Applying AD in the Finance Industry, and an Operator Overloading tool for "Handwritten" Adjoint
Acknowledgements

▶ Joint work with Johannes Lotz and Klaus Leppkes from STCE Aachen
▶ Work funded by NAG
AD in Finance Industry
So what is NAG?
Main products/services
Who are our clients?
Numerical Algorithms Group

- So what is NAG?
- Main products/services
- Who are our clients?
- NAG Library gets embedded in user code
Numerical Algorithms Group

- So what is NAG?
- Main products/services
- Who are our clients?
- NAG Library gets embedded in user code
- NAG Library is closed source
Why is quantitative finance interested in (adjoint) AD?
Main task of quant finance: pricing and risk management of financial instruments

- Call/Put option, American option, Parisian option, Russian option, best-of option, basket option, swaption, ...
Main task of quant finance: pricing and risk management of financial instruments

Call/Put option, American option, Parisian option, Russian option, best-of option, basket option, swaption, ...

Postulate model for asset dynamics
Main task of quant finance: pricing and risk management of financial instruments

- Call/Put option, American option, Parisian option, Russian option, best-of option, basket option, swaption, ...
- Postulate model for asset dynamics
- Estimate model parameters
Main task of quant finance: pricing and risk management of financial instruments

- Call/Put option, American option, Parisian option, Russian option, best-of option, basket option, swaption, ...

- Postulate model for asset dynamics

- Estimate model parameters

- Price option (Monte Carlo, PDEs, FFT, etc) using NAG routines
Once you’ve sold an option, what then?
A sold option is a liability

- Sell call option on Apple stock
- Everyone goes iPhone mad
- Stock shoots through the roof
- Bank has to deliver stock at strike price
- How??
A sold option is a liability
▶ Sell call option on Apple stock
▶ Everyone goes iPhone mad
▶ Stock shoots through the roof
▶ Bank has to deliver stock at strike price
▶ How??

Sounds like gambling?
In theory, it’s not!

For any option $V(t, S_t)$ on asset $S$, there exists an actionable trading strategy $(\Delta_t, A_t)$ such that

$V(t, S_t) = \Delta_t S_t + A_t$ for all time $t$ regardless of value of $S_t$

Strategy requires no cash injection after time 0

Magic value $\Delta_t = \frac{\partial V}{\partial S}(t, S_t)$ which changes continuously in time
Finance Industry

- AD used in hedging
- Can’t hedge continuously
- Becomes equivalent to Taylor series expansion
- Typically second order terms also included
Remember the credit crunch?

- People totally ignored credit-worthiness of counterparties
- I sell option to you, buy same option from Bank
- In theory, I have no risk
- What's the problem?
Remember the credit crunch?
What caused it?
Finance Industry

- Remember the credit crunch?
- What caused it?
- People totally ignored credit-worthiness of counterparties
Finance Industry

- Remember the credit crunch?
- What caused it?
- People totally ignored credit-worthiness of counterparties
- I sell option to you, buy same option from Bank
Finance Industry

- Remember the credit crunch?
- What caused it?
- People totally ignored credit-worthiness of counterparties
- I sell option to you, buy same option from Bank
- In theory, I have no risk
Finance Industry

- Remember the credit crunch?
- What caused it?
- People totally ignored credit-worthiness of counterparties
- I sell option to you, buy same option from Bank
- In theory, I have no risk
- What’s the problem?
Finance Industry

Regulators now require CVA = Credit Value Adjustment

- Expected loss if counterparty goes bankrupt
Regulators now require CVA = Credit Value Adjustment

- Expected loss if counterparty goes bankrupt
- Problem: trade set huge, includes complex options (expensive, lots of inputs)
Regulators now require CVA = Credit Value Adjustment

- Expected loss if counterparty goes bankrupt
- Problem: trade set huge, includes complex options (expensive, lots of inputs)
- Massive computational task, has to be done frequently (e.g. daily)
Regulators now require CVA = Credit Value Adjustment

- Expected loss if counterparty goes bankrupt
- Problem: trade set huge, includes complex options (expensive, lots of inputs)
- Massive computational task, has to be done frequently (e.g. daily)

Regulators require CVA to be hedged!
AD of NAG Routines

Typical CVA calculation involves many calls to NAG routines

- Need derivatives
- 100s to 1000s of inputs, hence adjoint AD
- NAG provide adjoint (discrete and continuous) versions of Library routines

- We use dco/c++, dco/fortran and handwritten internally
- Some customers use dco/c++, some don’t
- Requires quasi-standard adjoint interface
Getting banks to use AD
AD in Banks

- Vast majority of quants not aware of AD before 2006
- A very few “got it” straight away (handwritten or in-house tools)

The rest did what banks often do: wait and see
AD in Banks

Helps to understand quant libraries a little

- 100,000s to 1,000,000s of lines of heavily templated C++
- Some banks never withdraw code
- Code has to be reviewed, changes have to be signed off. Putting AD through the code base would mean entire code base has to be reviewed and signed off – impossible, too costly
- Changes have to be validated: AD ≠ FD
Hard to get adjoint AD message across

- Explain AD by overloading, only for people to say

  Yes but that was on $f(x) = \exp(x^3)$, I've got $f(x) = \sin(x^2)$

- For a while, was a common belief that “you always run out of memory”. Even now, difficult to convince people memory use can be effectively controlled

- desire to see AD done on “my code” before engaging further
And last, but not least, there is the Regulator, who giveth with the one hand and taketh with the other.
AD in Banks

So how do banks do AAD?
AD in Banks

- Some do all by hand: libraries of handwritten adjoints
- They say it’s not so bad, you get used to it
- Everyone else uses operator overloading AD tools such as dco/c++

Helpful to understand banks’ view of 3rd party software

- Some don’t use any: do everything in-house
- Others are not allowed to use any FOSS
- Any code not from an approved vendor has to be tested and certified by quants or validation teams (so nothing is free)
- Often cheaper to buy or write from scratch than adopt FOSS
What Scares Banks About AD
What Scares Banks About AD

Split sources
What Scares Banks About AD

- Split sources
- Memory
What Scares Banks About AD

- Split sources
- Memory
- Complexity
For an AD tool to successfully target a large quant library it must:

- Handle the whole of C++
- Integrate fully into the bank’s code base, build systems and testing harness
- **Be documented!** If it’s not documented, it doesn’t exist
- Provide full, flexible and intuitive access to the “tape” to allow checkpointing, user-defined adjoints, local bumping, etc
- Have a means of handling closed source libraries
- Not blow out the compile time
- Increasingly (though still very niche) provide GPU support
Doing AD on GPU Accelerated Code
GPUs in Finance

- Mainly Monte Carlo on GPU
Doing AD on GPU accelerated code not new

- Gremse et al (Comp. Phys. Comm. 2016) in image reconstruction
- Primal code is series of linear algebra operations (execute on GPU)
- AD tool differentiates analytically, easy GPU adjoint
- Very nice technique, very good performance
AD on GPU

Doing AD on GPU accelerated code not new

► Gremse et al (Comp. Phys. Comm. 2016) in image reconstruction
► Primal code is series of linear algebra operations (execute on GPU)
► AD tool differentiates analytically, easy GPU adjoint
► Very nice technique, very good performance

Only works if primal code expressable as linear algebra
AD on GPU

Doing AD on GPU accelerated code not new

- Gremse et al (Comp. Phys. Comm. 2016) in image reconstruction
- Primal code is series of linear algebra operations (execute on GPU)
- AD tool differentiates analytically, easy GPU adjoint
- Very nice technique, very good performance

Only works if primal code expressable as linear algebra

- In finance this is sufficiently rare even in Monte Carlo that not worth pursuing

- Local volatility model $X_{t+\Delta}$ is non-linear function of $X_t$
- Andersen QE scheme even worse
- GPU code must be “general purpose” with no special structure
AD on GPU

- Target language is CUDA: massively parallel C++11 language
- Tape on GPU is tricky
- 30,000+ threads, small GDDR memory
- Branches lead to cache-unfriendly access
- Unstructured memory access in tape playback
- Indirection bad

No one does CUDA for fun!! It’s an accelerator, it has to accelerate

Solutions must prioritise performance
The Primal Code and Handwritten Adjoint
Local volatility model is one of the fundamental models in math finance.

Governing SDE is

\[ dX_t = rX_t \, dt + \sigma(t, X_t)X_t \, dW_t \]

Local volatility not known analytically, can only sample it.

For MC path generation, typically use Euler discretisation

\[ X_{i+1} = rX_i \Delta t + \sigma_i(X_i)X_iZ\sqrt{\Delta t} \]

where each \( \sigma_i \) is a cubic spline, \( Z \) is normal r.v

Difficulty from AD perspective is the spline.
Cubic Spline

- Cubic spline is a mapping

\[(x, \lambda[1 : n], c[1 : n]) \mapsto y\]

- Given input \(x\), find \(j = j(x)\) such that \(\lambda[j] \leq x < \lambda[j + 1]\) and then

\[
y \leftarrow \begin{cases} \lambda[j(x)] \\ \vdots \\ \lambda[j(x) + 5] \\ c[j(x)] \\ \vdots \\ c[j(x) + 3] \end{cases}
\]
START

Stage 1: Setup

Stage 2: Monte Carlo

GPU Monte Carlo Kernel

Stage 3: Payoff

End forward run

Copy to GPU

Extract data from dco types

Insert into new dco types

Output: price = y

set y(I) = 1

Start reverse run

FINISH

dco tape

dco tape

Insert adjoints into tape

Copy output adjoints to CPU

Copy to GPU

Copy output to CPU

Extract adjoints from tape

GPU Adjoint Kernel
Don’t think I can “get used to” makings handwritten GPU adjoints
A Better Way?
Better Way?

Want a tool to help make adjoints of CUDA kernels

- Limited resources, expensive branching (avoid polymorphism)
- Programs are massively multi-threaded, memory limited: 100KB global memory per thread not uncommon
- Passive program data often more than 60% graphics memory
- Hardware highly cache sensitive
- Accesses to GDDR are slow: reduce as much as possible
- Race conditions in adjoint will be norm, not exception: have to handle efficiently
- Have to cope with C++11 code
Dealing with race conditions

- What is most common piece of thread-private memory?

- But GPUs don't like using stack

- What is other thread-private memory space?

- What are registers?

- Use stack/register file as much as possible

- Avoids race conditions

- Limits memory use
A Better Way?

Dealing with race conditions

- What is most common piece of thread-private memory?
- But GPUs don’t like using stack
A Better Way?

Dealing with race conditions

▶ What is most common piece of thread-private memory?
▶ But GPUs don’t like using stack
▶ What is other thread-private memory space?
A Better Way?

Dealing with race conditions

- What is most common piece of thread-private memory?
- But GPUs don’t like using stack
- What is other thread-private memory space?
- What are registers?
Dealing with race conditions

- What is most common piece of thread-private memory?
- But GPUs don’t like using stack
- What is other thread-private memory space?
- What are registers?

Use stack/register file as much as possible

- Avoids race conditions
- Limits memory use
A Better Way?

Implication of no-tape/stack-tape

- Stack/registers fixed at compile time, registers non-addressible
- Adjoint code fully known at compile time (as much of it as possible)
- Enforce single assignment: variables written only once

GPUs have lots of different memory spaces

- Must save variables (checkpoint) when overwritten non-linearly
- The tool should leave this to user
- Everything not saved is recomputed
A Better Way?

OK, so far so good, but how??
A Better Way?

C++ is not one, but three languages

▶ Procedural language of C
▶ Object oriented language
▶ Meta-programming language (templates)

Template meta-programming language is Turing complete
For straight-line code, producing an adjoint (ignoring race conditions) is completely algorithmic.
A Better Way?

For straight-line code, producing an adjoint (ignoring race conditions) is completely algorithmic.

*Can we not write a meta-program which would create the adjoint for a block of straight line code?*
A Better Way?

For straight-line code, producing an adjoint (ignoring race conditions) is completely algorithmic.

*Can we not write a meta-program which would create the adjoint for a block of straight line code?*

Turing complete $\neq$ straightforward or *practical*
A Better Way

Is possible, but not straightforward

- Easy to blow out compile time and compiler memory use
- How do you profile a meta-program?
A Better Way

Is possible, but not straightforward

- Easy to blow out compile time and compiler memory use
- How do you profile a meta-program?
- Not easy to instantiate adjoint as efficient as handwritten

Basic blocks not enough: control flow

- Function calls
- If-statements
- For loops

Implemented in dco/map = MetaAdjoint Programming
A Better Way

Is possible, but not straightforward

▶ Easy to blow out compile time and compiler memory use
▶ How do you profile a meta-program?
▶ Not easy to instantiate adjoint as efficient as handwritten
▶ End up relying heavily on compiler’s optimisations
A Better Way

Is possible, but not straightforward

- Easy to blow out compile time and compiler memory use
- How do you profile a meta-program?
- Not easy to instantiate adjoint as efficient as handwritten
- End up relying heavily on compiler’s optimisations

Basic blocks not enough: control flow

- Function calls
- If-statements
- For loops
- Not exposable to template engine

Implemented in dco/map = Meta Adjoint Programming
template<typename AD>
void spline(int n, const AD* lamda, const AD *c,
    const AD & x, AD &s)
{
    int j = ...  // Find correct place in arrays
    const auto k1 = lamda[j+1]; const auto k2 = lamda[j+2];
    const auto k3 = lamda[j+3]; const auto k4 = lamda[j+4];
    const auto k5 = lamda[j+5]; const auto k6 = lamda[j+6];
    const auto c0 = c[j];    const auto c2 = c[j+1];
    const auto c3 = c[j+2];

    const auto e2 = x - k2; const auto e3 = x - k3;
    const auto e4 = k4 - x; const auto e5 = k5 - x;
    const auto c1 = ((x-k1)*c2+e4*c0)/(k4-k1);
    const auto c5 = (e2*c3+e5*c2)/(k5-k2);
    const auto c7 = (e3*c[j+3]+(k6-x)*c3)/(k6-k3);
    const auto c4 = (e2*c5+e4*c1)/(k4-k2);
    const auto c6 = (e3*c7+e5*c5)/(k5-k3);
    s = (e3*c6+e4*c4)/(k4-k3);
}
void makePaths(int N, int M, const AD &x0, const AD & dt, 
    const AD &r, int n, const AD knots[], const AD cs[], 
    const double Z[], AD x[], double ckpoint[] ) 
{
    DCO_FOR(p, 0, N-1) {
        AD xi = x0, xii;
        DCO_FOR(i, 0, M-1) 
        {
            AD sigma;
            DCO_CALL( evalSpline<AD>(n, knots, cs, xi, sigma) );

            xii = xi + (r - 0.5*sigma*sigma)*dt
                    + sigma*sqrt(dt)*Z[i+p*M];
            DCO_STORE_CP(i) {
                ckpoint[i+p*M] = dco_map::value(xi);
            }
            xi = xii;
        }
        DCO_LOAD_CP(i) {
            dco_map::value(xi) = ckpoint[i+p*M];
        }
    }
    DCO_ENDFOR;
    x[p] = xi;
} DCO_ENDFOR
### dco/map on Harmonic Test Function

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc5.4</td>
</tr>
<tr>
<td>pass.</td>
<td>113</td>
</tr>
<tr>
<td>hand</td>
<td>244 (2x)</td>
</tr>
<tr>
<td>dco/map</td>
<td>563 (5x)</td>
</tr>
<tr>
<td>tape</td>
<td>1,180 (11x)</td>
</tr>
</tbody>
</table>
## dco/map on Local Volatility Monte Carlo Kernel

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc</td>
<td>clang</td>
<td>icc</td>
<td>nvcc</td>
</tr>
<tr>
<td>pass.</td>
<td>1,406</td>
<td>1,461</td>
<td>1,530</td>
<td>18</td>
</tr>
<tr>
<td>hand</td>
<td>2,800 (2x)</td>
<td>2,997 (2x)</td>
<td>2,580 (1.7x)</td>
<td>89 (4.9x)</td>
</tr>
<tr>
<td>dco/map</td>
<td>3,025 (2.2x)</td>
<td>3,031 (2x)</td>
<td>7,560 (5x)</td>
<td>83 (4.6x)</td>
</tr>
<tr>
<td>tape</td>
<td>11,011 (7x)</td>
<td>13,579 (9.3x)</td>
<td>12,520 (8x)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Windows</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VS2013</td>
<td>clang</td>
<td>Intel2015</td>
<td>nvcc</td>
</tr>
<tr>
<td>pass.</td>
<td>1,510</td>
<td>1,172</td>
<td>1,421</td>
<td>20</td>
</tr>
<tr>
<td>hand</td>
<td>1,992 (1.3x)</td>
<td>1,906 (1.6x)</td>
<td>1,876 (1.3x)</td>
<td>90 (4.5x)</td>
</tr>
<tr>
<td>dco/map</td>
<td>10,241 (7x)</td>
<td>4,384 (3.7x)</td>
<td>11,671 (8x)</td>
<td>85 (4.2x)</td>
</tr>
<tr>
<td>tape</td>
<td>24,000 (16x)</td>
<td>16,125 (16x)</td>
<td>18,833 (13x)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## dco/map on Different Local Volatility Code

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc</td>
<td>clang3.8</td>
</tr>
<tr>
<td>pass.</td>
<td>360</td>
<td>105</td>
</tr>
<tr>
<td>hand</td>
<td>1,130 (3x)</td>
<td>275 (2.5x)</td>
</tr>
<tr>
<td>dco/map</td>
<td>740 (2x)</td>
<td>370 (3.5x)</td>
</tr>
<tr>
<td>tape</td>
<td>1,100 (3x)</td>
<td>425 (4x)</td>
</tr>
<tr>
<td></td>
<td>740 (2x)</td>
<td>370 (3.5x)</td>
</tr>
<tr>
<td>tape</td>
<td>1,100 (3x)</td>
<td>425 (4x)</td>
</tr>
<tr>
<td></td>
<td>VS2013</td>
<td>clang3.8</td>
</tr>
<tr>
<td>pass.</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>hand</td>
<td>200 (2x)</td>
<td>192 (1.9x)</td>
</tr>
<tr>
<td>dco/map</td>
<td>860 (8.6x)</td>
<td>385 (3.8x)</td>
</tr>
<tr>
<td>tape</td>
<td>1610 (16x)</td>
<td>500 (5x)</td>
</tr>
</tbody>
</table>
Conclusion

Use cases for dco/map

- dco/map is not a drop-in replacement like dco/c++
- Would not use it on large code bases
- Use on key computational kernels
- Use where memory is highly constrained (primal using 90% RAM)
- Use on GPUs

Tool still needs some work, things still to understand

Thank you