The OPM Dense Automatic Differentiation Framework

Andreas Lauser

June 1, 2016

Overview

1. Automatic Differentiation
2. The OPM Dense-AD Implementation
3. Performance

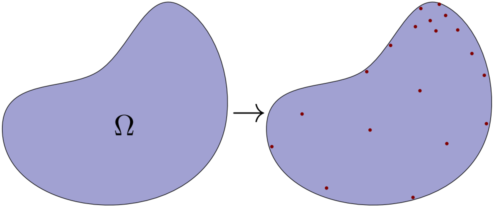
Overview

1. Automatic Differentiation
2. The OPM Dense-AD Implementation
3. Performance

(Forward) Automatic Differentiation

- For any differentiable function $\mathrm{f}(x_1, \ldots, x_n)$ always also evaluate the derivatives with regard to a given set of variables
- i. e., for any given evaluation point $\mathbf{x} = (x_1, \ldots, x_n)$, compute $\mathrm{f}(\mathbf{x}), \partial_{x_1}\mathrm{f}(\mathbf{x}), \ldots, \partial_{x_n}\mathrm{f}(\mathbf{x})$
- Provide operators and "primitive" functions to define composite functions (like residuals of PDEs)

Sparse and Dense Automatic Differentiation



Scalar field: Function f : $\Omega \mapsto \mathcal{R}$ were $\Omega \subseteq \mathcal{R}^d$ for some $d \in \mathcal{N}$
Usual approximation approach:

- Define the value of the field at a finite number of *degrees of freedom* (DOFs)
- Interpolate in-between
  - Allows approximation of a field using a finite number of values $f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)$

“Sparse” Automatic Differentiation

- Choose the full function f(Ω), and derivatives w.r.t. all variables of the DOFs $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in Ω
- Since for any given DOF *i*, the discretized version of f($\mathbf{x}_i$) only depends on the values of a small number of neighbors, most of the derivatives are zero
  - Sparse storage required because *n* is usually large
- Since *n* is correlated with the spatial domain size, it is only known at runtime
  - Dynamic memory management is required
- In OPM, this approach is implemented by `Opm::AutoDiffBlock`

“Dense” Automatic Differentiation

- Do not assume sparsity
- For reasonable performance, the number of derivatives per evaluation must be small
- Ideally, it is specified at compile time
    - Only allows a fixed number of derivatives

Compile-Time Dense-AD: Advantages

- No need for setup of a sparsity pattern
- Number of derivatives does not need to be stored at runtime
- No need for dynamic memory management
- Easier for the compiler to take advantage of SIMD instructions
    - Potentially better performance than sparse-AD

Compile-Time Dense-AD: Disadvantages

From a “reservoir simulator’s” point of view:

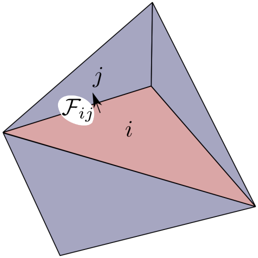
- May require fundamental changes to the linearization algorithm (cf. last year’s talk)
- The number of variables per DOF is fixed
  - Not easily applicable if the number of conservation quantities is specified at runtime

Dense-AD for Finite-Volume-Methods

An "all domain" function function $r(\Omega)$ can be linearized using compile time dense-AD:

- Loop over all DOFs $\mathbf{x}_i$ of the domain
- Compute the residual $r_i(\mathbf{x}_i)$ of the current DOF and the derivatives $\partial_i r_j(\mathbf{x}_i)$ for the all DOFs j (this results in a dense system of equations of size n)
- For PDEs, the stencil is small; i.e. most derivatives of DOFs are zero and the dense system of equations thus is small
- Store the result in a sparse global Jacobian matrix and a global residual vector (this means to sum up the respective entries)

Dense-AD for Finite-Volume-Methods (cont)



Evaluation of the residual of the whole stencil can be done efficiently with finite volume discretizations:

- The storage and source terms are evaluated with their derivatives w.r.t. to primary variables of DOF *i*
- For DOF *i*, $-\mathcal{F}_{ij}$ are summed up
- The residuals of the stencil's other DOFs *j* are only affected by the fluxes from *i* to *j*:
  - $\partial_i \mathrm{r}_j = \partial_i \mathcal{F}_{ij}$

Overview

The OPM Dense-AD Implementation

- Idea: Make function evaluations behave as much as primitive floating point values as possible
  - Implemented by `Opm::LocalAd::Evaluation<Scalar, numDeriv>`
  - Provides arithmetic, comparison and assignment operators
  - Optimized variants of all operators if one operand is a primitive floating point value
    - E.g. assignment of constant value *c* to any `Evaluation` object `f` is interpreted as constant function $f(\mathbf{x}) = c$
  - Common functions from `<cmath>` are available
  - `Evaluation` objects can be transparently used as scalars for Dune's linear algebra classes (e.g. `Dune::FieldVector` and `Dune::FieldMatrix`)
- Nesting possible: `Evaluation` can be used as scalar values for other `Evaluation` objects

Math Toolboxes

- Often code should work the same for primitive floating-point values and `Evaluation`
- Sometimes only the values (without derivatives) are of interest
- “Decaying” `Evaluation` objects sometimes needed:
  - Ignore derivatives if left-hand-side is primitive, else pass through the right-hand-side
- `<cmath>` only deals with primitive values

Math Toolboxes (II)

Solution:

- Templateize on the type of scalar values which is supposed to be used (can be an `Evaluation` or a primitive floating point type)
- Introduce the concept of a “math toolbox”:
  - Template class with specializations on `Evaluation` objects and on primitive floating point values
  - Provides access to the value of an object
  - Provides a defined way to decay objects
  - Provides the most common functions of `<cmath>`

Example: $f(x) = \sin(x)$

```
template <class Eval>
Eval fn(const Eval& x) {
    return Opm::MathToolbox<Eval>::sin(x);
}

int main() {
    std::cout << fn(3.1415/5) << std::endl;

    typedef Opm::LocalAd::Evaluation<double, 1> Eval;
    Eval x(3.1415/5); x.derivatives[0] = 1.0;
    Eval y = fn(x);
    std::cout << y.value << " | " << y.derivatives[0]
              << std::endl;
    return 0;
}
```

This prints:

```
0.58777
0.58777 | 0.809028
```

Example: $\sin(xy)\cos(x^{2.5})$

```
template <class Eval>
Eval fn(const Eval& x, const Eval& y) {
    typedef Opm::MathToolbox<Eval> Toolbox;
    return Toolbox::sin(x*y)*Toolbox::cos(Toolbox::pow(x, 2.5));
}
```

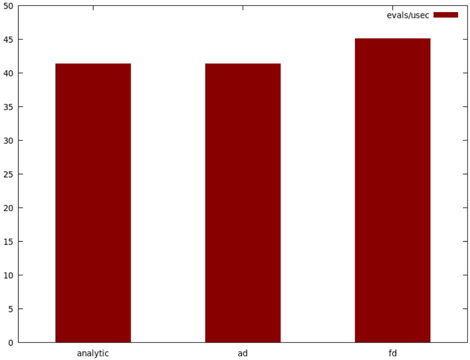
All other code stays identical! (after adjusting for the second variable)

Overview

Micro-Benchmarks[1]: $f(x) = \sin(x)$

$$\partial_x f(x) = \cos(x)$$

Computations of $\partial_x f(x)$ and $f(x)$ per μ-sec on a i7-5930K CPU @ 3.5GHz:



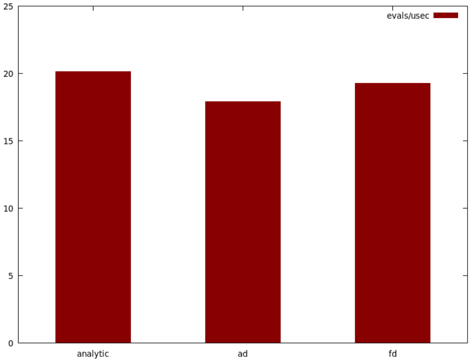
Number of evaluations (with derivatives) per μ-second

[1] `https://poware.org/aibi7osa/ubencheval.tar.gz`

μ-Benchmarks II: $f(x,y) = \sin(xy)\cos(x^{2.5})$

$$\partial_x f(x,y) = \sin(xy)\left(y\cos(x^{2.5}) - 2.5\,x^{1.5}\sin(x^{2.5})\right)$$

$$\partial_y f(x,y) = x\cos(xy)\cos(x^{2.5})$$



Number of evaluations (with derivatives) per μ-second

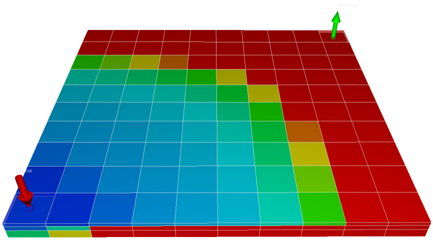
Performance: Reservoir Simulators

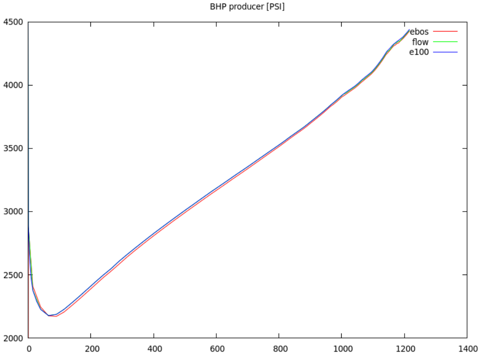
OPM provides two simulators, ebos and flow:

- flow uses the sparse AD approach
- ebos uses the dense AD approach
- Grid, deck processing, and material framework identical
- Code for the simulators themselves is completely disjoint
- Amount of resources that have been applied to flow is significantly larger
  - flow has seen quite a bit more performance related work
  - ebos should only be considered as an advanced prototype

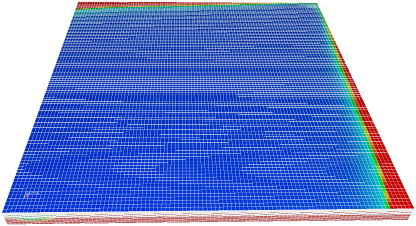
The following results should be taken with a grain of salt!

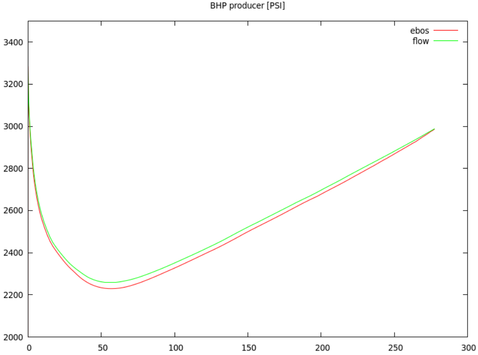
Boring Problem: SPE-1



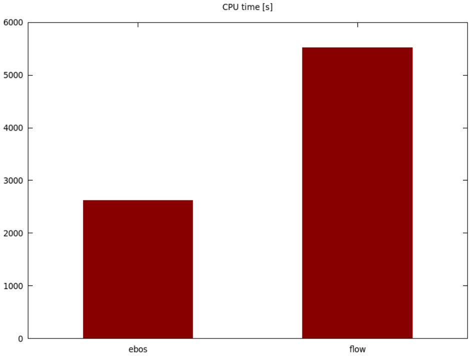


Large Boring Problem: Refined SPE1



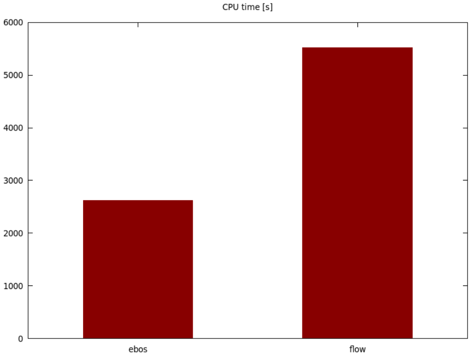


Performance: Apples and Bananas



Overall time of the refined SPE1 problem. flow/ebos ratio: 2.11

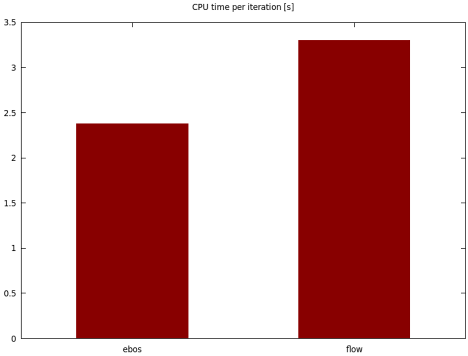
Performance: Apples and Bananas



Overall time of the refined SPE1 problem. flow/ebos ratio: 2.11

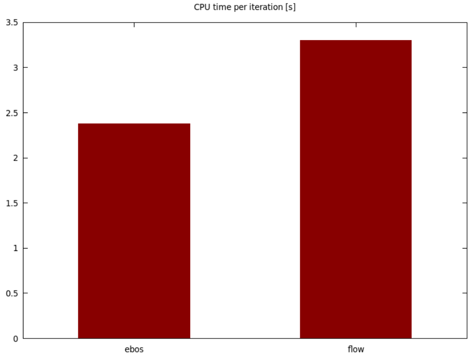
- flow required more iterations

Performance: Apples and Oranges



Time per Newton-iteration of the refined SPE1 problem. flow/ebos ratio: 1.39

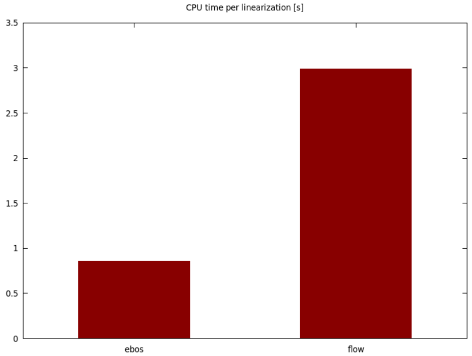
Performance: Apples and Oranges



Time per Newton-iteration of the refined SPE1 problem. flow/ebos ratio: 1.39

- The linear solver used by flow is more performant

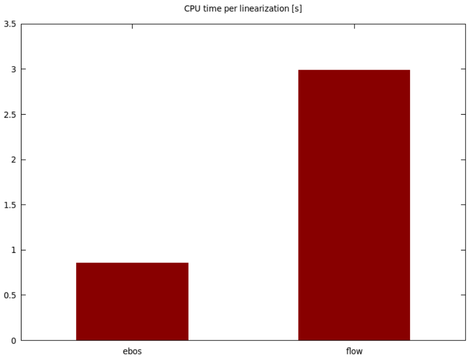
Performance: Green Apples and Red Apples



Linearization time per Newton iteration for the refined SPE1 problem.

flow/ebos ratio: 3.50

Performance: Green Apples and Red Apples



Linearization time per Newton iteration for the refined SPE1 problem.

flow/ebos ratio: 3.50

- Remember: ebos should considered to be "just" an advanced prototype!

Summary

- Automatic differentiation allows to conveniently evaluate a function together with its derivatives
- For discretized PDEs, AD can be used “globally” or “locally”
- The “global” approach leads to sparse data structures
- Compile-time dense-AD is more limited, but
  - Much simpler
  - Convection-diffusion-type equations can be linearized
  - Seems to perform better for linearizing convection-diffusion-type equations (if used in conjunction with a suitable linearization procedure)

Summary

- Automatic differentiation allows to conveniently evaluate a function together with its derivatives
- For discretized PDEs, AD can be used “globally” or “locally”
- The “global” approach leads to sparse data structures
- Compile-time dense-AD is more limited, but
  - Much simpler
  - Convection-diffusion-type equations can be linearized
  - Seems to perform better for linearizing convection-diffusion-type equations (if used in conjunction with a suitable linearization procedure)

Thank you for your attention.