HPC AAD-Compiler C++ Software Library User Guide

MatLogica LTD, London, UK

May 4, 2024

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

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

Version: 1.0

1 Overview

We start by explaning 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 binary[2] 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).

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

[2] Currently 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, SIMD[3] 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 succesfully 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.

[3]Single 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

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

or coordinate-wise:

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

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

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

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 `StartRecoring` 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.

[4] or 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 in 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 SIMD[5] to compute the values of the function at multiple points in one pass.

  Note that in the second line we envoke 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);
```

[5] Single 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 continiously compounded interest rate. We denote the expected present value by

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

where $\mathbb{E}$ stands for the expectation and the distibution 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 option[6] 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
```

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

3.3 Example 3. Monte Carlo

We demonstate 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 following[7]:

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

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::AADCArgument random_sample_arg(random_sample.
markAsInputNoDiff());
 aadc::AADCArgument S0_arg(S0.markAsDiff());
 aadc::AADCArgument K_arg(K.markAsDiff());
 aadc::AADCArgument r_arg(r.markAsDiff());
 aadc::AADCArgument vol_arg(vol.markAsDiff());
 aadc::AADCArgument 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 $S(0), K, r$).

[7] Again, 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 `nim_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

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

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

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

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 $\overline{f_i}$ (in this example $i = 1$ or $i = 2$) and setting all the other $\overline{f_j}$, $j \neq i$ 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[8]. 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 $\overline{f_1} = 1$ and $\overline{f_2} = 0$ and in the second line $\overline{f_1} = 0$ and $\overline{f_2} = 1$.

- ```
  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]
  ```

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

```
            << "," << 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

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

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 *iint* type. One has to use the function `get(m_vals,i_s)`[9] 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[10] 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();
```

[9] An analogue of `m_vals[i_s]`.

[10] 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.

```
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::AADCWorkSpace<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

```
******************************
Code size forward : 600928
Code size reverse : 1083661
Work array size   : 16
Stack size        : 5940
Const data size   : 235
Checkpoint size   : 11
******************************
f(3,2)=3.55006
df/dx=0.836758
df/dy=1.25514
```

Without Checkpointing

```
******************************
Code size forward : 597584
Code size reverse : 1080325
Work array size   : 16
Stack size        : 59994
Const data size   : 235
Checkpoint size   : 0
******************************
f(3,2)=3.55006
df/dx=0.836758
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

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

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

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::AADCArgument x_index = aad_x.markAsInputNoDiff();
    aadc::AADCArgument a_index = aad_a.markAsDiff();

        aad_f=funMath(aad_x,aad_a);

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

```
    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;
```

```
    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;
```

```
using namespace aadc;

typedef __m256d mmType;
std::shared_ptr<aadc::AADCFunctions<mmType> >   aad_funcs;
std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws;
aadc::AADCArgument 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[11]. 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 apporach AADC has been succesfully integrated into large opensource projects like QuantLib[12] and ORE[13]

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 *iIf*. 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

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

[12] https://www.quantlib.org/.

[13] 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

$$\text{stochastic } \mathit{ibool} \to \mathit{bool}$$

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[14] 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

[14] https://en.cppreference.com/w/cpp/language/operator_logical

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:

```
#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[15].

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

```
template<typename mdouble>
class MyAnalytics  {
public:

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

  mdouble var1, var2;
};
```

Modified user code

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

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

  mdouble var1, var2;
};
```

Now, establish a link between the two versions of the `MyAnalytics` class calling

```
aadc::VReg::AddLink<MyAnalytics<idouble>, MyAnalytics<double>>();
```

[15] 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 debuger:

- Add `#include <aadc/aadc_debug.h>`.
- Choose the point in the code and the variable you want to scrutinize. Wrap this variable[16] 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::AADCResult payoff_arg(payoff.markAsOutput());
```

[16] `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    1.00000000000034 [rv,adj]
AADCDebug::0000000000::Adj[0]:FD_left   1.00000000000165 [rv,adj]
AADCDebug::0000000000::Adj[0]:FD_right  1.00000000000165 [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[17] 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 NaN $* 0 =$ 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[18]. 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 NaN $* 0 =$ 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);
```

[17] https://en.wikipedia.org/wiki/Finite_difference_coefficient

[18] Equivalently, recompile with the option `{aadc::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[19] will set the debugger breakpoint to the location of the variable with index 73.

7 Advanced Methods

We 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 oveloaded 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;
}
```

[19] Equivalently, recompiling with the option `{aadc::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[20] 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[21].

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

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

   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

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

[21] D. Goloubentsev, E. Lakshtanov, *Remarks on stochastic automatic adjoint differentiation and financial models calibration.*

```
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);
```

```
            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
        LBFGSParam<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[22], so we recommend to avoid using `std::vector<mmType>` and use `aadc::mmVector<mmType>` instead. Examples can be found in A.11.

[22] 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_indx)`
  `double& val(const AADCResult& var_indx)`
  `mmType& val(const AADCArgument& var_indx)`
  `mmType& val(const AADCResult& var_indx)`
  Give the vector value of the variable with reference `var_indx`.
- `mmType& diff(const AADCArgument& var_indx)`
  Gives the vector value of the adjoint variable with reference `var_indx`.
- `AADCWorkSpace& setVal(const AADCArgument& var_indx, const mmType& val)`
  Attributes the vector value of the variable with reference `var_indx`. 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::AADCArgument> VectorArg;
typedef std::vector<aadc::AADCResult> VectorRes;
typedef std::vector<aadc::AADCScalarArgument> 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)
```

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 Checkpoiting 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 (Exampes 8 and 10), Section 6 (Example 11) and Section 7 (Examples 12 and 13).

A.1 Example 1. Hello, World!

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

void exampleHelloWorld()
{
    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
  ////////////////////
    // MANUAL Initializing the AADC Library
    typedef __m256d mmType;
  aadc::AADCFunctions<mmType> aad_funcs;
    // MANUAL
    // MANUAL StartRecording and inputs
    aad_funcs.startRecording();
    aadc::AADCArgument x_arg(x.markAsInput());
    aadc::AADCArgument y_arg(y.markAsInput());
    aadc::AADCArgument z_arg(z.markAsInput());
    // MANUAL
    // Here the calculations are simple, but in practice this
    // call can travel through a lot of calls to the
    // object-oriented code.
    // MANUAL Output
    f=std::exp(x/y+z);
    aadc::AADCResult f_res(f.markAsOutput());
    aad_funcs.stopRecording();
    // MANUAL
    // MANUAL Execution context
    std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
   createWorkSpace());
    // MANUAL
  ///////////////////////
    // Execute the function
```

```
44  ////////////////////

46  // set initial values for variables x, y & z
      ws->setVal(x_arg, _mm256_set_pd(1.0, 2.0, 3.0, 4.0));
48    ws->setVal(y_arg, 2.0);
      ws->setVal(z_arg, _mm256_set_pd(0.1, 0.2, 0.3, 0.4));
50
    // execute the forward function
52    aad_funcs.forward(*ws);

54  // check the results for the forward function, these
      // should match the original f=exp(x/y+z)
56    for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi
     ) {
          std::cout
58            << "avx["<<avxi<<"]"
              // MANUAL Setting input variables
60            << " f(" << ws->valp(x_arg)[avxi]
              << ","   << ws->valp(y_arg)[avxi]
62            << ","   << ws->valp(z_arg)[avxi]
              << ")="  << ws->valp(f_res)[avxi]
64            // MANUAL
          << std::endl;
66    }

68  // prepare for the reverse pass, set f() equal to 1
      // MANUAL Initialize adjoints
70    ws->setDiff(f_res, 1.0);
      // MANUAL
72
    // execute the reverse pass
74    // MANUAL Execute reverse
      aad_funcs.reverse(*ws);
76    // MANUAL

78  // now we have all requested derivatives and can print them out
    // (below we print only the first element of the __m256d
80    // type)
      for (uint64_t avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi
     ) {
82        std::cout
              << "avx["<<avxi<<"]"
84            << " df/dx="  << ws->diffp(x_arg)[avxi] << "," << "df
     /dy="  << ws->diffp(y_arg)[avxi] <<  ","<< "df/dz="  << ws->
     diffp(z_arg)[avxi]
          << std::endl;
86    }
      std::cout << "Example 1 is done" << std::endl;
88 }

90
  int main() {
92
      exampleHelloWorld();
```

```
94
      return 0;
96 }
```

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
  template<typename mdouble>
14 mdouble BlackScholes(
      mdouble S0, mdouble K, mdouble r, mdouble vol, mdouble T
16 ) {
      return S0*std::cdf_normal((std::log(S0/K) + T*(r + 0.5 * (std
     ::pow(vol, 2))))/
18     (vol*std::sqrt(T)))-std::exp(-r*T)*K*std::cdf_normal((std::
     log(S0/K)+
      T*(r-0.5*(std::pow(vol, 2))))/(vol*std::sqrt(T)));
20 }

22 // MANUAL Finite Differences derivative
  // Computes the Finite Differences derivative
24 // with respect to the first variable S0
  template<typename mdouble>
26 mdouble BlackScholesFD(
      mdouble S0, mdouble K, mdouble r, mdouble vol, mdouble T,
     mdouble h
28 ) {
      return (BlackScholes(S0 + h, K, r, vol, T) - BlackScholes(S0,
      K, r, vol, T))/h;
30 }
  // MANUAL
32

34 void exampleBlackScholes()
  {
36     typedef __m256d mmType;
    aadc::AADCFunctions<mmType> aad_funcs;
38
      idouble S0,K,r,vol,T,BS;
40
      S0 = 100.0;
42     K = 110.0;
```

```
    r = 0.1;
44  vol = 0.25;
    T = 1.0;
46
    aad_funcs.startRecording();
48
    aadc::AADCArgument S0_arg(S0.markAsInput());
50  aadc::AADCArgument K_arg(K.markAsInput());
    aadc::AADCArgument r_arg(r.markAsInput());
52  aadc::AADCArgument vol_arg(vol.markAsInput());
    aadc::AADCArgument T_arg(T.markAsInput());
54
        BS=BlackScholes(S0,K,r,vol,T);
56
aadc::AADCResult BS_res(BS.markAsOutput());
58
    aad_funcs.stopRecording();
60
    std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(aad_funcs.
   createWorkSpace());
62
    ws->setVal(S0_arg, 100.0);
64  ws->setVal(K_arg, 110.0);
    ws->setVal(r_arg, _mm256_set_pd(0.1, 0.1, 0.01, 0.1));
66  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));
68
    aad_funcs.forward(*ws);
70
    ws->setDiff(BS_res, 1.0);
72
    aad_funcs.reverse(*ws);
74  // Print out the values of the function and derivatives with
   respect to S0, K, r.
    for (int i=0; i<4; ++i) {
76      std::cout <<  " BS(" <<ws->valp(S0_arg)[0]
            << "," << ws->valp(K_arg)[i]
78          << "," << ws->valp(r_arg)[i]
            << "," << ws->valp(vol_arg)[i]
80          << "," << ws->valp(T_arg)[i]
            << ")="<< ws->valp(BS_res)[i]
82          << ".  dBS/dS0="  << ws->diffp(S0_arg)[i]
            << ";  dBS/dK="  << ws->diffp(K_arg)[i]
84          << ";  dBS/dr="  << ws->diffp(r_arg)[i]
        << std::endl;
86  }
    // Print out the Finite Differences derivatives
88  // with respect to the first variable S0 and with h=0.000001
    std::cout << "dBS/dS0(100,110,0.1,0.25,1)=" <<
90  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)=" <<
92  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)=" <<
94  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)=" <<
96  BlackScholesFD(100.0,110.0,0.1,0.5,2.0, 0.000001) << std::
    endl;

98 }

100 int main() {

102     exampleBlackScholes();

104     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  // MANUAL Adjusting the number of simulations
    int num_sim=1000000;
17  int count = aadc::mmSize<mmType>();
    int num_sim_avx=(int)floor(num_sim/count);
19  // MANUAL

21  aadc::AADCFunctions<mmType> aad_funcs;

23  aad_funcs.startRecording();
    // MANUAL Introducing the variables
25  aadc::AADCArgument random_sample_arg(random_sample.
    markAsInputNoDiff());
    aadc::AADCArgument S0_arg(S0.markAsDiff());
27  aadc::AADCArgument K_arg(K.markAsDiff());
    aadc::AADCArgument r_arg(r.markAsDiff());
29  aadc::AADCArgument vol_arg(vol.markAsDiff());
    aadc::AADCArgument T_arg(T.markAsDiff());
31  // MANUAL

33  // MANUAL Recording the math operations
    idouble S_T = S0 * std::exp((r - vol*vol / 2) * T +
```

```
35      vol * std::sqrt(T) * random_sample);
        idouble price = std::exp(-r*T) * std::max(S_T-K, 0.);
37
        aadc::AADCResult price_res(price.markAsOutput());
39      aad_funcs.stopRecording();
        // MANUAL
41
        std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(aad_funcs.
      createWorkSpace());
43      // MANUAL Integral sums
        mmType integral_S0(mmSetConst<mmType>(0)), integral_K(
      mmSetConst<mmType>(0)),
45      integral_r(mmSetConst<mmType>(0)), integral(mmSetConst<mmType
      >(0));
        // MANUAL
47      // MANUAL Initialize the loop
        for (int i=0; i<num_sim_avx; ++i) {
49
            double* _x = (ws->valp(random_sample_arg));
51          for (int ci=0; ci<count; ++ci) {
                _x[ci]= norm_distrib(gen);
53          }

55          aad_funcs.forward(*ws);
            // MANUAL
57          integral=aadc::mmAdd(ws->val(price_res), integral);
            // MANUAL Initialize and backward pass
59          ws->resetDiff();
            ws->setDiff(price_res, 1.0);
61
            aad_funcs.reverse(*ws);
63          integral_S0=mmAdd(ws->diff(S0_arg), integral_S0);
            integral_K=mmAdd(ws->diff(K_arg), integral_K);
65          integral_r=mmAdd(ws->diff(r_arg), integral_r);
            // MANUAL
67      }
        // MANUAL Print out the results
69      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=" <<
71      aadc::mmSum(integral_K) / num_sim << "; df/dr=" <<
        aadc::mmSum(integral_r) / num_sim << std::endl;
73      // MANUAL
    }
75
    int main() {
77
        exampleMonteCarlo();
79
        return 0;
81  }
```

Ex3MonteCarlo.cpp

A.4 Example 4. Jacobian

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

using namespace aadc;

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

    idouble x,y,z,f1,f2;

    aad_funcs.startRecording();

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

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

  aadc::AADCResult f1_res(f1.markAsOutput());
  aadc::AADCResult f2_res(f2.markAsOutput());

    aad_funcs.stopRecording();

    std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(aad_funcs.
   createWorkSpace());
    // MANUAL Setting input variables
    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));
    // MANUAL
    aad_funcs.forward(*ws);
    // MANUAL Initialize adjoints
    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));
    // MANUAL

    aad_funcs.reverse(*ws);
  // MANUAL Print the derivatives
    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]
            << "," << 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]
53          << ";  df" << dfi <<"/dz="  << ws->diffp(z_arg)[i]
        << std::endl;
55  }
    // MANUAL
57 }

59 int main() {

61     exampleJacobianFull();

63     return 0;
   }
```

Ex4Jacobian.cpp

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

```
 #include <iostream>
2 #include <random>
  #include <aadc/idouble.h>
4 #include <aadc/aadc.h>
  // MANUAL PiecewiseCurve
6 #include <aadc/ibool.h>
  #include <aadc/iint.h>
8
  using namespace aadc;
10 using namespace aadcArrayOps;

12 template<typename mdouble>
  class PiecewiseCurve {
14 public:
     PiecewiseCurve(const std::vector<double>& nodes)
16       : m_s(nodes)
    {}
18
    void init(const std::vector<double>& vals) {
20      m_vals.resize(m_s.size()+1);
        for (int i=0; i<m_vals.size(); i++) {
22          m_vals[i]=vals[i];
        }
24  }

26  mdouble interpolatedValue (const mdouble& s) {
        auto i_s = lower_bound(m_s, s);
28      return get(m_vals,i_s);
    }
30
    std::vector<mdouble> m_vals;
32  std::vector<double> m_s;
  };
34 // MANUAL
```

```
36
  template<typename mdouble>
38 mdouble funMath(
       mdouble& x, PiecewiseCurve<mdouble>& m_curve
40     ) {
       return m_curve.interpolatedValue(x);
42 }

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

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

60   aadc::AADCFunctions<mmType> aad_funcs;

62     PiecewiseCurve<idouble> aad_curve(curve_nodes);
       aad_curve.init(curve_vals2);
64
       idouble x,f;
66
       aad_funcs.startRecording();
68
       aadc::AADCArgument x_arg(x.markAsInput());
70     std::vector<aadc::AADCArgument> curve_val_args(aad_curve.
     m_vals.size());
       for (int i=0; i<aad_curve.m_vals.size(); ++i) {
72         curve_val_args[i] = aad_curve.m_vals[i].markAsDiff();
       }
74
           f=funMath(x,aad_curve);
76
       aadc::AADCResult f_res(f.markAsOutput());
78     aad_funcs.stopRecording();

80     std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
     createWorkSpace());

82     mmVector<mmType> D_integral(aad_curve.m_vals.size());
       std::fill(D_integral.begin(),D_integral.end(),mmSetConst<
     mmType>(0));
84     double integral(0);

86     for (int i=0; i<aad_curve.m_vals.size(); ++i) {
```

```
            ws->setVal(curve_val_args[i], mmSetConst<mmType>(
    curve_vals2[i]));
88      }

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

92      for (int ci=0; ci<count; ++ci) {
          auto _x = (ws->valp(x_arg));
94          _x[ci] = normal_distrib(gen);
        }
96
        aad_funcs.forward(*ws);
98
        integral+= aadc::mmSum(ws->val(f_res));
100
        ws->resetDiff();
102     ws->setDiff(f_res, 1.0);
        aad_funcs.reverse(*ws);
104
        for (int i=0; i<aad_curve.m_vals.size(); ++i) {
106         D_integral[i]=aadc::mmAdd(ws->diff(curve_val_args[i])
    , D_integral[i]);
        }
108  }

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

118 }

120 int main() {

122     exampleInterpolation();

124     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()
```

```
{
    typedef __m256d mmType;
  aadc::AADCFunctions<mmType> aad_funcs;
    // MANUAL Checkpointing effectiveness
    idouble x(0),y(1),z,f;


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

    std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
   createWorkSpace());

    std::cout << "************************************" << std
   ::endl;
    std::cout << "Code size forward : " << aad_funcs.
   getCodeSizeFwd() << std::endl;
    std::cout << "Code size reverse : " << aad_funcs.
   getCodeSizeRev() << std::endl;
    std::cout << "Work array size   : " << aad_funcs.
   getWorkArraySize() << std::endl;
    std::cout << "Stack size        : " << aad_funcs.getStackSize
   () << std::endl;
    std::cout << "Const data size   : " << aad_funcs.
   getConstDataSize() << std::endl;
    std::cout << "Checkpoint size   : " << aad_funcs.
   getNumCheckPoints() << std::endl;
    std::cout << "************************************" << std::
   endl;
    // MANUAL
    double x_init(3),y_init(2);

    ws->setVal(x_ind,mmSetConst<mmType>(x_init));
    ws->setVal(y_ind,mmSetConst<mmType>(y_init));

    aad_funcs.forward(*ws);

  double *_f = (ws->valp(f_ind));
    std::cout
        << " f(" << x_init
        << "," << y_init
        << ")="<< _f[0]
    << 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 // MANUAL Threads and function declaration
  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 }
  // MANUAL
24 // MANUAL MultiThread
  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();
34  aadc::AADCArgument x_index = aad_x.markAsInputNoDiff();
    aadc::AADCArgument a_index = aad_a.markAsDiff();
36
        aad_f=funMath(aad_x,aad_a);
38
    aadc::AADCResult f_index = aad_f.markAsOutput();
40  aad_funcs.stopRecording();

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

44  auto threadWorker = [&] (
        double &integral,
46      double &d_integral,
        const int i
48  ) {
        std::mt19937_64 gen;
50      std::uniform_real_distribution<> uni_distrib(0.0, 1.0);
        gen.seed(i*17+31);
52
        std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(
    aad_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;
60      double* _rand_mm = toDblPtr(rand_mm);
            for (int ci=0; ci<avx_count; ++ci) {
62              _rand_mm[ci]= uni_distrib(gen);
            }
64          ws->setVal(x_index, rand_mm);

66          aad_funcs.forward(*ws);
            integral_mm = aadc::mmAdd(ws->val(f_index),
    integral_mm);
68
            ws->resetDiff();
70          ws->setDiff(f_index, 1.0);

72          aad_funcs.reverse(*ws);
            d_integral_mm = aadc::mmAdd(ws->diff(a_index),
    d_integral_mm);
74      }

76  d_integral = mmSum(d_integral_mm);
        integral   = mmSum(integral_mm);
78  };
    // MANUAL
80  // MANUAL Partial sums
```

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

}
// MANUAL

int main() {

    exampleMultithread();

    return 0;
}
```

Ex7MultiThread.cpp

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

```
#include <Eigen/Core>
#include <LBFGS.h>
#include <iostream>
#include <aadc/aadc.h>
// MANUAL Declarations and recording
using Eigen::VectorXd;
using namespace LBFGSpp;
using namespace aadc;

typedef __m256d mmType;
std::shared_ptr<aadc::AADCFunctions<mmType> >   aad_funcs;
std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws;
aadc::AADCArgument 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();
}
// MANUAL

// MANUAL Defining the function
double calcGrad(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];
}
// MANUAL

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

```
67
        LBFGSSolver<double> solver(param);
69      record();

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

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

83
    int main() {
85
        exampleMinimization();
87
        return 0;
89  }
```

Ex8Minimization.cpp

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

```
1   // MANUAL MyAnalytics
    #ifndef _MY_ANALYTICS_H
3   #define _MY_ANALYTICS_H

5   #include "aadc/idouble.h"

7
    namespace UserNameSpace {
9
        class MyAnalytics {
11      public:
            // ...
13
            mdouble val(const mdouble& a);
15
            mdouble var1, var2;
17      };


19
    };
21

23  #ifndef AADC_PASS
        #define AADC_PASS
25      #undef _MY_ANALYTICS_H
```

```
    namespace AAD {
27      #include "MyAnalytics.h"
    };
29  #undef AADC_PASS
#endif
31
#endif // _MY_ANALYTICS_H
33 // MANUAL
```

MyAnalytics.h

```
1 // MANUAL MyAnalytics
#include "MyAnalytics.h"
3
// MANUAL Virtual method user code
5 namespace UserNameSpace {

7   mdouble MyAnalytics::val(const mdouble& a) {
      return (a+1)/a;
9   }

11 };

13 // MANUAL

15 #ifndef AADC_PASS
      #define AADC_PASS
17    namespace AAD {
        #include "MyAnalytics.cpp"
19    };
      #undef AADC_PASS
21 #endif
// MANUAL
```

MyAnalytics.cpp

```
#include <iostream>
2 #include <unordered_map>
#include <aadc/idouble.h>
4 #include <aadc/ibool.h>
#include <aadc/iint.h>
6 #include <aadc/aadc.h>
#include <aadc/vreg.h>
8
using namespace aadc;
10
template<typename mdouble>
12 class MyAnalytics : AADC_VREG_BASE {
public:
14    MyAnalytics() { AADC_VREG_OBJ_ADD }
      AADC_VREG_SWITCH;
16
      virtual mdouble val(const mdouble& a) {
18      return (a+1)/a;
      };
20
```

```
    mdouble var1, var2;
};


void exampleCastingDirect()
{
    MyAnalytics<double> my_analytics;
    MyAnalytics<idouble>* aad_my_analytics = (MyAnalytics<idouble
    >*)(&my_analytics);

    // MANUAL Virtual method link
    aadc::VReg::AddLink<MyAnalytics<idouble>, MyAnalytics<double
    >>();
    // MANUAL
    aadc::VReg::SwitchAll();

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

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

    aad_funcs.startRecording();
    aadc::AADCArgument x_arg(x.markAsInput());
    aadc::AADCArgument y_arg(y.markAsInput());
    aadc::AADCArgument z_arg(z.markAsInput());

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

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

    std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
    createWorkSpace());

    // Executing stage
    ws->setVal(x_arg,mmSetConst<mmType>(1));
    ws->setVal(y_arg,mmSetConst<mmType>(2));
    ws->setVal(z_arg,mmSetConst<mmType>(3));

    aad_funcs.forward(*ws);

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

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

    std::cout
        << "df/dx="  << ws->diffp(x_arg)[0]
        << std::endl << "df/dy=" << ws->diffp(y_arg)[0]
```

```
72          << std::endl << "df/dz=" << ws->diffp(z_arg)[0]
      << std::endl;
74

76 }

78 int main() {

80     exampleCastingDirect();

82     return 0;
  }
```

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
) {
    double dt = (next_t-current_t);
    vtype vol=vol_obj(current_value, current_t);
    vtype next_value = current_value * (
        1 + (-vol*vol / 2+ rate(current_t))*dt
        +  vol * std::sqrt(dt) * random_sample
    );
    return next_value;
}

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

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

void exampleOptionPricing()
{
    int num_threads=4;
    typedef __m256d mmType;  // __mm256d and __mm512d are
    supported

    double init_vol(0.005), init_asset(100.), init_rate(0.3);

    std::vector<Time> t(100 , 0.);
    int num_time_steps=t.size();
    for (int i=1; i<num_time_steps; i++) t[i]=i*0.01;

    double strike=95.;
    int num_mc_paths=1600;
    int AVXsize = aadc::mmSize<mmType>();
    int paths_per_thread = num_mc_paths / (AVXsize*num_threads);
    //assert(num_mc_paths % (num_threads *AADC::getNumAvxElements
    <mmType>()) == 0);

```

```
    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::AADCArgument rate_arg(aad_rate.markAsInput());
99      aadc::AADCArgument asset_arg(aad_asset.markAsInput());
        aadc::AADCArgument vol_arg(aad_vol.markAsInput());
101
        BankRate<idouble> rate_obj(aad_rate);
103     AssetVolatility<idouble> vol_obj(aad_vol);

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

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

115 auto threadWorker = [&] (
        double& total_price
117     , double& vega
        , double& delta
119     , double& rho
    ) {
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))
127         , mm_delta(mmSetConst<mmType>(0))
            , 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) {
            for (int j=0; j<num_time_steps; j++) {
135             for (int c=0; c<AVXsize; c++) toDblPtr(randoms[j
    ])[c] = normal_distrib(gen);
            }
```

```
137             setAVXVector(*ws, random_arg, randoms);
                ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
139             ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
    ;
                ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));
141
                aad_funcs.forward(*ws);
143             mm_total_price = mmAdd(ws->val(payoff_arg),
    mm_total_price);
                //std::cout << ws->val(payoff_arg)[1] << " Val\n";
145             ws->setDiff(payoff_arg, 1.0);
                aad_funcs.reverse(*ws);
147
                mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
149             mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
                mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
151         }

153         total_price = aadc::mmSum(mm_total_price) / num_mc_paths;
            rho = aadc::mmSum(mm_rho) / num_mc_paths;
155         delta = aadc::mmSum(mm_delta) / num_mc_paths;
            vega = aadc::mmSum(mm_vega) / num_mc_paths;
157     };

159     std::vector<std::unique_ptr<std::thread>> threads;
        for(int i=0; i< num_threads; i++) {
161         threads.push_back(
                std::unique_ptr<std::thread>(
163                 new std::thread(
                        threadWorker
165                     , std::ref(total_prices[i])
                        , std::ref(vegas[i])
167                     , std::ref(deltas[i])
                        , std::ref(rhos[i])
169                 )
                )
171         );
        }
173     for(auto&& t: threads) t->join();

175     std::cout << "Price " << compwiseSum(total_prices) << "\n";
        std::cout << "Delta " << compwiseSum(deltas) << "\n";
177     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>
#include <random>
#include <thread>
// MANUAL Pragma
#define AADC_ALLOW_TO_PASSIVE_BOOL
#include <aadc/aadc.h>
// MANUAL
#include <aadc/aadc_matrix.h>

using namespace aadc;
typedef double Time;
typedef idouble vtype;

enum optionTypes {
    Asian,
    Digital,
    European
};

class BankRate {
public:
    BankRate (vtype _rate) : rate(_rate) {}
    vtype operator () (const Time& t) const {return rate;}
    ~BankRate() {}
public:
    vtype rate;
};

class AssetVolatility {
public:
    AssetVolatility (vtype _vol) : vol(_vol) {}
    vtype operator () (const vtype asset, const Time& t) const {
    return vol;}
    ~AssetVolatility() {}
public:
    vtype vol;
};

vtype simulateAssetOneStep(
    const vtype current_value
    , Time current_t
    , Time next_t
    , const BankRate& rate
    , const AssetVolatility& vol_obj
    , const vtype& random_sample
) {
    double dt = (next_t-current_t);
    vtype vol=vol_obj(current_value, current_t);
    vtype next_value = current_value * (
        1 + (-vol*vol/2 + rate(current_t))*dt
        +  vol * std::sqrt(dt) * random_sample
    );
```

```
52      return next_value;
  }
54
  vtype onePathPricing (
56      vtype asset
        , double strike
58      , const std::vector<Time>& simulation_times
        , const std::vector<Time>& averaging_base_times
60      , const BankRate& rate_obj
        , const AssetVolatility& vol_obj
62      , const std::vector<vtype>& random_samples
        , const optionTypes payoff_type = optionTypes::Digital
64  ) {
        vtype one_path_price = vtype(0.);
66      for (int t_i = 0; t_i < simulation_times.size()-1; ++t_i) {
            asset = simulateAssetOneStep(asset, simulation_times[t_i
        ], simulation_times[t_i+1], rate_obj, vol_obj, random_samples[
        t_i]);
68          // MANUAL Example no changes 2
            if (payoff_type == optionTypes::Asian && std::count(
        averaging_base_times.begin(), averaging_base_times.end(),
        simulation_times[t_i+1])) {
70              one_path_price += asset;
            }
72          // MANUAL
        }
74
        // Choice A: Causes compilation errors since the operator &&
        is not defined for ibool
76      // MANUAL Stochastic condition
        // if (strike>asset && payoff_type == optionTypes::Digital)
        one_path_price = 1;
78      // MANUAL

80      //Choice B:  AADC-compatible analog of Choice A:
        /*
82      // MANUAL Changed Stochastic condition
        {
84          using namespace aadcBoolOps;
            one_path_price = iIf (asset>strike && payoff_type ==
        optionTypes::Digital, 1 ,0);
86      }
        // MANUAL
88      */

90     // Choice C: The best choice in this particular situation:
        jumps are substituted by smoothings if derivatives are
        required.
        // MANUAL Smoothed Stochastic condition
92      if (payoff_type == optionTypes::Digital) one_path_price = (1
        + tanh((asset-strike) * 1e+1)) / 2;
        // MANUAL Example no changes 1
94      if (payoff_type == optionTypes::European) one_path_price =
        max(asset-strike,vtype(0.));
```

```
    // MANUAL
    if (payoff_type == optionTypes::Asian) one_path_price /=
    double(simulation_times.size());
    return exp(-rate_obj(simulation_times.back()) *
    simulation_times.back()) * one_path_price;
    // MANUAL
}

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

void exampleOptionPricingWithoutTemplates()
{
    int num_threads=4;
    typedef __m256d mmType;  // __mm256d and __mm512d are
    supported

    double init_vol(0.2), init_asset(100.), init_rate(0.3);

    std::vector<Time> simulation_time_steps(100 , 0.);
    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;



    std::vector<Time> averaging_base_times;
    for (int i=1; i<num_time_steps; i=i+7) averaging_base_times.
    push_back(simulation_time_steps[i]);

    double strike=100.;
    int num_mc_paths=1000000;
    std::size_t AVXsize = aadc::mmSize<mmType>();
    std::size_t paths_per_thread = num_mc_paths / (AVXsize*
    num_threads);

    aadc::AADCFunctions<mmType> aad_funcs;
    // Use the declaration below to set the debugger breakpoints
    to stochastic variable idouble-to-double conversions
    /*
    // MANUAL Positive number of conversions
    aadc::AADCFunctions<mmType> aad_funcs( {
        {AADC_BreakOnActiveBoolConversion, 1}
    });
    // MANUAL
    */
    std::vector<idouble> aad_random_samples(num_time_steps, 0.);
    idouble aad_rate(init_rate), aad_vol(init_vol), aad_asset(
    init_asset);
    aadc::VectorArg random_arg;
```

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

146         aadc::AADCArgument rate_arg(aad_rate.markAsInput());
            aadc::AADCArgument asset_arg(aad_asset.markAsInput());
148         aadc::AADCArgument vol_arg(aad_vol.markAsInput());

150         BankRate rate_obj(aad_rate);
            AssetVolatility vol_obj(aad_vol);
152         // MANUAL Wrapping many variables
            idouble payoff = onePathPricing(
154             aad_asset, strike, simulation_time_steps,
        averaging_base_times,
                rate_obj, vol_obj, aad_random_samples, optionTypes::
        Digital
156         );
            aadc::AADCResult payoff_arg(payoff.markAsOutput());
158     aad_funcs.stopRecording();
        // MANUAL
160
        // MANUAL Conversions
162   aad_funcs.printPassiveExtractLocations(std::cout, "
        OptionPricingWithoutTemplates");
        // MANUAL
164
        std::vector<double> total_prices(num_threads, 0.)
166         , deltas(num_threads, 0.)
            , vegas(num_threads, 0.)
168         , rhos(num_threads, 0.)
        ;
170
        auto threadWorker = [&] (
172         double& total_price
            , double& vega
174         , double& delta
            , double& rho
176     ) {
            std::mt19937_64 gen;
178         std::normal_distribution<> normal_distrib(0.0, 1.0);
            std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(
        aad_funcs.createWorkSpace());
180
            mmType mm_total_price(mmSetConst<mmType>(0))
182             , mm_vega(mmSetConst<mmType>(0))
                , mm_delta(mmSetConst<mmType>(0))
184             , mm_rho(mmSetConst<mmType>(0))
            ;
186
            mmVector<mmType> randoms(num_time_steps);
188
            for (int mc_i = 0; mc_i < paths_per_thread; ++mc_i) {
```

```
190             for (int j=0; j<num_time_steps; j++) {
                    for (int c=0; c<AVXsize; c++) toDblPtr(randoms[j
    ])[c]=normal_distrib(gen);
192             }
                setAVXVector(*ws, random_arg, randoms);
194             ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
                ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
    ;
196             ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));

198             aad_funcs.forward(*ws);
                mm_total_price=mmAdd(ws->val(payoff_arg),
    mm_total_price);
200             ws->setDiff(payoff_arg, 1.0);
                aad_funcs.reverse(*ws);
202
                mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
204             mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
                mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
206         }

208         total_price=aadc::mmSum(mm_total_price)/num_mc_paths;
            rho=aadc::mmSum(mm_rho)/num_mc_paths;
210         delta=aadc::mmSum(mm_delta)/num_mc_paths;
            vega=aadc::mmSum(mm_vega)/num_mc_paths;
212     };

214     std::vector<std::unique_ptr<std::thread>> threads;
        for(int i=0; i< num_threads; i++) {
216         threads.push_back(
                std::unique_ptr<std::thread>(
218                 new std::thread(
                        threadWorker
220                     , std::ref(total_prices[i])
                        , std::ref(vegas[i])
222                     , std::ref(deltas[i])
                        , std::ref(rhos[i])
224                 )
                )
226         );
        }
228     for(auto&& t: threads) t->join();

230     std::cout << "Price " << compwiseSum(total_prices) << "\n";
        std::cout << "Delta " << compwiseSum(deltas) << "\n";
232     std::cout << "Vega " << compwiseSum(vegas) << "\n";
        std::cout << "Rho " << compwiseSum(rhos) << "\n";
234 }

236 int main() {

238     exampleOptionPricingWithoutTemplates();

240     return 0;
```

```
}
```

Ex11OptionPricingWithoutTemplates.cpp

A.12 Example 12. Option Pricing with Debug

```
#define AADC_CHECK_VALUES
#include <iostream>
#include <random>
#include <thread>

#include <aadc/ibool.h>
#include <aadc/aadc.h>
#include <aadc/aadc_matrix.h>
// MANUAL Add debug
#include <aadc/aadc_debug.h>
// MANUAL
using namespace aadc;
typedef double Time;

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

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

template<class vtype>
vtype simulateAssetOneStep(
    const vtype current_value
    , Time current_t
    , Time next_t
    , const BankRate<vtype>& rate
    , const AssetVolatility<vtype>& vol_obj
    , const vtype& random_sample
) {
    double dt = (next_t-current_t);
    vtype vol=vol_obj(current_value, current_t);
    vtype next_value = current_value * (
        1 + (-vol*vol / 2+ rate(current_t))*dt
        +  vol * std::sqrt(dt) * random_sample
```

```
49      );
        return next_value;
51  }

53  template<class vtype>
    vtype onePathPricing_debug (
55      vtype asset
        , double strike
57      , const std::vector<Time>& t
        , const BankRate<vtype>& rate_obj
59      , const AssetVolatility<vtype>& vol_obj
        , const std::vector<vtype>& random_samples
61  ) {
        AADC_PRINT_DEBUG_COMMENT("collateral update");
63      asset=AADC_PRINT(asset);
        for (int t_i = 0; t_i < t.size()-1; ++t_i) {
65          asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1],
       rate_obj, vol_obj, random_samples[t_i]);
        }
67      return exp(-rate_obj(t.back()) * t.back()) * std::max(asset -
        strike, 0.);
    }
69
    inline double compwiseSum(const std::vector<double>& vec)
71  {
        double sum(0.);
73      for(int i=0; i<vec.size(); i++) sum+=vec[i];
        return sum;
75  }

77  void exampleOptionPricingDebug()
    {
79      int num_threads = 1;
        typedef __m256d mmType;  // __mm256d and __mm512d are
       supported
81
        double init_vol(0.005), init_asset(100.), init_rate(0.3);
83
        std::vector<Time> t(10 , 0.);
85      int num_time_steps=t.size();
        for (int i=1; i<num_time_steps; i++) t[i]=i*0.01;
87
        double strike=95.;
89      int num_mc_paths=1600;
        int AVXsize = aadc::mmSize<mmType>();
91      int paths_per_thread = num_mc_paths / (AVXsize*num_threads);
        //assert(num_mc_paths % (num_threads * getNu mAvxElements<
       mmType>()) == 0);
93
        aadc::AADCFunctions<mmType> aad_funcs({
95                      {aadc::AADC_InstrumentCode, 0}
                //    ,{aadc::AADC_DebugStopRecordingAt, 7}
97      });
```

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

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

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

113         aadc::AADCArgument rate_arg(aad_rate.markAsInput());
            aadc::AADCArgument asset_arg(aad_asset.markAsInput());
115         aadc::AADCArgument vol_arg(aad_vol.markAsInput());

117         BankRate<idouble> rate_obj(aad_rate);
            AssetVolatility<idouble> vol_obj(aad_vol);
119
            idouble payoff = aad_rate;
121        //payoff = payoff/0;
            // MANUAL Debug modification 1
123         payoff = onePathPricing_debug(aad_asset, strike, t,
       rate_obj, vol_obj, aad_random_samples);
            // MANUAL
125         // MANUAL Debug modification 2
            payoff += sqrt(0.0000000001*payoff);
127         // MANUAL

129         payoff = AADC_PRINT(payoff);
            aadc::AADCResult payoff_arg(payoff.markAsOutput());
131     aad_funcs.stopRecording();
        uint64_t total_print_vars(AADCGetTotalDebugPrints());
133     std::cout << "Total debug prints : " << total_print_vars <<
       std::endl;


135
        std::vector<double> total_prices(num_threads, 0.)
137         , deltas(num_threads, 0.)
            , vegas(num_threads, 0.)
139         , rhos(num_threads, 0.)
        ;
141
        auto threadWorker = [&] (
143         double& total_price
            , double& vega
145         , double& delta
            , double& rho
147     ) {
```

```
        std::mt19937_64 gen;
149     std::normal_distribution<> normal_distrib(0.0, 1.0);
        std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws(
    aad_funcs.createWorkSpace());
151
        mmType mm_total_price(mmSetConst<mmType>(0))
153         , mm_vega(mmSetConst<mmType>(0))
            , mm_delta(mmSetConst<mmType>(0))
155         , mm_rho(mmSetConst<mmType>(0))
        ;
157
        mmVector<mmType> randoms(num_time_steps);
159
        for (int mc_i=0; mc_i < paths_per_thread; ++mc_i) {
161         for (int c=0; c<AVXsize; c++) {
                for (int j=0; j<num_time_steps; j++) toDblPtr(
    randoms[j])[c] = normal_distrib(gen);
163         }
            setAVXVector(*ws, random_arg, randoms);
165         ws->setVal(rate_arg, mmSetConst<mmType>(init_rate));
            ws->setVal(asset_arg, mmSetConst<mmType>(init_asset))
    ;
167         ws->setVal(vol_arg, mmSetConst<mmType>(init_vol));
            aad_funcs.forward(*ws);
169         mm_total_price=mmAdd(ws->val(payoff_arg),
    mm_total_price);

171         ws->setDiff(payoff_arg, 1.0);
            aad_funcs.reverse(*ws);
173         // MANUAL Call debug
            if (mc_i==0) debugTestAdjointsUsingFD(std::cout, *ws,
     aad_funcs, payoff_arg, true, true);
175         // MANUAL
            mm_vega=mmAdd(ws->diff(vol_arg), mm_vega);
177         mm_delta=mmAdd(ws->diff(asset_arg), mm_delta);
            mm_rho=mmAdd(ws->diff(rate_arg), mm_rho);
179     }

181     total_price=aadc::mmSum(mm_total_price) / num_mc_paths;
        rho=aadc::mmSum(mm_rho) / num_mc_paths;
183     delta=aadc::mmSum(mm_delta) / num_mc_paths;
        vega=aadc::mmSum(mm_vega) / num_mc_paths;
185  };

187  std::vector<std::unique_ptr<std::thread>> threads;
     for(int i=0; i < num_threads; i++) {
189     threads.push_back(
            std::unique_ptr<std::thread>(
191             new std::thread(
                    threadWorker
193                 , std::ref(total_prices[i])
                    , std::ref(vegas[i])
195                 , std::ref(deltas[i])
                    , std::ref(rhos[i])
```

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

    std::cout << "Price " << compwiseSum(total_prices) << "\n";
    std::cout << "Delta " << compwiseSum(deltas) << "\n";
    std::cout << "Vega " << compwiseSum(vegas) << "\n";
    std::cout << "Rho " << compwiseSum(rhos) << "\n";
}

int main() {

    exampleOptionPricingDebug();

    return 0;
}
```

Ex12OptionPricingDebug.cpp

A.13 Example 13. Using AADC for external functions

```
#include "aadc/aadc.h"
#include "aadc/aadc_ext.h"
#include "aadc/idouble.h"
#include <iostream>

using namespace aadc;

// MANUAL myfunc
inline double myfunc(const double& x, const double& y) {
  return x*y;
}
// MANUAL
// MANUAL Wrapper
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]);
```

```
30    }
      // MANUAL
32    // MANUAL Reverse using FD
      template<class mmType>
34    void reverse(const mmType* v, mmType* d) const {
          // code to calculate the local gradient and update the
       adjoint variables.
36        double h(0.00001);
          for (int avxi = 0; avxi < aadc::mmSize<mmType>(); ++avxi) {
38            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);
40            toDblPtr(d[xi])[avxi] += toDblPtr(d[resi])[avxi] * d1;
              toDblPtr(d[yi])[avxi] += toDblPtr(d[resi])[avxi] * d2;
42        }
      };
44  private:
      ExtVarIndex xi, yi;
46    ExtVarIndex resi;
    };
48  // MANUAL
    // MANUAL Overloaded myfunc
50  inline idouble myfunc(const idouble& x, const idouble& y) {
      idouble res;
52
      aadc::addConstStateExtFunction(std::make_shared<
       MyFuncAADWrapper>(res, x, y));
54
      return res;
56  }
    // MANUAL
58
    template<typename mdouble>
60  mdouble funMath(mdouble& x, mdouble& y) {
        mdouble c=3.0;
62      return myfunc(x,c+y);
    }
64
    void exampleExternalFunc()
66  {
         typedef __m256d mmType;
68      aadc::AADCFunctions<mmType> aad_funcs;

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

72      aad_funcs.startRecording();
        aadc::AADCArgument x_arg(x.markAsInput());
74      aadc::AADCArgument y_arg(y.markAsInput());

76        f=funMath(x,y);
```

```
78      aadc::AADCResult f_res(f.markAsOutput());
        aad_funcs.stopRecording();
80
        std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
       createWorkSpace());
82
        ws->setVal(x_arg,mmSetConst<mmType>(1));
84      ws->setVal(y_arg,mmSetConst<mmType>(2));

86      aad_funcs.forward(*ws);

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


94
        ws->setDiff(f_res, 1.0);
96      aad_funcs.reverse(*ws);

98      std::cout
            << "df/dx=" << (ws->diffp(x_arg))[0]
100         << std::endl << "df/dy=" << (ws->diffp(y_arg))[0]
            << std::endl;
102 }

104 int main() {

106     exampleExternalFunc();

108     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  // MANUAL Integral class
    class Integral {
13      typedef __m256d mmType;
    public:
15      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();
    }
    // MANUAL
    // MANUAL Computing the integrals
    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);

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

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

```
69          // MANUAL
            // MANUAL Calculating the integral sum
71          for (int i=0; i<num_sims_count; ++i) {

73              ws->setVal(v_index[1],mmSetConst<mmType>(x[0]));
                ws->setVal(v_index[2],mmSetConst<mmType>(x[1]));
75
          double *_x = (ws->valp(v_index[0]));
77              for (int ci=0; ci<count; ++ci) {
                    _x[ci]= uni_distrib(gen);
79              }

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

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

89              D_integral1=mmAdd(ws->diff(v_index[1]), D_integral1);
                D_integral2=mmAdd(ws->diff(v_index[2]), D_integral2);
91          }

93          grad[0] = mmSum(D_integral1)/num_sims;
            grad[1] = mmSum(D_integral2)/num_sims;
95
            return 0.5*((_integral1-3)*(_integral1-3)+ (_integral2-5)
      *(_integral2-5));
97       }
        // MANUAL
99  private:
        int num_sims=1000; //divisible by avx count
101     int count = aadc::mmSize<mmType>();
        int num_sims_count=num_sims/count;
103
        std::shared_ptr<aadc::AADCFunctions<mmType> >   aad_funcs;
105     std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws;
        std::vector<aadc::AADCArgument> v_index;
107     std::vector<aadc::AADCResult> res_index;
    };
109 // MANUAL Call the LBFGS++ library
    //Then we just need to set up parameters, create a solver object,
        provide an initial guess, and run the minimization function.
111 void exampleExpectationsMSE()
    {
113     // Set up parameters
        LBFGSParam<double> param;
115     param.epsilon = 1e-6;
        param.max_iterations = 100;
117     // Create a solver and function object

119     LBFGSSolver<double> solver(param);
        Integral fun;
```

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

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

133 int main() {

135     exampleExpectationsMSE();

137     return 0;
    }
```

Ex14ExpectationsMSE.cpp

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

```
    #include <iostream>
2
    #include <Eigen/Dense>
4
    #define AADC_ALLOW_TO_PASSIVE_BOOL
6
    #include <aadc/aadc.h>
8   #include <aadc/aadc_eigen.h>
    #include <aadc/aadc_matrix.h>
10
    using namespace aadc;
12
    ////////////////////////////////////////////////////////
14  //
    // This example demonstrates how to record the Eigen routines
16  // using AADC.
    // For matrices $A$ and $B$ it records (A*B).norm(),
18  // calculates the gradient using AADC and
    // compares it with the finite difference (F.D.) derivatives
20  // with respect to A[row,col].
    //
22  ////////////////////////////////////////////////////////

24  void exampleEigen()
    {
26      typedef __m256d mmType;
        // At the end we compare the AADC-derivative and the
28      // base-derivative of (A*B).norm() with respect to
        // A[row,col].
30      double h = 0.001;
```

```
    int row = 3, col = 2;

    Eigen::Matrix<idouble, Eigen::Dynamic, Eigen::Dynamic> A,B;
    A.resize(4,4);
    B.resize(4,2);
    B << 1,2,3,4,5,6,7,8;
    std::vector<aadc::AADCScalarArgument> A_args;

    aadc::AADCResult norm_res;
  aadc::AADCFunctions<mmType> aad_funcs;

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

    std::shared_ptr<aadc::AADCWorkSpace<mmType>> ws(aad_funcs.
   createWorkSpace());

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

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

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

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

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

int main() {

    exampleEigen();
```

```
80      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 <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::AADCFunctions<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 << "aadc-J[" << y_position << "," << x_position <<
    "] = ";

81      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
    int main() {
99
        exampleComplex();
101
        return 0;
103 }
```

Ex16Complex.cpp