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 ($+ * -/$, $\exp, \sin, ...$) 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)

---

[1]theory says 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)
- *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 (SIMD instructions)
- this is not an exception, but seems to occur for every "numerically intense" code section (see below for a second example)

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 optimization can be applied to non-native types)
- e.g. a convolution engine for bermudan swaptions is 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

---

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

Source Code Transformation

- generate adjoint code at compile time, which hopefully 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 (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

Steps to use 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

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

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

```cpp
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 (non-transformed code): 4.2 ms[3]
- pricing + gradient $\in \mathbb{R}^{105}$: 25.6 ms
- additional stuff[4]: 6.2 ms
- adjoint calculation multiple: 6.1x (7.6x including add. stuff)
- common, practical target for the adjoint multiple: 5x - 10x

---

[3]Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz, single threaded
[4]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, 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[5]

$$\text{Black}(\sigma_1) - \text{Npv(LGM)}(\zeta_1) = 0$$
$$...$$
$$\text{Black}(\sigma_n) - \text{Npv(LGM)}(\zeta_n) = 0$$

with

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

---

[5]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

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

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)
- matrix inversion and multiplication is cheap
- the additional computation time is quite small (see the example above)

Summary

- global instrumentation (via typedefs) with active variables can lead to performance 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, an example in terms of a bermudan swaption engine was given using OpenAD/F

Questions / Discussion

thank you for your attention