HOW TO USE AUTODIFF

SEHYOUN AHN

1. Initial Setup

If you are reading this document, I believe you already want to implement automatic differentiation instead of other methods of numerical differentiation, so I will jump right into how to use it. For comparison between automatic differentiation and other forms of numerical differentiation, see section 7 below.

1.1. **Compiling mex Files.**

This program uses C acceleration to improve on the (intermediate) memory usage and speed of a portion of the program (the biggest difference will be from matrix-vector multiplication). Hence, if the size of the problem is small or the speed does not matter too much, the program can be used without compiling c files. Before this step, C/C++ compiler that is accepted by MATLAB mex needs to be installed. List of compatible compilers can be found at `http://www.mathworks.com/support/compilers/R2015b/index.html` Once, a proper compiler has be installed; run `<compile_mex_files.m>` in MATLAB. This will compile the C code for local OS using the compatible compiler.

1.2. **Adding Class Folder at Startup.**

MATLAB defines class structure with folders. Hence, the `@myAD` needs to be in its search path for the automatic differentiation to work. One way to do this is by just adding the parent folder location of the `@myAD` folder at the beginning of each session. For example, if the `@myAD` folder is in `<\home\username\scripts\>` Then

```
path(path,'\home\username\scripts\');
```

will add the correct parent folder to the search path.

However, this requires the addition of the path in every session, and this can be fixed permanently by adding the path permanently to the path file by calling

```
savepath
```

and you would not have to add the path again with each new session.[1]

2. A Quick Tutorial

Suppose you want to find the derivative of $f(\vec{x})$ at some point $\vec{x}_0$, $D_{\vec{x}}f|_{\vec{x}_0}$. Then, you would first initialize the values of $\vec{x}$'s at the point $\vec{x}_0$ by calling

```
x=myAD(x_0);
```

which will create a dual number variable x with values of x_0. Then, you can do the operation on the variable x by calling the function f.

Date: January 2016.

[1]In some platforms, you might not be able to `savepath` unless you run with heightened privileges. Matlab will throw an error/warning message with further instructions.

```
y=f(x);
```

Then, y will contain both the functional value $f(\vec{x}_0)$, and the derivative $D_{\vec{x}} f|_{\vec{x}_0}$. You can access them by calling

```
getvalues(y);
getderivs(y);
```

This is it! Numerically computing the derivative does not require any coding beyond initializing the variables as a dual number (with the initial call of myAD).

2.1. **Example.**

For example, consider $f(x_1, x_2, x_3) = (x_1^2, x_2^3, x_1 \cdot x_3)$ at point $\vec{x}_0 = (2, 6, 4)$. The following code will compute the derivative of f at $\vec{x}_0$, and set the variable A as the derivative matrix.

```
x=myAD([2;6;4]);
y=[x(1)^2;x(2)^3;x(1)*x(3)];
A=getderivs(y);
```

Automatic differentiation will handle more complex combination of operations, as long as each operations are supported. For example, since `spdiags` and matrix-vector multiplication are supported.

```
B=spdiags(y,1,3,3);
z=B*x;
C=getderivs(z);
```

will compute the derivative of $B\vec{x}$ respect to $\vec{x}$ at $\vec{x}_0$.

2.2. **Supported Functions.**

The list of supported operations are given in the following table. Since matlab has a lot of functions, not all functions are implemented. If you need functions not yet supported, e-mail sehyouna@princeton.edu.[2]

2.2.1. *Algebraic Operations.*

`+` (plus)
`-` (minus)
`.*` (times)
`.^` (power)
`./` (rdivide)
`.\` (ldivide)
`abs`
`exp`
`log`
`sqrt`

2.2.2. *Matrix Operations/Functions.*

`'` (ctranspose)
`*` (mtimes)
`\` (mldivide)
`/` (mrdivide)
`[A;B]` (vertcat) where `A` and `B` are matrices
`[A,B]` (horzcat) where `A` and `B` are matrices

[2]Disclaimer: Obviously, not all functions can be implemented.

`var = A(i:j,k:l)` (subref) where `A` is a matrix[3]
`A(i:j,k:l) = var` (subasgn) where `A` is a matrix
`cumprod`
`cumsum`
`diff`
`length`
`max`
`min`
`repmat`
`reshape`
`size`
`sort`
`spdiags`

2.2.3. *Trignometric Functions.*

`acos`
`asin`
`atan`
`cos`
`sin`
`tan`
`tanh`

2.2.4. *Logical Operations.*

`==`
`>=`
`<=`
`>`
`<`
`isnan`

2.2.5. *Etc.*

`disp`
`end`
`fsolve`[4]

3. Non-Standard Syntax

Some function calls require syntactic decisions since there is not a canonical way to do so for the function. There is only one function (fsolve for multiple variables) that require an addition syntactic requirement, but this list of functions might increase in the future. For any function not leasted here, calling the funcition for dual numbers should be the same as that of the usual function call for real numbers.

3.1. **fsolve.**

Given $f(\vec{x},\vec{y}) : \mathbb{R}^{n+m} \Rightarrow \mathbb{R}^m$ with $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^m$. The function $f(\cdot)$ needs to have the first n-dimension as the unknown values that `fsolve` will be solving for. This is not too restrictive as a new intermediate function can be

[3]`end` can be used for subsetting, e.g., `A(2:end)`
[4]Check technical notes below to see how to implement vector values problems

made with the variables reordered so that `fsolve` is solving for the first n variables. Given a function with correct ordering, the syntax of calling `fsolve` is

```
fsolve(f,x0,y,...)
```

where `f` is the function handle, `x0` is the initial guess, and `y` is the given parameters/variables (with `fsolve` options following y). This is more clear with an example. Suppose x_1, x_2 are defined implicitely by

$$f(x_1, x_2, x_3; a) = \begin{pmatrix} x_1^2 + x_2 + x_3 + a^2 \\ x_1 + x_1x_2 + ax_3 + \sin(x_3) \end{pmatrix} = \vec{0}$$

and we want to find the derivatives of x_1 and x_2 with respect to variables z_1, z_2 and z_3, where $x_3 = z_1z_2 + z_3$ and $a = z_1 + z_2$ at a point $\vec{z} = (1, 2, 3)$.

```
z=myAD([1;2;3]);
f = @(x) [x(1)^2+x(2)+x(3)+x(4)^2;...
          x(1)+x(1)*x(2)+x(3)*x(4)+sin(x(3))];
y = fsolve(f,[0;0],[z(1)*z(2)+z(3);z(1)+z(2)]);
y_values = getvalues(y);
y_derivs = getderivs(y);
```

Note that even if a is interpreted a parameter of the function (instead of a variable) in the problem, if it is a dual-number, then the function f needs to be treated as a 4-dimensional function with a as a variable.

4. Points of Potential Speed Gain

This code was optimized/tuned for our problem in mind. For example, it saves the derivative matrix as a sparse matrix because our problem results in a very sparse derivative matrices. Instead of trying to guess and tune for hypothetical case, I have included a list of points of potential speed gains.

- (To be implemented in a near future) Scalar-matrix multiplication can be C accelerated.
- (To be implemented in a near future) I allocate an auxillary storage (of size <number of rows>) to do a linear-time sorting in matdrivXvecval.c, but this can be changed. The new algorithm will behave better if the resulting vector (from $A\vec{b}$) is sparse.
- If your problem results in a non-sparse derivative matrix, it will be more efficient to use full matrices instead of sparse matrices. (You will not be able to use C acceleration provided with the package if you use full representation.)
- Because I believe people do not use cumsum and cumprod that much, I just opted not to optimize this code too much. They are included for completeness, but they can be further optimized. `cumprod` uses double loops. It probably can be improved on (or at least written in C). Also, for the second dimension of cumsum/cumprod I just transpose, cumsum in first dimension and then transpose back to get the result.
- b/A calls either ./ if A is a scalar, or (A'/b')' if A is a matrix. The latter does spurious transposes. Therefore, if possible, the problem should be reformulated to avoid b/A where A is a matrix

- Concatenations is done through loops because the varargin object can contain both `myAD` and double array objects. This can be made faster by making the matrices directly if possible instead of calling concatenation.

5. License

This program is based on <Automatic Differentiation for Matlab> package by Martin Fink with license:

With a few majors changes being:

- Derivative matrices are stored as sparse matrices instead of full matrices
- Matrices can be defined directly, and other sparse matrix operations, e.g., `spdiags`, are implemented
- Matrices multiplication and backslash are implemented
- fsolve is implemented
- New C acceleration is written for matrix-vector multiplication
- C acceleration is rewritten to for sparse storage.

6. How Automatic Differentiation Works

To be completed.

7. Comparison to Other Forms of Numerical Differentiation

To be completed.