Algorithmic Differentiation Term Project Presentation

Ahamad Imtiaz Khan
MSc Student
ID:001188188

Department of Mathematics and Computer Science
University of Lethbridge

18th December, 2015

OUTLINE

Algorithmic Differentiation and other related methods

Project Works

Algorithmic Differentiation and other related methods

- Algorithmic differentiation is a method to compute derivatives.
- Derivative is rate of change of any quantity.
- There are other methods for computing derivatives like:
  - Symbolic Differentiation
  - Numerical differentiation using difference quotients

Problems with other methods

- Symbolic Differentiation:
  - When we work with number in computer symbolic differentiation is not a choice.
- Numerical differentiation using difference quotients:
  - This method incurs truncation error.
    $\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
    We get this from Taylor expansion:
    $f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots$
    We ignore the terms $\frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots$ and get
    $f'(x) = \frac{f(x+h)-f(x)}{h}$

Algorithmic Differentiation

- In algorithmic differentiation we use exact equations to calculate derivatives so it does not involve with any truncation or cancellation error.
- 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.
- Then we compute derivative using one of the two modes of AD
  - Forward mode AD
  - Reverse mode AD

Projects

- Implement a subset of MINPACK benchmark problems using Neidinger valder algorithmic differentiation package to compute their Jacobian.
- Extend Neidinger valder algorithmic differentiation package to incorporate reverse accumulation.

Implementation of MINPACK benchmark problems

- We implemented the set of MINPACK benchmark problems using Neidinger valder algorithmic differentiation package.
- 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.

Implementation of valder Package in C++

- Valder AD package is already implemented in MATLAB/Octave.
- I implemented 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.

Methods of valder.cpp class

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/(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);

Test functions in SSA form

- In Static single assignment form each variable is assigned exactly once.
- Every variable is defined before it is used.
- We converted the test functions of MINPACK benchmark problems. For Example 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
    $$\theta(x_1, x_2) = \begin{cases} \frac{1}{2\pi} arctan\left(\frac{x_2}{x_1}\right) & \text{if } x_1 > 0 \\ \frac{1}{2\pi} arctan\left(\frac{x_2}{x_1}\right) + 0.5 & \text{if } x_1 < 0 \end{cases}$$

Test functions in SSA form

```
//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;
```

Implementation of MINPACK benchmark problems

- 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.
- 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.

Extending Neidinger valder algorithmic differentiation package to incorporate reverse accumulation

- Neidinger implemented forward mode AD using MATLAB/Octave.
- We had to change this valder.m package for implementing reverse mode AD.
- In reverse mode AD there are two passes.
  - Forward pass
  - Reverse pass

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.

```
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
```

Forward Pass

- mtimes(u,v) funtion for reverse AD in valder.m

```
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
```

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.
- For tracking the dependency we had to assign position for each valder object.
- We formed an extended Jacobian matrix and from backward substitution which is reverse pass we got the derivatives.

Thank You