib(0.0, 1.0);
50 gen.seed(i*17+31);

52 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(
aadc_funcs.createWorkspace());

54 mmType integral_mm(mmSetConst<mmType>(0)),d_integral_mm(
mmSetConst<mmType>(0));

56 for (int i=0; i<num_sim_avx; ++i) {

```

```

58 mmType rand_mm;
 double* _rand_mm = toDblPtr(rand_mm);
60 for (int ci=0; ci<avx_count; ++ci) {
 _rand_mm[ci]= uni_distrib(gen);
62 }
 ws->setVal(x_index, rand_mm);
64
 aad_funcs.forward(*ws);
 integral_mm = aadc::mmAdd(ws->val(f_index),
integral_mm);

66
 ws->resetDiff();
 ws->setDiff(f_index, 1.0);
70
 aad_funcs.reverse(*ws);
 d_integral_mm = aadc::mmAdd(ws->diff(a_index),
d_integral_mm);
72 }
74
 d_integral = mmSum(d_integral_mm);
 integral = mmSum(integral_mm);
76 };
78
 std::vector<std::unique_ptr<thread>> threads;
80 for(int i=0; i< num_treads; i++) {
 threads.push_back(
82 std::unique_ptr<thread>(
 new thread(
84 threadWorker
 , std::ref(integral_c[i])
86 , std::ref(d_integral_c[i])
 , i
88)
)
90);
 }
92 for(auto&& t: threads) t->join();

94 for(int i=0; i< num_treads; i++) {
 integral+=integral_c[i];
96 d_integral+=d_integral_c[i];
 }
98
 integral/=num_sim;
100 d_integral/=num_sim;

102 std::cout << "F(2)=" << integral << std::endl;
 std::cout << "F'(2)=" << d_integral << std::endl;
104 }
106
int main() {
108

```

```

 exampleMultithread();
110 return 0;
112 }

```

Ex7MultiThread.cpp

## A.8 Example 8. Interfacing AADC to a minimization library

```

#include <Eigen/Core>
2 #include <LBFGS.h>
#include <iostream>
4 #include <aadc/aadc.h>

6 using Eigen::VectorXd;
using namespace LBFGSpp;
8 using namespace aadc;

10 typedef __m256d mmType;
std::shared_ptr<aadc::AADCFunctions<mmType>> aad_funcs;
12 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws;
aadc::AADCArgument x_ind,y_ind;
14 aadc::AADCResult f_ind;

16 void record() {
 aad_funcs = std::make_shared<aadc::AADCFunctions<mmType>>();
18
 idouble x,y,f;
20
 aad_funcs->startRecording();
22 x_ind=x.markAsInput();
 y_ind=y.markAsInput();
24
 f=(x-1)*(x-1)+(y-2)*(y-2);
26
 f_ind=f.markAsOutput();
28 aad_funcs->stopRecording();

30 ws = aad_funcs->createWorkspace();
}
32

34 double calcGrad(const VectorXd& x, VectorXd& grad) {

36 ws->setVal(x_ind,x[0]);
 ws->setVal(y_ind,x[1]);
38
 (*aad_funcs).forward(*ws);
40

42 ws->resetDiff();
 ws->setDiff(f_ind, 1.0);
44

```

```

 (*aad_funcs).reverse(*ws);
46
 double *_dx = (ws->diffp(x_ind));
48 double *_dy = (ws->diffp(y_ind));
 double *_f = (ws->valp(f_ind));
50
 grad[0] = _dx[0];
52 grad[1] = _dy[0];
 return _f[0];
54 }

56 //Now we just need to set up parameters, create a solver object,
 provide an initial guess, and run the minimization function.

58 void exampleMinimization()
 {
60 // Set up parameters
 LBFSGParam<double> param;
62 param.epsilon = 1e-6;
 param.max_iterations = 100;
64 // Create the solver object

66 LBFSSolver<double> solver(param);
 record();
68
 // Initial guess
70 VectorXd x = VectorXd::Zero(2);
 // x to be overwritten to the best point found
72 double fx;
 int niter = solver.minimize(calcGrad, x, fx);
74
 std::cout << niter << " iterations" << std::endl;
76 std::cout << "x = \n" << x.transpose() << std::endl;
 std::cout << "f(x) = " << fx << std::endl;
78 }

80
82 int main() {
84 exampleMinimization();
86 return 0;
}

```

Ex8Minimization.cpp

## A.9 Example 9. Reusing non-AAD objects. Global idouble Casting

```

#ifndef _MY_ANALYTICS_H
2 #define _MY_ANALYTICS_H

4 #include "aadc/idouble.h"

```



## A.9 Example 9. Reusing non-AAD objects. Global idouble Casting

```

6 namespace UserNameSpace {
8 class MyAnalytics {
9 public:
10 // ...
12 mdouble val(const mdouble& a);
14 mdouble var1, var2;
15 };
16
17 };
18
19 #ifndef AADC_PASS
20 #define AADC_PASS
21 #undef _MY_ANALYTICS_H
22 namespace AAD {
23 #include "MyAnalytics.h"
24 };
25 #undef AADC_PASS
26 #endif
27
28 #endif // _MY_ANALYTICS_H

```

MyAnalytics.h

```

1 #include "MyAnalytics.h"
2
3
4
5 namespace UserNameSpace {
6
7 mdouble MyAnalytics::val(const mdouble& a) {
8 return (a+1)/a;
9 }
10
11 };
12
13 #ifndef AADC_PASS
14 #define AADC_PASS
15 namespace AAD {
16 #include "MyAnalytics.cpp"
17 };
18 #undef AADC_PASS
19 #endif

```

MyAnalytics.cpp

```

1 #include <iostream>
2 #include <unordered_map>
3 #include <aadc/idouble.h>
4 #include <aadc/ibool.h>
5 #include <aadc/iint.h>
6 #include <aadc/aadc.h>
7 #include <aadc/vreg.h>

```

```

9 using namespace aadc;

11 template<typename mdouble>
 class MyAnalytics : AADC_VREG_BASE {
13 public:
 MyAnalytics() { AADC_VREG_OBJ_ADD }
15 AADC_VREG_SWITCH;

17 virtual mdouble val(const mdouble& a) {
 return (a+1)/a;
19 };

21 mdouble var1, var2;
 };
23

25 void exampleCastingDirect()
 {
27 MyAnalytics<double> my_analytics;
 MyAnalytics<idouble>* aad_my_analytics = (MyAnalytics<idouble>
 >*)&my_analytics;
29
 aadc::VReg::AddLink<MyAnalytics<idouble>, MyAnalytics<double>
 >>();
31 aadc::VReg::SwitchAll();

33 typedef __m256d mmType;
 aadc::AADCFuncs<mmType> aad_funcs;
35
 idouble x(0),y(1),z(0),f;
37
 aad_funcs.startRecording();
39 aadc::AADCArument x_arg(x.markAsInput());
 aadc::AADCArument y_arg(y.markAsInput());
41 aadc::AADCArument z_arg(z.markAsInput());

43 f=std::exp(x/y+z)+(*aad_my_analytics).val(x);

45 aadc::AADCResult f_res(f.markAsOutput());
 aad_funcs.stopRecording();
47

 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
 createWorkspace());
49
 // Executing stage
51 ws->setVal(x_arg,mmSetConst<mmType>(1));
 ws->setVal(y_arg,mmSetConst<mmType>(2));
53 ws->setVal(z_arg,mmSetConst<mmType>(3));

55 aad_funcs.forward(*ws);

57 std::cout
 << " f(" << (ws->valp(x_arg))[0]

```

```

59 << ", " << (ws->valp(y_arg))[0]
60 << ", " << (ws->valp(z_arg))[0]
61 << ")=" << (ws->valp(f_res))[0]
 << std::endl;
63
 ws->setDiff(f_res, 1.0);
65 aad_funcs.reverse(*ws);

67 std::cout
 << "df/dx=" << ws->diffp(x_arg)[0]
69 << std::endl << "df/dy=" << ws->diffp(y_arg)[0]
 << std::endl << "df/dz=" << ws->diffp(z_arg)[0]
71 << std::endl;

73 }
75
 int main() {
77
 exampleCastingDirect();
79
 return 0;
81 }

```

Ex9Casting.cpp

## A.10 Example 10. European Option Monte-Carlo Pricing

```

1 #include <iostream>
 #include <aadc/ibool.h>
3 #include <aadc/aadc.h>
 #include <random>
5 #include <aadc/aadc_matrix.h>
 #include <thread>
7
 using namespace aadc;
9 typedef double Time;

11 template<class vtype>
 class BankRate {
13 public:
 BankRate (vtype _rate) : rate(_rate) {}
15 vtype operator () (const Time& t) const {return rate;}
 ~BankRate() {}
17 public:
 vtype rate;
19 };

21 template<class vtype>
 class AssetVolatility {
23 public:
 AssetVolatility (vtype _vol) : vol(_vol) {}
25 vtype operator () (const vtype asset, const Time& t) const {
 return vol;}
 ~AssetVolatility() {}

```

```

27 public:
 vtype vol;
29 };

31 template<class vtype>
 inline vtype simulateAssetOneStep(
33 const vtype current_value
 , Time current_t
35 , Time next_t
 , const BankRate<vtype>& rate
37 , const AssetVolatility<vtype>& vol_obj
 , const vtype& random_sample
39) {
 double dt = (next_t-current_t);
41 vtype vol=vol_obj(current_value, current_t);
 vtype next_value = current_value * (
43 1 + (-vol*vol / 2+ rate(current_t))*dt
 + vol * std::sqrt(dt) * random_sample
45);
 return next_value;
47 }

49 template<class vtype>
 inline vtype onePathPricing (
51 vtype asset
 , double strike
53 , const std::vector<Time>& t
 , const BankRate<vtype>& rate_obj
55 , const AssetVolatility<vtype>& vol_obj
 , const std::vector<vtype>& random_samples
57) {
 for (int t_i = 0; t_i < t.size()-1; ++t_i) {
59 asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1],
 rate_obj, vol_obj, random_samples[t_i]);
 }
61 return exp(-rate_obj(t.back()) * t.back()) * std::max(asset -
 strike, 0.);
 }

63 inline double compwiseSum(const std::vector<double>& vec)
65 {
 double sum(0.);
67 for(int i=0; i<vec.size(); i++) sum+=vec[i];
 return sum;
69 }

71 void exampleOptionPricing()
 {
73 int num_threads=4;
 typedef __mm256d mmType; // __mm256d and __mm512d are
supported
75
 double init_vol(0.005), init_asset(100.), init_rate(0.3);
77

```

```

std::vector<Time> t(100 , 0.);
79 int num_time_steps=t.size();
for (int i=1; i<num_time_steps; i++) t[i]=i*0.01;
81
double strike=95.;
83 int num_mc_paths=1600;
int AVXsize = aadc::mmSize<mmType>();
85 int paths_per_thread = num_mc_paths / (AVXsize*num_threads);
//assert(num_mc_paths % (num_threads *AADC::getNumAvxElements
<mmType>()) == 0);
87
aadc::AADCFunctions<mmType> aad_funcs;
89
std::vector<idouble> aad_random_samples(num_time_steps, 0.);
91 idouble aad_rate(init_rate), aad_vol(init_vol), aad_asset(
init_asset);
aadc::VectorArg random_arg;
93
aad_funcs.startRecording();
95 // ark the vector of random variables as an input only.
No adjoints for those variables.
markVectorAsInput(random_arg, aad_random_samples, false);
97
aadc::AADCArument rate_arg(aad_rate.markAsInput());
99 aadc::AADCArument asset_arg(aad_asset.markAsInput());
aadc::AADCArument vol_arg(aad_vol.markAsInput());
101
BankRate<idouble> rate_obj(aad_rate);
103 AssetVolatility<idouble> vol_obj(aad_vol);

idouble payoff= onePathPricing(aad_asset, strike, t,
105 rate_obj, vol_obj, aad_random_samples);
aadc::AADCResult payoff_arg(payoff.markAsOutput());
107 aad_funcs.stopRecording();

std::vector<double> total_prices(num_threads, 0.)
, deltas(num_threads, 0.)
111 , vegas(num_threads, 0.)
, rhos(num_threads, 0.)
113 ;

auto threadWorker = [&] (
115 double& total_price
, double& vega
117 , double& delta
, double& rho
119) {
121 std::mt19937_64 gen;
std::normal_distribution<> normal_distrib(0.0, 1.0);
123 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(
aad_funcs.createWorkspace());

125 mmType mm_total_price(mmSetConst<mmType>(0))
, mm_vega(mmSetConst<mmType>(0))

```

## A.10 Example 10. European Option Monte-Carlo Pricing

```

127 , mm_delta(mmSetConst<mmType>(0))
128 , mm_rho(mmSetConst<mmType>(0))
129 ;

131 mmVector<mmType> randoms(num_time_steps);

133 for (int mc_i = 0; mc_i < paths_per_thread; ++mc_i) {
134 for (int j=0; j<num_time_steps; j++) {
135 for (int c=0; c<AVXsize; c++) toDblPtr(randoms[j
136])[c] = normal_distrib(gen);
137 }
138 setAVXVector(*ws, random_arg, randoms);
139 ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
140 ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
141 ;

142 ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));

143 aad_funcs.forward(*ws);
144 mm_total_price = mmAdd(ws->val(payoff_arg),
mm_total_price);
145 //std::cout << ws->val(payoff_arg)[1] << " Val\n";
146 ws->setDiff(payoff_arg, 1.0);
147 aad_funcs.reverse(*ws);

148 mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
149 mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
150 mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
151 }

152 total_price = aadc::mmSum(mm_total_price) / num_mc_paths;
153 rho = aadc::mmSum(mm_rho) / num_mc_paths;
154 delta = aadc::mmSum(mm_delta) / num_mc_paths;
155 vega = aadc::mmSum(mm_vega) / num_mc_paths;
156 };

157 std::vector<std::unique_ptr<std::thread>> threads;
158 for(int i=0; i< num_threads; i++) {
159 threads.push_back(
160 std::unique_ptr<std::thread>(
161 new std::thread(
162 threadWorker
163 , std::ref(total_prices[i])
164 , std::ref(vegas[i])
165 , std::ref(deltas[i])
166 , std::ref(rhos[i])
167)
168)
169);
170 };

171 for(auto&& t: threads) t->join();

172 std::cout << "Price " << compwiseSum(total_prices) << "\n";
173 std::cout << "Delta " << compwiseSum(deltas) << "\n";
174 std::cout << "Vega " << compwiseSum(vegas) << "\n";

```

```

 std::cout << "Rho " << compwiseSum(rhos) << "\n";
179 }

181 int main() {

183 exampleOptionPricing();

185 return 0;
}

```

Ex10OptionPricing.cpp

## A.11 Example 11. Option Pricing without templates

```

#include <iostream>
2 #include <random>
#include <thread>
4
#define AADC_ALLOW_TO_PASSIVE_BOOL
6 #include <aadc/aadc.h>
#include <aadc/aadc_matrix.h>
8
using namespace aadc;
10 typedef double Time;
typedef idouble vtype;
12
enum optionTypes {
14 Asian,
 Digital,
16 European
};
18
class BankRate {
20 public:
 BankRate (vtype _rate) : rate(_rate) {}
22 vtype operator () (const Time& t) const {return rate;}
 ~BankRate() {}
24 public:
 vtype rate;
26 };

28 class AssetVolatility {
public:
30 AssetVolatility (vtype _vol) : vol(_vol) {}
 vtype operator () (const vtype asset, const Time& t) const {
 return vol;}
32 ~AssetVolatility() {}
public:
34 vtype vol;
};
36
vtype simulateAssetOneStep(
38 const vtype current_value
 , Time current_t
40 , Time next_t

```

```

 , const BankRate& rate
42 , const AssetVolatility& vol_obj
 , const vtype& random_sample
44) {
 double dt = (next_t-current_t);
46 vtype vol=vol_obj(current_value, current_t);
 vtype next_value = current_value * (
48 1 + (-vol*vol/2 + rate(current_t))*dt
 + vol * std::sqrt(dt) * random_sample
50);
 return next_value;
52 }

54 vtype onePathPricing (
 vtype asset
56 , double strike
 , const std::vector<Time>& simulation_times
58 , const std::vector<Time>& averaging_base_times
 , const BankRate& rate_obj
60 , const AssetVolatility& vol_obj
 , const std::vector<vtype>& random_samples
62 , const optionTypes payoff_type = optionTypes::Digital
) {
64 vtype one_path_price = vtype(0.);
 for (int t_i = 0; t_i < simulation_times.size()-1; ++t_i) {
66 asset = simulateAssetOneStep(asset, simulation_times[t_i]
, simulation_times[t_i+1], rate_obj, vol_obj, random_samples[
t_i]);
 if (payoff_type == optionTypes::Asian && std::count(
averaging_base_times.begin(), averaging_base_times.end(),
simulation_times[t_i+1])) {
68 one_path_price += asset;
 }
70 }

72 // Choice A: Causes compilation errors since the operator &&
is not defined for ibool
 //if (strike>asset && payoff_type == optionTypes::Digital)
 one_path_price = 1;

74 //Choice B: AADC-compatible analog of Choice A:
76 /*
 {
78 using namespace aadcBoolOps;
 one_path_price = iIf (asset>strike && payoff_type ==
optionTypes::Digital, 1 ,0);
80 }
 */

82 // Choice C: The best choice in this particular situation:
jumps are substituted by smoothings if derivatives are
required.
84 if (payoff_type == optionTypes::Digital) one_path_price = (1
+ tanh((asset-strike) * 1e+1)) / 2;

```



```

86 if (payoff_type == optionTypes::European) one_path_price =
max(asset-strike,vtype(0.));
 if (payoff_type == optionTypes::Asian) one_path_price /=
double(simulation_times.size());
88 return exp(-rate_obj(simulation_times.back()) *
simulation_times.back()) * one_path_price;
}
90
inline double compwiseSum(const std::vector<double>& vec)
92 {
 double sum(0.);
94 for(int i=0; i<vec.size(); i++) sum+=vec[i];
 return sum;
96 }

98 void exampleOptionPricingWithoutTemplates()
{
100 int num_threads=4;
 typedef __m256d mmType; // __mm256d and __mm512d are
supported
102
 double init_vol(0.2), init_asset(100.), init_rate(0.3);
104
 std::vector<Time> simulation_time_steps(100 , 0.);
106 std::size_t num_time_steps = simulation_time_steps.size();
 for (int i=1; i<num_time_steps; i++) simulation_time_steps[i
]=i*0.01;
108
110
 std::vector<Time> averaging_base_times;
112 for (int i=1; i<num_time_steps; i=i+7) averaging_base_times.
push_back(simulation_time_steps[i]);

114 double strike=100.;
 int num_mc_paths=1000000;
116 std::size_t AVXsize = aadc::mmSize<mmType>();
 std::size_t paths_per_thread = num_mc_paths / (AVXsize*
num_threads);
118
 aadc::AADCFUNCTIONS<mmType> aad_funcs;
120 // Use the declaration below to set the debugger breakpoints
to stochastic variable idouble-to-double conversions
/*
122 aadc::AADCFUNCTIONS<mmType> aad_funcs({
 {AADCBreakOnActiveBoolConversion, 1}
124 });
 */
126 std::vector<idouble> aad_random_samples(num_time_steps, 0.);
 idouble aad_rate(init_rate), aad_vol(init_vol), aad_asset(
init_asset);
128 aadc::VectorArg random_arg;

```

```

130 aad_funcs.startRecording();
 // Mark the vector of random variables as an input only.
 No adjoints for those variables.
132 markVectorAsInput(random_arg, aad_random_samples, false);

134 aadc::AADCArument rate_arg(aad_rate.markAsInput());
 aadc::AADCArument asset_arg(aad_asset.markAsInput());
136 aadc::AADCArument vol_arg(aad_vol.markAsInput());

138 BankRate rate_obj(aad_rate);
 AssetVolatility vol_obj(aad_vol);
140

 idouble payoff = onePathPricing(
142 aad_asset, strike, simulation_time_steps,
 averaging_base_times,
 rate_obj, vol_obj, aad_random_samples, optionTypes::
 Digital
144);
 aadc::AADCResult payoff_arg(payoff.markAsOutput());
146 aad_funcs.stopRecording();

148 aad_funcs.printPassiveExtractLocations(std::cout, "
 OptionPricingWithoutTemplates");

150 std::vector<double> total_prices(num_threads, 0.)
 , deltas(num_threads, 0.)
152 , vegas(num_threads, 0.)
 , rhos(num_threads, 0.)
154 ;

156 auto threadWorker = [&] (
 double& total_price
158 , double& vega
 , double& delta
160 , double& rho
) {
162 std::mt19937_64 gen;
 std::normal_distribution<> normal_distrib(0.0, 1.0);
164 std::shared_ptr<aadc::AADCWorkspace<mmType> > ws(
 aad_funcs.createWorkspace());

166 mmType mm_total_price(mmSetConst<mmType>(0))
 , mm_vega(mmSetConst<mmType>(0))
168 , mm_delta(mmSetConst<mmType>(0))
 , mm_rho(mmSetConst<mmType>(0))
170 ;

172 mmVector<mmType> randoms(num_time_steps);

174 for (int mc_i = 0; mc_i < paths_per_thread; ++mc_i) {
 for (int j=0; j<num_time_steps; j++) {
176 for (int c=0; c<AVXsize; c++) toDblPtr(randoms[j
])[c]=normal_distrib(gen);
 }

```

## A.11 Example 11. Option Pricing without templates

```

178 setAVXVector(*ws, random_arg, randoms);
179 ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
180 ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
181 ;
182 ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));
183
184 aad_funcs.forward(*ws);
185 mm_total_price=mmAdd(ws->val(payoff_arg),
mm_total_price);
186 ws->setDiff(payoff_arg, 1.0);
187 aad_funcs.reverse(*ws);
188
189 mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
190 mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
191 mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
192 }
193
194 total_price=aadc::mmSum(mm_total_price)/num_mc_paths;
195 rho=aadc::mmSum(mm_rho)/num_mc_paths;
196 delta=aadc::mmSum(mm_delta)/num_mc_paths;
197 vega=aadc::mmSum(mm_vega)/num_mc_paths;
198 };
199
200 std::vector<std::unique_ptr<std::thread>> threads;
201 for(int i=0; i< num_threads; i++) {
202 threads.push_back(
203 std::unique_ptr<std::thread>(
204 new std::thread(
205 threadWorker
206 , std::ref(total_prices[i])
207 , std::ref(vegas[i])
208 , std::ref(deltas[i])
209 , std::ref(rhos[i])
210)
211)
212);
213 }
214 for(auto&& t: threads) t->join();
215
216 std::cout << "Price " << compwiseSum(total_prices) << "\n";
217 std::cout << "Delta " << compwiseSum(deltas) << "\n";
218 std::cout << "Vega " << compwiseSum(vegas) << "\n";
219 std::cout << "Rho " << compwiseSum(rhos) << "\n";
220 }
221
222 int main() {
223
224 exampleOptionPricingWithoutTemplates();
225
226 return 0;
227 }

```

Ex11OptionPricingWithoutTemplates.cpp

## A.12 Example 12. Option Pricing with Debug

```

#define AADC_CHECK_VALUES
2 #include <iostream>
 #include <random>
4 #include <thread>

6 #include <aadc/ibool.h>
 #include <aadc/aadc.h>
8 #include <aadc/aadc_matrix.h>
 #include <aadc/aadc_debug.h>
10
 using namespace aadc;
12 typedef double Time;

14 template<class vtype>
 class BankRate {
16 public:
 BankRate (vtype _rate) : rate(_rate) {}
18 vtype operator () (const Time& t) const {return rate;}
 ~BankRate() {}
20 public:
 vtype rate;
22 };

24 template<class vtype>
 class AssetVolatility {
26 public:
 AssetVolatility (vtype _vol) : vol(_vol) {}
28 vtype operator () (const vtype asset, const Time& t) const {
 return vol;}
 ~AssetVolatility() {}
30 public:
 vtype vol;
32 };

34 template<class vtype>
 vtype simulateAssetOneStep(
36 const vtype current_value
 , Time current_t
38 , Time next_t
 , const BankRate<vtype>& rate
40 , const AssetVolatility<vtype>& vol_obj
 , const vtype& random_sample
42) {
 double dt = (next_t-current_t);
44 vtype vol=vol_obj(current_value, current_t);
 vtype next_value = current_value * (
46 1 + (-vol*vol / 2+ rate(current_t))*dt
 + vol * std::sqrt(dt) * random_sample
48);
 return next_value;
50 }

```

```

52 template<class vtype>
 vtype onePathPricing_debug (
54 vtype asset
 , double strike
56 , const std::vector<Time>& t
 , const BankRate<vtype>& rate_obj
58 , const AssetVolatility<vtype>& vol_obj
 , const std::vector<vtype>& random_samples
60) {
 AADC_PRINT_DEBUG_COMMENT("collateral update");
62 asset=AADC_PRINT(asset);
 for (int t_i = 0; t_i < t.size()-1; ++t_i) {
64 asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1],
 rate_obj, vol_obj, random_samples[t_i]);
 }
66 return exp(-rate_obj(t.back()) * t.back()) * std::max(asset -
 strike, 0.);
 }

68 inline double compwiseSum(const std::vector<double>& vec)
70 {
 double sum(0.);
72 for(int i=0; i<vec.size(); i++) sum+=vec[i];
 return sum;
74 }

76 void exampleOptionPricingDebug()
 {
78 int num_threads = 1;
 typedef __mm256d mmType; // __mm256d and __mm512d are
supported
80
 double init_vol(0.005), init_asset(100.), init_rate(0.3);
82
 std::vector<Time> t(10 , 0.);
84 int num_time_steps=t.size();
 for (int i=1; i<num_time_steps; i++) t[i]=i*0.01;
86
 double strike=95.;
88 int num_mc_paths=1600;
 int AVXsize = aadc::mmSize<mmType>();
90 int paths_per_thread = num_mc_paths / (AVXsize*num_threads);
 //assert(num_mc_paths % (num_threads * getNumAvxElements<
mmType>()) == 0);
92
 aadc::AADCFunctions<mmType> aad_funcs({
94 {aacdc::AADC_InstrumentCode, 0}
 // ,{aacdc::AADC_DebugStopRecordingAt, 7}
96 });

98 std::vector<idouble> aad_random_samples(num_time_steps, 0.);
 idouble aad_rate(init_rate), aad_vol(init_vol), aad_asset(
init_asset);
100 aadc::VectorArg random_arg;

```

```

102 std::mt19937_64 gen;
103 std::normal_distribution<> normal_distrib(0.0, 1.0);
104 for (int i=0; i< aad_random_samples.size(); i++) {
105 aad_random_samples[i] = markVariableAsRandomInput<double>
106 >(normal_distrib(gen));
107 }

108 aad_funcs.startRecording();
109 // Mark the vector of random variables as an input only.
110 // No adjoints for those variables.
111 markVectorAsInput(random_arg, aad_random_samples, false);

112 aadc::AADCArument rate_arg(aad_rate.markAsInput());
113 aadc::AADCArument asset_arg(aad_asset.markAsInput());
114 aadc::AADCArument vol_arg(aad_vol.markAsInput());

115 BankRate<idouble> rate_obj(aad_rate);
116 AssetVolatility<idouble> vol_obj(aad_vol);

117 idouble payoff = aad_rate;
118 //payoff = payoff/0;

119 payoff = onePathPricing_debug(aad_asset, strike, t,
120 rate_obj, vol_obj, aad_random_samples);
121 payoff += sqrt(0.0000000001*payoff);

122 payoff = AADC_PRINT(payoff);
123 aadc::AADCResult payoff_arg(payoff.markAsOutput());
124 aad_funcs.stopRecording();
125 uint64_t total_print_vars(AADCGetTotalDebugPrints());
126 std::cout << "Total debug prints : " << total_print_vars <<
127 std::endl;

128

129

130

131

132 std::vector<double> total_prices(num_threads, 0.)
133 , deltas(num_threads, 0.)
134 , vegas(num_threads, 0.)
135 , rhos(num_threads, 0.)
136 ;

137

138 auto threadWorker = [&] (
139 double& total_price
140 , double& vega
141 , double& delta
142 , double& rho
143) {
144 std::mt19937_64 gen;
145 std::normal_distribution<> normal_distrib(0.0, 1.0);
146 std::shared_ptr<adc::ADCWorkspace<mmType> > ws(
147 aad_funcs.createWorkspace());

148 mmType mm_total_price(mmSetConst<mmType>(0))
149 , mm_vega(mmSetConst<mmType>(0))

```

```

150 , mm_delta(mmSetConst<mmType>(0))
151 , mm_rho(mmSetConst<mmType>(0))
152 ;
153
154 mmVector<mmType> randomnesses(num_time_steps);
155
156 for (int mc_i=0; mc_i < paths_per_thread; ++mc_i) {
157 for (int c=0; c<AVXsize; c++) {
158 for (int j=0; j<num_time_steps; j++) toDblPtr(
159 randomnesses[j])[c] = normal_distrib(gen);
160 }
161 setAVXVector(*ws, random_arg, randomnesses);
162 ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
163 ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
164 ;
165 ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));
166 aad_funcs.forward(*ws);
167 mm_total_price=mmAdd(ws->val(payoff_arg),
168 mm_total_price);
169
170 ws->setDiff(payoff_arg, 1.0);
171 aad_funcs.reverse(*ws);
172
173 if (mc_i==0) debugTestAdjointsUsingFD(std::cout, *ws,
174 aad_funcs, payoff_arg, true, true);
175
176 mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
177 mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
178 mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
179 }
180
181 total_price=aadc::mmSum(mm_total_price) / num_mc_paths;
182 rho=aadc::mmSum(mm_rho) / num_mc_paths;
183 delta=aadc::mmSum(mm_delta) / num_mc_paths;
184 vega=aadc::mmSum(mm_vega) / num_mc_paths;
185 };
186
187 std::vector<std::unique_ptr<std::thread>> threads;
188 for(int i=0; i < num_threads; i++) {
189 threads.push_back(
190 std::unique_ptr<std::thread>(
191 new std::thread(
192 threadWorker
193 , std::ref(total_prices[i])
194 , std::ref(vegas[i])
195 , std::ref(deltas[i])
196 , std::ref(rhos[i])
197)
198)
199);
200 }
201 for(auto&& t: threads) t->join();
202
203 std::cout << "Price " << compwiseSum(total_prices) << "\n";

```

```

200 std::cout << "Delta " << compwiseSum(deltas) << "\n";
 std::cout << "Vega " << compwiseSum(vegas) << "\n";
202 std::cout << "Rho " << compwiseSum(rhos) << "\n";
 }
204
 int main() {
206 exampleOptionPricingDebug();
208
 return 0;
210 }

```

Ex12OptionPricingDebug.cpp

## A.13 Example 13. Using AADC for external functions

```

#include "aadc/aadc.h"
2 #include "aadc/aadc_ext.h"
#include "aadc/idouable.h"
4 #include <iostream>

6 using namespace aadc;

8
 inline double myfunc(const double& x, const double& y) {
10 return x*y;
 }
12
 class MyFuncAADWrapper : public aadc::ConstStateExtFunc {
14 public:
 MyFuncAADWrapper(idouable& res, const idouable& x, const idouable&
 y)
16 : xi(x), yi(y)
 , resi(res)
18 {
 // This code is executed during the recording stage.
20 res.val = myfunc(x.val,y.val);
 }
22
 template<typename mmType>
24 void forward(mmType* v) const {
 // This code is executed during the forward pass.
26 // Note that we should use a mutex lock here if myfunc is
 not multithread safe.
 for (int avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi)
28 toDb1Ptr(v[resi])[avxi] = myfunc(toDb1Ptr(v[xi])[avxi],
 toDb1Ptr(v[yi])[avxi]);
 }
30
 template<class mmType>
32 void reverse(const mmType* v, mmType* d) const {
 // code to calculate the local gradient and update the
 adjoint variables.
34 double h(0.00001);
 for (int avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi) {

```



```

36 double d1 = (myfunc(toDblPtr(v[xi])[avxi] + h, toDblPtr(
 (v[yi])[avxi]) - myfunc(toDblPtr(v[xi])[avxi] - h, toDblPtr(v[
 yi])[avxi])) / (2 * h);
 double d2 = (myfunc(toDblPtr(v[xi])[avxi], toDblPtr(v[
 yi])[avxi] + h) - myfunc(toDblPtr(v[xi])[avxi], toDblPtr(v[yi
])[avxi] - h)) / (2 * h);
38 toDblPtr(d[xi])[avxi] += toDblPtr(d[resi])[avxi] * d1;
 toDblPtr(d[yi])[avxi] += toDblPtr(d[resi])[avxi] * d2;
40 }
 };
42 private:
 ExtVarIndex xi, yi;
44 ExtVarIndex resi;
 };
46 inline idouble myfunc(const idouble& x, const idouble& y) {
48 idouble res;

50 aadc::addConstStateExtFunction(std::make_shared<
 MyFuncAADWrapper>(res, x, y));

52 return res;
 }
54 template<typename mdouble>
56 mdouble funMath(mdouble& x, mdouble& y) {
 mdouble c=3.0;
58 return myfunc(x,c+y);
 }
60 void exampleExternalFunc()
62 {
 typedef __m256d mmType;
64 aadc::AADCFunctions<mmType> aad_funcs;

66 idouble x(0),y(1),f;

68 aad_funcs.startRecording();
 aadc::AADCArument x_arg(x.markAsInput());
70 aadc::AADCArument y_arg(y.markAsInput());

72 f=funMath(x,y);

74 aadc::AADCResult f_res(f.markAsOutput());
 aad_funcs.stopRecording();
76 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
 createWorkspace());
78 ws->setVal(x_arg,mmSetConst<mmType>(1));
80 ws->setVal(y_arg,mmSetConst<mmType>(2));

82 aad_funcs.forward(*ws);

```

```

84 std::cout
 << " f(" << (ws->valp(x_arg))[0]
86 << "," << (ws->valp(y_arg))[0]
 << ")=" << (ws->valp(f_res))[0]
88 << std::endl;

90
91 ws->setDiff(f_res, 1.0);
92 aad_funcs.reverse(*ws);

94 std::cout
 << "df/dx=" << (ws->diffp(x_arg))[0]
96 << std::endl << "df/dy=" << (ws->diffp(y_arg))[0]
 << std::endl;
98 }

100 int main() {

102 exampleExternalFunc();

104 return 0;
 }

```

Ex13ExternalFunc.cpp

## A.14 Example 14. AAD for Mean Square Error of Expectations

```

1 #include <Eigen/Core>
 #include <LBFGS.h>
3 #include <iostream>
 #include <aadc/aadc.h>
5 #include <random>

7
 using Eigen::VectorXd;
9 using namespace LBFGSpp;
 using namespace aadc;
11
 class Integral {
13 typedef __m256d mmType;
 public:
15 Integral () {
 idouble a,b,x,f0,f1,f2;
17
 v_index.resize(3);
19 res_index.resize(2);

21 aad_funcs = std::make_shared<aadc::AADCFunctions<mmType>
 >();

23 aad_funcs->startRecording();
 v_index[0]=x.markAsInput();
25 v_index[1]=a.markAsInput();

```

## A.14 Example 14. AAD for Mean Square Error of Expectations

```

27 v_index[2]=b.markAsInput();
29 f0=a*x+b*x*x;
31 f1=a*f0;
33 f2=std::exp(f0);
35 res_index[0]=f1.markAsOutput();
37 res_index[1]=f2.markAsOutput();
39 aad_funcs->stopRecording();
41 ws = aad_funcs->createWorkspace();
43 }
45 double operator()(const VectorXd& x, VectorXd& grad)
47 {
49 std::mt19937_64 gen;
51 std::uniform_real_distribution<> uni_distrib(0.0, 1.0);
53 mmType integral1(mmSetConst<mmType>(0)),
55 integral2(mmSetConst<mmType>(0)),
57 D_integral1(mmSetConst<mmType>(0)),
59 D_integral2(mmSetConst<mmType>(0));
61 for (int i=0; i<num_sims_count; ++i) {
63 ws->setVal(v_index[1],mmSetConst<mmType>(x[0]));
65 ws->setVal(v_index[2],mmSetConst<mmType>(x[1]));
67 double *_x = (ws->valp(v_index[0]));
69 for (int ci=0; ci<count; ++ci) {
71 _x[ci]= uni_distrib(gen);
73 }
75 (*aad_funcs).forward(*ws);
77 integral1=mmAdd(ws->val(res_index[0]), integral1);
79 integral2=mmAdd(ws->val(res_index[1]), integral2);
81 }
83 double _integral1(mmSum(integral1));
85 double _integral2(mmSum(integral2));
87 for (int i=0; i<num_sims_count; ++i) {
89 ws->setVal(v_index[1],mmSetConst<mmType>(x[0]));
91 ws->setVal(v_index[2],mmSetConst<mmType>(x[1]));
93 double *_x = (ws->valp(v_index[0]));
95 for (int ci=0; ci<count; ++ci) {
97 _x[ci]= uni_distrib(gen);
99 }
101 (*aad_funcs).forward(*ws);

```

## A.14 Example 14. AAD for Mean Square Error of Expectations

```

ws->resetDiff();
81
ws->setDiff(res_index[0], _integral1-3);
83
ws->setDiff(res_index[1], _integral2-5);

85
(*aad_funcs).reverse(*ws);

87
D_integral1=mmAdd(ws->diff(v_index[1]), D_integral1);
D_integral2=mmAdd(ws->diff(v_index[2]), D_integral2);
89
}

91
grad[0] = mmSum(D_integral1)/num_sims;
grad[1] = mmSum(D_integral2)/num_sims;
93

return 0.5*((_integral1-3)*(_integral1-3)+ (_integral2-5)
*(_integral2-5));
95
}
private:
97
int num_sims=1000; //divisible by avx count
int count = aadc::mmSize<mmType>();
99
int num_sims_count=num_sims/count;

101
std::shared_ptr<aacdc::AADCFUNCTIONS<mmType> > aad_funcs;
std::shared_ptr<aacdc::AADCWorkspace<mmType> > ws;
103
std::vector<aacdc::AADCArgument> v_index;
std::vector<aacdc::AADCResult> res_index;
105 };

107 //Then we just need to set up parameters, create a solver object,
// provide an initial guess, and run the minimization function.
void exampleExpectationsMSE()
109 {
// Set up parameters
111
LBFGSPParam<double> param;
param.epsilon = 1e-6;
113
param.max_iterations = 100;
// Create a solver and function object
115

LBFGSSolver<double> solver(param);
117
Integral fun;
// Initial guess
119
VectorXd x = VectorXd::Zero(2);
// x to be overwritten to the best point found
121
double fx;
int niter = solver.minimize(fun, x, fx);
123

std::cout << niter << " iterations" << std::endl;
125
std::cout << "x = \n" << x.transpose() << std::endl;
std::cout << "f(x) = " << fx << std::endl;
127
}

129 int main() {

131
exampleExpectationsMSE();

```

```

133 return 0;
 }

```

Ex14ExpectationsMSE.cpp

## A.15 Example 15. Recording the Eigen routines using AADC

```

#include <iostream>
2
#include <Eigen/Dense>
4
#include <aadc/aadc.h>
6 #include <aadc/aadc_eigen.h>
#include <aadc/aadc_matrix.h>
8
using namespace aadc;
10
//////////
12 //
// This example demonstrates how to record the Eigen routines
14 // using AADC.
// For matrices A and B it records (A*B).norm(),
16 // calculates the gradient using AADC and
// compares it with the finite difference (F.D.) derivatives
18 // with respect to A[row,col].
//
20 //////////

22 void exampleEigen()
{
24 typedef __m256d mmType;
// At the end we compare the AADC-derivative and the
26 // base-derivative of (A*B).norm() with respect to
// A[row,col].
28 double h = 0.001;
int row = 3, col = 2;
30
Eigen::Matrix<idouble, Eigen::Dynamic, Eigen::Dynamic> A,B;
32 A.resize(4,4);
B.resize(4,2);
34 B << 1,2,3,4,5,6,7,8;
std::vector<aadc::AADCScalarArgument> A_args;
36
aadc::AADCResult norm_res;
38 aadc::AADCFunctions<mmType> aad_funcs;

40 aad_funcs.startRecording();
for(auto& x : A.resaped()) A_args.push_back(x.
markAsScalarInput());
42 idouble norm = (A*B).norm();
norm_res = norm.markAsOutput();
44 aad_funcs.stopRecording();

```

```

46 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
createWorkspace());

48 std::vector<double> inputs(A.rows() * A.cols());
for (int i=0; i < inputs.size(); i++) inputs[i] = i*i;

50
setScalarVectorInput(*ws, A_args, inputs);
52 aad_funcs.forward(*ws);
std::cout << "aadc-value " << ws->valp(norm_res)[0] << "\n";

54
// Check that the aadc-value is correct.
56 Eigen::Matrix<idouble, Eigen::Dynamic, Eigen::Dynamic> A_base
(A.rows(), A.cols());
for (int i=0; i < inputs.size(); i++) A_base.reshapeed()[i] =
inputs[i];
58 idouble base = (A_base*B).norm();
std::cout << "base-value " << base << "\n";

60
// Compute the gradient using AADC:
62 ws->setDiff(norm_res,1);
aad_funcs.reverse(*ws);

64
// Compute the base derivative:
66 A_base(row, col) += h;
std::cout << "base-derivative by A[" << row << "," << col <<
"] = "
68 << ((A_base * B).norm() - base) / h << ", \n"
 << "corrsp. aadc-derivative: "
70 << ws->diff(A_args[col * A.rows() + row]) / aadc::mmSize
<mmType>() << "\n"
;
72 }

74 int main() {

76 exampleEigen();

78 return 0;
}

```

Ex15Eigen.cpp

## A.16 Example 16. Using Eigen with complex *idouble*

```

1 #include <iostream>
#include <iomanip>
3
#define _USE_MATH_DEFINES
5 #include <cmath>

7 #include <Eigen/Dense>
#include <Eigen/unsupported/Eigen/FFT>
9
#include <aadc/aadc.h>

```

```

11 #include <aadc/aadc_eigen.h>
 #include <aadc/icomplex.h>
13 #include <aadc/aadc_matrix.h>

15 using namespace aadc;

17 ///
 // This example implements the AADC-recording of
19 // Eigen::FFT.fft(), applies it to sin(4x)+cos(5x),
 // computes the gradient (at a certain frequency) and
21 // compares it to the finite difference (F.D.) derivative
 // (at a certain x).
23 ///

25 void exampleComplex()
 {
27 typedef __m256d mmType;
 double pi = M_PI;
29 int N = 32;
 Eigen::FFT<idouble> fft;
31
 Eigen::Matrix<std::complex<idouble>, Eigen::Dynamic, 1> G_in,
 G_out;
33 G_in.resize(N);
 for (int i=0; i<N; i++) {
35 // G_in will be marked as Input(), so nothing depends on
 // these values later on:
37 G_in(i) = std::sin(2 * pi / N * i * 5) / N;
 }
39
 aadc::VectorArg real_in_args, imag_in_args;
41 aadc::VectorRes real_out_res, imag_out_res;
 aadc::AADCFuncions<mmType> aad_funcs;
43 aad_funcs.startRecording();
 for (int i=0; i<N; i++) {
45 real_in_args.push_back(G_in(i).real().markAsInput());
 imag_in_args.push_back(G_in(i).imag().markAsInput());
47 }
 fft.fwd(G_out, G_in);
49 for (int i=0; i<N; i++) {
 real_out_res.push_back(G_out(i).real().markAsOutput());
51 imag_out_res.push_back(G_out(i).imag().markAsOutput());
 }
53 aad_funcs.stopRecording();

55 std::shared_ptr<aadc::AADCWorkspace<mmType>> ws(aad_funcs.
 createWorkspace());
 for (int i=0; i<N; i++) {
57 double x = 2 * pi / N * i;
 ws->setVal(real_in_args[i], (std::sin(5 * x) + std::cos(4
 * x)) / N);
59 ws->setVal(imag_in_args[i], 0);
 }
61 aad_funcs.forward(*ws);

```

```

std::cout << "non-zero frequencies: \n";
63 for (int i=0; i<N; i++) {
 if (fabs(ws->valp(real_out_res[i])[0]) + fabs(ws->valp(
imag_out_res[i])[0]) > 1e-10) {
65 std::cout
 << i << std::setprecision(2) << " "
67 << std::left << ws->valp(real_out_res[i])[0] << "
 "
 << std::left << ws->valp(imag_out_res[i])[0] << "
\n"
69 ;
 }
71 }
// We wish to get the gradient of G_out[y_position]:
73 int y_position = 4;
ws->setDiff(real_out_res[y_position], 1);
75 aad_funcs.reverse(*ws);
// Display and compare with F.D. derivative only at one point
77 // called the x_position:
int x_position = 1;
79 std::cout << "aad-J[" << y_position << "," << x_position <<
"] = ";

std::cout << " " << ws->diffp(real_in_args[x_position])[0] <<
\n";
////////////////////////////////////
83 // Compare with the F.D. derivative:
double h = 0.001;
85 idouble base = ws->valp(real_out_res[y_position])[0];
for (int i=0; i<N; i++) {
87 double x = 2*pi/N*i;
 G_in(i) = (std::sin(5 * x) + std::cos(4 * x)) / N;
89 }
G_in(x_position) += h;
91 fft.fwd(G_out, G_in);
std::cout << "base-J[" << y_position
93 << "," << x_position << "] = "
 << (G_out(y_position).real() - base) / h << "\n"
95 ;
}
97
99 int main() {
101 exampleComplex();
103 }

```

Ex16Complex.cpp