Intro to Automatic Differentiation with CoDiPack

Max Aehle

Chair for Scientific Computing, TU Kaiserslautern

Mar 22$^{\text{th}}$, 2021

Problem.

Given a computer program that computes $f : \mathbb{R}^n \to \mathbb{R}^m$ function, construct a computer program that computes the derivatives alongside.
E. g. $f(x) = x^2 + 1$, $f'(x)\big|_{x=4} =?$

Relevance of derivatives

- Uncertainty Quantification: Gradients are sensitivities
- Gradient-based optimization: Gradient points in the direction of steepest ascent

Solution with Finite Difference Quotients

e. g. $f'(x)\big|_{x=4} \approx \frac{(4.001^2+1)-(4^2+1)}{0.001} = 8.001$

Solution with Automatic Differentiation

The program is a sequence of elementary operations, for which we know differentiation rules.

Replace `double` **and overload** $+$, $\cdot$, $\sqrt{\ }$, sin, ...**!**

- **forward mode:** (simpler, good for few inputs)
  Each variable stores **value** and **gradient** w. r. t. all input variables, operators act on both: e. g. for primal code `c = a * b`,

```
c.val  = a.val * b.val;
c.grad = a.grad * b.val + a.val * b.grad;
```

- **reverse mode:** (good for few outputs e. g. **optimization**, memory-intensive)
  Record all operations on a **tape** and play it backwards.
  For each variable, compute the derivatives of all outputs w. r. t. it.

Demonstration with CoDiPack

C++ header-only library for Automatic Differentation, based on the operator-overloading approach

Lead Developers: Max Sagebaum, Johannes Blühdorn, Tim Albring

`https://www.scicomp.uni-kl.de/software/codi/`

Demonstration: Primal program

```
#include <iostream>

int main(int nargs, char** args) {
  double x = 4.0, y;

  y = x * x + 1;

  std::cout << "f(4.0) = " << y << std::endl;
  std::cout << "df/dx(4.0) = " << 2*x << std::endl;

}
```

Demonstration: Forward AD

```
#include <iostream>
#include "../CoDiPack/include/codi.hpp"

int main(int nargs, char** args) {
  codi::RealForward x = 4.0, y;
  x.setGradient(1.0);

  y = x * x + 1;

  std::cout << "f(4.0) = " << y << std::endl;
  std::cout << "df/dx(4.0) = " << y.getGradient() << std::endl;

}
```

Demonstration: Reverse AD

```
#include <iostream>
#include "../CoDiPack/include/codi.hpp"

int main(int nargs, char** args) {
  codi::RealReverse x = 4.0, y;

  codi::RealReverse::TapeType& tape = codi::RealReverse::getGlobalTape();
  tape.setActive();
  tape.registerInput(x);

  y = x * x + 1;

  tape.registerOutput(y);
  tape.setPassive();
  y.setGradient(1.0);
  tape.evaluate();

  std::cout << "f(4.0) = " << y << std::endl;
  std::cout << "df/dx(4.0) = " << x.getGradient() << std::endl;

}
```

Is that everything?

In general, we just have to replace `double` by a `codi`-type everywhere, including numerical libraries etc.

Is that everything?

In general, we just have to replace `double` by a `codi`-type everywhere, including numerical libraries etc.

But: Concerning **numerical algorithms** like

- solving linear systems by an iterative scheme like DROP-TVS
- fixed-point iteration
- ...,

algorithm-dependent adjustments will be necessary.

Example for special treatment of numerical algorithm: $y = A^{-1} \cdot x$ in forward mode

- **Primal code**:

```
double* y = linsolve<double>(A, x);
```

- **Do not differentiate the numerical algorithm** like this:

```
RealForward* y = linsolve<RealForward>(A, x);
```

- Instead, **find an equation/algorithm for the gradients:** product rule $\rightsquigarrow (\frac{\partial}{\partial \text{ input } i}A)y + A(\frac{\partial}{\partial \text{ input } i}y) = \frac{\partial}{\partial \text{ input } i}x$, thus

```
y.vals = linsolve<double>(A.vals, x.vals);
for(i=0; i<nInputVars; i++)
  y.grads[i] = linsolve<double>( A.vals,
             x.grads[i] - A.grads[i]*y.vals );
```

$\Rightarrow$ **Let us find out when the pipeline prototype is ready.**

Application-independent limitations and best practices

- C++ header-only library, compile with `--std=c++11`.
  Not accessible from other languages.
- Avoid C-style `malloc`, `free`, `memcpy`.
- `codi`-type must be used instead of `double`, in libraries also
  $\rightsquigarrow$ maybe we can avoid to differentiate **ROOT**
- Support for parallelisation with **MPI** (MeDiPack) and **OpenMP** (OpDiLib).
- Partial support for **CUDA**.
- **Separation of algorithm and I/O** is helpful here as well, so that no dependencies are overlooked and gradients can be stored alongside values.