Automatic Differentiation beyond typedef and operator overloading

Peter Caspers

Quaternion

November 30, 2015

Table of contents

AD in a nutshell

- compute derivatives by symbolic differentiation of an operation sequence $(+ * -/, \exp, \sin, ...)$ using the chain rule
- derivatives can be propagated in forward and reverse (adjoint) mode
- results are exact up to machine precision, also for higher order derivatives
- law of cheap gradient in adjoint mode (complexity $=$ constant low multiple[1] of one function evaluation)
- implementation by hand coded derivative code, operator overloading tools instrumenting code at runtime or source code transformation generating derivative code at compile time

[1] theory claims that the multiple is bounded by 4

The typedef approach

- just says `typedef CppAD::AD<double> Real`
- it is a bit more complicated than that
- QuantLibAdjoint project (CompatibL / Alex Sokol), with additional logic (Tapescript project)
- *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[2]
- the compiler does not seem to apply certain optimizations to the active type that are possible for the native double (SIMD instructions)
- this is not an exception, but seems to occur for every "numerically intense" code section (see below for a second example)

[2] different OO tools my show different penalties; a MinimalWrapper consisting of a double and a native MinimalWrapper - pointer (which is null always), the penalty is around 2.0x

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
- currently not finalized, 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/`

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[3] 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

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

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 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 [4] and LGM swaption engine[5] available
- Build via `make` (AD support library) or `make plain` (without OpenAD - transformation, for testing)

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

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

LGM Bermudan swaption convolution engine

- core computation can be implemented in around 200 lines
- native interface only using doubles and arrays of doubles
- 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 Sat Oct 31 09:52:42

make
openad -c -m rj lgm.f90
openad log: openad.2015-10-31_09:52:42.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 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 driver_lgm.o -c driver_lgm.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 Sat Oct 31 09:52:50

U:%*-  *compilation*   All L1     (Compilation:exit [0])
Compilation finished
```

LGM Bermudan swaption convolution engine

- C++ wrapper is a normal QuantLib pricing engine
- precomputes the values for the Fortran interface
- 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[6] (non-transformed code): 4.2 ms
- pricing + gradient $\in \mathbb{R}^{105}$: 25.6 ms
- additional stuff[7]: 6.2 ms
- adjoint calculation multiple: 6.1x (7.6x including add. stuff)
- common, practical target for the adjoint multiple: 5x - 10x

[6] Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz, single threaded

[7] transformation of gradient w.r.t. model parameters to usual vegas, see below

Where to use AD and where not

- apply AD only to differentiable programs
- avoid to push it through solvers, optimizers, etc.
- instead use the implicit function theorem to convert gradients w.r.t. calibrated 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 even necessary from a practical viewpoint

Calibration of LGM model

Calibration to n swaptions[8]

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

with

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

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

Implicit function theorem

Locally, there exists a unique g

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

and

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

Pasting the vega together

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

- 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 similar to 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 CppAD

Questions / Discussion

thank you for your attention