Automatic Differentiation (AD) in Octave

December 2007

Thomas Kasper thomaskasper@gmx.net

Table of Contents

1 Concept 1

2 Octave AD-Extension 3
2.1 License Information and Disclaimer 3
2.2 Prerequisites 3
2.3 New Features 3
2.4 Download and Installation 3
2.5 Testsuite 3

3 Using AD in Octave 4
3.1 User Interface and Class Gradient 4
3.2 Sparse Storage Mode 5
3.3 Complex-valued Domains 6

4 Limitations 7

5 Index 8
5.1 Functions by Category 8
5.1.1 Overloaded Operators 8
5.1.2 Utility Functions 8
5.1.3 Overloaded Functions 8
5.2 Functions Alphabetically 9
5.2.1 __ga__ 9
5.2.2 gradabs 10
5.2.3 gradacos 10
5.2.4 gradacosh 10
5.2.5 gradasin 10
5.2.6 gradasinh 10
5.2.7 gradatan 10
5.2.8 gradatanh 10
5.2.9 gradconj 11
5.2.10 gradcos 11
5.2.11 gradcosh 11
5.2.12 gradcot 11
5.2.13 gradcumprod 11
5.2.14 gradcumsum 11
5.2.15 gradexp 11
5.2.16 gradfind 12
5.2.17 gradimag 12
5.2.18 gradinit 12
5.2.19 gradlog 12
5.2.20 gradlog10 13
5.2.21 gradprod 13
5.2.22 gradreal 13
5.2.23 gradsin 13
5.2.24 gradsinh 13
5.2.25 gradsqrt 13
5.2.26 gradsum 13

5.2.27 gradtan 14
5.2.28 gradtanh 14
5.2.29 isgradient 14
5.2.30 use_sparse_jacobians 14

1 Concept

A wide range of numerical problems can be efficiently solved using derivatives in one way or the other. While from a strictly mathematical point of view the derivative is a well-defined object, its computation is anything but trivial.

A classical approach is finite differences. Let f be differentiable at some point x. Clearly, for a certain h small enough

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

The problem with finite differences is twofold. One issue – probably the more important one – is accuracy. Being necessarily an approximation, its quality largely depends on a sensible choice of h. Large values, obviously, make for a poor estimate of the actual derivative; small ones, on the other hand, are prone to computational artefacts such as cancellation. While there are strategies to cope with this dilemma they normally do so – and that is the second concern – at the expense of additional evaluations of your function. Central differences, for instance, requires a total of $2n$ evaluations, where n is the dimension of the domain space. If, to make matters worse, the computation is carried out within an iterative loop, you forfeit a good deal of the algorithmic efficiency that may have motivated the use of derivatives in the first place.

The concept of Automatic Differentiation is altogether different from the above. Unlike finite differences, it provides a means to *analytically* compute the derivative of a function at a given inner point of its domain. A straightforward approach – the one implemented by the extension – is to introduce a new data-type, often referred to in the literature as differential number or gradient. Basically, this is a compound of the value itself and the associated derivative. The fundamental idea is to define the common operators on the set of differential numbers according to the well-know rules of elementary calculus. Hence, multiplication becomes

$$* : \begin{pmatrix} x \\ \dot{x} \end{pmatrix}, \begin{pmatrix} y \\ \dot{y} \end{pmatrix} \mapsto \begin{pmatrix} xy \\ \dot{x}y + x\dot{y} \end{pmatrix}$$

Likewise, the addition of two differential numbers would have to be

$$+ : \begin{pmatrix} x \\ \dot{x} \end{pmatrix}, \begin{pmatrix} y \\ \dot{y} \end{pmatrix} \mapsto \begin{pmatrix} x+y \\ \dot{x}+\dot{y} \end{pmatrix}$$

and so on for the remaining cases. Now consider that a function, in practice, is implemented by a computer program, which in turn is made up of discrete instructions. Control flow may bifurcate depending on switch-statements, but, no matter how complex its structure, eventually it is a sequence of elementary operations. By overloading all or most of these in the above described manner you create an ideally complete algebra of differential numbers, where

$$f(\begin{pmatrix} x \\ 1 \end{pmatrix}) = \begin{pmatrix} f(x) \\ D_x f(x) \end{pmatrix}$$

Thus, all you have to do is create an initial gradient and pass it on to the computer program, which will then construct the derivative $D_x f(x)$ along with the output $f(x)$ simultaneously.

With AD you elude the two principal drawbacks of numerical differentiation outlined previously. First of all, it is more reliable in that you no longer have to worry about approximation errors. Although accuracy, of course, is ultimately bounded by machine precision, it *can* make a difference if you get 16 instead of, say, 10 correct figures. The other advantage is maybe less apparent and of minor relevance to most users. However, in cases where cost is a non-negligible factor, it may be not indifferent that the number of evaluations does not scale with the problem size. Whether your function depends on 5 or, say, 500 variables, one pass will do either way. Due to the computational overhead implied by every single operation this comes at the price of a slowed-down execution during that single pass. We shall rely on vectorized code for a good performance here.

Today Automatic Differentiation is a widely used technology in both industry and academic sience. Implementations cover almost every language or application commonly used for numerical computations, the most popular being Fortran, C, and MATLAB. For further discussion of the topic and relevant links see, for instance, `http://www-sop.inria.fr/tropics/ad/whatisad.html`, the INRIA site dedicated to AD.

2 Octave AD-Extension

2.1 License Information and Disclaimer

2.2 Prerequisites

- GNU Octave 3.0

The maintainer will do his best to retain upward compatibility, though at the current pace of releases this seems an audacious promise.

2.3 New Features

- Easy installation thanks to Octave's new packaging system
- Enhanced testsuite and improved documentation
- Support for n-d arrays allowing gradients of arbitrary dimensions
- Handling of minimum norm solutions for over- and underdetermined linear systems
- Implementation of gradients by (complex) sparse matrix operations

2.4 Download and Installation

The current release (‘ad-0.9.19.tar.gz’, as of this writing) is available for download as a gzipped archive at http://home.cs.tum.edu/~kasper/ad/index.html#download.

Install by typing `pkg install ad-x.x.x.tar.gz` at the octave prompt. For advanced options and general information about the new package manager invoke the online documentation with `help pkg` or consult the Octave-Forge website at http://octave.sf.net.

2.5 Testsuite

It is recommended that you run the integrated testscript after installation to make sure the entire AD-functionality is available to you. Be prepared that it takes a couple of seconds before the results are reported.

```
octave:1> fid = fopen ("ad.log", "wt");
octave:2> __ga__ (fid)
PASSES 289 out of 289 tests
```

Note that the vast majority of tests are statistical and involve randomly generated data. This may occasionally result in noise exceeding the specified tolerance. Do let me know if any of the tests repeatedly fail on your system. Before you report a bug, however, please check the log does not already classify it as a known issue.

3 Using AD in Octave

3.1 User Interface and Class Gradient

The function `D` provides an intuitive interface to AD-functionality while hiding the ugly and potentially confusing details from the user. Let us have a look at its signature:

```
octave:3> help D
-- Function File: [Y, J] = D (F, X, VARARGIN)
   Evaluate F for a given input X and compute the jacobian, such that

                d
   J(i,j) = ----- Y(i) where Y = F (X, VARARGIN{:})
              dX(j)

   If X is complex, the above holds for the directional derivatives
   along the real axis

   Derivatives are computed analytically via Automatic Differentiation


See also: use_sparse_jacobians.
```

Note that *F* must be a function handle, not a character string. A simple use-case scenario could look as follows. Suppose you want to find a root of the non-linear function

```
octave:4> function y = foo (x)
> y(1) = 100 * (x(1) - x(2)^2)^2;
> y(2) = (1 - x(1))^2;
> y = y(:) / 2;
> endfunction
```

One way to go about this is to iteratively refine a random initial guess by Newton-steps

```
octave:5> x = rand (2, 1);
octave:6> for k = 1:30, [y, J] = D (@foo, x); x = x - J \ y; endfor
octave:7> x, res = norm (foo (x))
x =
    1.0000
    1.0000

res = 1.1696e-009
```

Note that `D` is a mere convenience function which wraps up the steps outlined in the introductory section. Thus, `[y, J] = D (@F, x)` essentially is a shortcut for

```
result = F (gradinit (x));
y = result.x;
J = result.J;
```

With `gradinit` you specify the independent variables to differentiate with respect to along with their initial values. In the resulting gradient you find the argument `x` augmented by the jacobian which evaluates to the identity matrix of size `numel (x)`

```
octave:5> g = gradinit ([-1; 2])
g =

value =
  -1
   2

(partial) derivative(s) =

   1   0
   0   1
```

Gradients represent a class of their own and are listed as such by the interpreter. Use `isgradient` for type-checking:

```
octave:8> who -long g
*** local user variables:

  Prot Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
   rwd g           2x1                         48  gradient

Total is 2 elements using 48 bytes

octave:9> isgradient (g)
ans = 1
```

Each of the two members (value and partial derivatives) can be accessed by suffixing the variable with ".x" and ".J" respectively. (For obvious reasons, however, they should only be read out and never be assigned to directly.) Analytical expressions in one or more variables of type gradient automatically evaluate to gradients:

```
octave:10> foo (g)
ans =

value =

   1250
      2

(partial) derivative(s) =

   -500   2000
     -2     -0
```

At any time their members satisfy `g.J(i,j) = d/dx(j)[g.x(i)]`, with `x(j)` the variables previously passed to `gradinit`. Beware that this relation is independent of shape and extends to arrays of arbitrary dimension. Thus, operations which preserve the linear order of elements (like reshape, for instance, or a transposal on column-vectors) do not alter the jacobian.

3.2 Sparse Storage Mode

You may ask that partial derivatives be stored as a sparse matrix by invoking `use_sparse_jacobians` with a nonzero value. As with increasing dimension jacobians tend to be sparsely occupied, doing so may eventually pay off in terms of both memory consumption and speed.

```
octave:11> use_sparse_jacobians (1);
octave:12> [y, J] = D (@cumprod, reshape (1:9, 3, 3), 2)

y =
      1    4   28
      2   10   80
      3   18  162

J =

Compressed Column Sparse (rows = 9, cols = 9, nnz = 18)

  (1, 1) -> 1
  (4, 1) -> 4
  (7, 1) -> 28
  (2, 2) -> 1
  (5, 2) -> 5
  (8, 2) -> 40
  (3, 3) -> 1
  (6, 3) -> 6
  (9, 3) -> 54
  (4, 4) -> 1
  (7, 4) -> 7
  (5, 5) -> 2
  (8, 5) -> 16
  (6, 6) -> 3
  (9, 6) -> 27
  (7, 7) -> 4
  (8, 8) -> 10
  (9, 9) -> 18
```

This is a best effort service, however, and there is no guarantee as to whether the returned jacobian will in fact be sparse. It certainly helps when the involved operands are:

```
octave:13> A = rand (6); b = rand (6);
octave:14> x = sparse (A) \ gradinit (b);
octave:15> spy (x.J, 0.5), issparse (x.J)
ans = 1
```

3.3 Complex-valued Domains

Although primarily designing for functions with a real domain, the author does not think fit to impose any restriction here. Users should bear in mind though, when working with complex input, that what they get is the directional derivative along the real axis. It then may – or may not, for that matter – coincide with *the* derivative, depending on whether the function is locally holomorphic or not.

```
octave:16> [z, dz] = D (@abs, 1 + i)
z = 1.4142
dz = 0.70711
```

4 Limitations

Beware that operator overloading is frail when it comes to interfacing with low-level routines. If you are in the habit of writing good portions of code in C++ or Fortran as DLD-functions – and there may well be good a reasons for it –, you will definitely run into trouble. The same caveat applies even to some functions of the Octave core API, in which case you should incur an error message like the one below:

```
octave:17> gamma (gradinit (4))
error: AD-rule unknown or function not overloaded
```

One might consider adding rules as the need arises. On the other hand, balancing the benefit against the extra effort, it is often more reasonable to fall back on numerical differentiation for less common operations and use `numgradient` instead. In any event, the algebra provided by the extension makes no claim for completeness and there certainly would be no point in trying.

5 Index

5.1 Functions by Category

5.1.1 Overloaded Operators

+	no restriction
-	no restriction
*	no restriction
/	operand 2 must have maximal rank
ldiv	operand 1 must have maximal rank
pow	both operands must be scalar or, if op1 is square, op2 must be a non-negative integer. This implies that in the latter case op2 cannot be a gradient, since int $Z = \emptyset$
.*	no restriction
./	no restriction
elpow	no restriction

5.1.2 Utility Functions

__ga__	Testscript for the gradient algebra implemented by the package AD
D	Evaluate F for a given input x and compute the jacobian
gradinit	Create a gradient with value x and derivative `eye`(`numel`(x))
isgradient	Return 1 if x is a gradient, otherwise return 0
use_sparse_jacobians	Query or set the storage mode for AD

5.1.3 Overloaded Functions

gradabs	overloads built-in mapper ‘abs’ for a gradient X
gradacos	overloads built-in mapper ‘acos’ for a gradient X
gradacosh	overloads built-in mapper ‘acosh’ for a gradient X
gradasin	overloads built-in mapper ‘asin’ for a gradient X
gradasinh	overloads built-in mapper ‘asinh’ for a gradient X
gradatan	overloads built-in mapper ‘atan’ for a gradient X
gradatanh	overloads built-in mapper ‘atanh’ for a gradient X
gradconj	overloads built-in mapper ‘conj’ for a gradient X
gradcos	overloads built-in mapper ‘cos’ for a gradient X
gradcosh	overloads built-in mapper ‘cosh’ for a gradient X
gradcot	overloads mapping function ‘cot’ for a gradient X

gradcumprod	overloads built-in function 'cumprod' for a gradient X
gradcumsum	overloads built-in function 'cumsum' for a gradient X
gradexp	overloads built-in mapper 'exp' for a gradient X
gradfind	overloads built-in function 'find' for a gradient X
gradimag	overloads built-in mapper 'imag' for a gradient X
gradlog	overloads built-in mapper 'log' for a gradient X
gradlog10	overloads built-in mapper 'log10' for a gradient X
gradprod	overloads built-in function 'prod' for a gradient X
gradreal	overloads built-in mapper 'real' for a gradient X
gradsin	overloads built-in mapper 'sin' for a gradient X
gradsinh	overloads built-in mapper 'sinh' for a gradient X
gradsqrt	overloads built-in mapper 'sqrt' for a gradient X
gradsum	overloads built-in function 'sum' for a gradient X
gradtan	overloads built-in mapper 'tan' for a gradient X
gradtanh	overloads built-in mapper 'tanh' for a gradient X

5.2 Functions Alphabetically

5.2.1 __ga__

__ga__ (*name*, *varargin*) Function File
__ga__ (*fid*) Function File

Testscript for the gradient algebra implemented by the package AD

If the first argument is a character string, assert functionality *name* complies with the specification. Otherwise run a set of predefined tests and report failures to the stream *fid* (defaulting to *stderr*)

Intended use is:

```
fid = fopen ("errors.log", "wt");
__ga__ (fid)
⇒ PASSES [#] out of [#] tests ([#] expected failures)
```

See also: test

[*y*, *J*] = **D** (*F*, *x*, *varargin*) Function File

Evaluate *F* for a given input *x* and compute the jacobian, such that

$$J_{i,j} = \frac{\partial y_i}{\partial x_j}, \qquad y = F(x, \texttt{varargin}\{:\})$$

If *x* is complex, the above holds for the directional derivatives along the real axis

Derivatives are computed analytically via Automatic Differentiation

See also: use_sparse_jacobians

5.2.2 gradabs

gradabs (*x*) Mapping Function

overloads built-in mapper `abs` for a gradient *x*

See also: abs

5.2.3 gradacos

gradacos (*x*) Mapping Function

overloads built-in mapper `acos` for a gradient *x*

See also: acos

5.2.4 gradacosh

gradacosh (*x*) Mapping Function

overloads built-in mapper `acosh` for a gradient *x*

See also: acosh

5.2.5 gradasin

gradasin (*x*) Mapping Function

overloads built-in mapper `asin` for a gradient *x*

See also: asin

5.2.6 gradasinh

gradasinh (*x*) Mapping Function

overloads built-in mapper `asinh` for a gradient *x*

See also: asinh

5.2.7 gradatan

gradatan (*x*) Mapping Function

overloads built-in mapper `atan` for a gradient *x*

See also: atan

5.2.8 gradatanh

gradatanh (*x*) Mapping Function

overloads built-in mapper `atanh` for a gradient *x*

See also: atanh

5.2.9 gradconj

gradconj (*x*) Mapping Function

overloads built-in mapper `conj` for a gradient *x*

See also: conj

5.2.10 gradcos

gradcos (*x*) Mapping Function

overloads built-in mapper `cos` for a gradient *x*

See also: cos

5.2.11 gradcosh

gradcosh (*x*) Mapping Function

overloads built-in mapper `cosh` for a gradient *x*

See also: cosh

5.2.12 gradcot

gradcot (*x*) Mapping Function

overloads mapping function `cot` for a gradient *x*

See also: cot

5.2.13 gradcumprod

y = **gradcumprod** (*x*) Function File

y = **gradcumprod** (*x*, *dim*) Function File

overloads built-in function `cumprod` for a gradient *x*

See also: cumprod

5.2.14 gradcumsum

y = **gradcumsum** (*x*) Function File

y = **gradcumsum** (*x*, *dim*) Function File

overloads built-in function `cumsum` for a gradient *x*

See also: cumsum

5.2.15 gradexp

gradexp (*x*) Mapping Function

overloads built-in mapper `exp` for a gradient *x*

See also: exp

5.2.16 gradfind

gradfind (x) Function File

overloads built-in function `find` for a gradient x

See also: find

5.2.17 gradimag

gradimag (x) Mapping Function

overloads built-in mapper `imag` for a gradient x

See also: imag

5.2.18 gradinit

g = **gradinit** (x) Loadable Function

Create a gradient with value x and derivative `eye`(`numel`(x))

Substituting $x \mapsto g$ in an analytical expression F depending on x will then produce at once $F(x)$ and the jacobian $DF(x)$. See example below:

```
a = gradinit ([1; 2]);
b = [a.' * a; 2 * a]
⇒
b =

value =

  5
  2
  4

(partial) derivative(s) =

  2  4
  2  0
  0  2
```

Members can be accessed by suffixing the variable with .x and .J respectively

See also: use_sparse_jacobians

5.2.19 gradlog

gradlog (x) Mapping Function

overloads built-in mapper `log` for a gradient x

See also: log

5.2.20 gradlog10

gradlog10 (*x*) Mapping Function

overloads built-in mapper `log10` for a gradient *x*

See also: log10

5.2.21 gradprod

y = **gradprod** (*x*) Function File
y = **gradprod** (*x*, *dim*) Function File

overloads built-in function `prod` for a gradient *x*

See also: prod

5.2.22 gradreal

gradreal (*x*) Mapping Function

overloads built-in mapper `real` for a gradient *x*

See also: real

5.2.23 gradsin

gradsin (*x*) Mapping Function

overloads built-in mapper `sin` for a gradient *x*

See also: sin

5.2.24 gradsinh

gradsinh (*x*) Mapping Function

overloads built-in mapper `sinh` for a gradient *x*

See also: sinh

5.2.25 gradsqrt

gradsqrt (*x*) Mapping Function

overloads built-in mapper `sqrt` for a gradient *x*

See also: sqrt

5.2.26 gradsum

y = **gradsum** (*x*) Function File
y = **gradsum** (*x*, *dim*) Function File

overloads built-in function `sum` for a gradient *x*

See also: sum

5.2.27 gradtan

gradtan (*x*) Mapping Function

overloads built-in mapper `tan` for a gradient *x*

See also: tan

5.2.28 gradtanh

gradtanh (*x*) Mapping Function

overloads built-in mapper `tanh` for a gradient *x*

See also: tanh

5.2.29 isgradient

isgradient (*x*) Loadable Function

Return 1 if *x* is a gradient, otherwise return 0

5.2.30 use_sparse_jacobians

val = **use_sparse_jacobians** () Loadable Function
val = **use_sparse_jacobians** (*new_val*) Loadable Function

Query or set the storage mode for AD. If nonzero, gradients will try to store partial derivatives as a sparse matrix