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 the operation sequence 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)

[1] theoretically bounded by 4

The typedef approach

- just says `typedef CppAD::AD<double> Real`
- it is a bit more complicated than that
- QuantLibAdjoint project, driven by CompatibL
- *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
- the compiler does not seem to apply certain optimizations to the active type that are possible for the native double (vectorization)

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`

Expensive gradients with operator overloading

- both approaches use operator overloading tools (like CppAD)
- for numerically intense algorithms, we observe dramatic performance loss (because less agressive optimizations can be applied by the compiler to non-native types)
- e.g. a convolution engine for bermudan swaptions may be 80x slower[2] 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, because operator overloading is not attractive any more

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

Source Code Transformation

- Generate adjoint code at compile time, which yields 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 of the computational model (XAIF)
- OpenAD/F adds a Fortran 90 front end
- Open Source, proven on large scale real-world models
- `http://www.mcs.anl.gov/OpenAD`

Strategy

- 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

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)
```

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;
```

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 Tue Oct 27 15:41:44

make
openad -c -m rj lgm.f90
openad log: openad.2015-10-27_15:41:44.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 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 Tue Oct 27 15:41:52
U:%*-  *compilation*   All L1     (Compilation:exit [0])
Compilation finished
```

Performance

- 10y bermudan swaption, yearly callable
- 49 grid points per expiry
- single pricing (non-transformed function): 4.2 ms
- pricing + full adjoint vector $\in \mathcal{R}^{105}$ w.r.t. $(H(\cdot), \zeta(\cdot), P(0, \cdot))$: 25.6 ms
- additional stuff[3]: 6.2 ms
- adjoint calculation multiple: 6.1x (7.6x including "stuff")

[3] gradients for calibrating swaptions, transformation of gradient w.r.t. model parameters to common 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), even technically necessary for the SCT approach shown before

Calibration of LGM model

Calibration to n swaptions:

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

with

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

Implicit function theorem

Locally, there exists a unique g

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

and

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

Pasting the market vega together

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

Summary

Questions / Discussion