Automatic Differentiation in Matlab

Quick Start.

Suppose you are using `fminunc` to minimize a function `fun` of several variables, but the function does not compute derivatives. You would call `fminunc` like this:

```
x = fminunc(fun,x0,options,p1,p2,...)
```

where the `options` structure tells `fminunc` that you are not computing derivatives, and generally wait a long time. However, `fminunc` works faster if it has derivatives for the function. Thanks to a technique called automatic differentiation, it may be sufficient to change the call to this:

```
x = fminunc('autodiff',x0,options,fun,p1,p2,...)
```

in order to provide `fminunc` with derivatives of your function. The `autodiff` function takes `x0` as the first argument, the original function `fun` as its second argument, then the additional arguments to `fun`, and computes the value and derivative of `fun`.

The Automatic Derivative Object.

The automatic derivative object (**adiff**) is an object which contains a column vector and a jacobian of that vector with respect to some free variables. An **adiff** object is created with a call to `adiff`:

```
a = adiff(x)
```

where `x` is the point in $\Re^n$ at which you want to evaluate a function and its derivatives. The vector of free variables `x` used to initialize the adiff object will be called the *root* vector. This is always taken to be a column vector, and will be transposed if necessary to ensure this. One then uses the **adiff** object in calculations (where it behaves much like a column vector) and retrieves the function value and derivatives at the end of the calculation.

For example, the Rosenbrock function of two variables can be evaluated at (1,2) (and the derivative automatically calculated) as follows:

```
x = adiff([1 2]);
rosen = 100*(x(1)^2-x(2))^2+(x(1)-1)^2;
```

`rosen` is an **adiff** object because it is the result of calculations involving the **adiff** object `x`, and it contains the value of the rosenbrock function at (1,2) and the derivative at that point. To retrieve the value and derivative of `rosen`, we can use the function `adiffget`, as follows:

```
rosen_value = adiffget(rosen,'value');
rosen_deriv = adiffget(rosen,'derivative');
```

or, more succinctly,

```
[rosen_value,rosen_deriv] = adiffget(rosen);
```

We end up with `rosen_value = 100` and `rosen_deriv =[ -400   200 ].` The `rosen_deriv` is a row vector giving [∂`rosen_value`/∂`x(1)`, ∂`rosen_value`/∂`x(2)`] at `x` = `[1,2]`.

Note that it only makes sense to do calculations with **adiff** objects that share the same root. Any violation of this will raise an error. For example

```
x1 = adiff([1 2]);
x2 = adiff([3 4]);
y  = x1+x2;
```

will raise the error `"adiff objects must come from the same root"` when the **adiff** objects `x1` and `x2` are added, because the first **adiff** object is a function of two variables with values 1 and 2 (the root of `x1` is [1,2]), while the second **adiff** object is a function of two variables, but with different values 3 and 4 (i.e. the root of `x2` is [3,4]). Generally it is best to define the **adiff** object once, with as many variables as are needed, and then extract parts of it using subscripting.

How Automatic Differentiation Works.

Automatic differentiation uses the chain rule to compute derivatives (the so-called *forward* mode of automatic differentiation). Suppose we have two column vectors `a` and `b`, which are functions of a vector `x`, and jacobians `Ja` and `Jb`, where

```
Ja(i,j) = ∂a(i)/∂x(j), and
Jb(i,j) = ∂b(i)/∂x(j).
```

Then the product `a.*b` (where defined) has, by the chain rule, the jacobian

```
J = diag(a)*Jb+diag(b)*Ja.
```

Similarly, the function `cos(a)` has jacobian

```
J = diag(-sin(a))*Ja
```

The **adiff** object merely encapsulates the chain rule for these and many other cases in a Matlab object.

Subscripting the ADIFF Object.

We have already met some subscripting when computing the Rosenbrock function above. In general, you can subscript an **adiff** object in any way that you can subscript a column vector, so long as the result is a column vector. For example:

```
y = adiff(1:10);
y(3:4)  = y(3:4).^2;
y(1:6)  = 4*y(1:6)+y(5:10)+y(9)/y(10);
y(5:10) = [];
```

The end result is that `y` is a four-element vector with values `[28 33 62 91]'` and jacobian

```
4     0     0     0     1     0     0     0     0.1     -0.9
0     4     0     0     0     1     0     0     0.1     -0.9
0     0    24     0     0     0     1     0     0.1     -0.9
0     0     0    32     0     0     0     1     0.1     -0.9
```

(which can be retrieved with a call to `adiffget`). The elements of the jacobian are ∂`y(i)`/∂`root(j)`, where `root(j)` is the j-th free variable defined when `y` was initialized (i.e. the vector `1:10`)

Sparse ADIFF Objects.

There are cases where the number of variables is very large and the jacobian has many zeros. In this case, a sparse **adiff** object may be more efficient. A sparse **adiff** object can be created by adding the string `'sparse'` to a call to `adiff`, e.g.

```
y = adiff(1:300,'sparse');
```

A sparse **adiff** object stores the jacobian as a sparse matrix. You can switch between sparse and full **adiff** objects by using the overloaded functions `sparse` and `full`, and test for sparseness using the overloaded function `issparse`, e.g.

```
% y is an adiff object
if issparse(y)
      y = full(y);
else
      y = sparse(y)
end
```

A sparse **adiff** object may occasionally become full as a result of Matlab operations.

Matlab Operators.

ADIFF objects are designed to act like column vectors as much as possible. Most operations that are valid for column vectors will work with **adiff** objects. The operators `+`, `-`, `.*`, `./`, `*`, `/`, and `\` have all been defined to work with **adiff** objects and/or doubles. Both power operators `.^` and `^` are available, but `^` is simply mapped to `.^`.

It is possible to take the transpose of an **adiff** object. However, this is of limited usefulness, and may cause problems. It is provided mainly so that one can compute quadratic or bilinear forms with **adiff** objects, e.g. `x'*A*x` will work if `x` is an **adiff** object and `A` a square matrix. Otherwise it is best avoided.

Finally, **adiff** objects may be vertically concatenated, and mixed with doubles while doing this. That is, if `x` is an **adiff** object, one may compute `[0; x; x.^2;(1:3)']`, and this is another **adiff** object. Any constant numbers in the vertical concatenation will create zero rows in the resultant jacobian. Horizontal concatenation is not supported.

Matlab Functions and ADIFF Objects.

It is possible to apply a large number of Matlab functions to **adiff** objects. If `x` is an **adiff** object, the following function calls have all been defined to work with **adiff** objects:

```
abs(x), cos(x), sin(x), tan(x), acos(x), asin(x), atan(x),
exp(x), log(x), log10(x), cosh(x), sinh(x), tanh(x), sqrt(x)
length(x), size(x), sum(x), max(x), max(x,y), min(x), min(x,y)
```

The following functions are also available as overrides for **adiff** objects:

`diff(x)`	: this computes `x(2:end)-x(1:end-1)`
`diff(x,n)`	: the n-th difference
`double(x)`	: is the same as `adiffget(x,'value')`
`norm(x)`	: is the same as `sqrt(sum(x.^2) )`

In addition, a number of functions implemented as m-files (such as `mean`, `std`, `var`) will work with **adiff** objects as arguments.

Complex Numbers.

The “root” of an **adiff** object must be a real vector, but you can construct complex functions from it. For example,

```
      x = adiff([1 2]);
      y = x(1)+i*x(2);
```

creates a complex function (`y`) of a real valued vector `x`. The following Matlab functions for complex numbers will also work with complex valued **adiff** objects:

```
      real, imag, isreal, conj, abs, fft, ifft
```

Note that `fft` and `ifft` are overrides for **adiff** objects, and are much slower than the built-in fft function. They also assume that their input is a column vector with a length that is an exact power of 2. If not, the vector needs to be padded with zeros.

Quick Start Revisited.

Autodiff is a function defined as:

```
   function [x,dx] = autodiff(x,func,varargin)

   if nargout==1
      x = feval(func,x,varargin{:});
   else
      [x,dx] = adiffget(feval(func,adiff(x),varargin{:}));
      dx = dx';
   end
```

It checks to see if there is only one output required (in which case fminunc is asking for the function value only) and if so, it returns the function value. If two output arguments are required, it calls the function but passes it an **adiff** object. The return value is an **adiff** object. The function value and derivative are then retrieved from it and returned as separate variables.

There is also `autodiff_sparse`, which is like autodiff but creates a sparse **adiff** object.

Some Examples.

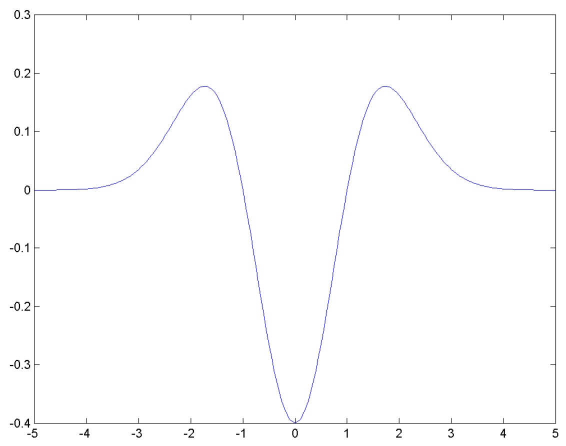
These examples show the **adiff** object in use.

1. How a Gaussian depends on its parameters.

We can use the following code to compute the derivative of a gaussian with respect to its space-constant sigma. The following code

```
   sigma = adiff(1);
   x = (-5:0.05:5)';
   gauss = exp(-x.^2/2/sigma.^2)/sqrt(2*pi)/sigma;
   [y,ydsigma] = adiffget(gauss);
   plot(x,ydsigma)
```

yields this plot, showing the derivative of a gaussian function with respect to sigma at each point x: (note that `x` needs to be a column vector, so that the resultant **adiff** object `gauss` is also a column vector. If `x` were a row vector, an error would be raised).



2. The Ackley Function.

This function (defined in D. H. Ackley. "A connectionist machine for genetic hillclimbing". Boston: Kluwer Academic Publishers, 1987) is interesting because it has multiple minima.

```
x = adiff([1 2 3]);
a = 20; b = 0.2; c = 2*pi;
s1 = 0; s2 = 0;  % s1 and s2 are double here
n = length(x);
for i=1:n;
   s1 = s1+x(i)^2;       % s1, s2 are converted to adiff objects
   s2 = s2+cos(c*x(i));  % in the loop
end
% y is an adiff object
y = -a*exp(-b*sqrt(1/n*s1))-exp(1/n*s2)+a+exp(1);
[y,dydx] = adiffget(y);
```

This yields the value `y = 7.0165` and `dydx = [0.4007 0.8014  1.2020]` at `x = [1 2 3]`.

The Speed of Automatic Differentiation.

The **adiff** class is likely to be faster than approximate derivatives computed using finite differences, and more accurate. However, automatic differentiation using the **adiff** class can be a lot slower than hand-coded derivatives, so if hand-coding is easy, the **adiff** class should be avoided.

Even so, the speed advantage to hand-coding has to be weighed against the ease of using **adiff** objects to compute derivatives automatically. **adiff** objects can always be usefully used to check the correctness of hand coded derivatives.

Installation.

Copy the `autodiff` directory to the desired location and add it and the subdirectory `autodiff/@adiff` to the Matlab search path. Note that this class has been developed using Matlab R11. More modern features such as function handles are not used, and any recent syntax or operator changes may lead to failures in the code. Please contact me if this occurs.

Contact.

For queries and bug reports, please e-mail william.mcilhagga@googlemail.com, clearly stating where and how the problem occurred and the Matlab version. A script generating the problem would be useful too.