Handwritten Adjoints by Operator Overloading

C++11 meta-program creates adjoints of arbitrary procedural codes

Jacques du Toit (Numerical Algorithms Group), Johannes Lotz (RWTH Aachen) and Klaus Leppkes (RWTH Aachen)

Algorithmic Differentiation: Handwritten vs Runtime

- Handwritten adjoint : most efficient, but laborious to write, error-prone and have two sets of source to be kept in sync.
- Runtime : primal code executed through a tool which builds execution graph (tape) at runtime and computes adjoint from this. Very flexible, only one source, but has runtime overheads and potentially massive memory use.

Runtime AD tools such as `dco/c++` are most popular due to flexibility and productivity. Memory use of `dco/c++` can be constrained as much as desired, and runtime overheads are small.

Runtime AD Tools are not Always Appropriate

Sometimes handwritten adjoints are preferable, or are the only option:

- Accelerators such as GPUs: thousands of threads, limited RAM and branching control logic makes taping hard or infeasible
- "Static" library routines (e.g. spline evaluation) which don't change and are heavily used
- Performance-critical kernels, e.g. Monte Carlo
- Applications where available memory is constrained

Compile-time Adjoints Through Operator Overloading

With C++11 it is possible to write a meta-program based overloading tool which uses the platform C++ compiler (gcc, cl, nvcc, etc.) to instantiate an adjoint code at compile time. If the tool is designed carefully, **and the platform compiler is good enough**, then the resulting code can be optimized to the same level as a handwritten adjoint. We implemented this in a new `dco/c++` type called **dco_ntr**.

Multi-dimensional Local Volatility Test Code

We applied the tool to an FX basket code: 10 factor local volatility model driving a basket of 10 FX rates, local volatility surfaces represented by 1D splines. The model has ≈ 450 inputs, mainly the market observed implied volatility quotes. Price and full gradient are computed on GPU or CPU (serial only).

Adjacent code snippets show the Monte Carlo kernel. Minor details are omitted for brevity. The function `compute` is called from CPU code or GPU kernels and instantiated with `dco_ntr` scalar and array types or normal (`float, float*`) types. It stores Monte Carlo paths in `d_logS` and checkpoints local volatilities in `d_locVolCKP` which are used to speed up the adjoint run.

Local Volatility Monte Carlo Source Code

```
template<class FP, class ARRAY>
void getVol(const FP& x, int n, const ARRAY& knots,
            const ARRAY& coeffs, const ARRAY& extrap, FP& sigma) {
   // Linearly extrapolate in the wings
   if(x <= knots[3]) {
      sigma = extrap[0] + extrap[1]*(x - knots[3]);
   } else if(x >= knots[n]) {
      sigma = extrap[2] + extrap[3]*(x - knots[n]);
   } else {
      sigma = evalSpline(x, n, knots, coeffs);
   }
}
template<FP, class ARRAY, class OARRAY>
void compute(const ARRAY& d_S0, const ARRAY& d_rd,  ... [snip]
    ..., const float * d_Z, OARRAY d_logS, float *d_locVolCKP) {
  const auto S0 = d_S0[dim];       const auto rd = d_rd[0];
  const auto rf = d_rf[dim];       const auto dt = d_dt[0];
  const auto sqrtdt = sqrt(dt);    const auto logS0 = log(S0);

  DCO_FOR(FP, i, 0, nTimeSteps-1) // Time step loop
  {
    auto id_Z      = d_Z + i*nPaths;
    auto id_logS   = d_logS   + dim*stride + i*nPaths;
    auto id_locVol = d_locVol + dim*stride + i*nPaths;
    // Get thread-safe high performance input subarrays of size ns & 4
    auto knots  = dco_ntr::input_subarray(d_knots + i*ns, ns);
    auto coeffs = dco_ntr::input_subarray(d_coeffs + i*ns, ns);
    auto extrap = dco_ntr::input_subarray(d_extrap + i*4, 4);
    FP sigma(0);  // Local vol, computed or loaded from checkpt
    DCO_FOR(FP, p, threadId, nPaths-1, nthds) // Path loop
    {
      FP logSi(0);
      DCO_IF(FP, i==0 ) {
        logSi = logS0;
      } DCO_ELSE {
        logSi = id_logS[-nPaths + p];
      }
      DCO_ENDIF
      const auto Si = exp( logSi );  dco_ntr::loop_var<FP> zcorr(0);
      DCO_CKP_CALL(FP, getVol(Si, ns, knots, coeffs, extrap, sigma));
      DCO_FOR(FP, j, 0, dim) {
        zcorr += d_cholCorr[j] * id_Z[p+j*stride];
      }
      DCO_ENDFOR
      const auto inc = (rd-rf-0.5*sigma*sigma)*dt + sqrtdt*sigma*zcorr;
      id_logS[p] = logSi + inc;
    }
    DCO_FOR_STORE_CKP(p) {
      // In forward run, store locvol checkpoint
      id_locVolCKP[p] = dco_ntr::value(sigma);
    } DCO_FOR_LOAD_CKP(p) {
      // In adjoint run, load locvol checkpoint
      sigma = id_locVolCKP[p];
    }
    DCO_ENDFOR  // End path loop
  }
  DCO_ENDFOR   // End time step loop
}
```

Results for Local Volatility Monte Carlo Kernel

The table below gives runtimes (in ms) of the Monte Carlo kernel (passive), the handwritten adjoint (handwritten), the adjoint via the new `dco_ntr` type, and the adjoint via `dco/c++` taping. Figures in brackets are adjoint time as multiple of passive time.

Linux			
	clang 3.6	gcc 4.7	nvcc 7.5
passive	1,461	1,406	18
handwritten	2,997 (2x)	2,808 (2x)	89 (4.9x)
`dco_ntr`	3,031 (2x)	3,025 (2.2x)	83 (4.6x)
`dco/c++` tape	13,579 (9.3x)	11,011 (7.2x)	N/A
Windows			
	clang 3.8	VS2015	Intel2015
passive	1,172	1,510	1,421
handwritten	1,906 (1.6x)	1,992 (1.3x)	1,876 (1.3x)
`dco_ntr`	4,384 (3.7x)	10,241 (6.9x)	11,671 (6.8x)
`dco/c++` tape	16,125 (16x)	24,025 (15.9x)	18,833 (16x)

Results are all on the same machine equipped with a K20c graphics card. Note that CUDA 7.5 does not support C++11 on Windows. The clang compiler is truly impressive on both platforms. Intel 15.0 on Linux gives `dco_ntr` runtime of 9,725ms. Although results on Windows are not as good as on Linux, the memory use on all platforms is always as efficient as a handwritten adjoint.

Benefits of New `dco/c++` Type

- Single "natural looking" source for both primal and adjoint
- Primal code can be changed and adjoint is always in sync
- Adjoint runtime often as fast as handwritten, and much faster than tape
- Memory use as efficient as handwritten adjoint
- The `dco_ntr` type works with any C++11 compiler

The new type system is under development and will be part of a future `dco/c++` release. NAG encourages anyone interested in the technology to contact us as supported early access can be arranged and would be beneficial to both parties.

Acknowledgments

NAG would like to thank Prof. Dr. David Bommes of RWTH Aachen for making code available during early testing.