The New User Interface of ADiMat and How to Use it with DAE Solvers in Matlab and Octave

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

[×] Fachgebiet Scientific Computing
TU Darmstadt

[*] Institute for Scientific Computing
RWTH Aachen University

Twelfth European Workshop on Automatic Differentiation
Humboldt-Universität zu Berlin
December 9, 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

RWTHAACHEN UNIVERSITY

Overview

ADiMat

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

New in ADiMat

- Reverse mode
- Second, alternative forward mode implementation
- Server for source transformation
- **High-level user interface**

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

Forward mode in ADiMat

There are two alternative FM implementations in ADiMat

- admDiffFor: first order and second order (experimental)
- admDiffVFor: first order
- Some more on the differences 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 $\mathrm{J} \cdot \mathrm{S}_{\mathsf{FM}}$
- S $=$ 1: Shortcut for $\mathrm{S}_{\mathsf{FM}} = \mathrm{I}_{n_a+n_b}$, 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 $S_{\mathsf{RM}} \cdot J$

- Basic support for recomputation (on the function call level)
  - Directive <recompute>g</recompute>
  - Recompute following call to function g
- Several stack implementations
  - 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)

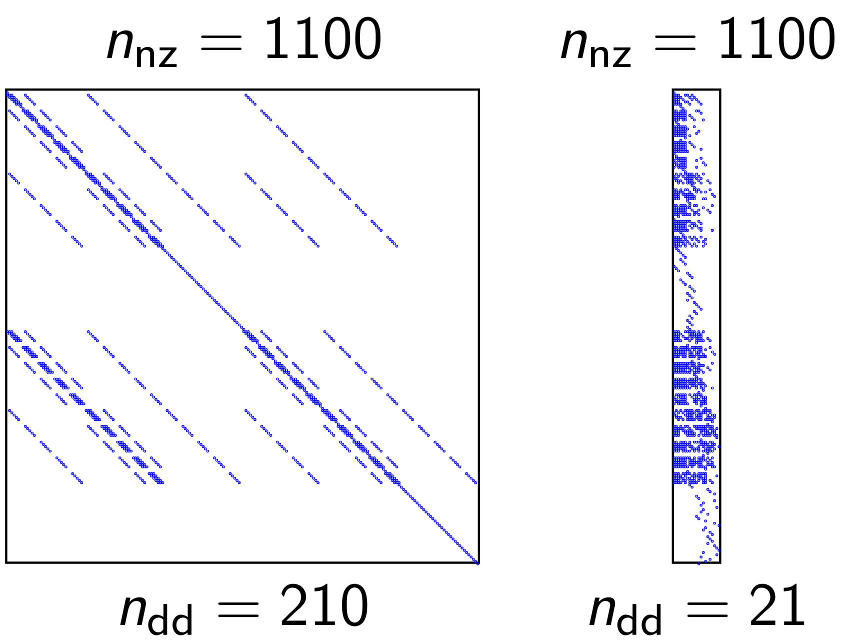
J = admDiffFor(@f, S, a, b, admOptions('i', 1, 'd', [1, 2]))

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

Compressed Jacobian computation

Often Jacobian J is sparse

- *Sparsity exploitation* can reduce the number of derivative directions n_{dd}
- Non-zero (NZ) pattern P of J must be known



Example (Compressed Jacobian computation with ADiMat)

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

- P: Non-zero pattern P
- J: Full Jacobian J returned as sparse matrix
- cpr: Curtis-Powell-Reed heuristic

Alternatives to AD in ADiMat

ADiMat provides two non-AD methods to compute derivatives

- Can use the same options shown before, including sparsity exploitation

Example (Finite difference (FD) method)

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

- Central, forward, and backward, up to fourth order derivatives, also higher accuracy order stencils

Example (Complex variable method [LYNESS AND MOLER 1967])

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

Example (Nested application of ADiMat)

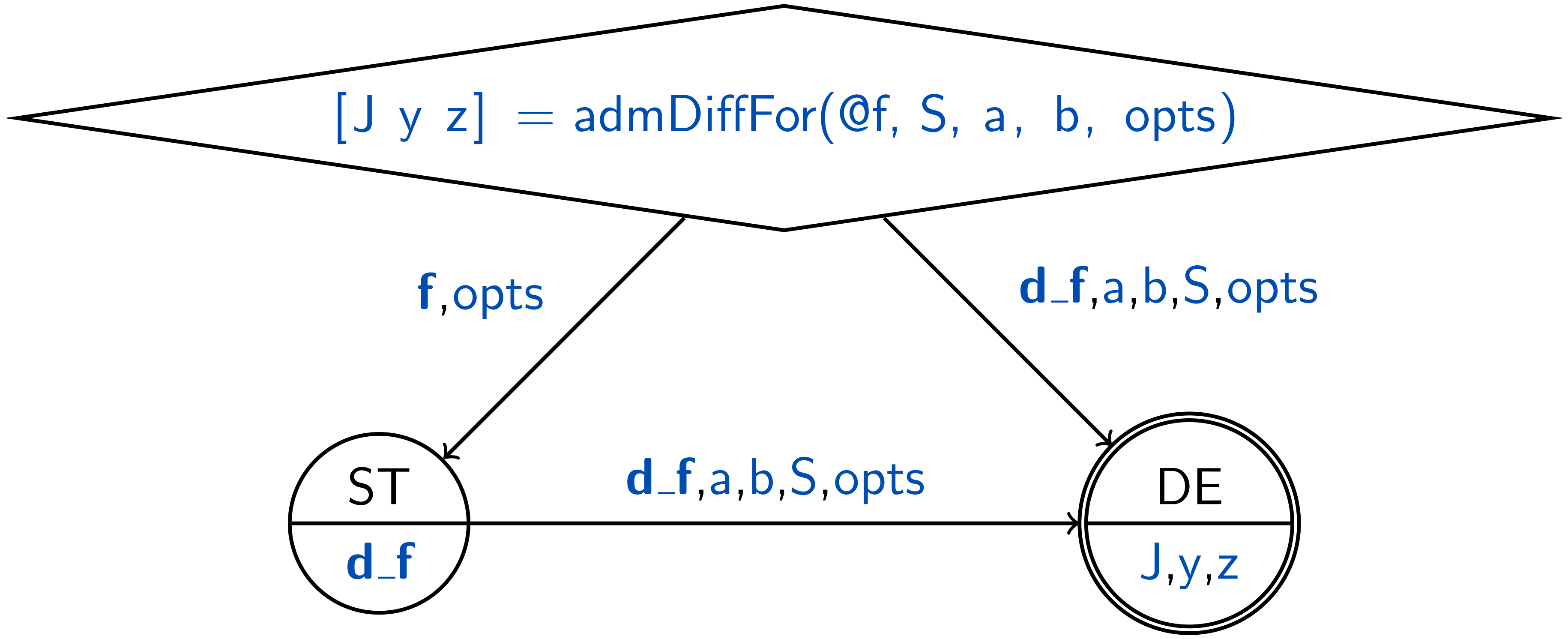
```
Hessian = admDiffComplex(@admDiffFVor, 1, @f, 1, x, y, ...
admOptions('i', 3:4))
```

- AD over AD will in general not work

How ADiMat works

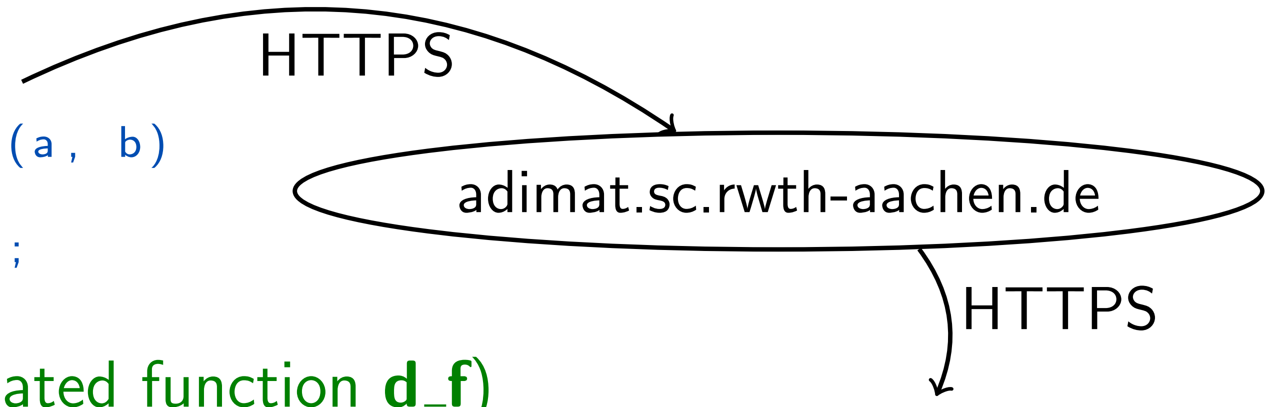
admDiffFor(@f, S, a, b, opts) works in two steps

- 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



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

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
```

- 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 $n_{\mathrm{dd}} > 1$

Vector mode

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

Performance

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

Non-vectorized code: Multiphase flow in porous media

[Büsing et al. 2011]

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, 75 function calls

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, 27 function calls, admDiffComplex is not applicable

🟦	Linear increase
🟧	Amortization by deriv. classes
🟩	Amortization by vectorized deriv. code

Matlab: ode15s

Matlab has ODE solvers, e.g. ode15s, which can also solve index 1 DAEs [Shampine, Reichelt, and Kierzenka 1999]

- Solves $\mathrm{M}(t, \mathbf{y})\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$
- This is a DAE when *mass matrix* $\mathrm{M}(t, \mathbf{y})$ is singular.
- Inputs required: function $\mathbf{f}$, initial $\mathbf{y}_0$, start and end times t_0 and t_f, mass matrix or function
- Consistent initial $\dot{\mathbf{y}}_0$ automatically computed (or via option)
- Use AD: User can provide a handle to function that computes the Jacobian of $\mathbf{f}$ w.r.t. $\mathbf{y}$

Octave: DASSL, DASPK and ODE package

Octave provides interfaces to the DAE solvers DASSL [Petzold 1982] and DASPK [Brown, Hindmarsh, and Petzold 1999]

- Solve equation $0 = \mathbf{f}(\mathbf{y}, \dot{\mathbf{y}}, t)$
- Inputs required: $\mathbf{f}$, $\mathbf{y}_0$, $\dot{\mathbf{y}}_0$, vector of time points $\mathbf{t}$
- Option to compute consistent initial conditions
- Use AD: User can provide a handle to function that computes the Jacobian $\mathrm{J}_c = \frac{\partial \mathbf{f}}{\partial \mathbf{y}} + c\frac{\partial \mathbf{f}}{\partial \dot{\mathbf{y}}}$ for a given c

There is also an ODE package which provides drop-in replacements for Matlab's ODE solvers

Example: Binary Distillation Column

Model of 42 differential and 83 algebraic equations [Diehl 2001]

- Adapted version found in the Nonlinear Model Library from hedengred.net (Distillation 4)
- Coded as `function xdot = distill (t,y,mode)`

Distillation example with Matlab and AD

The call to ode15s looks like this:

```
opts=odeset('Mass', distill(t,y_0,'mass'),...
            'MassSingular', 'yes',...
            'MStateDependence', 'none', ...
            'JPattern', distill(t,y_0,'jpat'));
[t,y] = ode15s(@distill, [0 tf], y_0, opts);
```

Use AD: set opts.Jacobian to a function that computes $J = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}$

```
adopts = admOptions('i', 2);
adopts.JPattern = distill(t, y_0, 'jpat');
opts.Jacobian = @(t, y) ...
    admDiffVFor(@distill,@cpr,t,y,'',adopts);
```

- Pass the non-zero pattern of J to ADiMat instead of ode15s

Distillation example with Octave and DASSL

First, a wrapper function to adapt distill to dassl interface

```
function res = dassl_distill(y,ydot,t)
  global M
  res = M*ydot - distill(t,y,'');
```

Use AD: give DASSL a second function handle, for J_c:

```
  global JPat
  adopts = admOptions('i', 1:2);
  adopts.JPattern = JPat;
  adopts.coloringFunction = 'cpr';

  seed = [speye(numel(y))
              c .* speye(numel(y))];
  jac = admDiffVFor(@dassl_distill, seed, y, ...
      ydot, t, adopts);
```

$$S = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \\ c & & \\ & \ddots & \\ & & c \end{pmatrix}$$

- This way $J_c = J \cdot S$ will be compressed

Binary Distillation Column results

- J compressed to 68, J_c to 86 columns
- n_F: calls to distill , n_∂: calls to derivative function
- t: solve time

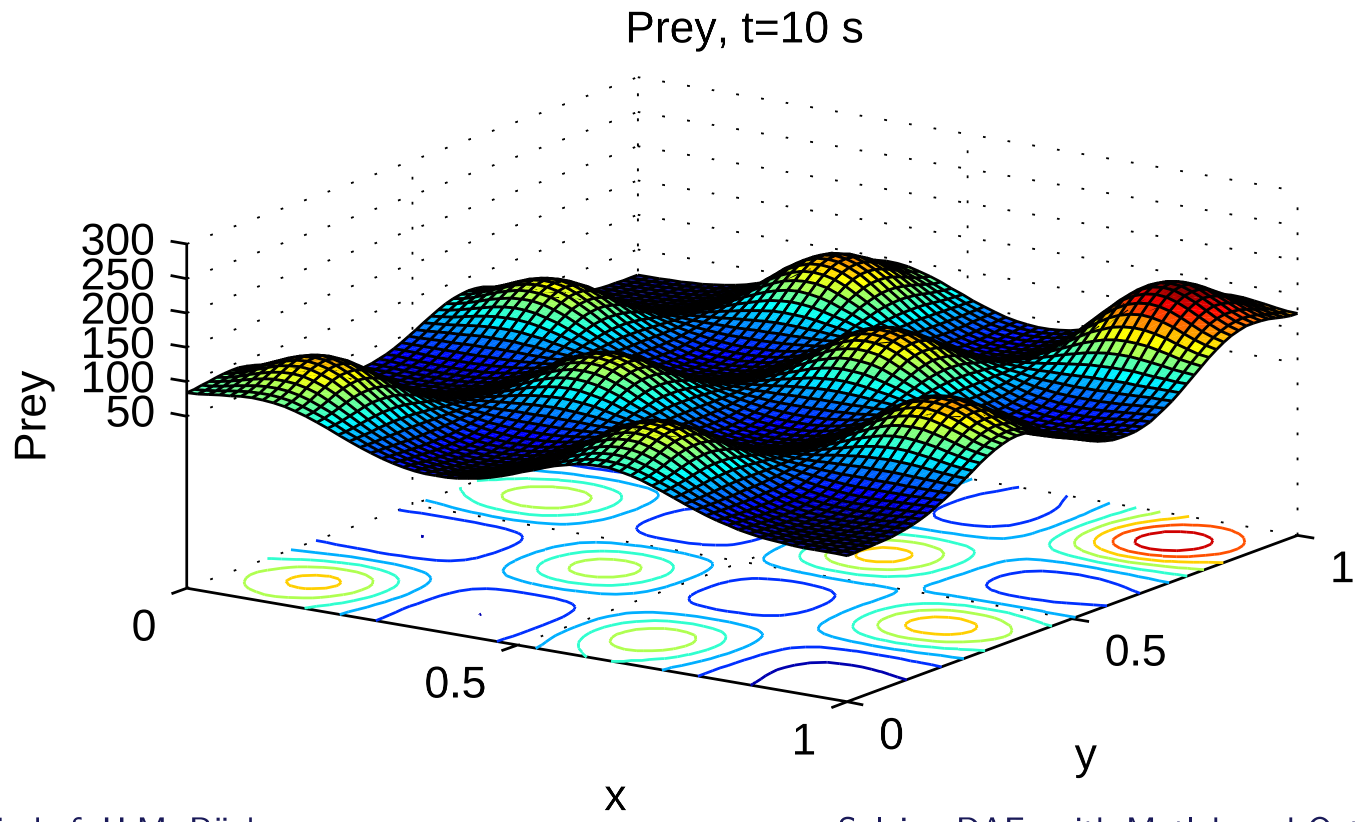
	Derivative	n_F	n_∂	t/s
Matlab	–	109	–	0.35
	FD	41	1	0.60
	VFor	41	1	0.88
Octave	–	1791	–	40.8
	FD	41	14	59.7
	VFor	41	14	28.6

distill : 442 LOC, 148 statements, 1 function, 17 function calls
Lenovo T420s: Core i5-2520M @2.5 GHz, Linux 3.0.0, Matlab R2011a, Octave 3.2.4, and ADiMat 0.5.6-3150

Example: Food web with DASPK and Octave

From DASPK paper [Brown, Hindmarsh, and Petzold 1999]

- One prey, one predator species on $(L + 2) \times (L + 2)$ grid
- No preconditioning
- F: Loops over grid, matrix operations for species interaction
- Jacobian J_c can be compressed to 11 columns



Example: Food web with DASPK and Octave

- t_F: time in F, t_∂: time in derivative function

Derivative	L	n_F	n_∂	t/s	t_F/s	t_∂/s
FD	20	741	53	126	38.1	75.8
VFor	20	921	49	133	47.3	73.7
FD	40	1111	57	850	207	338
VFor	40	995	56	848	189	340
FD	60	917	55	3556	375	828
VFor	60	925	51	3321	371	760

F: 58 LOC, 35 statements, 2 functions, 16 function calls
Lenovo T420s

- For $L = 20$ and $L = 60$ FD results in less calls to F
- In all cases less derivative evaluations with AD

Conclusion

New user interface for ADiMat

- Makes sophisticated AD features easily accessible
  - 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
  - Tell us which builtin functions you miss

Solving DAEs with AD

- AD runtime is comparable to numerical methods

- Lawrence F. Shampine, Mark W. Reichelt, and Jacek A. Kierzenka
  Solving Index 1 DAEs in MATLAB and Simulink
  SIAM Review, Vol 41, 1999

- Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
  Using Krylov Methods in the Solution of Large-Scale Differential Algebraic Systems
  SIAM Journal on Scientific Computing, Volume 15 (6), 1994