AD Master Class 3

Testing and Validation

Viktor Mosenkis

viktor.mosenkis@nag.co.uk

13 August 2020

nag®

Experts in numerical algorithms
and HPC services

AD Masterclass Schedule and Remarks

AD Masterclass Schedule

- ☐ 30 July 2020 | Why the need for Algorithmic Differentiation?
- ☐ 6 August 2020 | How AD works
- ☐ 13 August 2020 | Testing and validation
- ☐ 20 August 2020 | Pushing performance using SIMD vectorization
- ☐ 27 August 2020 | Bootstrapping validated adjoints on real-world codes

Remarks

- ☐ Please submit your questions via the questions panel at any time during this session, these will be addressed at the end.
- ☐ A recording of this session, along with the slides will be shared with you in a day or two.

Dialogue

We want this webinar series to be interactive (even though it's hard to do)

- We want your feedback, we want to adapt material to your feedback
- Please feel free to contact us via email to ask questions at any time
- We'd love to reach out offline, discuss what's working, what to spend more time on
- For some orgs, may make sense for us to do a few bespoke sessions

Blog:
https://www.nag.com/blog/algorithmic-differentiation-masterclass-series-2

Outcomes

- Learn how to validate (check for errors) AD code
- Discuss typical errors made when applying AD with operator overloading tool
- See changes to the user code that are required to apply operator overloading AD tool
- Implications for code structure and build system

What can go wrong when applying AD?

- Wrong driver e.g.
  - wrong seeding
  - inputs of the function are not const (only adjoint)
  - forget to register input variables (anchor as root notes in the DAG)
- AD tool applied incorrectly e.g.
  - haven't activated (replaced with AD types) all required variables
  - wrong code changes
  - messed up the tape, e.g
    - check-pointing (symbolic adjoints)
    - path-wise adjoints (MC)
    - parallelization
    - black box routines
- Reference results by FD are difficult to verify
- non-differentiability
- high memory requirements for the adjoint model

How do we test AD code using TLM and ADM

- Compute Jacobian with
  - ☐ tangent-linear and adjoint model
  - ☐ compare the results
  - ☐ Ok for small Jacobians
  - ☐ slow unit testing may if you have lots of unit tests with moderately sized Jacobians

- Compute tangent-adjoint identity

$$\bar{x}^T \cdot \dot{x} = (\underbrace{F'(x)^T \cdot \bar{y}}_{=\bar{x}})^T \cdot \dot{x} = \underbrace{\bar{y}^T \cdot F'(x)}_{=\bar{x}^T} \cdot \dot{x} = \bar{y}^T \cdot \underbrace{(F'(x) \cdot \dot{x})}_{=\dot{y}} = \bar{y}^T \cdot \dot{y}$$

  - ☐ fast (only one execution of tangent-linear and adjoint model)
  - ☐ reliable
  - ☐ PS: in theory you need to check with several seed vectors, but no-one ever does this

What parts can we test with TLM and ADM?

We can test

- driver
- parallelization
- Advanced adjoint techniques that operate directly on the DAG
  - checkpointing/symbolic adjoints
  - illegal memory access to the tape
  - running out of memory

But we can't check the code changes we performed to activate the entire data flow. Is it really hard to activate the code? Just replace the data types... Well what about

- routines that work with `void` pointers. You need to fix them.
- treat correctly data that is written and then read from file.
- 3rd party dependencies (another internal library you just don't want to touch for now)

What models do we need to validate ADM

Statement "I've activated my entire data path correctly" is very strong and we need FD to check this! That is why we need

- Finite differences
- Tangent-linear model
- Adjoint model

to test the adjoint model

Testing AD code

- Tangent vs. FD
  - ☐ full Jacobian (for cheap Jacobians)
  - ☐ single entries of the Jacobian
  - ☐ Jacobian-vector products (not advisable problems with FD accuracy)
  - ☐ Jacobian-vector products with small number of non-zero entries
- Adjoint vs. TLM
  - ☐ full Jacobian (for cheap Jacobians)
  - ☐ single entries of the Jacobian
  - ☐ Jacobian-vector products (Jacobian matrix is required)
  - ☐ tangent-adjoint identity
- Adjoint vs. FD
  - ☐ same as for TLM but problems with FD accuracy for Jacobian-vector products and tangent-adjoint identity

Procedure to apply AD

- Downscale your problem (if possible)
- Implement FD
- Replace all floating point variables with corresponding
  - tangent-linear and
  - adjoint types
- Ensure that your original values are correct with both tangent-linear and adjoint types. Helps to find
  - wrong code changes
  - unstable primal code
- Write the driver for the tangent-linear model and compare vs. FD
- Write the driver for the adjoint model and compare vs. tangent-linear (and FD)

Common pitfalls: Inputs to the function must be const.

Inputs to the function must be const.

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){ // x is written  }
3
4  int main(){
5    DCO_ADJOINT_TYPE *x, y;
6    ...
7    for (int i=0; i<n; i++)
8    DCO_MODE::global_tape->register_variable(x[i]);
9    foo(n, x, y);
10   DCO_MODE::global_tape->register_output_variable(y);
11   dco::derivative(y)=1.0;
12   DCO_MODE::global_tape->interpret_adjoint();
13
14   for (int i=0; i<n; i++) J[i] = dco::derivative(x[i]);
15 }
```

Common pitfalls: Aliasing (FIX)

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){ // x is written  }
3
4  int main(){
5    DCO_ADJOINT_TYPE *x, *x_in, y;
6    ...
7    for (int i=0; i<n; i++){
8      DCO_MODE::global_tape->register_variable(x_in[i]);
9      x[i] = x_in[i]; //x_in is pure input
10   }
11   foo(n, x, y);
12   DCO_MODE::global_tape->register_output_variable(y);
13   dco::derivative(y)=1.0;
14   DCO_MODE::global_tape->interpret_adjoint();
15   //use x_in to access derivatives
16   for (int i=0; i<n; i++) J[i] = dco::derivative(x_in[i]);
17 }
```

Common pitfalls: Not activating all variables

Through not activating all variables you can loose some dependencies. In this example, we are not interested in derivatives with of output `z` but it must be active (of dco/c++ type). Compiler can help you.

```
1  void foo(T& x, T& y, double& z) {
2    z = dco::value(sin(x)); //explicit use of dco::value
3    y = cos(x) * z;
4  }
5  int main(){
6    DCO_ADJOINT_TYPE x, y;
7    double z;
8    ...
9    DCO_MODE::global_tape->register_variable(x);
10   foo(x, y, z);
11   ...
12   std::cout << "dy/dx = "dco::derivative(x); //wrong
13 }
```

Common pitfalls: Not activating all variables (FIX)

Through not activating all variables you can loose some dependencies. In this example, we are not interested in derivatives with of output `z` but it must be active (of dco/c++ type). Compiler can help you.

```
1  void foo(T& x, T& y, T& z) {
2    z = sin(x); //now active
3    y = cos(x) * z;
4  }
5  int main(){
6    DCO_ADJOINT_TYPE x, y;
7    DCO_ADJOINT_TYOE z;
8    ...
9    DCO_MODE::global_tape->register_variable(x);
10   foo(x, y, z);
11   ...
12   std::cout << "dy/dx = "dco::derivative(x); //correct
13 }
```

Code Implications: Template arguments

Modern operator overloading AD tools use expression templates to perform some calculations at compile time (e.g. statement level preaccumulation). To do this `decltype(x*y) !=decltype(x)`. Implications for templates: compiler can't infer the template parameter, thus you get a compile time error.

```
1  template <typename T>
2  void foo(T& x, T& y){ ... }
3
4  int main(){
5    DCO_TLM_OR_ADJ_TYPE x, y
6    ...
7    foo(x*x, y);
8    ...
9  }
```

Code Implications: Template arguments (FIX)

Possible fix is to create an auxiliary variable, store the result of the operation in this variable and use it to call the function.

```
1  template <typename T>
2  void foo(T& x, T& y){ ... }
3
4  int main(){
5    DCO_TLM_OR_ADJ_TYPE x, y
6    // create a seperate variable
7    DCO_TLM_OR_ADJ_TYPE tmp = x*x;
8    //and use it call the function
9    foo(tmp, y);
10   ...
11 }
```

Code Implications: Black-box routines

There exist four ways to deal with black-box routines in your code

- use tangent/adjoint version of the routine from its vendor
- use symbolic derivative
- use FD to compute the derivative of the black-box routine
- replace with open source one (not always an option)

For the first three options your AD tool has to allow you to:

- create a checkpoint
- create a custom callback
- directly update adjoint values in the tape

This is non-trivial, we will deal with it when we talk about symbolic adjoints. Alternatively, ask your vendor to provide you with tangent/adjoint versions of the code, using your AD tool.

Common pitfalls: Parallelization MPI

For tangents you must ensure to communicate both value and tangent component. E.g.

```
1 double *data;
2 MPI_Send(data,count, MPI_DOUBLE, dest, Tag, COMM);
```

should be replaced with

```
1 DCO_TANGENT_TYPE *data;
2 MPI_Send(data,count, MPI_DOUBLE_COMPLEX, dest, Tag, COMM);
```

`MPI_DOUBLE_COMPLEX` can be replaced with your own data type consisting of two doubles

Common pitfalls: Parallelization MPI

For adjoints use AMPI Library. E.g.

```
1 double *data;
2 MPI_Send(data,count, MPI_DOUBLE, dest, Tag, COMM);
```

should be replaced with

```
1 DCO_ADJOINT_TYPE *data;
2 AMPI_Send(data,count, MPI_DOUBLE, dest, Tag, COMM);
```

AMPI ensure correct reversal of the communication.

Common pitfalls: Parallelization OpenMP

Tangents should work more or less out of the box. You just need to implement the operators for some specific OpenMP directives e.g.

- reductions
- atomics

But compiler will tell you this.

Adjoints with OpenMP are more complicated. During tape interpretation

- shared reads become shared writes inherently a race condition
- AAD tapes are not thread safe across all tools (too slow)
- there are other ways of dealing with race conditions. We will look at this at later master class once we talk about Monte Carlo.

Code Structure: Put differentiated code in one routine

Ideally we should put the whole differentiated code in one routine. E.g.

```
1  void foo(const DCO_T *a, const DCO_T *b, const double
       *ap, DCO_T *y, double *yp){ ... }
```

- easier to write the driver
- separation from driver and function
- easy to reuse the code for different models (primal/FD, TLM, ADM)

Might be not so easy achievable for your code. You can mix the driver with the actual differentiated code, but this makes it more complicated to write correct driver and reuse the code.

Build system implications

As already discussed the code should support all three models

- primal/FD
- tangent-linear
- adjoint

Goals for our AD-enabled build system:

- reduce operational risk: one definition for the algorithm
- Limit compilation time for individual source files as much as possible
- Maximise build level parallelism

Build system implications

Are templates (look at the blog) the solution?

Templates

- are rules for creating code (hence meta-programming)
- separate the algorithm from the data the algorithm operates on
- do not produce any object code, unless they are instantiated with actual types (e.g. `float`, `double`, `dco::ga1s<double>::type`, etc...)
- are placed in header files and included everywhere, so that they can be instantiated with whatever type the user requires.

Doing templates naively will violate 2nd and 3rd goals.

Build system implications

Should we then move the implementation to `.cpp` file. This violates 1st goal:

- separate implementation for each AD model (duplicate the algorithm)
- `typedef` makes it very hard to produce a library with all three models, since `typedef` can only have one value at a time.
- deal with naming of the object files in the build system

Build system implications

Our recommendation: move the implementation to `impl.hpp`. `impl.hpp` is included only in three different `.cpp` files that contain the instantiations for primal/FD, TLM and ADM.

- whole implementation in the `impl.hpp`
- code is reused, so only one implementation to maintain
- we have three times more `.cpp` files for the models, but they can be build in parallel
- the templated code for each model is included/compiled only in once.

Summary

In this Class we learned

- How to validate AD code
- Typical errors made when applying AD with operator overloading tool
- What changes to the user code are required to apply operator overloading AD tool
- Code structure and build system

AD Master Class 4: Pushing performance using SIMD vectorization

In the next part we will

- learn tangent and adjoint vector mode and there applications
- see how SIMD vectorization can speed up the tangent vector mode
- continue discussing tangent (vector) vs. adjoint (vector) mode.

You will see a survey on your screen after exiting from this session.
We would appreciate your feedback.

We are now moving on the Q&A Session