Accurate Derivatives for Computational Engineering with Automatic Differentiation for Matlab (ADiMat)

Johannes Willkomm[×] Christian H. Bischof[×]
H. Martin Bücker[*]

[×]Fachbereich Informatik
TU Darmstadt

[*]Institute for Scientific Computing
RWTH Aachen University

2nd International Conference on Computational Engineering
October 6, 2011

RWTH AACHEN UNIVERSITY

Overview

Automatic Differentiation for Matlab

- Automatic Differentiation (AD)
  - Automatically differentiate numeric programs
  - Available for e.g. C/C++, Fortran, Python, Matlab
  - Accurate derivatives at arbitrary points
  - See `http://www.autodiff.org`
- Matlab
  - Matlab is popular for prototyping and rapid development
  - Can also solve large problems efficiently
  - Is easy to learn
  - Provides a wealth of high-level builtin functions
  - Open-source alternative: GNU Octave
- **ADiMat**: **A**utomatic **Di**fferentiation for **Mat**lab
  - ... and for Octave
  - `http://www.sc.rwth-aachen.de/adimat`
  - Developed by André Vehreschild first, now by me
- Other Matlab AD tools: TOMLAB/MAD and INTLAB

Where derivatives are needed

Whenever dealing with non-linear problems in numerics, derivatives are necessary

Simulation

- PDE-Solvers
  - Finite Element, Finite Volume, or Finite Difference schemes
- Non-linear equation systems
  - Newton method
  - Need *Jacobian* matrix

Optimization

- Inverse problems, Shape optimization, Parameter estimation
- Non-linear optimization
  - Need *Gradient* of cost/objective function
  - Possibly second-order derivative information

Derivatives of Matlab functions

Consider Matlab **function** [y z] = **f**(a, b)

- Jacobian matrix of derivatives:

$$J = \frac{\partial(y,z)}{\partial(a,b)} = \left(\begin{array}{c|c} \frac{\partial y}{\partial a} & \frac{\partial y}{\partial b} \\ \hline \frac{\partial z}{\partial a} & \frac{\partial z}{\partial b} \end{array}\right)$$

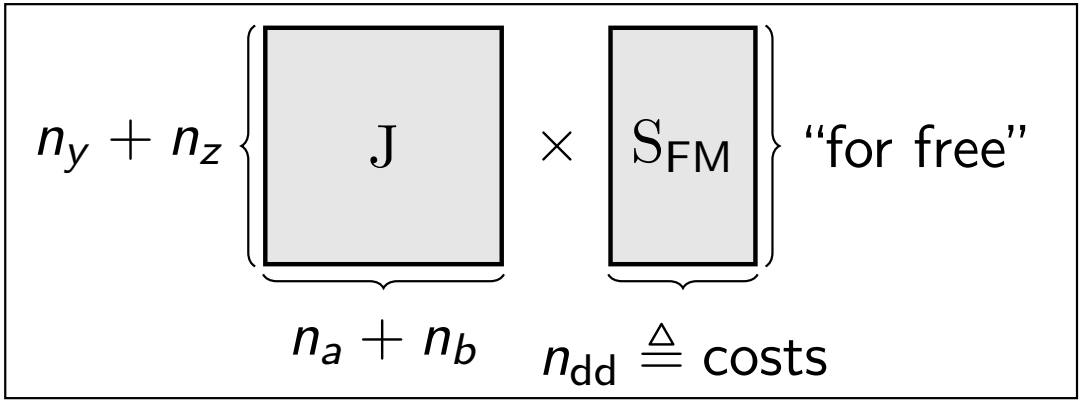
$$= \left(\begin{array}{ccc|ccc} \frac{\partial y_1}{\partial a_1} & \cdots & \frac{\partial y_1}{\partial a_{n_a}} & \frac{\partial y_1}{\partial b_1} & \cdots & \frac{\partial y_1}{\partial b_{n_b}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{n_y}}{\partial a_1} & \cdots & \frac{\partial y_{n_y}}{\partial a_{n_a}} & \frac{\partial y_{n_y}}{\partial b_1} & \cdots & \frac{\partial y_{n_y}}{\partial b_{n_b}} \\ \hline \frac{\partial z_1}{\partial a_1} & \cdots & \frac{\partial z_1}{\partial a_{n_a}} & \frac{\partial z_1}{\partial b_1} & \cdots & \frac{\partial z_1}{\partial b_{n_b}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n_y}}{\partial a_1} & \cdots & \frac{\partial z_{n_y}}{\partial a_{n_a}} & \frac{\partial z_{n_y}}{\partial b_1} & \cdots & \frac{\partial z_{n_y}}{\partial b_{n_b}} \end{array}\right) \in \mathbb{C}^{(n_y+n_z)\times(n_a+n_b)}$$

- a_i is a(i), i-th component of multidimensional array a
- n_a is number of components in a

AD in a nutshell

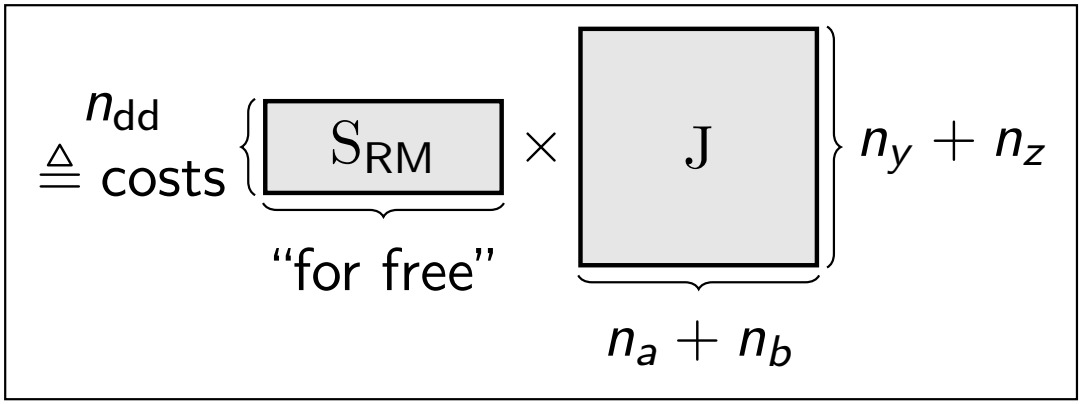
Forward Mode (FM)

- Computes $\mathrm{J} \cdot \mathrm{S}_{\mathrm{FM}}$, with *seed matrix* $\mathrm{S}_{\mathrm{FM}} \in \mathbb{C}^{(n_a+n_b)\times n_{\mathrm{dd}}}$
  - n_{dd}: Number of derivative directions
- Time: $T_\partial / T_f < c \cdot n_{\mathrm{dd}}$
- Memory: $M_\partial / M_f < c \cdot n_{\mathrm{dd}}$



Reverse Mode (FM)

- Computes $\mathrm{S}_{\mathrm{RM}} \cdot \mathrm{J}$, $\mathrm{S}_{\mathrm{RM}} \in \mathbb{C}^{n_{\mathrm{dd}}\times(n_y+n_z)}$
- Time: $T_\partial / T_f < c \cdot n_{\mathrm{dd}}$
- Memory: $M_\partial \approx T_f$, without *checkpointing*



Forward mode in ADiMat

There are two alternative FM implementations in ADiMat

- Some more on that later...

Example (FM with ADiMat)

```
[J y z] = admDiffFor(@f, S, a, b)
[J y z] = admDiffVFor(@f, S, a, b)
```

- S: Seed matrix S_{FM}
- J: Jacobian matrix product $\mathrm{J} \cdot \mathrm{S}_{\mathrm{FM}}$
- S = 1: Shortcut for $\mathrm{S}_{\mathrm{FM}} = \mathrm{I}_{n_a+n_b}$, conforming identity matrix
  - Compute full Jacobian J
- y, z: Function results returned by AD process

Reverse mode in ADiMat

Example (RM with ADiMat)

[J y z] = admDiffRev(@f, S, a, b)

- S: Seed matrix S_{RM}
- J: Jacobian matrix product $\mathrm{S}_{\mathrm{RM}} \cdot \mathrm{J}$

- RM evaluates derivatives in reverse direction of program flow
  - Chain rule of differentiation is associative
  - Have to store every single variable value, plus program flow
  - Can use *recomputation* to do better
- RM can be used to efficiently compute gradients
  - Costs do not depend on number of input parameters
  - RM computes *discrete adjoints* ("first discretize, then differentiate")
- Data storage options in ADiMat:
  - Keep data in memory
  - Write data to file (asynchronously)

Options

Example (Passing options to ADiMat)

opts = admOptions(name1, value1, name2, value2, ...)

- Create options structure

J = admDiffFor(@f, S, a, b, opts)

- Pass options structure as last argument
  - Works with any number of function arguments

Example (Specifying *independent* and *dependent* variables)

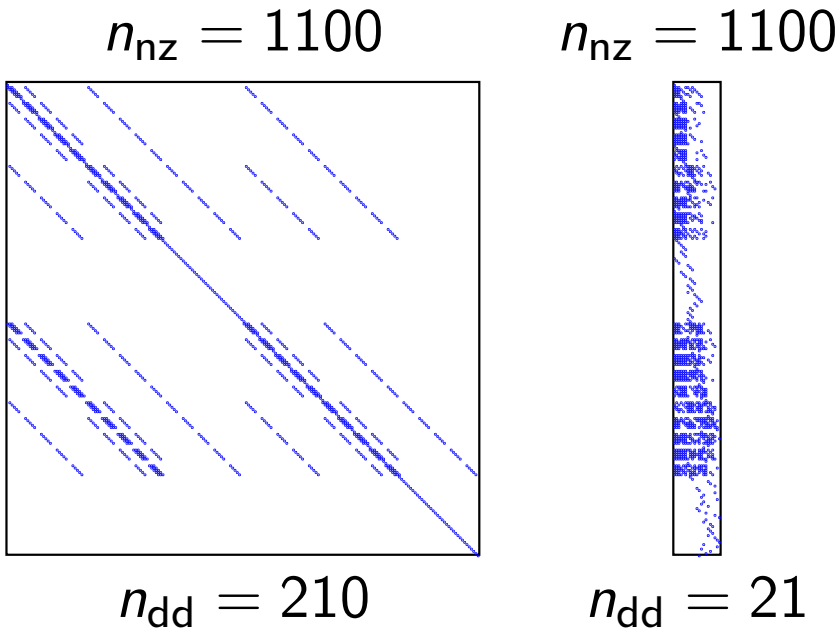
J = admDiffFor(@f, S, a, b, admOptions('i', 1, 'd', [1, 2]))

- J: left "half" of Jacobian, i.e. $\frac{\partial(y,z)}{\partial a} \cdot S_{FM}$
- Independent variables: first parameter a
- Dependent variables: both output parameters y and z
- 'i', 'd' are shortcuts for fields independents, dependents
- b is treated as a constant parameter

Compressed Jacobian computation

Often Jacobian J is sparse

- *Sparsity exploitation* can reduce the required n_{dd}
- Non-zero (NZ) pattern P of J (a bit matrix) must be known



Example (Compressed Jacobian computation with ADiMat)

```
opts = admOptions('jac_nzpattern', P)
J = admDiffFor(@f, @cpr, a, b, opts)
```

- P: Non-zero pattern P
- J: Full Jacobian J returned as sparse matrix
- cpr: "Compression" heuristic function (Curtis-Powell-Reed)

Alternatives to AD in ADiMat

ADiMat provides two non-AD methods to compute derivatives

Example (Finite difference (FD) method)

```
[JFD y z] = admDiffFD(@f, S, a, b)
```

Example (Complex variable method [Lyness and Moler 1967])

```
[JCV y z] = admDiffComplex(@f, S, a, b)
```

- Can use the same options shown before, including sparsity exploitation
- Will skip details in the talk ▸ Skip details

Finite difference method

Example (Finite difference (FD) method)

[JFD y z] = admDiffFD(@f, S, a, b)

- JFD: product $\mathrm{J} \cdot \mathrm{S}_{\mathrm{FM}}$, as in FM of AD
- Uses central FD by default

$$\frac{\mathrm{d}\, f}{\mathrm{d}\, x} \approx \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h}$$

- Forward and backward FD via option

- Accuracy half the machine precision, with right h (option)
- Not useful near discontinuities

Complex variable method

Example (Complex variable method [Lyness and Moler 1967])

[JCV y z] = admDiffComplex(@f, S, a, b)

- JCV: product $\mathrm{J} \cdot \mathrm{S}_{\mathrm{FM}}$, as in FM of AD
- Uses

$$\frac{\mathrm{d}\, f}{\mathrm{d}\, x} \approx \frac{\Im[f(x + \mathrm{i}\epsilon)]}{\epsilon}$$

- Accuracy as with AD, full machine precision, any tiny ϵ will do
- Function f must be *real analytic*
- Time: $T_\partial / T_f < c \cdot n_{\mathrm{dd}}$
- Potential problems when program flow changes

Usage example

Example (Function **f**)

```
function [y z] = f(a, b)
  y = a + sqrt(b);
  z = sqrt(a) .* b;
```

Example (Run **f**)

```
>> a = [9 16];
>> b = [25 36];
>> [y z] = f(a, b)
y = 14    22
z = 75   144
```

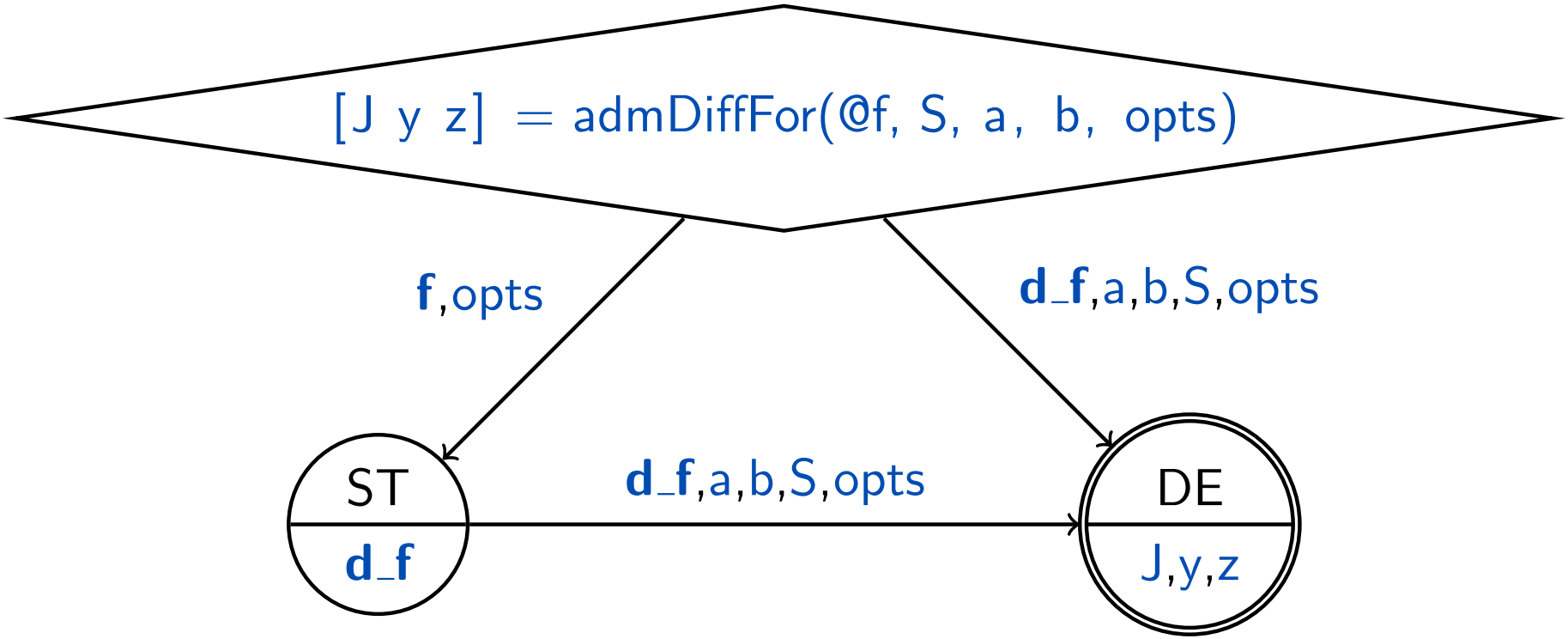
Example (AD of **f**)

```
>> J = admDiffFor(@f, 1, a, b)
J = 1.0000         0    0.1000         0
         0    1.0000         0    0.0833
    4.1667         0    3.0000         0
         0    4.5000         0    4.0000
```

How ADiMat works

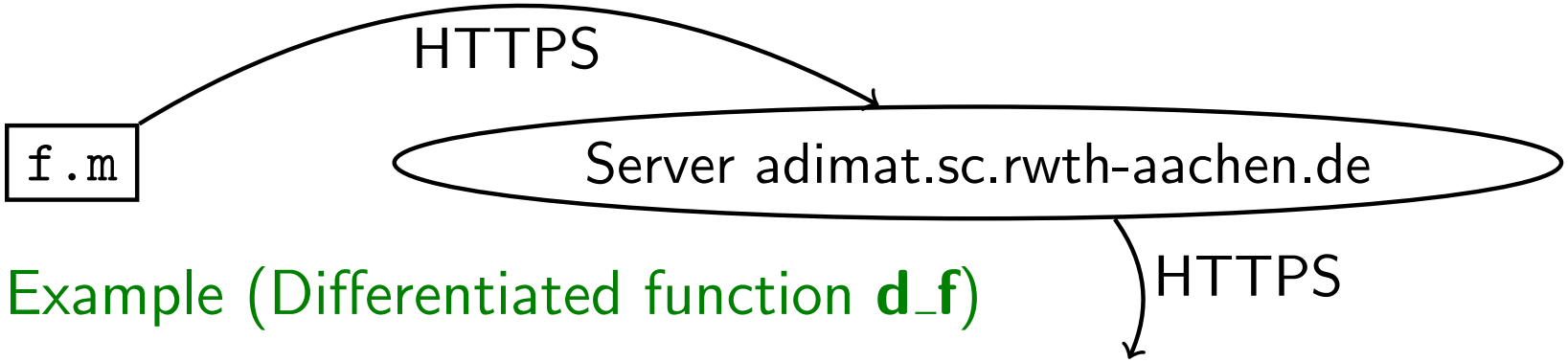
admDiffFor(@f, S, a, b, opts) works in two steps

- Source transformation (ST)
  - Differentiate the source code, produce function **d_f**
  - Only done if necessary
- Derivative evaluation (DE)
  - Actually compute the numbers, calling **d_f**



Source Transformation

Source code is differentiated by transformation server



Example (Differentiated function **d_f**)

```
function [d_y y d_z z] = d_f(d_a, a, d_b, b)
    [d_tmpca1 tmpca1] = diff_sqrt(d_b, b);
    d_y = opdiff_sum(d_a, d_tmpca1);
    y = a + tmpca1;
    [d_tmpca1 tmpca1] = diff_sqrt(d_a, a);
    d_z = opdiff_emult(d_tmpca1, tmpca1, d_b, b);
    z = tmpca1 .* b;
end
```

- Aside: This is actually the code produced by admDiffVFor
- Redone, when source code or options are modified

Derivative Evaluation

Run differentiated code **d_f** to evaluate derivatives

- Derivative arguments d_a, d_b are created from a, b and S
- Extract derivatives from outputs d_y, d_z and create J

Scalar mode

- **d_f** is called n_{dd} times
- Derivative variable d_a is double array with same shape as a
  - Fast execution, but redundant computations when $n_{\text{dd}} > 1$

Vector mode

- Single call of **d_f**
- Derivative variable d_a is *derivative class* object
  - Object internally holds $n_{\text{dd}} \cdot n_a$ derivative values
  - Dispatching of overloaded operators at runtime, hence slow
- admDiffVFor: d_a is double array with $n_{\text{dd}} \cdot n_a$ items
  - Overloaded operators replaced by function calls
  - Only possibility for vector mode in Octave

Why source transformation

The source transformation step in ADiMat is difficult, but:

- ST can analyze the code
  - *Activity analysis*
    - Do not differentiate code that does not influence result
    - Derivative code becomes shorter
    - Avoid unnecessary derivative computations
  - *Optimizations*
    - Can apply optimizations to code before running it
    - Before and after differentiation
  - May consider more than a single expression/statement
  - We lack type information, though
- ST can create code that suits the needs of AD
  - E.g. replace calls of overloaded operators by function calls
  - Shift work from runtime to transformation and "compile" (to intermediate code) time
- We have a flexible ST framework for Matlab that might be useful for other transformation tasks (ideas?)

Performance

Factor T_∂ / T_f, For: admDiffFor, /D: double deriv., /O: deriv. are objects

Non-vectorized code: Two-phase flow in porous media

Henrik Büsing, c.f. ICCE talk "Using exact Jacobians in an implicit ..."

n_{dd}	For/D	For/O	VFor	Rev/D	Rev/O	FD	Complex
1	2.15	17.2	3.15	18.6	58.3	2.03	1.02
10	21.2	26.4	3.19	185	67.9	20.4	10.2
100	211	119	3.28	1853	213	203	102

$T_f = 13.3$s, f: 761 LOC, 372 statements, 16 functions

Vectorized code: 1D Burgers PDE Solver

Research code by Micheal Herty

n_{dd}	For/D	For/O	VFor	Rev/D	Rev/O	FD	Complex
1	3.43	42.4	9.87	32.7	129	2.03	–
10	32.4	97.3	32.5	332	235	20.1	–
100	324	674	304	3350	1348	201	–

$T_f = 1.64$s, f: 169 LOC, 57 statements, 6 functions, admDiffComplex is not applicable

Linear increase

Amortization by deriv. classes

Amortization by vectorized deriv. code

Conclusion

ADiMat, AD tool for Matlab and Octave

- Is easy to use
- Provides sophisticated features
  - Two alternative implementations of the forward mode of AD
  - Reverse mode of AD
  - Support for compressed Jacobian computation
- Try it out
  - `http://www.sc.rwth-aachen.de/adimat`
- Please do report bugs
  - ADiMat is far from complete
  - We need feedback to enhance ADiMat to suit your needs