University of Lethbridge

Algorithmic Differentiation Term Project Report

Ahamad Imtiaz Khan
ID:001188188

December 18, 2015

1.Introduction:

Algorithmic differentiation is a method to compute derivatives. Derivative is actually rate of change of any quantity. We need to know rate of change of quantities in algebraic and differential equations as well as in many other applications. Derivatives play central role in many sectors like sensitivity analysis , inverse problem and design optimization. Numerical differentiation using difference quotients can be used to calculate derivatives. The problem with numerical differentiation using difference quotients it involves approximation error. In algorithmic differentiation we use exact equations to calculate derivatives so it does not involve with any truncation or cancellation error. Let us discuss what happens in numerical differentiation using difference quotients.We know that the derivative of a function $f(x)$ is:

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$$

We get the right hand side of the equation from Taylor expansion.

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \frac{f'''(x_0)}{3!}(x-x_0)^3 + \cdots$$

Putting $x_0 = x$ and $x = x+h$ in Taylor expansion we get:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots$$

Here we ignore the terms $\frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots$ and get

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

As the terms $\frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots$ are ignored so numerical differentiation using difference quotients incurs truncations errors. Symbolic expression is another method for computing derivative. Problem with it is it takes a lot of memory. So we will use algorithmic differentiation for computing derivatives. We were taught two modes of algorithmic differentiation in our course. Those are

i. Forward mode AD

ii. Reverse mode AD

In AD we form an evaluation trace of a mathematical equation. Evaluation trace contains a sequence of mathematical variable definitions. No variable is assigned value more than once and not more than one operator is used in right hand side of the assignment. Actually evaluation trace is a sequence of mathematical equations as well as sequence of numerical values. Sometimes we call it code list.
We completed two projects using AD. We used object-oriented programming for these projects. The projects are as follows:

i. Implement a subset of MINPACK benchmark problems using Neidinger valder algorithmic differentiation package to compute their Jacobian.

ii. Extend Neidinger valder algorithmic differentiation package to incorporate reverse accumulation or higher-dimensional tensors.

Now I will discuss about these two projects.

2.Implementing a subset of MINPACK benchmark problems using Neidinger valder algorithmic differentiation package to compute their Jacobian:

Neidinger valder algorithmic differentiation package computes derivatives using forward mode AD. I am giving here an example how forward mode AD works. Let's assume we have three functions:

$f_1 = 3x - \cos yz - \frac{1}{2}$
$f_2 = x^2 - 81(y + 0.1)^2 + \sin z + 10.6$
$f_3 = e^{-xy} + 20z + \frac{10\pi - 3}{3}$

Jacobian matrix of the functions will be like:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{pmatrix}$$

The computational graph of the functions is given in Figure 1.
The weights of the edges are local derivatives. For each independent variable we have one or more paths to the dependent variables. We calculate product of weights of edges of each path and find summation of the paths from one independent variable to one dependent variable. By this way we get suppose $\frac{\partial f_1}{\partial x}$. All other partial derivatives are calculated in same way. Finally we compute Jacobian which is as follows:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{pmatrix}$$

$=$

$$\begin{pmatrix} 3 & z \sin yz & y \sin yz \\ 2x & -162(y + 0.1) & \cos z \\ -ye^{-xy} & -xe^{-xy} & 20 \end{pmatrix}$$

If we put any value for x, y and z we will get a Jacobian matrix. The result we get using valder.m package is same.

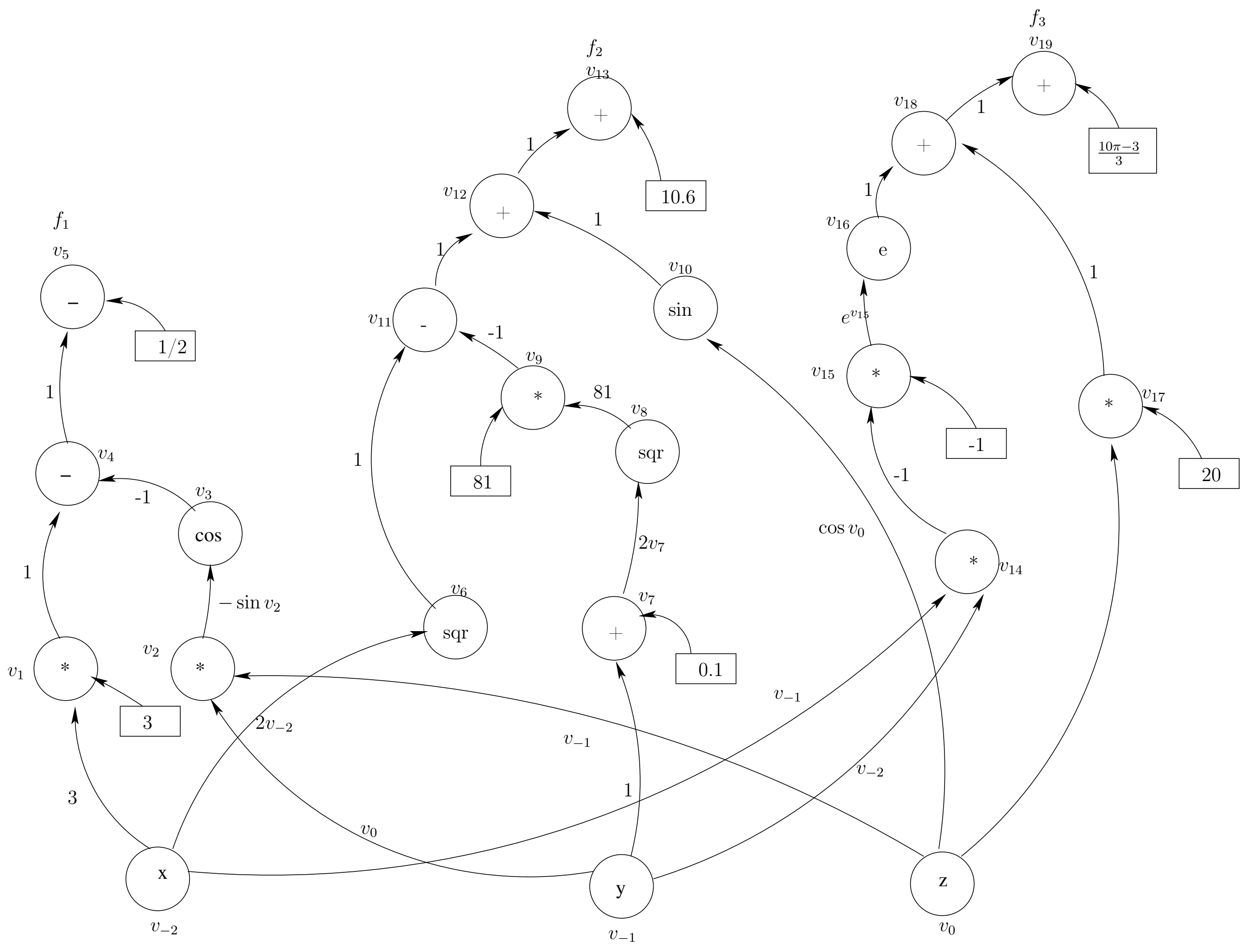

Figure 1: Computational graph of f_1, f_2 and f_3

We implemented the set of MINPACK benchmark problems using Neidinger valder algorithmic differentiation package. Our task was to compute Jacobian matrix of the test functions. The MINPACK benchmark problems we got gives their Jacobian as output but they implicitly calculated the derivatives. Our task was to implement valder AD package so that we will not need to compute derivatives in computing Jacobian. Valder AD package is already implemented in MATLAB/Octave. My task was to implement it using C++ programming language. Neidinger used OOP with operator overloading. I implemented the Neidinger valder package in C++ using OOP. Operator and function overloading was used in valder.cpp class. The functions are stated bellow:

```
valder operator+(valder u, valder v);
valder operator+(valder u, double v);
valder operator+(double u, valder v);
valder operator-(valder u);
valder operator-(valder u, valder v);
valder operator-(valder u, double v);
valder operator-(double u ,valder v);
valder operator*(valder u, valder v);
valder operator*(valder u, double v);
valder operator*(double u, valder v);
valder operator/(valder u, valder v);
valder operator/(valder u, double v);
valder operator/(double u, valder v);
valder pow(valder u, valder v);
valder pow(valder u, double v);
valder pow(double u, valder v);
valder exp(valder u);
valder log(valder u);
valder sqrt(valder u);
valder sin(valder u);
valder cos(valder u);
valder tan(valder u);
valder asin(valder u);
valder atan(valder u);
valder abs(valder u);
```

a. Test functions in SSA form: We converted the test functions in Static Single Assignment form. In SSA form variables are split into instances. Each variable is assigned exactly once. SSA form has advantages in optimization. Here is an example of a test function and the code list for that test function:
Helical valley function

$$f_1(x) = 10[x_3 - 10\theta(x_1, x_2)]$$
$$f_2(x) = 10[(x_1^2 + x_2^2)^{1/2} - 1]$$
$$f_3(x) = x_3$$

where

$$\theta(x_1, x_2) = \begin{cases} \frac{1}{2\pi} arctan\left(\frac{x_2}{x_1}\right) & \text{if } x_1 > 0 \\ \frac{1}{2\pi} arctan\left(\frac{x_2}{x_1}\right) + 0.5 & \text{if } x_1 < 0 \end{cases}$$

Code list for Helical valley function is as follows:

```
//read x1,x2 and x3
v_2 = x1;
v_1 = x2;
v0 = x3;
v1 = v_1/v_2;
v2 = atan(v1);
v3 = p*v2;
v4 = v3+0.5;
if(x1 > 0)
{
        v5 = 10*v3;
        v6 = v0-v5;
        f1 = 10*v6;
}
if(x1 < 0)
{
        v7 = 10*v4;
        v8 = v0-v7;
        f1 = 10*v8;
}
v9 = pow(v_2,2);
v10 = pow(v_1,2);
v11 = v9+v10;
v12 = sqrt(v11);
v13 = v12-1;
f2 = 10*v13;
f3 = v0;
```

The variables such as v_2, v_1, v0 etc. are actually valder class objects. At each assignment the corresponding overloaded function is called and value and derivatives are calculated. so at final assignment we got function value and gradient vectors in f1, f2 and f3 object. From the gradient vectors we got Jacobian of these functions. We converted the test functions of MINPACK benchmark problems similar ways and using valder.cpp class calculated the derivatives and finally got Jacobian matrix for the test functions. The example stated above has fixed number of functions. But some test functions have variable number of functions. In those case instead of declaring a single valder object I declared arrays of object. In this way the test functions were converted to SSA form and computed Jacobian.

3.Exteningd Neidinger valder algorithmic differentiation package to incorporate reverse accumulation:

Richard D. Neidinger implemented forward mode AD using MATLAB/Octave. Neidinger used OOP with operator overloading in implementing forward mode AD. valder.m package calculates function value as well as it's derivatives. We had to change this valder.m package for implementing reverse mode AD. In reverse mode AD there are two passes. First one is forward pass and then reverse pass.

a. Forward Pass: In forward pass we calculated immediate partial derivative. valder.m does not calculate immediate partial derivatives. So we had to change valder.m package so that it can calculate immediate partial derivative so that we can proceed towards reverse mode AD. A small example is given bellow: The function that overloads multiplication operator in forward mode AD is:

```
function h = mtimes(u,v)
          sprintf('**  mtimes   **');
         %VALDER/MTIMES overloads multiplication * with at least one
            valder object argument
         if ~isa(u,'valder') %u is a scalar
            h = valder(u*v.val, u*v.der);
         elseif ~isa(v,'valder') %v is a scalar
            h = valder(v*u.val, v*u.der);
         else
            h = valder(u.val*v.val, u.der*v.val + u.val*v.der);
         end
end
```

For calculating immediate partial derivatives we changed this function as follows:

```
function h = mtimes(u,v)
           sprintf('**  mtimes   **');
           %VALDER/MTIMES overloads multiplication * with at least one
              valder object argument
          if ~isa(u,'valder') %u is a scalar
                         h = valder(u*v.val, [0 u 0 v.pos]);
          elseif ~isa(v,'valder') %v is a scalar
             h = valder(v*u.val, [v 0 u.pos 0]);
          else
                h = valder(u.val*v.val, [v.val u.val u.pos v.pos]);
          end
end
```

In the same way we modified the other functions used for operator overloading. In forward pass we calculated and stored immediate partial derivatives each variable of evaluation trace.

b. Reverse Pass: We formed an extended Jacobian matrix that stores the immediate partial derivatives. In forward pass when a row variable has an immediate dependency on column variable we saved the immediate partial derivative in that location. We assign a position (in this case row position) for each variable in code list and as rhs of each code can have at most 2 operands/variables, their position are also stored in corresponding row. From this process we get immediate dependency of row variables and column variables in forming extended Jacobian matrix. The we subtracted an identity matrix of same size from this matrix. Backward substitution which is actually the reverse pass will result in solution i.e. the derivatives.
We tested number of formulas using our extension and it gives correct result.

4.Discussion:

What we get from AD it is also possible to achieve using symbolic differentiation. But when we work with numerical data symbolic differentiation doesn't make much sense. Numerical differentiation using difference quotients incurs truncation error. So AD can be appropriate method in calculating derivatives. We implemented the above mentioned projects using forward mode AD and reverse mode AD. We converted the MINPACK benchmark problems to SSA form and computed Jacobian using valder package. There are advantages of using SSA form. They are needed to be tested. The implementation of reverse mode AD my modifying valder package is also needed to be tested.