Automatic differentiation beyond typedef and operator overloading

Peter Caspers

Quaternion Risk Management

November 30, 2015

Table of contents

AD in a nutshell 1/3

- for a computer program $f : \mathbb{R}^n \to \mathbb{R}^m$, compute $\partial_x f$
- ... by looking at the program's sequence of basic operations $(+ - */, \exp \ldots)$ and using basic calculus in each step
- ... and stitching everything together with the chain rule

AD in a nutshell 2/3

- results are exact up to machine precision, also for higher order derivatives
- implementation:
    - operator overloading instrumenting the double type[1]
    - source code transformation tools[2]
    - coding by hand

[1]e.g. CppAD, ADOL-C, Adept, dco, proprietary tools

[2]e.g. ADIC, OpenAD/F

AD in a nutshell 3/3

- local jacobians can be propagated forward ($x \rightsquigarrow y$) (that's intuitive) or backward [3] ($y \rightsquigarrow x$) in a dual or *adjoint* fashion
- one *forward* sweep yields one directional derivative of your choice of the vector of output variables
- one *reverse* sweep yields the gradient w.r.t. all input variables of one linear combination of the output variables
- the complexity for one (forward or reverse) sweep is a constant, low multiple of the complexity for one function evaluation[4]
- in particular: **law of cheap gradient** !

[3] `https://quantlib.wordpress.com/2015/08/09/backward-automatic-differentiation-explained/`

[4] theory tells us that the multiple in adjoint mode is bounded by 4

Adjoint mode example

- program $f : \mathbb{R}^n \to \mathbb{R}$: $y = \exp\left(\prod_{i=0}^n x_i\right) \sin\left(\prod_{i=0}^n x_i\right)$
- imagine n to be large, like 1000
- evaluation complexity: $n + 4 = O(n)$ operations $\in \{*, \exp, \sin\}$
- goal: compute $\partial_x f \in \mathbb{R}^n$
- finite difference approach: $(n+1)(n+4) = O(n^2)$ operations in addition to the evaluation

Adjoint mode example - distance 1 nodes

- init $\partial_y y = 1$
- first break down is $y = u * v$, $u = \exp\left(\prod_{i=0}^n x_i\right)$, $v = \sin\left(\prod_{i=0}^n x_i\right)$
- $\partial_u y = \partial_y y \partial_u y = v$, $\partial_v y = \partial_y y \partial_v y = u$
- 2 operations assuming we have performed a forward sweep, so we
  - know the value of u and v and
  - have built the computational graph and
  - know the "analytics" for the local derivatives
- we are not overly pedantic on how to count the operations in this example here ...

Adjoint mode example - distance 2 nodes

- second break down $u = \exp(x), v = \sin(x)$
- $\partial_x u = \exp(x), \partial_x v = \cos(x)$
- $\partial_x y = \partial_u y \partial_x u + \partial_v y \partial_x v = \sin(x)\exp(x) + \exp(x)\cos(x)$
- again, we know x from the forward sweep
- 5 operations (total operations count 7)

Adjoint mode example - distance 3 nodes

- third break down $x = x_0 h_0$
- $\partial_{x_0} x = h_0, \partial_{h_0} x = x_0$
- $\partial_{x_0} y = \partial_x y \partial_{x_0} x = [\sin(x)\exp(x) + \exp(x)\cos(x)]h_0$
- $\partial_{h_0} y = \partial_x y \partial_{x_0} h_0 = [\sin(x)\exp(x) + \exp(x)\cos(x)]x_0$
- ... we know h_0 from the forward sweep ...
- 4 operations (total operations count 11)

Adjoint mode example - nodes with distance n+2

- continue like in the third break down until we arrive at $h_{n-1} = x_n$
- $\partial_{x_i} y = [\sin(\prod x_i) \exp(\prod x_i) + \exp(\prod x_i) \cos(\prod x_i)] \prod_{j \neq i} x_i$
- $4(n-2)$ operations from the third break down on
- total operations count $4(n-1) + 7 = 4n + 3$
- one function evaluation was $n + 4$ operations
- naive approach for gradient calculation was $(n+1)(n+4)$ operations

Table of contents

The typedef approach

- just says `typedef CppAD::AD<double> Real`
- it is a bit more complicated than that
- QuantLibAdjoint (CompatibL), with additional logic (tapescript)
- *AD-or-not-AD* decision at compile time and globally, i.e. no selective activation of variables

Matrix multiplication with (sleeping) active doubles

```
Matrix_t<T> A(1024, 1024);
Matrix_t<T> B(1024, 1024);
...
Matrix_t<T> C = A * B;
```

- `T = double`: 764 ms
- `T = CppAD::AD<double>`: 8960 ms
- penalty: 11.7x
- note that we do not get anything for that (AD is disabled)
- this is not an exception, but seems to occur for every "numerically intense" code section (see below for a second example)

Active doubles vs. native doubles 1/2

- for a `MinimalWrapper` consisting of a `double` and a pointer `MinimalWrapper*` (set to `nullptr` always), the penalty is around 2.1x
- for this gcc generates *scalar* double instructions (mulsd, addsd)
- for the native `double` gcc generates *packed* double instructions (mulpd, addpd)[5]
- in addtion the more involved data layout of the `MinimalWrapper` (placing a pointer after each native `double`) leads to more instructions in the innermost loop

[5] with -O3

Active doubles vs. native doubles 2/2

- (current) compilers seem to generate more instructions and possibly less efficient instructions for non-native double wrappers
- memory consumption will go up, too
- it is not clear what the "best possible" OO tool can achieve, but probably it will be something between 2x and 12x
- 2x is already too much, if we do not get anything for that
- we can easily avoid this useless overhead

The template approach

- introduce templated versions of relevant classes (e.g. `Matrix_t`)
- for backward compatibility, `typedef Matrix_t<Real> Matrix`
- it is a bit more complicated than that
- allows mixing of active and native classes, as required, i.e. activation of variables in selected parts of the application only
- work in progress[6], but basic IRD stuff works (like yield and volatility termstructures, swaps, CMS coupons, GSR model)
- `https://github.com/pcaspers/quantlib/tree/adjoint`
- `https://quantlib.wordpress.com/tag/automatic-differentiation/`

[6] conversion rate $\approx$ 2000 LOC / day (manual + an Elisp-little-helper)

Expensive gradients with operator overloading

- the typedef as well as the template approach use operator overloading tools (like CppAD)
- for numerically intense algorithms, we observe dramatic performance loss (because less optimization can be applied to non-native types)
- e.g. a convolution engine for bermudan swaptions is **80x slower**[7] in adjoint mode compared to one native-double pricing
- if AD is actually not needed, the template approach is the way out, otherwise we need other techniques

[7] see `https://quantlib.wordpress.com/2015/04/14/adjoint-greeks-iv-exotics`

Table of contents

Source Code Transformation

- generate adjoint code at compile time, which may yield better performance
- however, does not work out of the box like OO tools
- no mature tool for C++ (ADIC 2.0 = "OpenAD/Cpp" under development)
- needs specific preparation of code before it can be applied

OpenAD/F

- OpenAD is a language independent AD backend working with abstract xml representations (XAIF) of the computational model
- OpenAD/F adds a Fortran 90 front end
- Open Source, proven on large scale real-world models
- `http://www.mcs.anl.gov/OpenAD`

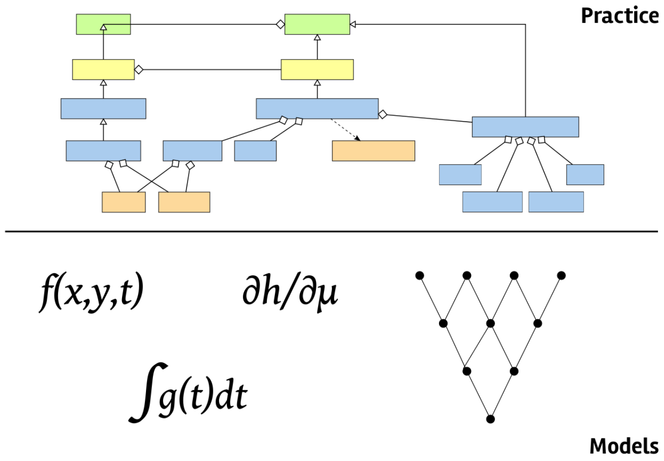
From QuantLib to SCT

- isolate the core computational code and reimplement it in Fortran
- use OpenAD/F to generate adjoint code, build a separate support library from that
- use a wrapper class on the QuantLib side to communicate with the support libary
- minimal library example [8] and LGM swaption engine[9] available
- build via `make` (AD support library) or `make plain` (without OpenAD - transformation, for testing)

[8] `https://github.com/pcaspers/quantlib/tree/master/QuantLibOAD/simplelib`

[9] `https://github.com/pcaspers/quantlib/tree/master/QuantLibOAD/lgm`

By the way ... different motivation, but same idea ?



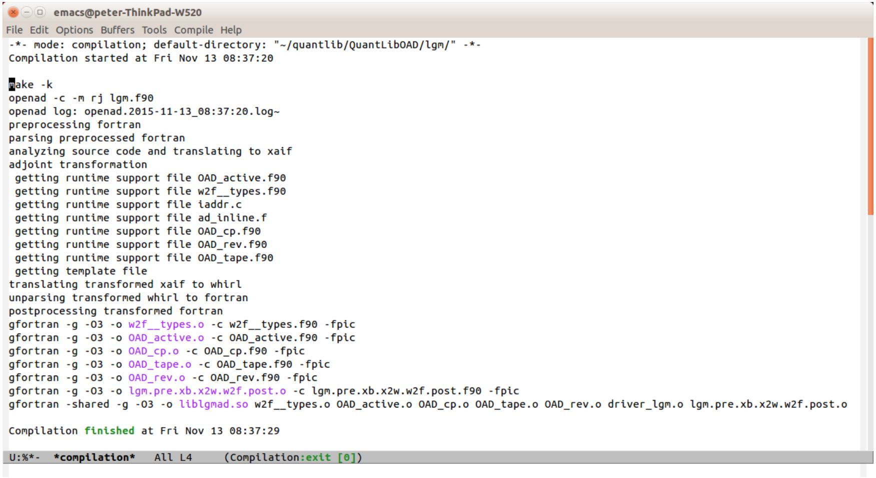
(taken from from Luigi's talk at the 11th FI conference, 2015, Paris)

LGM Bermudan swaption convolution engine

- core computation can be implemented in around 200 lines
- native interface only using (arrays of) doubles and integers
- input: relevant times $\{t_i\}$, model $\{(H(t_i), \zeta(t_i), P(0, t_i)\}$, Termsheet, codified as index lists $\{k_i, l_i, ...\}$
- output: npv, gradient w.r.t. $\{(H(t_i), \zeta(t_i), P(0, t_i)\}$

```
subroutine lgm_swaption_engine(n_times, times, modpar, n_expiries, &
     expiries, callput, n_floats, &
     float_startidxes, float_mults, index_acctimes, float_spreads, &
     float_t1s, float_t2s, float_tps, &
     fix_startidxes, n_fixs, fix_cpn, fix_tps, &
     integration_points, stddevs, res)
```

Building the AD support library


emacs@peter-ThinkPad-W520
File Edit Options Buffers Tools Compile Help

```
-*- mode: compilation; default-directory: "~/quantlib/QuantLibOAD/lgm/" -*-
Compilation started at Fri Nov 13 08:37:20

make -k
openad -c -m rj lgm.f90
openad log: openad.2015-11-13_08:37:20.log~
preprocessing fortran
parsing preprocessed fortran
analyzing source code and translating to xaif
adjoint transformation
 getting runtime support file OAD_active.f90
 getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
 getting runtime support file ad_inline.f
 getting runtime support file OAD_cp.f90
 getting runtime support file OAD_rev.f90
 getting runtime support file OAD_tape.f90
 getting template file
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
gfortran -g -O3 -o w2f__types.o -c w2f__types.f90 -fpic
gfortran -g -O3 -o OAD_active.o -c OAD_active.f90 -fpic
gfortran -g -O3 -o OAD_cp.o -c OAD_cp.f90 -fpic
gfortran -g -O3 -o OAD_tape.o -c OAD_tape.f90 -fpic
gfortran -g -O3 -o OAD_rev.o -c OAD_rev.f90 -fpic
gfortran -g -O3 -o lgm.pre.xb.x2w.w2f.post.o -c lgm.pre.xb.x2w.w2f.post.f90 -fpic
gfortran -shared -g -O3 -o liblgmad.so w2f__types.o OAD_active.o OAD_cp.o OAD_tape.o OAD_rev.o driver_lgm.o lgm.pre.xb.x2w.w2f.post.o

Compilation finished at Fri Nov 13 08:37:29

U:%*-  *compilation*   All L4     (Compilation:exit [0])
```

LGM Bermudan swaption convolution engine

- C++ wrapper is a normal QuantLib pricing engine
- precomputes the values and organizes them in arrays for the Fortran core
- invokes the Fotran routine
- stores the npv and the adjoint gradient as results

```
void LgmSwaptionEngineAD::calculate() const {
    // collect data needed for core computation routine
    ...
    // join all dates and fill index vectors
    ...
    // call core computation routine and set results

    lgm_swaption_engine_ad_(&ntimes, &allTimes[0], &modpar[0], &nexpiries, ...
                    &integration_pts, &std_devs, &res, &dres[0]);
    ...
    results_.value = res;
    results_.additionalResults["sensitivityTimes"] = allTimes;
    results_.additionalResults["sensitivityH"] = H_sensitivity;
    results_.additionalResults["sensitivityZeta"] = zeta_sensitivity;
    results_.additionalResults["sensitivityDiscount"] = discount_sensitivity;
```

Performance

- 10y bermudan swaption, yearly callable
- 49 grid points per expiry
- single pricing[10] (non-transformed code): 4.2 ms
- pricing + gradient $\in \mathbb{R}^{105}$: **25.6 ms**[11]
- additional stuff[12]: 6.2 ms
- adjoint calculation multiple: **6.1x** (7.6x including add. stuff)
- common, practical target for the adjoint multiple: 5x - 10x

[10] Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz, using one thread
[11] to achieve this, the runtime configuration of OpenAD/F has to be modified
[12] transformation of gradient w.r.t. model parameters to usual vegas, see below

How not to use AD

- avoid to record tapes that go through solvers, optimizers, etc.[13]
    - instead use the implicit function theorem to convert gradients w.r.t. calibrated (model) variables to gradients w.r.t. market variables
    - this is more efficient, less error prone (e.g. `Bisection` produces zero derivatives always, optimizations may produce bogus derivatives depending on the start value)
    - in the case of SCT applied as above this is even necessary from a practical viewpoint
- apply AD only to differentiable programs (replace a digital payoff for example by a call spread)
- avoid to record *long* tapes (e.g. for *all* paths of a MC simulation), reuse a tape recorded (in a *tape-safe* way) on one path

[13]not to be confused with feeding AD - derivatives of the target function to optimizers like Levenberg-Marquardt or Newton-style solvers

Calibration of LGM model

To illustrate the usage of the implicit function theorem, consider the calibration to n swaptions[14]

$$\begin{aligned} \mathsf{Black}(\sigma_1) - \mathsf{Npv}_{\mathsf{LGM}}(\zeta_1) &= 0 \\ &\dots \\ \mathsf{Black}(\sigma_n) - \mathsf{Npv}_{\mathsf{LGM}}(\zeta_n) &= 0 \end{aligned}$$

with

$$\frac{\partial \mathsf{Npv}_{\mathsf{LGM}}}{\partial \zeta} = \mathsf{diag}(\nu_1, \dots, \nu_n), \text{ all } \nu_i \neq 0 \tag{1}$$

[14] recall that $\zeta(t)$ is the accumulated model variance up to time t

Implicit function theorem

Locally, there exists a unique g

$$g(\sigma_1, ..., \sigma_n) = (\zeta_1, ..., \zeta_n) \tag{2}$$

and

$$\frac{\partial g}{\partial \sigma} = \left(\frac{\partial \mathsf{Npv}_{\mathsf{LGM}}}{\partial \zeta} \right)^{-1} \frac{\partial \mathsf{Black}}{\partial \sigma} \tag{3}$$

Informally, $g = \zeta(\sigma)$ and

$$\frac{\partial \zeta}{\partial \sigma} = \frac{\partial \zeta}{\partial \mathsf{NPV}} \frac{\partial \mathsf{NPV}}{\partial \sigma} = \left(\frac{\partial \mathsf{NPV}}{\partial \zeta} \right)^{-1} \frac{\partial \mathsf{NPV}}{\partial \sigma} \tag{4}$$

Pasting the vega together

$$\frac{\partial \mathsf{Npv}_{\mathsf{Berm}}}{\partial \sigma} = \frac{\partial \mathsf{Npv}_{\mathsf{Berm}}}{\partial \zeta}\frac{\partial \zeta}{\partial \sigma} = \frac{\partial \mathsf{Npv}_{\mathsf{Berm}}}{\partial \zeta}\left(\frac{\partial \mathsf{Npv}_{\mathsf{Calib}}}{\partial \zeta}\right)^{-1}\frac{\partial \mathsf{Black}}{\partial \sigma}$$

- the components can be calculated analytically (calibrating swaptions' market vegas) or using the ad engine (calibrating swaptions' ζ-gradient, but this is much cheaper than for the bermudan case)
- matrix inversion and multiplication is cheap
- the additional computation time is quite small (see the example above, the addtional costs are the same as for 1.5x original NPV calculations)

Summary

- global instrumentation (via typedefs) with active variables can lead to performance (and memory) issues
- selective / mixed instrumentation (via templates) solves the issue, but leaves problems when AD is required for numerically intense parts of the code
- source code transformation can solve this issue, we gave an example in terms of a bermudan swaption engine transformed using OpenAD/F yielding an adjoint multiple of **6.1** compared to **80** with operator overloading (using CppAD)

Table of contents

Questions / Discussion

Thank you for your attention