# AD_example

July 9, 2018

## 1 Algorithmic Differentiation in tensorflow

Below is the analytic result for the evaluation and derivative of an arbitrary function. The first thing printed is the function evaluated at the chosen parameter values, and the second is the gradient with respect to each variable evaluated at those values.

In [2]:
```
import tensorflow as tf
import numpy as np

#Analytical result
xa = 1.
ya = 2.
fa = xa * np.sin(ya) + (xa / ya)
dfa_dxa = np.sin(ya) + 1 / ya
dfa_dya = xa * np.cos(ya) - xa / (ya*ya)
print(fa)
print(dfa_dxa, dfa_dya)
```

```
1.4092974268256817
1.4092974268256817 -0.6661468365471424
```

Next, we will do the same thing utilizing tensorflow. The outputs are in the same order as above.

In [3]:
```
#Tensorflow example#
#define tensorflow variables
x = tf.placeholder(tf.float64)
y = tf.placeholder(tf.float64)

#define function
f = x * tf.sin(y) + (x / y)

#create gradient object
grad = tf.gradients(f, [x, y])

#create session
```

```
        s = tf.Session()
        val = s.run([f, grad], {x: [1], y: [2]} )
        s.close

        print(np.array(val[0]))
        print(np.array(val[1]))
```

```
[1.40929743]
[[ 1.40929743]
 [-0.66614684]]
```

Lastly, this is an example of finding dx/dA for the equation b = Ax

```
In [4]: #Tensorflow matrix example#
        #A*x = b, calculating derivative dx_dA(i,j) = sum_l dx(l)_dA(i,j)..... I think
            #returns sum(dy/dx) for each independent variable, x
        #Build tensorflow objects
        A = tf.placeholder(dtype = tf.float32, shape=(3,3)) #A matrix of placeholder vals
        b = tf.constant([[1.0],[2.0],[3.0]])    #b vector
        #x = A^-1 * b
        x = tf.matmul(tf.matrix_inverse(A),b)

        grad = tf.gradients(x, A)

        s = tf.Session()
        val = s.run([x, grad], {A : [[5.0,1.0,1.0],[1.0,10.0,1.0],[1.0,1.0,15.0]]})
        s.close

        print(np.array(val[0]))
        print(np.array(val[1]))
```

```
[[0.1301939 ]
 [0.16897504]
 [0.18005538]]
[[[-0.02272082 -0.02948872 -0.03142241]
  [-0.01009814 -0.0131061  -0.01396551]
  [-0.00649166 -0.00842535 -0.00897783]]]
```

Note the output of the gradient is the sum(dy/dx) for each independent variable, x. I believe this was done because tensorflow was designed to perform back propogation and the full jacobian is not needed. Currently, tensorflow does not have an implementation for jacobians, but there are some workarounds you can find on stackexchange, etc.

```
In [5]: A = tf.placeholder(dtype=tf.float64, shape = (3,3))
        b = tf.constant([[1.0],[2.0],[3.0]],dtype=tf.float64)

        x0 = tf.random_uniform((3,1),dtype=tf.float64)
```

```
        r0 = b - tf.matmul(A,x0)
        p0 = r0
        dotr0 = tf.matmul(r0, r0, transpose_a = True)
        for i in range(30):
            Ap = tf.matmul(A,p0)
            pAp = tf.matmul(p0, Ap, transpose_a = True)
            a = dotr0/pAp
            x1 = x0 + tf.scalar_mul(tf.reshape(a,[]),p0)
            r1 = r0 - tf.scalar_mul(tf.reshape(a,[]),Ap)
            dotr1 = tf.matmul(r1, r1, transpose_a = True)
            b = dotr1/dotr0
            p1 = r1 + tf.scalar_mul(tf.reshape(b,[]),p0)
            x0 = x1
            r0 = r1
            p0 = p1
            dotr0 = dotr1

        grad = tf.gradients(x1, A)

        s = tf.Session()
        val = s.run([x1, grad], {A : [[5.0,1.0,1.0],[1.0,10.0,1.0],[1.0,1.0,15.0]]})
        s.close

        print(np.array(val[0]))
        print(np.array(val[1]))
```

```
[[0.13019391]
 [0.16897507]
 [0.1800554 ]]
[[[-0.02272082 -0.02948872 -0.03142241]
  [-0.01009814 -0.0131061  -0.01396552]
  [-0.00649166 -0.00842535 -0.00897783]]]
```

Lastly, this is an example using the conjugate gradient algorithm to solve a linear system of equations, and then find the gradient dx_dA