The NAG Algorithmic Differentiation Portfolio

January 2017

Jacques du Toit
Johannes Lotz

Experts in numerical software and
High Performance Computing

NAG Algorithmic Differentiation Portfolio

- AD tools
  - `dco/c++`, `dco/map` and `dco/fortran`
  - `dco/matlab`$^\alpha$ and `dco/python`$^\alpha$
  - Other languages possible
- Commercial support for tools
- AD consulting services
- NAG AD Library

NAG AD Library

NAG AD Library

- Adjoint (algorithmic/symbolic), tangent and second order adjoint versions of NAG Library routines
- Can be used with any AD tool, including handwritten adjoints
- Useful if need robust adjoints of complex/tricky numerical routines
  - Constrained non-linear programming that is not iterated to convergence
  - Nearest correlation matrix routines
  - Iterative solvers such as CG, GMRES, Multigrid, ...
- Relevant checkpointing to constrain memory
- Checkpointing exposed to users so that memory can be managed

NAG AD Library

Why might this be interesting?

- For complex numerical routines (e.g. NLP) making a robust discrete adjoint is costly
- For other routines (e.g. NCM, CG, GMRES) making efficient adjoint requires good understanding of underlying mathematics

These can be done, but many customers feel it's not good use of their time

NAG AD Library specifically designed to be tool-agnostic

Consulting and Commercial Support for AD Tools

Commercial Support for our AD Tools

NAG doesn't leave you on your own

- We have dedicated team working full time on our AD products
  - On-demand paid resource, not professor/post-doc who'll reply when they have time ....
- We provide excellent customer support (timely, comprehensive, authoritative)
  - Help with using tools/products
  - Help applying AD tools to difficult sections of code (memory management, numerics, etc)
  - Can give advice on most suitable functionality to use, etc
- Implement bug fixes
- Implement feature requests
- Provide comprehensive documentation

AD Consulting Services

For larger pieces of work customers don't want to do themselves. Can be anything, but typical examples are

- Apply our own AD tools to customer's code
- Help customer develop efficient checkpointing strategies
- Develop symbolic or mixed discrete/symbolic adjoints of numerical components
- Develop solutions for adjoints of mixed-language applications
- Develop solutions for differentiating through client's own iterative solvers
- … etc

NAG and RWTH Aachen have more than a decade of experience applying AD to financial and scientific HPC codes

dco/c++

- Why use an AD tool?

dco/c++

dco/c++ = Derivative Code by Overloading

- Operator overloading tool based on a tape
  - Why operator overloading?
- Arbitrary order tangents and adjoints
- Very clean API
- Designed to support checkpointing and “user adjoints”
- Expression templates give local partials at compile time
- Highly optimised and cache-efficient tape structure
- Has a host of specialised functionality, a lot of which is unique
- Easy interfacing with other languages (e.g. Matlab, Python, Fortran) for adjoints of these codes or mixed-source codes

dco/c++

Key features of dco/c++ (there are many others we'll discuss)

- Easy to use
- Cross platform (Windows, Linux)
- Handles whole of C++98: next major release will support C++11
- Very fast
- Very flexible: tool does not assume that adjoints are easy!
- Supports parallelism
- Checkpointing allows memory to be constrained almost arbitrarily
- Developed and tested to industrial software engineering standards
- Fully documented and commerically supported

dco/c++ is Battle Proven

dco/c++ represents over 15 man years of R&D and has been "proven in battle"

- Incorporated into core quant libraries of several large banks
- Applied to MIT Global Circulation Model (ocean and weather)
- Applied to OpenFOAM
- Applied to PETSc
- German Aerospace Centre DG solver *padge* (built on *deal.II*)
- London Queen Mary University fluid dynamics code *gpde*

Using `dco/c++`: Tangent Mode

Tangent Mode

Recall for function F tangent mode computes

$$y = F(x)$$
$$y^{(1)} = \left\langle \frac{\partial F}{\partial x}, x^{(1)} \right\rangle$$

so getting whole gradient/Jacobian is $O(n)$

Only way to get a feeling for `dco/c++` ease of use is to see it used

Tangent Mode

```
template<class FP>
FP foo(FP a, FP std::vector<FP> &x) {
    ... C++98 code ...
}

int main() {
    using FP = double;
    FP a; std::vector<FP> x;
    // Load input data

    FP y = foo(a, x);
}
```

```cpp
template<class FP>
FP foo(FP a, std::vector<FP> &x) {
   ... C++98 code ...
}

int main() {
   // dco/c++ tangent type
   using FP = dco::gt1s<double>::type;

   FP a; std::vector<FP> x;
   // Load input data

   // Seed tangent
   dco::derivative(x[7]) = 1;

   FP y = foo(a, x);

   // Read value
   double v = dco::value(y);
   // Harvest Gradient(7)
   double t = dco::derivative(y);
}
```

Second Order Tangent Mode

Second order tangent model of a function F computes

$$
\begin{aligned}
y &= F(x) \\
y^{(1)} &= \left\langle \frac{\partial F}{\partial x}, x^{(1)} \right\rangle \\
y^{(2)} &= \left\langle \frac{\partial F}{\partial x}, x^{(2)} \right\rangle \\
y^{(1,2)} &= \left\langle \frac{\partial^2 F}{\partial^2 x}, x^{(1)}, x^{(2)} \right\rangle + \left\langle \frac{\partial F}{\partial x}, x^{(1,2)} \right\rangle
\end{aligned}
$$

for inputs $x, x^{(1)}, x^{(2)}$ and $x^{(1,2)}$. Getting whole Hessian is $O(n^2)$

In `dco/c++` higher order is done through recursive instantiation

```
template<class FP> FP foo(FP a, std::vector<FP> &x) { ... }

int main() {
   // dco/c++ second order tangent type
   using FP = dco::gt1s< dco::gt1s<double>::type >::type;

   FP a; std::vector<FP> x;
   // Load input data

   // Seed x^(1)
   dco::derivative( dco::value(x[7]) ) = 1;
   // Seed x^(2)
   dco::value( dco::derivative(x[3]) ) = 1;

   FP y = foo(a, x);

   // Read value
   double v = dco::value( dco::value(y) ;
   // Harvest Hessian(7,3)
   double t = dco::derivative(dco::derivative(y)); // y^(1,2)
}
```

Using `dco/c++`: Adjoint Mode

Adjoint Mode

Recall that the adoint model of a function F computes

$$y = F(x)$$

$$x_{(1)} = x_{(1)} + \left\langle y_{(1)}, \frac{\partial F}{\partial x} \right\rangle$$

for inputs x and $y_{(1)}$ so getting whole gradient/Jacobian is $O(m)$

```
template<class FP> FP foo(std::vector<FP> x) { ... }

int main() {
   using MODE = dco::ga1s<double>;
   using FP    = MODE::type;
   std::vector<FP> x;
   // Load inputs

  MODE::global_tape  = MODE::tape_t::create();
  MODE::tape_t &tape = MODE::global_tape;

  // Register inputs with the tape
  tape->register_variable(x);

  FP y = foo(x);

  tape->register_output_variable(y);
  // Seed adjoint, interpret tape and harvest adjoints
  dco::derivative(y) = 1;
  tape->interpret_adjoint();

  double v = dco::value(y);
  double t = dco::derivative( x[7] );
}
```

Second Order Adjoint Mode

The second order adjoint model of a function F computes

$$\begin{aligned}
y &= F(x) \\
y^{(2)} &= \left\langle F(x), x^{(2)} \right\rangle \\
x_{(1)} &= x_{(1)} + \left\langle y_{(1)}, \frac{\partial F}{\partial x} \right\rangle \\
x_{(1)}^{(2)} &= x_{(1)}^{(2)} + \left\langle y_{(1)}^{(2)}, \frac{\partial F}{\partial x} \right\rangle + \left\langle y_{(1)}, \frac{\partial^2 F}{\partial x^2}, x^{(2)} \right\rangle
\end{aligned}$$

for inputs $x, x^{(2)}, y_{(1)}^{(2)}$ and $y_{(1)}$. Getting the whole Hessian is $O(mn)$

```
int main() {
   using MODE = dco::ga1s< dco::gt1s<double>::type >;
   using FP   = MODE::type;
   std::vector<FP> x;
   // Load inputs

   MODE::global_tape = MODE::tape_t::create();
   MODE::tape_t &tape = MODE::global_tape;
   tape->register_variable(x);

   // Seed tangent x^(2)
   dco::derivative( dco::value( x[7] ) )  = 1;

   FP y = foo(x);

   // Register output, set adjoint y_(1), interpret tape
   tape->register_output_variable(y);
   dco::value( dco::derivative(y) ) = 1;
   tape->interpret_adjoint();

   // Read value
   double v = dco::value( dco::value(y) );
   // Harvest Hessian(7,3)
   double t = dco::derivative( dco::derivative(x[3]) );
}
```

Using dco/c++ with Handwritten Adjoints

External Adjoint Interface

Making adjoints of non-trivial code is not easy

- We know this, we have first hand experience
- Naive application of any overloading tool may run out of memory
- Not everything *should* be handled automatically by the tool
  - Implicit function theorem
  - Symbolic (continuous) adjoints
  - Handwritten discrete adjoints
  - Special handling of certain numerical procedures

`dco/c++` handles all these cases with an *external adjoint interface*

External Adjoint Interface

Suppose we have a handwritten adjoint for a function

```
template<class FP>
void func(const std::vector<FP> &x, FP &y)
{ ... }

template<class FP>
void a1func(const std::vector<FP> &x, std::vector<FP> &a1x,
            FP &y, FP a1y)
{ ... }
```

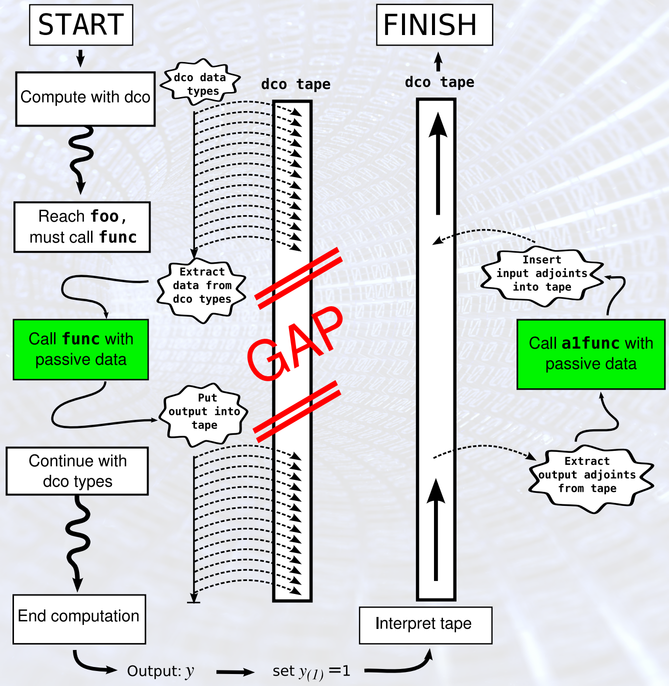
Suppose `func` is called in our code

```
template<class FP>
FP foo(std::vector<FP> &x)
{
    FP y;
    func(x, y);
    return y;
}
```

External Adjoint Interface

We want to use dco/c++ for `foo` but not `func`

- Don't want to tape through `func`
- We want to use handwritten adjoint `a1func`


START
Compute with dco
Reach foo, must call func
Call func with passive data
Continue with dco types
End computation
Output: y
set $y_{(1)}=1$
dco data types
dco tape
Extract data from dco types
GAP
Put output into tape
FINISH
dco tape
Insert input adjoints into tape
Call a1func with passive data
Extract output adjoints from tape
Interpret tape

```cpp
template<class FP> FP foo(std::vector<FP> &x)
{
   // Get dco mode and create external adjoint object
   using MODE = dco::mode<FP>;
   using EAO = MODE::external_adjoint_object_t;
   EAO* cp = MODE::global_tape->create_callback_object<EAO>();

   // Register inputs - get back passive data
   using BASE = MODE::value_t;
   std::vector<BASE> xp( x.size() );
   xp = cp->register_input( x );
   // Can write arbitrary data to checkpoint (FIFO)
   cp->write_data(xp);

   // Run func passively, i.e. not with dco data types
   BASE yp;
   func(xp, yp);

   // Register the outputs - returns dco data type
   FP y = cp->register_output(yp);
   // Write callback function to tape & return dco type
   MODE::global_tape->insert_callback(cpfoo<MODE>, cp);
   return y;
}
```

```
template<class MODE>
void cpfoo(MODE::external_adjoint_object_t *cp)
{
   using BASE = MODE::value_t;
   // Read data from checkpoint (FIFO)
   const auto &xp = cp->read_data< std::vector<BASE> >();

   // Declare adjoints of inputs
   std::vector<BASE> a1xp( x.size(), 0 );
   // Get adjoints of outputs
   BASE a1y = cp->get_output_adjoint();

   // Call handwritten adjoint
   BASE y;
   a1func(xp, a1xp, y, a1y);

   // Write the adjoints into the tape
   cp->increment_input_adjoint( a1xp );
}
```

Checkpointing to Conserve Memory

Checkpointing

Being able to "switch off" `dco/c++` is an important feature

- Allows hand-written user adjoints and *checkpointing*
- Trade flops for memory
- Recompute parts of program to keep tape small

Consider the following program

$$x \xrightarrow{f_1} x_1 \xrightarrow{f_2} x_2 \xrightarrow{f_3} x_3 \xrightarrow{f_4} x_4 \xrightarrow{f_y} y$$

Suppose not enough memory to tape the whole computation

Checkpointing

- At each x_1, x_2, x_3, use external adjoint interface to write a *checkpoint* c_i of computation to the tape
- Checkpoint is sufficient state to restart computation from that point

$$\begin{tikzcd}
x \arrow[r, "f_1"] & x_1 \arrow[r, "f_2"] \arrow[d] & x_2 \arrow[r, "f_3"] \arrow[d] & x_3 \arrow[r, "f_4"] \arrow[d] & x_4 \arrow[r, "f_y"] & y \\
& c_1 & c_2 & c_3 & &
\end{tikzcd}$$

- Run f_1, f_2, f_3, f_4 passively, i.e. without using `dco/c++` types
- Run f_y actively (recording tape) and interpret to get adjoints $\overline{x_4}$
- Restore c_3 and run f_4 actively, interpret and get $\overline{x_3}$
- Restore c_2 and run f_3 actively, interpret and get $\overline{x_2}$, etc
- Tape only ever as big as needed for a *single* function f_i
- But effectively compute f_1, f_2, f_3, f_4 twice, so more flops

Checkpointing

Literature on checkpointing for AD is quite big

- Main question is where to place and how big to make tape
- For *ensembles* (e.g. Monte Carlo paths) treat several paths at once
  - Try to keep tape in L1/L2 cache
  - Can exploit parallelism (record multiple tapes in parallel)
- For evolutions (e.g. PDE solvers) typically checkpoint every k time steps
- Algorithm for optimal placement of checkpoints is known (revolve). Totally depends on user code features
- Research is underway to integrate this into `dco/c++`
  - Linux process forking (automatic checkpointing)
  - Adjoint code modules (Design Patterns)

Additional dco/c++ Functionality

Additional dco/c++ Functionality

- Vector mode adjoints and tangents
- Different tape types
- Activity analysis
- First and second order sparsity pattern detection
- Parallel adjoints through multiple thread-safe tapes
- Tape compression (Jacobian/gradient preaccumulation)
- Direct tape manipulation
- Adjoint MPI support (correct reversal of communication)
- Non-tape adjoint storage
- Combined debugging: finite diff. vs tangent vs adjoints
- ...

Vector Mode

- Consider vector-valued function

$$x \xrightarrow{F} y$$

  where $y \in \mathbb{R}^m$
- Adjoint model $F_{(1)}$ computes

$$x_{(1)} = x_{(1)} + \left\langle y_{(1)}, \frac{\partial F}{\partial x} \right\rangle$$

- To get whole gradient call $F_{(1)}$ m times with $y_{(1)}$ ranging over Cartesian basis vectors in $\mathbb{R}^m$

Vector Mode

$$x_{(1)} = x_{(1)} + \left\langle y_{(1)}, \frac{\partial F}{\partial x} \right\rangle$$

- $\frac{\partial F}{\partial x}$ independent of $y_{(1)}$, as is `dco/c++` tape
- Tape interpretation just computes inner product
- So we can compute all m inner products at once with a single tape interpretation
- Think of $y_{(1)}$ being a matrix instead of a vector
- For vector-valued functions this typically gives a respectable speedup vs multiple interpretations
- `dco/c++` has vector mode support for tangent and adjoint codes

Vector mode exposed through **`dco::ga1v`** and **`dco::gt1v`** types

Activity Analysis

- In large, complex, legacy codes, not always easy to tell which intermediate variables are active
- If we don't treat all as active, we risk wrong sensitivities
- If we treat all as active, we risk very large tape
- `dco/c++` supports activity analysis types
- Before writing local data to tape, checks that there is dependence on active input data
- Slightly slower recording, but can result in (much) smaller tapes

Sparsity Pattern Detection

- Pattern data types can reveal which elements of Jacobian/Hessians are *structurally zero*
- Structural zero means no data dependence between a given input/output pair, rather than a sensitivity that happens to be zero

```
template<class FP> void foo(FP x1, FP x2, FP &y1, FP &y2)
{
   y1 = x1*x2;  // y1 depends on x1 & x2 but deriv may be zero
                // if either x1=0 or x2=0

   y2 = x1*x1;  // structural zero: y2 doesn't depend on x2
}
```

Information useful when computing Jacobians/Hessians as can reduce number of tape interpretations

Parallel Adjoint

- Up to now we've only seen one *global tape*

```
dco::ga1s<double>::global_tape =
                              dco::ga1s<double>::tape_t::create();
```

- Global tape is thread safe (OpenMP) but incurs overheads
- dco/c++ also has thread local tapes
- Can be recorded and interpreted concurrently on different threads with no performance loss
- Useful for things like adjoints of Monte Carlo simulations

Tape Compression by Gradient Preaccumulation

- Sometimes we experience a "narrowing" in the computational graph

$$\begin{tikzcd}
x \arrow[r] & \cdots \arrow[r] & w_1 \arrow[r] & \cdots \arrow[r] & w_2 \arrow[r] & \cdots \arrow[r] & y \\
\mathbb{R}^{500} \arrow[u] & & \mathbb{R}^{10} \arrow[u] & & \mathbb{R}^{10} \arrow[u] & & \mathbb{R}^{70} \arrow[u]
\end{tikzcd}$$

- The graph between w_1 and w_2 may be complex and tape representation may be huge
- Collapsing this graph into a Jacobian requires only 100 storage elements
- Can dramatically reduce tape size
- `dco/c++` provides API for marking which sections of the program to collapse into preaccumulated Jacobians

Further dco/c++ Functionality

- Direct tape manipulation
- Adjoint MPI support (correct reversal of communication)
- Non-tape adjoint storage
- Combined debugging: finite diff. vs tangent vs adjoints
- Adjoint code modules (design patterns)

Coming in dco/c++ v4.0

- Code instrumentation for bidirectional dataflow analysis
- Linux only (low-level memory info)
  - Combined storage of active/passive data: zero-copy when calling functions passively
  - Fixed, iteration-independent adjoint memory size
  - Automatic spawing/reducing of per-thread tapes for OpenMP
  - Fully automatic checkpoint support (process spawning)

CUDA Support via `dco/map`

dco/map

dco/map = Meta Adjoint Programming

- Not just for CUDA, works with any C++11 compiler
- Designed to work properly (high performance) on CUDA GPUs
- Completely separate from dco/c++, but easy to use with it
- Tool for tape-free adjoints

dco/map

What's the problem with tape?

- Each thread needs its own tape
- 30,000 threads on GPU, only about 12GB RAM
- For GPU-intensive applications (XVA) highly likely to run out of GPU memory if tape all floating point operations
- Tricky to have efficient tape in presence of warp divergence (non-coalesced reads/writes)
- Taping will be much slower on GPU than recompute

dco/map

dco/map designed to address this

- Is a tape-free operator overloading AD tool
- Uses meta-programming to let C++11 compiler write the adjoint code *at compile time*
- Goal is to be as fast as hand-written discrete adjoint
  - On CUDA we more or less achieve this
  - On x86 we're not far off
- It's faster than tape on CPUs (typically quite a bit faster)

but dco/map is more demanding to use and requires more code changes than dco/c++

`dco/map`

To use `dco/map`

- Functions must have `void` return type
- All function arguments passed by reference
- All intermediate LHS must have type `const auto`
- Variables may not be overwritten
  - Linear overwriting handled automatically
  - Non-linear overwrites can be handled but require user input (storage)
- All control flow (conditionals, loops, function calls) must be implemented with `dco/map` macros
  - Macros just thin wrappers to factory methods, so nothing scary

```
template<class Active>
void foo(int n, const Active x[], Active &y)
{
   // Intermediate LHS of type const auto
   const auto a = sin(x[0])*x[1];

   // Outputs of control flow statements
   Active a, b(0), c, d;
   // A for-loop with simple linear over-writing
   MAP_FOR(Active, i, 2, n-1, 1) {
      b += x[i];
   } MAP_FOR_END;

   // Call function bar to compute c
   MAP_CALL(Active, bar(a, b, c));

   // Use an if-statement to compute d
   MAP_IF(Active, b < c) {
      d = exp(a)*b;
   } MAP_ELSE {
      d = c*cos(b-a);
   } MAP_IF_END;

   y = d*c*b*a;
}
```

```cpp
int main()
{
    using Active = dco_map::ga1s<double>::type;
    int n = 10;
    Active * x = new Active[n];
    // Populate x with data
    ...

    // Declare output
    Active y;
    // Seed adjoint
    dco_map::derivative(y) = 1;

    foo(n, x, y);

    // Read value
    double v = dco_map::value(y);
    // Read adjoint
    double t dco_map::derivative( x[7] );
}
```

dco/map Features

dco/map has the following features/functionality

- First order tangent and adjoint (second order in development)
- Produces single *unified code* for primal, tangent and adjoint
- Primal as fast as non dco/map primal
- Specialised high-performance array types to handle race conditions inherent in parallel adjoints
- Supports whole of C++11, cross platform
- API for storing things you don't want to recompute
- Easy integration with dco/c++ via external adjoint interface

dco/map currently in final stage client PoC for GPU XVA application: GPU adjoint factor is 2.6x

Our Value Proposition

NAG AD Value Proposition

- Industry leading AD consulting services
- Industry leading AD tools
  - Speed
  - Tape size
  - Functionality and flexibility
- The only (commercial) solution that can handle CUDA
- Dedicated commercial support for all our AD products

Thank you