Adding AD to existing projects

Laurent Hascoët
Paul Hovland
Jan Hückelheim
Sri Hari Krishna Narayanan

SIAM AN24 Student Days

What we usually start with

- Existing code spread over many files
- Non trivial build system
- Multiple build time configuration options
- Multiple top level routines and drivers
- Calls to libraries
- Extensive I/O, initialization
- Parallelism
- Many users and use cases not needing AD

Code Preparation

Analyze the code ...

- Start with a small case first
- Identify portion(s) and top level function(s) that need to be differentiated
- Identify independent and dependent variables
- Identify initialization code and I/O
- Identify portions that cannot be differentiated

Code Modification and Maintenance

Prepare the code ...

- Rewrite portions the cannot be differentiated
  - Special handling for solvers
  - Stubs for library calls
  - Code changes may need to *guarded* using preprocessor directives
- Write preprocessing scripts
- Add AD to existing build system
- Regression tests to ensure that changes for AD do not change primal output

Validation and Debug

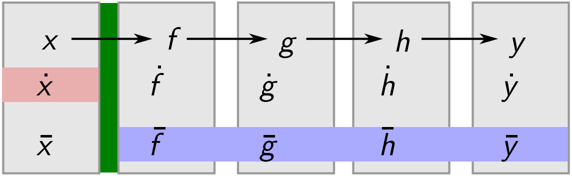
- Original Primal vs. AD Primal
- Finite Differences vs Forward mode[1]
- Forward mode vs Reverse mode
- Many tools provide feedback on problems.
- Start always with the simplest case possible

[1] See https://fluids.ac.uk/files/resources/Farrell_adjoint_slides.pdf p.6 for a more accurate method

The dot-product test

The dot-product of all active tangent variables with all active adjoint variables remains the same throughout the code.

$$y = h(g(f(x)))$$



Forward: $\dot{x}$

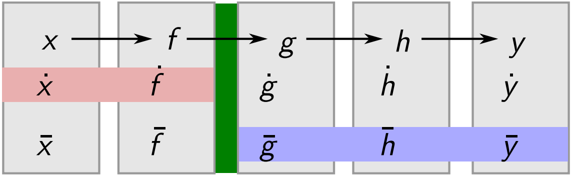
Reverse: $\partial f \cdot \partial g \cdot \partial h \cdot \bar{y}$

Dot-product: $\dot{x} \cdot \partial f \cdot \partial g \cdot \partial h \cdot \bar{y}$

The dot-product test

The dot-product of all active tangent variables with all active adjoint variables remains the same throughout the code.

$$y = h(g(f(x)))$$



Forward: $\dot{x} \cdot \partial f$

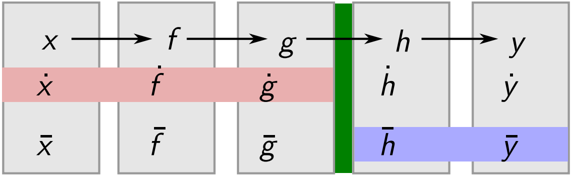
Reverse: $\partial g \cdot \partial h \cdot \bar{y}$

Dot-product: $\dot{x} \cdot \partial f \cdot \partial g \cdot \partial h \cdot \bar{y}$

The dot-product test

The dot-product of all active tangent variables with all active adjoint variables remains the same throughout the code.

$$y = h(g(f(x)))$$



Forward: $\dot{x} \cdot \partial f \cdot \partial g$

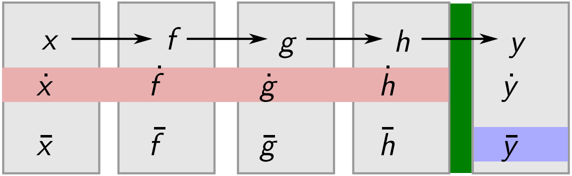
Reverse: $\partial h \cdot \bar{y}$

Dot-product: $\dot{x} \cdot \partial f \cdot \partial g \cdot \partial h \cdot \bar{y}$

The dot-product test

The dot-product of all active tangent variables with all active adjoint variables remains the same throughout the code.

$$y = h(g(f(x)))$$



Forward: $\dot{x} \cdot \partial f \cdot \partial g \cdot \partial h$

Reverse: $\bar{y}$

Dot-product: $\dot{x} \cdot \partial f \cdot \partial g \cdot \partial h \cdot \bar{y}$

Limitations of the dot-product test

1. **It only checks for a given seed.** The dot-product may look good if some errors cancel each other out. (Given some correct tangent-linear vector and a dot-product result, there's an infinite number of adjoint vectors with the same dot-product result).
   - Dot-product is a necessary, but not a sufficient condition for correctness (except for scalar functions). Sufficient condition for consistency: Dot-product test holds for a complete set of basis vectors.
2. **Results will differ by some small amount.** Is it roundoff, or a bug?
3. **It only checks for a given primal input.** If the dot-product works at some point, it may not work somewhere else.
4. It only finds discrepancies between forward and reverse, assuming that forward is correct