AUTOMATIC DIFFERENTIATION FOR FIRST AND SECOND DERIVATIVE FOR SOLVING NONLINEAR PROGRAMMING PROBLEMS

By

YING LIN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2011

To my parents

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my advisor Dr. Anil V. Rao, for his constant encouragement and support throughout this project. Especially, I thank him for his technical guidance that makes the accomplishment of my work be possible. His clear thinking, understanding, enthusiasm towards science all influence me to work harder. I would also thank my committee member Dr. Warran Dixon for his patience and his time on my thesis. His hard working also inspires me to work harder and dig deeper in my thesis work. I thank all my friends, who helped me a lot no matter in my thesis work or the mental support and consistent support throughout the work. I am especially in debt to Michael Patterson, who spent a lot of time in providing better understanding for my work. Finally, I would like to express my deep appreciation to my family. In particular my parents, their care, love, and motivation make me hold on until the last minute.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AUTOMATIC DIFFERENTIATION FOR FIRST AND SECOND DERIVATIVE FOR SOLVING NONLINEAR PROGRAMMING PROBLEMS

By

Ying Lin

August 2011

Chair: Anil V. Rao
Major: Mechanical Engineering

In this work, a new algorithm is proposed for automatic differentiation. The new algorithm uses object-oriented MATLAB to generate multi-dimensional derivatives. This work combines the forward mode with operator overloading to realize the first and second derivatives of a function. Automatic differentiation in this thesis can be applied for solving more complex function problems in a more efficient and precise manner, than when compared to the traditional differentiations. In order to get to know the features of this tool, an example of the arrowhead function is presented. The second example is a simple optimization problem with only two states. The last two nonlinear programming problems are more complex and sensitive with greater than three states. By providing the gradient value for the object and constraint functions and hessian value for the Lagrangian equation, those examples are solved with more effectiveness and preciseness. Compared to the traditional differentiations, which is tedious and error prone, automatic differentiation saves time for deriving optimization results with machine precision.

CHAPTER 1
INTRODUCTION

Motivation

Getting the derivatives of a function plays an important role in different areas that need numerical computation. There exists different ways of computing derivatives, such as hand-coding, finite differencing, symbolic differentiation and automatic differentiation which is a relatively new technology to generate the derivative value. (1) Hand-coding: Hand-coding means computing derivatives by hand. This procedure is tedious, error prone and time consuming. (2) Finite Differencing: One of the numerical methods to calculate differentiation is finite differencing, which has the general form of $\mathrm{f}(\mathrm{x}+\Delta\mathrm{x})-\mathrm{f}(\mathrm{x})$. Once it divides $\Delta\mathrm{x}$, we can get the differentiation quotient $\frac{\mathrm{f}(\mathrm{x}+\Delta\mathrm{x})-\mathrm{f}(\mathrm{x})}{\Delta\mathrm{x}}$. Hence, numerical calculation for differentiation is preferred over hand differentiation. Taylor Series is the expansion of finite differencing. A function that is infinitely differentiable in a neighborhood of point a, can be expressed using the Taylor series expansion. $\mathrm{f}(\mathrm{x})=\mathrm{f}(\mathrm{a})+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f^3(a)}{3!}(x-a)^3+\cdots+\frac{f^n(a)}{n!}(x-a)^n$. By using the higher order terms to approximate a function f(x), we can improve the precision. However, even the small errors in higher order differentiation values can cause big round-off errors to the entire approximation result. (3) Symbolic Differentiation: Symbolic differentiation is the process of taking the derivative of a string represented function with respect to the variables. For example,

$$f(x)=(x_1*x_2*x_3+x_4)*x_5 \tag{1-1}$$

The first order derivative of this function is

$$f'(x) = \begin{bmatrix} x_2 * x_3 * x_5 \\ x_1 * x_3 * x_5 \\ x_1 * x_2 * x_5 \\ x_5 \\ x_1 * x_2 * x_3 + x_4 \end{bmatrix} \quad (1\text{-}2)$$

The evaluation of f'(x) is not sufficient. Especially, when the problem is very complicated and with large size of variables, the description of the function derivatives will be very long. (4) Automatic differentiation: Automatic differentiation is a tool that relies on a computer program to execute the elementary operation, such as the binary function, by applying the chain rule to the composition of those elementary operations over and over again, yielding the final derivative value. Automatic differentiation has become an important tool for quick computation and error reduction in computing the derivative value.

Automatic differentiation finds a great deal of application in solving optimal control problems. Usually, an optimal control problem can be converted into a nonlinear programming problem (NLP). This NLP requires the gradient of the function to find the minimum or maximum point, which gives the point of optimality. If the user does not provide the gradient value of the function, the function will generate the gradient using finite difference approximation, which has poor precision. Since most of the functions in practical optimization problems have complex derivatives, it becomes very difficult for users to provide the gradient of each function. Hence, the automatic differentiation tool plays an important role in providing the efficient gradient values.

Apart from the optimization problems, automatic differentiation can also be used in sensitivity analysis, which is widely used in mathematics, physics, engineering and other areas.

Literature Review

In 1982, Philip Wolf developed a procedure in his paper [1] that tests a program used for computing the derivatives of given function and locates the errors in the program. He also mentioned that calculating the gradient is 1.5 times as expensive as calculating the function. Later, Andreas Griewank, in his paper [2], stated that the upper bounds for the ratio of expense for calculating the gradient and the function should be 5. He also introduced the basic mathematics and three automatic differentiation tools' implementations.

Automatic differentiation tool ADIFOR was developed in the paper [3] by Christian Bischof. ADIFOR stands for automatic differentiation in FORTRAN and it is a source transformation tool written in Fortran 77. This tool was capable for use in real time experiments. It was more accurate than the divided approximation and also reduced the time used for computing derivatives. An implementation of ADIFOR is discussed in [4]. This paper uses the automatic differentiation tool ADIFOR for solving systems of nonlinear equations and for parameter identification problems. Magana Jimenez [5] used ADIFOR 2.0 to provide efficient lie derivatives and Jacobians to calculate the control law and evaluate the stability of the control system. Meanwhile, the author also developed a user-friendly package called MAFC, which is MATLAB5.3-ADIFOR2.0-FORTRAN. This package is useful for analyzing closed-loop control system. The paper [6] applied the automatic differentiation in solving nonlinear programming problems. He also presents a heuristic approach to reduce the memory occupied for processing graphs, which are used for solving the large-scale problems. He also discussed the method to decide the sparsity pattern of hessian matrix. This work [7] implemented automatic differentiation into optimal

control problems. The original problems were reduced to nonlinear programming problems. This transformation was realized by using Runge-Kutta formulation.

In 1991, Andreas Griewank introduced an automatic differentiation tool ADOL-C [8]. It is a forward mode tool, which is written in C++ and implemented by using operator overloading. This tool could not only compute the first order derivative but also compute the higher order derivatives. Hence it was widely used for Automatic differentiation implementation, in areas requiring numerical computation. In [9], Kyung-Wook Jee implemented automatic differentiation tool (ADOL-C) to solve Intensity Modulated Radiation Therapy (IMRT) problem. These problems have multiple-objective, and they are solved using the multi-criteria approach. In order to make ADOL-C more efficient, David Juedes and Andreas Griewank [10] paralleled implemented the reverse mode evaluation on each node along with the forward mode evaluation. The paper by Kowarz Andreas [11] develops a technique that improves the application of operator overloading based automatic differentiation and allows higher serial performance of ADOL-C. This paper also developed an activity-tracking technique that allows compact internal function and simplifies automatic differentiation.

There are other automatic differentiation tools, such as ADMAT. In 1998, Arun Verma introduced an automatic differentiation tool, which can compute the derivative accurately and fast [12]. This tool used object oriented MATLAB and could compute the derivative of up to second order. One year later, he along with Thomas F Coleman developed ADMIT-1 [13]. This new tool made the computing of sparse Jacobian and Hessian matrices possible. Also ADMAT could be used as a plug-in tool with ADMIT-1 to compute the derivative matrices for nonlinear optimization. In 2006, Christian H. Bischof

developed the source transformation tool ADiMat for MATLAB with embedded macro language [14]. This macro language helped in generating more efficient derivatives. Also David M.Gay [15] developed RAD package which is a semiautomatic differentiation tool that compute a function and its gradient individually.

These gradients were then assembled individually “by hand” to occupy less memory than the normal automatic differentiation. The paper [16] introduces the AD tool implemented by the cross-country elimination, which is called OpenAD. And this paper applied this tool to construct unambiguous computational graphs. Cross-country is different from forward or reverse mode, it requires the code’s computational graph. EliAD also implements the cross-country, but it has the number restriction in its input code. OpenAD is the extension of EliAD. Phipps [17] used automatic differentiation package ADMC++ to provide the derivative for computing the periodic orbits in the system of hybrid differential-algebraic equations. ADMC++ was developed based on MATLAB language that can not only provide derivatives, but also provide taylor series coefficients. By combining the automatic differentiation tool with COSY INFINITY, Pavel Snopok [18] simulated the Tevatron accelerator and optimizes them into the same strength in all the arcs. Tadiouddine [19] studied computing Jacobian by using vertex elimination algorithm. This algorithm reorders the statements of Jacobian code, and not only reduces the memory, but also reduce the flops.

Overview of the Thesis

This work has developed a new Automatic Differentiation tool named “Hes”, which stands for hessian matrix. In particular, Hes focus on the operator overloading and forward mode. The aim of the work is to introduce the algorithm of the automatic differentiation tool. At the same time, this work studies three nonlinear programming

problems. From the results of those examples, the AD tool is proved to be efficient in solving nonlinear programming problems.

Thesis Outline

The outline of this paper is as follows. An introduction of the research motivation on automatic differentiation (AD) is given in Chapter 1. Chapter one also introduces the review of previous work, such as different AD tools, their implementation along with how they related to the optimal control. Chapter 2 has introduced basic mathematics and chain rule, which is used in automatic differentiation. It also gives brief introduction of the two ways of implementing automatic differentiation, such as source transformation and operator overloading. Also, two different modes of AD are described here. Chapter 3 first shows class name, member variables and functions being used in automatic differentiation. Meanwhile, it also introduces different types of functions along with their functionality. In Chapter 4, brief study of optimization and the way of how it is related to the nonlinear programming is presented. This chapter also shows the formulation of optimization, nonlinear programming, and Lagrangian that is used to solve the nonlinear programming problems. Four examples are given in Chapter 4. Those examples are organized from the simplest one with fewest variables to the complicated one with more variables. Chapter 5 is the conclusion of the whole thesis and the results showed in the example implementations. It also comes up with the direction of future work.

CHAPTER 2
BACKGROUND

The main purpose of this chapter is to study some basic mathematical concepts which are closely related to the automatic differentiation. Meanwhile, it introduces two different modes: forward and reverse mode; and two AD implementing method: operator overloading and source transformation. A brief description of the chain rule is also presented in this chapter.

Differentiation

In Mathematics, differentiation is the process of computing the derivative of the function. For a function y=f(x), where the output, y is called the dependent variables and the input, x is called the independent variable. Derivative measures the rate change of output y with respect to independent variable x. It can be represented as follows:

$$y = \lim_{x_1 \to x} \frac{y(x_1) - y(x)}{x_1 - x} \quad (2\text{-}1)$$

$$\text{Or } y = \frac{dy}{dx} \quad (2\text{-}2)$$

This equation shows the derivative is the slope value of the tangent drawn at point x1. In other words, it is about the rate change of the function with respect to the variables at a given point.

The derivative can be used to determine the critical points of a function. A critical point is the point where the derivative of the function equals zero. We may show this graphically using a simple two-dimensional example. The lateral line represents the input variable or independent variable. The vertical line represents the output variable or dependent variable. The capital letter, A, B and C are the different points on the curve line. The curve has different rates of change at A, B and C.

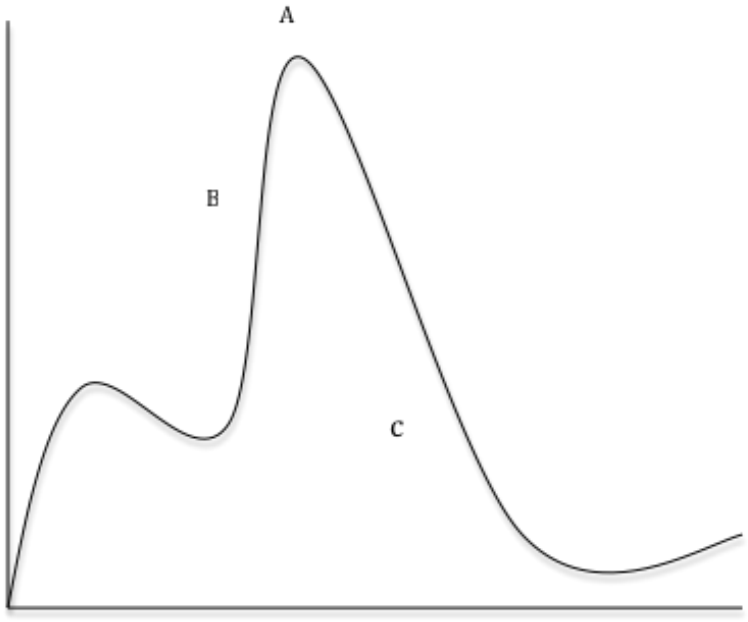

Figure 2-1. The curve showing the changing of the slope

From the graph above, the curve changes at different rates. At A, the rate of change of the function is zero. The function goes from increasing to decreasing at A, therefore A is a local maximum point. The automatic differentiation Algorithm developed in this work relies on determining the maximum or minimum values of most user supplied functions.

Jacobian Matrix

In calculus, Jacobian Matrix is a matrix that one vector taking partial derivative with respect to another vector. Assuming y is a vector with m variables, x is a vector with n variables. So the Jacobean matrix for y is denoted as

$$\begin{pmatrix} \frac{dy_1}{dx_1} & \cdots & \frac{dy_m}{dx_1} \\ \vdots & \ddots & \vdots \\ \frac{dy_1}{dx_n} & \cdots & \frac{dy_m}{dx_n} \end{pmatrix} \tag{2-3}$$

Suppose m=1, that is, y is a scalar. Then the Jacobian matrix is a column vector. If x is a scalar, and y is a vector, the Jacobian matrix will be a row vector. If both x and y are scalars, the result will be a scalar.

Hessian Matrix

In mathematics, Hessian Matrix is the matrix of one function taking second partial derivatives with respect to the entire variables. Suppose one function f with m variables, and is written in the form of f $f(x_1, x_2, \dots, x_m)$. Actually, hessian matrix is the Jacobian matrix taking derivative with respect to all the variables. In this case, output is a scalar, Jacobian matrix will be a column vector. Then, hessian Matrix is a square matrix

$$H(f)_{ij}(x) = d_i d_j f(x) \tag{2-4}$$

Where i and j are indices indicating row and column number and D is the derivative operation. Here the Hessian matrix will become as

$$\begin{bmatrix} \frac{d^2 f(x)}{dx_1^2} & \frac{d^2 f(x)}{dx_1 dx_2} & \cdots & \frac{d^2 f(x)}{dx_1 dx_m} \\ \frac{d^2 f(x)}{dx_2 dx_1} & \frac{d^2 f(x)}{dx_2^2} & \cdots & \frac{d^2 f(x)}{dx_2 dx_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{d^2 f(x)}{dx_m dx_1} & \frac{d^2 f(x)}{dx_m dx_2} & \cdots & \frac{d^2 f(x)}{dx_m^2} \end{bmatrix} \tag{2-5}$$

Ways of Implementing AD

There are two ways of implementing automatic differentiation which transform the function from computing function into the function that generates value and derivative. Those two ways are operator overloading and source transformation.

Operator overloading initially overloads the elementary operations of the function in an array. The derivatives of the elementary function are given in those functions. By taking the derivative of composite function with respect to the independent variable,

overall derivative is generated. It is hard to apply the reverse mode in this method. The reverse mode is explained in Chapter 3.

Source transformation is a way that reads in a program, determines which statement needs to be derivative and carries out the source code for computing the derivative. It augments the original source code of the function with instruction for calculating the derivatives. The source transformation can be applied by reverse mode as given in [20]

Different Modes

Forward Mode

Forward mode propagates derivatives with respect to the independent variables through the chain rule. In the forward mode, it breaks the complex function into the elementary functions. The program gives the derivatives of those elementary functions. By applying some operation rules, such as product rule, power rule, Quotient rule, reciprocal rule and so on, forward mode propagates the derivative of the function. For example, h(x)= f(g(x))+m (n(x)), G(x), n(x) is the function with the variable x, f and m is the function with variable g(x), n(x). Therefore, f (g(x)) is functional with the variable x.

$$f'\big(g(x)\big) = \frac{df}{dg}\frac{dg}{dx} \tag{2-6}$$

$$m'\big(n(x)\big) = \frac{dm}{dn}\frac{dn}{dx} \tag{2-7}$$

$$h'(x) = \Big(f\big(g(x)\big) + m\big(n(x)\big)\Big)' = f'(g(x)) + m'\big(n(x)\big) \tag{2-8}$$

So the total derivative of the equation should collect each element's value and derivative then compile them into the equation above.

$$w_1 = x;$$
$$w_2 = g(w_1);$$
$$w_3 = f(w_2);$$
$$w_4 = n(w_1);$$
$$w_5 = m(w_4);$$
$$w_6 = w_3 + w_5;$$

Taking derivative of the variables above,

$$w_1' = 1;$$
$$w_2' = g'(w_1) * w_1';$$
$$w_3' = f'(w_2) * w_2';$$
$$w_4' = n'(w_1) * w_1';$$
$$w_5' = m'(w_4') * w_4';$$
$$w_6' = w_3' + w_5';$$

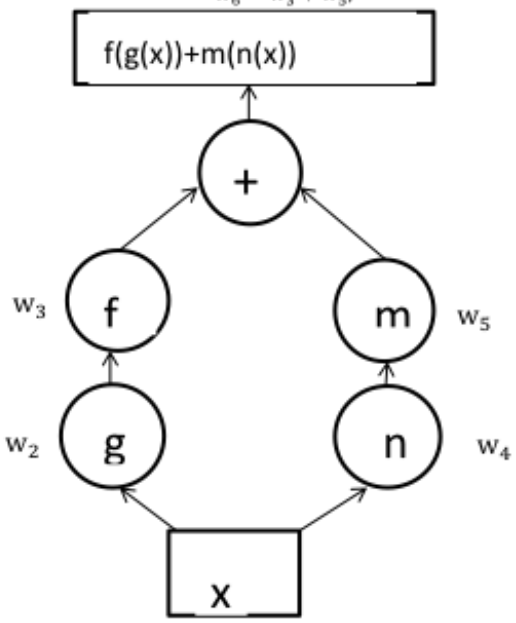

Figure 2-2. Shows the computation procedure of forward mode

The second derivative of this function is $h^{"}(x) = f"(g(x)) + m^{"}(n(x))$. In order to get the second derivative of the function, we need to first compute the second derivative of each element in the two sides of "+" by applying product rule and then substitute each value into the equation above respectively.

Reverse Mode

Reverse Mode follows the opposite direction as the forward mode does in computing the derivative. That is to say, it generates derivatives of the underlying function

through the opposite direction of the chain rule. For example f (g(x)), the chain rule to calculate the first derivative of this function is $f'\big(g(x)\big) = \frac{df}{dg}\frac{dg}{dx}$. In reverse mode, it will first get the value of $\frac{dg}{dx}$, then $\frac{dg}{dx}$, serving $\frac{df}{dg}$ as a seed. In order to illustrate the way reverse mode works, we use the following example. Suppose one composition function $f(x) = \sin(\cos(x))$

$$w_1 = x \tag{2-9}$$

$$w_2 = \cos(w_1) \tag{2-10}$$

$$w_3' = 1 \tag{2-11}$$

$$w_2' = w_3' * \frac{dw_3}{dw_2} = w_3' * \cos(w_2) = 1 * \cos(w_2) \tag{2-12}$$

It uses w_1, w_2, w_3 to represent

So it propagates derivatives from most outside of the function to the most inner of the function. And the final derivative of the function is taken with respect to the intermediate quantity.

Chain Rule

The chain rule is a mathematical formula that is used to compute the derivative of a composite function with more than two functions. For example, a function f maps a vector x with n-dimensions into the vector y with m-dimensions. The dimension of this function is $m \times n$. If f is a simple function with variable x, it can be written as f (x) where x is an argument with n entries. Normally, the first derivative of f(x) is $f'(x) = \frac{df}{dx}$. But we have created a new term $\frac{dx}{dx}$ that can be used as the stop target for the whole computation. AD stops evaluating the derivative of the function when it encounters the situation of taking the derivatives of independent variables with respect to the independent variables. For

example, in the following equation, the first derivative of f(x) is the product of the outer derivative$\frac{df}{dx}$ and the inner derivative$\frac{dx}{dx}$.

$$\nabla f(x) = \frac{df}{dx}\frac{dx}{dx} \tag{2-11}$$

For the first element of the right side of the equation, f is a vector with length m. When the derivative is taken derivative with respect to independent variable x, the output dimension will be increased. That is because if one of the elements in f takes the derivative with respect to n x, it will return a row vector with length n. Stacking up the total number of the m row vectors will result in the total dimension of m by n. So the dimension for the first element $\frac{df}{dg}$ in the equation above is .

If f is a composite function of g(x), where g(x) is the argument of f and x is the independent variable,$x \in R^n$, and f can be written as f(g(x)). The first derivative of function f with respect to x can be written as

$$f\ (g(x)) = \frac{df}{dg}\frac{dg}{dx} \tag{2-12}$$

Suppose f maps from $f \in R^m$to $x \in R^n$, the dimension for f(x) is . The dimension of $\frac{df}{dg}$ is $m \times n$. Therefore, the total dimension for the first derivative is $m \times n \times n$. The dimension of the first derivative equals the dimension of the function times the number of the input. Generally, the computed first derivative's dimension equals the dimension of output times the dimension of input.

Then the chain rule is applied to the second derivative. For the first simple function, the second derivative is literally taking the derivative of the target function twice. Based on

the first derivative, the second derivative is taking the derivative of the first derivative with respect to the independent variables. It can be written as

$$f"(x) = \frac{d}{dx} f'(x) = \frac{d}{dx}(\frac{df}{dx}\frac{dx}{dx}) = \frac{d^2 f}{dx^2}(\frac{dx}{dx})^2 + \frac{df}{dx}\frac{d^2 x}{dx^2} \quad (2\text{-}13)$$

As provided in the first derivative, the dimension of first derivative is $m \times n$. Suppose the first derivative is a function h (x) and the second derivative of the original function should be the first derivative of h (x). Applying the general rule for the dimensions as described above, the dimension of h'(x) is the dimension of function f times the dimension of the independent variables: $m \times n \times n$.

For the composite function problem f(g(x)), the second derivative of the function can be written as

$$f"(x) = \frac{d}{dx} f'(x) = \frac{d}{dx}(\frac{df}{dx}\frac{dx}{dx}) = \frac{d^2 f}{dx^2}(\frac{dx}{dx})^2 + \frac{df}{dx}\frac{d^2 x}{dx^2} \quad (2\text{-}14)$$

Because $\frac{d^2 x}{dx^2}$ is zero, only the first two terms left. Then the dimension of the second derivative equals the dimension of the function times the dimension of the independent variable squared or $m \times p \times n \times n$. In general, the product rule is applied in both first derivative and second derivatives.

CHAPTER 3
ALGORITHM

In this chapter, different AD algorithm components, such as binary functions, unary functions, operation functions and shaping functions, are described. Consider again the function f (x):$\mathrm{R}^n \to R^m$, the total dimension of this function is $\mathrm{R}^{m \times n}$. When evaluating this function, automatic differentiation will overload the different AD algorithm components.

Class, Variable, Member Functions, Properties

- Class: In this work, the class name is Hes, which stands for hessian.
- Variable: 1) Input variable: In this work, the input variable could either be a double precision variable or an object of class. 2) Output variable: The output variable can only be an object of the class Hes.
- Member functions: Class Constructor function, Composite function, Unary function, Binary function, Constant Variable function, Constant function, Reshaping function, Array Reference& Assignment function.

Figure 3-1 shows how these member functions work together with each other and propagate the overall output values. Suppose the input is a complex function, which contains both unary functions and binary functions. In the first step, user has supplied one input variable, and the input variable can be any type of variable. The constructor function will first evaluate the input variable, create one class object and return the object's properties as well. The program will call relative unary function and binary function to provide the derivative. Meanwhile, the shaping function, array reference and assignment function assist the whole computation process if necessary. In the end of the computation, the composite function collects all the results from the unary function, binary function and constructor function, and returns the final derivative value of the user supplied equation's derivative.

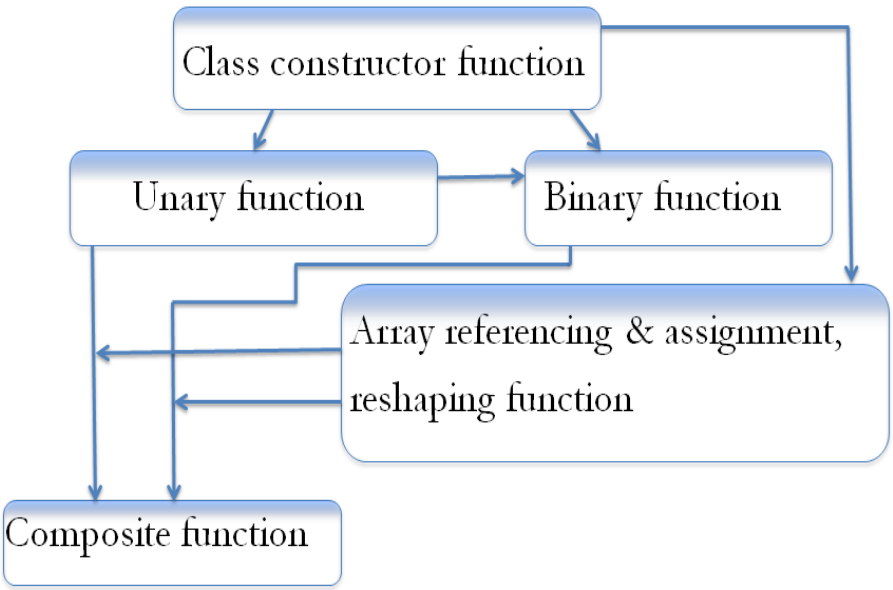

Figure 3-1. Relationship between different AD components

• **Properties**: The object stores the specification of the class; those properties are Value, derivative, s_nz, s_derivative, nderivs, s_nderivs. The fields, value and derivative stand for the value of the variables and the directional first derivative respectively. The field nderivs is the number of independent variables. s_nz stands for the nonzero values indicating the direction to which the second derivative goes. The field, s_derivative stands for the second derivative value. s_nderivs is the number of independent variables that the function takes second partial derivative with respect to. The order of these fields has to be fixed.

Inner Derivative

Inner derivative is taking derivative of the independent variables with respect to the independent variables themselves. The function will stop evaluating the derivative when it finishes taking the inner derivative. Assuming n-valued variables that are denoted as x,

the total dimension of the first derivative is $n \times n$. If the derivative of an independent variable is taken derivative with respect to other independent variables, it will result in zero. As a result, for the first derivative, the elements that lie in the diagonal are equal to one and the off-diagonal elements are equal to zero. Then the first derivative of x is the identity matrix.

For the second derivative, the dimension will be $n \times n \times n$; the second derivative is a three dimensional matrix, which can be described by a cubic as shown in Fig 3-2, with the length of each side equal to one of the dimension in the matrix. Look at the graph below, suppose O is the starting point. I, J and K represent each dimension of the derivative. The form of the second derivative is $\frac{d}{dx_i}\frac{dx_j}{dx_k}$. Only when i=j=k, for example, in the diagonal direction of the cube, will the value of the second derivative be nonzero.

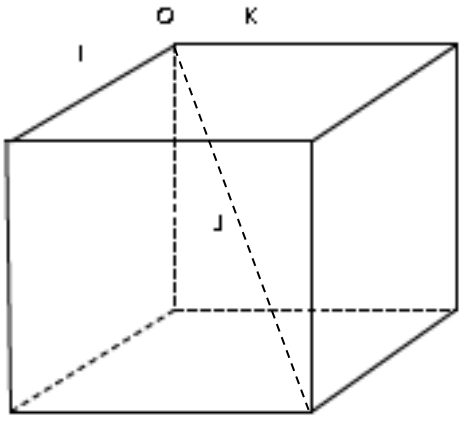

Figure 3-2. Visualized second derivative of a matrix

In automatic differentiation, all the first and second derivatives are stored sparsely. sparse function is used to create a sparse matrix for derivatives or transform the original matrix into a sparse matrix. There are three different ways to use the sparse function:

1) sparse (A)

The sparse function acts like a convertor, directly converting a matrix A into a sparse matrix. The example below has created one diagonal matrix. Then, it uses the sparse function to reconstruct the matrix. The result has been shown below in Figure 3-3. The two numbers in each parenthesis are the indices of rows and columns where the elements are nonzero, respectively.

```
>> m=eye(5)

m =

   1   0   0   0   0

   0   1   0   0   0

   0   0   1   0   0

   0   0   0   1   0

   0   0   0   0   1

>> sparse(m)

ans =

  (1,1)       1

  (2,2)       1

  (3,3)       1

  (4,4)       1

  (5,5)       1
```

Figure 3-3. Results of the sparse function

In this example, the sparse function squeezes out all the zero elements, leaving only the nonzero elements. The dimension of the matrix does not change. The numbers in the bracket shows the number of the row and column where the nonzero elements are.

2) sparse (m,n)

In this way, m and n are scalars indicating the size of the matrix. The utilization of this function entails an all-zeros sparse matrix with the size of $m \times n$.

3) sparse (i, j, s, m, n)

S is the vector that stores all the nonzero elements. The vectors i and j indicate the position of nonzero elements. The final size of the sparse matrix is created by the sparse function.

Since the sparse matrix can only be stored as two dimensional forms and the second derivative is a multi-dimensional matrix as discussed before; it is necessary to convert this 3D matrix into a two-dimensional concatenation of each nxn matrix. Therefore there would be n stacks of 2D matrices that consist of the second derivatives of each element.

Outer Derivative

The outer derivative is $\frac{df}{dg}\frac{dg}{dx}$ as show in the chain rule section, which is provided by the unary or binary functions. For non-constant functions, the outer derivative gives non-zero values. According to the chain rule, the derivative of the function is the derivative of the outer function times the derivative of inner function. For a non-scalar independent variable, its inner derivative is a sparse matrix. The outer derivative can either be a scalar, a vector or a matrix. To make the derivatives dimensionally compatible, the outer derivative and the inner derivative can be reshaped into vectors. When zero elements are

multiplied with any variable, the result is always zero. That is why we only extract the non-zero values from the inner derivative. In this way, the function can save time that would be used to calculate the zero results. The “find” function is used to locate the nonzero elements in a matrix and returns the indices of the rows and columns in which the nonzero elements stored. And the “find” function returns a vector as an output.

Class Constructor Function

The class constructor function of class Hes is used to create objects of the class Hes. It is stored under the path: hes/@hes. The class object stores specific manners of the class as a structure data type. It contains six fields: value, derivative, s_nz, s_derivative, nderivs, s_nderivs. Suppose x is an object of class hes. The syntax of accessing the properties of x are x.value , x.derivative and x.s_derivative whose values are the value of x, the first derivative of x and the second derivative of x, respectively. These specific properties are all defined in the class constructor function.

The input of the class constructor function could be an object of the Hes class or a double precision variable. The output of the function is an object of class hes. The constructor function needs to check whether the input is an object of the class, because the input data type is ambiguous. If the input is not of class Hes, then the output value component equals the value of the input. The other properties of the output are empty matrices. If the input is an object of the class, the constructor function uses dot operation to get the output value.

Again, suppose x is a Hes class object. The command x.value can be used to get the value stored in the object. If x is a scalar, then the first derivative with respect to x will be 1. If the x is a matrix, the function needs to take the derivative with respect to the entire set of variables. Each element’s first derivative will result in a row vector. For example,

assume the dimension of the matrix x is M x N, and the number of variables is p. For each element in the matrix, the first derivative will be a 1xp matrix with only one non-zero element. Then, each of these row vectors is reshaped into the column vectors of a diagonal matrix.

The second derivative behaves differently. In the class constructor function, if the input variable is a scalar, the result of the second derivative will be the same as that of the first derivative. The general form for the second derivative is$\frac{d^2y}{dx^2}(\frac{dx}{dx})^2$, and $\frac{dx}{dx}$ is a square identity matrix is. When$\frac{dx}{dx}$ takes the derivative with respect to an independent variable, each row vector in the matrix will become a square matrix.

For example, x is a 2×2 matrix that can be written as

$$\mathrm{x} = \begin{bmatrix} \mathrm{x1} & \mathrm{x2} \\ \mathrm{x3} & \mathrm{x4} \end{bmatrix} \tag{3-1}$$

If the first derivative of this matrix is

$$\nabla \mathrm{x} = \begin{bmatrix} \frac{\mathrm{dx1}}{\mathrm{dx}} & \frac{\mathrm{dx2}}{\mathrm{dx}} \\ \frac{\mathrm{dx3}}{\mathrm{dx}} & \frac{\mathrm{dx4}}{\mathrm{dx}} \end{bmatrix} \tag{3-2}$$

Then the second derivative of this matrix will be

$$\nabla^2 \mathrm{x} = \begin{bmatrix} \frac{\mathrm{d}}{\mathrm{dx}}\frac{\mathrm{dx1}}{\mathrm{dx}} & \frac{\mathrm{d}}{\mathrm{dx}}\frac{\mathrm{dx2}}{\mathrm{dx}} \\ \frac{\mathrm{d}}{\mathrm{dx}}\frac{\mathrm{dx3}}{\mathrm{dx}} & \frac{\mathrm{d}}{\mathrm{dx}}\frac{\mathrm{dx4}}{\mathrm{dx}} \end{bmatrix} \tag{3-3}$$

In the second derivative matrix, each element is the derivative of the first derivative with respect to the all of the independent variables. For example, the full form of $\frac{\mathrm{dx1}}{\mathrm{dx}}$ is$\left[\frac{\mathrm{dx1}}{\mathrm{dx1}}\frac{\mathrm{dx1}}{\mathrm{dx2}}\frac{\mathrm{dx1}}{\mathrm{dx3}}\frac{\mathrm{dx1}}{\mathrm{dx4}}\right]$. Then, taking the derivative of $\frac{\mathrm{dx1}}{\mathrm{dx}}$ with respect to the independent variable x again, it becomes a Hessian matrix as shown

$$\nabla^2 x_1 = \begin{bmatrix} \frac{d}{dx1}\frac{dx1}{dx1} & \frac{d}{dx2}\frac{dx1}{dx1} & \frac{d}{dx3}\frac{dx1}{dx1} & \frac{d}{dx4}\frac{dx1}{dx1} \\ \frac{d}{dx1}\frac{dx1}{dx2} & \frac{d}{dx2}\frac{dx1}{dx2} & \frac{d}{dx3}\frac{dx1}{dx2} & \frac{d}{dx4}\frac{dx1}{dx2} \\ \frac{d}{dx1}\frac{dx1}{dx3} & \frac{d}{dx2}\frac{dx1}{dx3} & \frac{d}{dx3}\frac{dx1}{dx3} & \frac{d}{dx4}\frac{dx1}{dx3} \\ \frac{d}{dx1}\frac{dx1}{dx4} & \frac{d}{dx2}\frac{dx1}{dx4} & \frac{d}{dx3}\frac{dx1}{dx4} & \frac{d}{dx4}\frac{dx1}{dx4} \end{bmatrix} \quad (3\text{-}4)$$

In the Hessian matrix above, there is only one non-zero element located in the first row and first column. The values of the other elements are zero since they are derived with respect to other variables. The second derivative matrix is a large matrix with four matrix blocks; each block is composed of the second partial derivatives of the element in the original matrix with respect to the independent variables. In order to generate the second derivative matrix, we have to use a loop to assign the values to the elements whose row and column indices are the same. The constructor function takes the product of the dimensions of the first derivative as the number of rows in the second derivative matrix. The number of columns in the second derivative matrix is the number of variables.

Unary Function

The automatic differentiation Hes creates unary functions, such as sin (x), cos (x), powers (x), which take only one hes class input, x, and return an object of class hes. The most common unary functions are the trigonometric functions. Take f (x) =sin (x) as an example where x is an independent variable. The first derivative of this function, $f'(x) = \frac{d\sin(x)}{dx}\frac{dx}{dx}$, $\frac{d\sin(x)}{dx}$ can be regarded as the outer derivative of the function, and $\frac{dx}{dx}$ as the inner derivative. In all unary functions, the number of independent variables determines the output size of the inner derivative. This unary function is defined in MATLAB as:

```
function y = sin(x)
```

Where x is the input and y is the output. This function defines all the properties of y, including the value, first derivative, second derivative and number of derivatives. The first derivative component of y is $\frac{dy}{dx}\frac{dx}{dx}$,which equals $\cos(\mathrm{x})$, and the second derivative component is $\frac{d^2y}{dx^2}$, which is $-\sin(\mathrm{x})$. The inner derivative is not evaluated in the unary function, but is defined in the constructor function. After defining all the properties of y, y is a structure. In automatic differentiation, there are only two types of inputs: double precision variables and the class object. As a result, the unary function has to call function "class" to convert y into an object of class.

```
y = class(y,'hes');
```

Next a randomized 2×3 matrix, x, is used as an input to test the unary function to see the output. The following graph, Figure 3-4, shows the value of x. After evaluating x in the Hes constructor function, the function returns an object of class, y. The graph shows the details of y's properties, including the value of y, the dimensions of y's first derivative, the value of the first derivative, dimensions of y's second derivative and the value of y's second derivative. The difference between the dimensions of first derivative and the second derivative, how the dimensions change from the oringinal matrix to the first derivative and to the second derivative are very clearly shown in Figure 3-4. Since the function y=sin(x) is needed to be evaluated, the program overloads the unary function sin(x). The results shown in the graph demonstrate that the dimensions of the first and second derivatives. The total number of derivatives do not change, because of the unary function's overloading. Unary function only has one input, its number of derivatives depends on the input's property of number of derivatives. Since the first and second derivative matrix dimension is enlarged, compared to the original matrix, the elements are

allocated, and the allocation of nonzero values in the first and second derivatives are also shown in the result.

```
>> y=hes(x)

HES Object

Value(s) of HES Object:
  0.9649   0.9706   0.4854
  0.1576   0.9572   0.8003

first Derivative
   6    6


Derivative(s) of HES Object:

sizeDerivative =

   2    3    6

  (1,1)       1
  (2,4)       1
  (1,8)       1
  (2,11)      1
  (1,15)      1
  (2,18)      1


Number of Derivatives:
   6

second Derivative

size_sDerivative =

   2    3    6    6

  All zero sparse: 2-by-108


Number of Derivatives:
   6
>> sin(y)

HES Object

Value(s) of HES Object:
  0.8220   0.8252   0.4665
```

Figure 3-4. Result of the function “sin” with the ramdon input value

```
  0.1570   0.8176   0.7176

first Derivative
   6    6


Derivative(s) of HES Object:

sizeDerivative =

   2    3    6

  (1,1)      0.5695
  (2,4)      0.9876
  (1,8)      0.5648
  (2,11)     0.5758
  (1,15)     0.8845
  (2,18)     0.6965


Number of Derivatives:
   6

second Derivative

size_sDerivative =

   2    3    6    6

  (1,1)     -0.8220
  (2,22)    -0.1570
  (1,44)    -0.8252
  (2,65)    -0.8176
  (1,87)    -0.4665
  (2,108)   -0.7176


Number of Derivatives:
   6
```

Figure 3-4. Continued

Constant Variable Function

When evaluating a constant input, the program will automatically call the hesconstant function. Hesconstant evaluates the constant value and returns an object of the class hes. In the constant function, the input could be an object of class, or a double precision variable. If input is an object of class, then input stores all the properties of class, including value, derivative, s_nz, s_derivative, nderivs, s_nderivs. The output is the

deep copy of the input. All the values of x and y's properties are the same. If the input is a double precision variable, then the value of input equals the output value component. Because x is a constant, the first and second derivatives of the output are all zero sparse matrix. Since input is a double precision variable, it does not have the properties of the class. Therefore, the derivatives of the output will be empty matrices.

Binary Function

The binary function is a function with two input variables. There are several binary functions in automatic differentiation, such as the plus and multiplication function. The Table 3-1 explains the gradient and hessian pattern of binary functions.

Table 3-1. The first and second pattern of different unary operations

Equation	First derivative	Second derivative
$z = x + y$	$z = x' + y'$	$z'' = x'' + y''$
$z = x - y$	$z = x' - y'$	$z'' = x'' - y''$
$z = x * y$	$z = x' * y + x * y'$	$z'' = x'' * y + 2 * x' * y' + x * y''$
$z = x.* y$	$z = x'.* y + x.* y'$	$z'' = x''.* y + 2 * x'.* y' + x.* y''$
$z = \frac{x}{y}$	$z' = \frac{x' * y - x * y'}{y^2}$	$z'' = \frac{(x'' * y - x * y'') * y - 2(x' * y - x * y')}{y^3}$
$z = x./y$	$z' = \frac{x'.* y - x.* y'}{y^2}$	$z'' = \frac{(x''.* y - x.* y'').* y - 2(x'.* y - x.* y')}{y^3}$

Addition and Subtraction

Addition and subtraction functions in AD are used to add or subtract two variables and take derivative value of the result. They take in two input variables and return one object of class hes. In a matrix, when doing addition or subtraction, the corresponding two entries are added or subtracted element by element. The input variables should have

identical sizes, or one of them should be a scalar. Addition and subtraction have the following form in MATLAB,

```
z=function_name(x,y)
```

Where x and y are the two inputs to the function, function_name, and z is the output of the function. For this function, there will be three different cases: only x is an object of the class; only y is an object of the class; or both of x and y are objects of the class. An example of the possible inputs for plus() is shown in Figure 3-5 below.

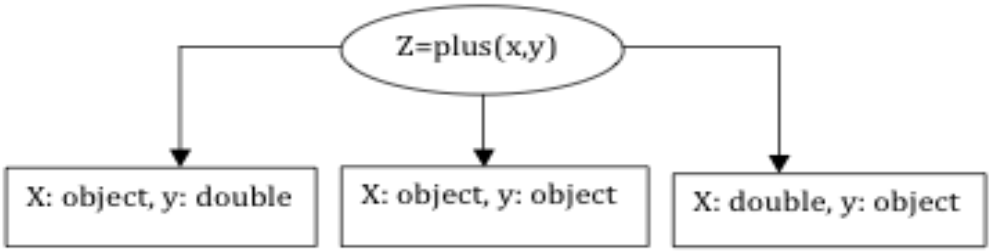

Figure 3-5. All the input cases for the plus function

Case1: x is an object of class and y is a double precision variable. In this case, x has the properties: value, derivative, s_nz, s_derivative, nderivs, and s_nderivs. As y is a double precision variable, the function does not evaluate at y.

The variable x could be a constant or an equation. If x is a constant, the constant function will be called to evaluate all of its properties and then return the property values to the output. The value component of the output of plus(x,y) is equivalent to the addition of the value components of x and y . The derivative of the output z is found by implementing the chain rule recursively until the evaluation reaches the derivative of the innermost function, which is taken with respect to the independent variable. Since y is not an object of the class the binary function only takes the derivative of x. Thus, the derivative of the output is equivalent to the derivative of x. The number of independent

variables in x decides the number of derivatives of z. Since x is a constant variable, the constant function will be overloaded. The constant function will generate the first and second derivatives of x, and then pass their values to the plus function.

There're also two scenarios when x is an equation. First, when x is a unary equation, such as a trigonometric function, it will first call the unary function to pass the outer derivative to the composite function. Then the composite function computes the final derivative of x. Second, when x is a binary equation, it will first call the binary function. As the binary function is composed of two more unary functions, it has to call the relevant unary functions to give the derivative values until it arrives at the innermost derivative.

Case 2: If y is an object of class "hes" and x is a double precision variable, then the situation is reversed from the Case 1. Now the plus function does not have to evaluate x, but has to define y's properties and pass them to the output z.

Case 3: if both x and y are objects of the class, the output z's value component depends on both x and y's value components: z.value=x.value+y.value. In this case, x or y could be a scalar or a matrix. If x is a scalar and y is a matrix. The first derivative of x would be a row vector, while the first derivative of y will be a multiple dimensional matrix. When adding the value component of x and y, each element in y's value component will add with x. The output size will be the same as y. When doing addition of two derivatives, we need to use function "repmat".

Repmat is a function, which takes a matrix as an input block and returns a larger matrix with a specified number of rows and columns. The input block can be an array or a matrix. The number of rows and columns of these blocks to be placed in the output matrix are given as inputs to the function. The repmat function converts the derivative

component of x into the same size as derivative component of y. Once the dimensions of x and y are compatible, x and y can be safely added together. In the following graph, we have created one random matrix with the size of2×3, $x \in R^{2\times3}$. We need the “repmat” function to create a two by three tiling of copied input matrix. After using the “repmat” function, the result will be a big matrix with repeated blocks.

```
>> x=rand(2,3)

x =

   0.9572   0.8003   0.4218
   0.4854   0.1419   0.9157

>> repmat(x,2,3)

ans =

   0.9572   0.8003   0.4218   0.9572   0.8003   0.4218   0.9572   0.8003   0.4218
   0.4854   0.1419   0.9157   0.4854   0.1419   0.9157   0.4854   0.1419   0.9157
   0.9572   0.8003   0.4218   0.9572   0.8003   0.4218   0.9572   0.8003   0.4218
   0.4854   0.1419   0.9157   0.4854   0.1419   0.9157   0.4854   0.1419   0.9157
```

Figure 3-6. Result of the function “repmat” with input of random matrix

If x and y are both matrices, the derivative component of output z is the addition of both the directional derivative component of x and y. When function adds x and y’s derivative, it must check whether x and y have the same independent variables. If they are taken derivative with respect to different independent variables with different sizes and the dimensions of variables will be different, that will lead to different dimensions of x and y’s derivative components. Also, the derivative of output z cannot be realized, because when z takes derivative, it can only do so with respect to one variable each time. For plus, both sides of ‘+’ sign should have the same dimension. If taking derivative of x

and y with respect to the same variable, the sizes of both sides of '+' sign are compatible. Thus, the number of derivative is the number of variable x or y are taking derivative with respect to. The number of variable for second derivative to be with respect to is the same.

Element-Wise Operation

Element-wise operations perform computation on the corresponding elements of the two matrices. They have the mathematical form of $z = x.*y$. In the element-wise multiplication, in order to be operationally compatible, both x and y must have the same size, except when one of them is a scalar. When x or y is a scalar, the scalar will multiply with each element in the matrix without changing the dimension of the matrix.

Figure 3-7 illustrates the form of the element-wise operation: take two variable inputs and return one output. In Figure 3-7, there are three cases existing for function "times": x is a double precision variable, y is an object of class; x is an object of class, y is double precision variable; x and y are both class objects. MATLAB's intrinsic function "isa" is used to determine whether x or y is an object of the class.

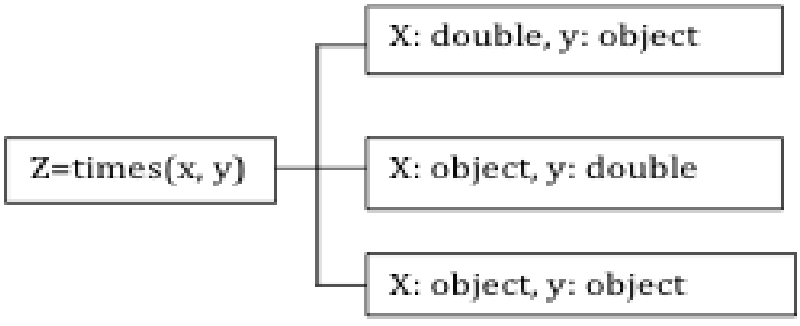

Figure 3-7. Different possibilities of inputs for function "times"

Case1: When x is a double precision variable and y is an object of class, the output, z, is obtained by adding the numerical x's value with the value property of y. The size of

y's first derivative is determined by the computation of $\frac{dy}{dx}\frac{dx}{dx}$, and it should be the dimension of y times the number of independent variables. The first derivative of z is written as $z' = x.* y'$ where x and y could be a scalar or a matrix.

If x is a double precision scalar, the output's first derivative equals x times each element in y's first derivative matrix. The same principle has to be implemented when computing the second derivative of the output.

If y is a scalar, taking derivative of y with respect to the independent variables will result in a row vector. Suppose the independent variable vector is $a \in R^n$, the first derivative of y is $[\frac{dy}{da_1}, \frac{dy}{da_2}, \ldots, \frac{dy}{da_n}]$. For the second derivative, the result will be a square matrix with the size of $n \times n$. Since x is not a scalar and the first derivative of y is a row vector, in order to multiply each element in x by each element in y's first derivative, x (:) is used to convert x into a column vector. The second derivative of output would be

. Much like calculating the first derivative, algebraic multiplication is applied to $z'' = x(:) * y''$ that computes the second derivative of z.

If both x and y are matrices, the equations of z's first and second derivatives are

$$z' = x.* y' \tag{3-5}$$

$$z'' = x.* y'' \tag{3-6}$$

The first derivative of y is a directional derivative: the derivative of y is the rate change of y with respect to a certain direction. Therefore, the function uses commands: repmat (x (:), 1, number of derivatives), to convert x into [x (:)…..x (:)], which has the same dimension of the first derivative of y. Suppose n represents the number of derivatives, the total number of x (:) is n and x (:) is a column vector with dimensions of 1*the product of the

dimension of x. When doing the second derivative, the same method above is used except the tiling dimension of input to repmat would be 1 and n^2.

Case2: When x is an object of class and y is a double precision variable the methods from case 1 are used with the roles of x and y reversed.

Case3: When x and y are both class objects, the output's value will be generated from the product of x and y's value components. The derivative of the output follows the product rule and can be written as:

$$z' = x'.* y + x.* y' \tag{3-7}$$

$$z=\text{x}''.* y + 2 * x' * y' + x.* y'' \tag{3-8}$$

If x is a scalar then the first derivative of z can be divided into two terms: term1=x.derivative.*y.value, term2=x.value.*y.derivative. Suppose the independent variable is a n-valued entry. For term1, the first derivative of x is a row vector with size n. Each element of x's first derivative is multiplied with the corresponding element of y. For term2, the value component of x is a scalar. Whether or not the first derivative of y is a matrix or a scalar, the first derivative of each element in y will multiply by x. The result of the computation will be the same size as the size of the first derivative of y. If y is a scalar then the methods used in case 1 are duplicated with the roles of x and y reversed.

If x and y are both matrices then the sizes of x and y need to be the same for element-wise operation. Suppose the dimensions of x and y are m x n and the size of the independent variable vector is p. The dimensions of term one above will be m x n x p, that is$R^{m\times n\times p}$. The direction of the first derivative is taken along the columns. Likewise, tiling the column vector y (:) follows the same direction as the derivative direction. After reshaping into a column vector, y is duplicated into a matrix with the number of columns

as the size of the derivative of x. When multiplying y with x's first derivative, the nonzero elements are computed. A similar method is used to deal with term 2.

The equation for the second derivative is $z'' = x''.*y + 2*x.*y + x.*y''$. The principle of taking the second derivative is similar to the principal of taking the first derivative. The only difference of the two computations is that there is an extra term in the computation of second derivative, which is 2*x.*y. The size of the extra term is m x n x p. However, the dimensions of the other terms in the equation are all m x n x p x p. In order to continue the whole computation, the sizes of all the terms should be compatible. So the binary function uses the "find" function to extract the nonzero elements in the second term and return their locations. "Find" returns the nonzero elements as a vector. After extracting the nonzero values from the matrix, according to their original location in the matrix, a sparse matrix is used to reorganize those values into the same size as first and third term without changing the location of the nonzero values.

All of the properties are stored in the output z. Due to features of operator overloading, the input can only be a double precision variable or an object of the class hes. And the function needs to return the data type of the class object. Therefore, in the end of the code, function "class" is utilized to convert z into an object of class. Figure 3-8 gives all the three cases when x and y are both class objects. These three cases are 1) x is a scalar and y is a matrix. 2) x is a matrix and y is a scalar. 3) x and y are both matrices. Under these three cases, the program carries out different methods to compute the derivative of output z. As in case two and three, the equations for computing first derivative value of z are different.

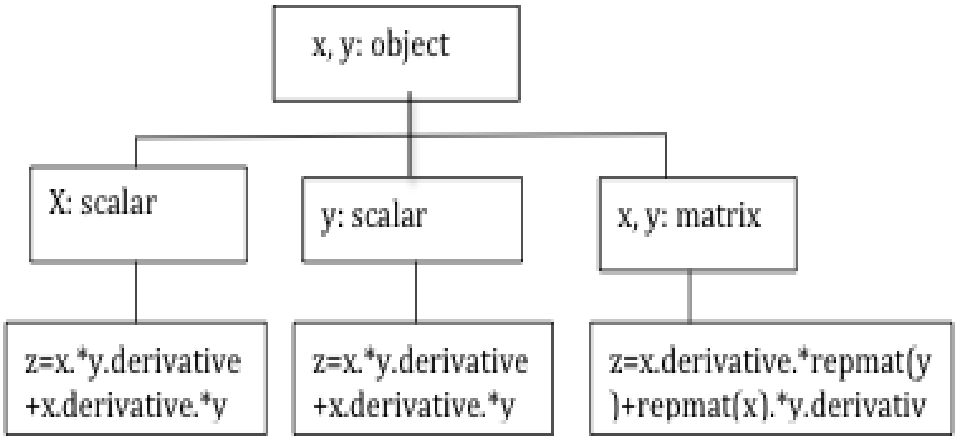

Figure 3-8. Different cases when x and y are both objects

Matrix Operation

There are some matrix operations in automatic differentiation, such as mtimes, mrdivide, mpower. Different from element-wise operation, matrix operation is the linear algebraic operation of matrix.

$$z = x_{ij} * y_{jk} \tag{3-9}$$

Equation 3-9 is about the matrix multiplication. The elements in row vector of x are multiplied with the associated elements in the same column in y. As indicated in the equation, the number of x's columns should be equal to the number of y's rows. The following equation shows the function mtimes' expression in MATLAB:

```
function z=mtimes(input1,input2)
```

If x or y is a scalar, then the operation process is the same as element-wise multiplication. The three cases of x and y as matrices are shown below in Figure 3-9. Specially, there are also there cases under x and y are matrices.

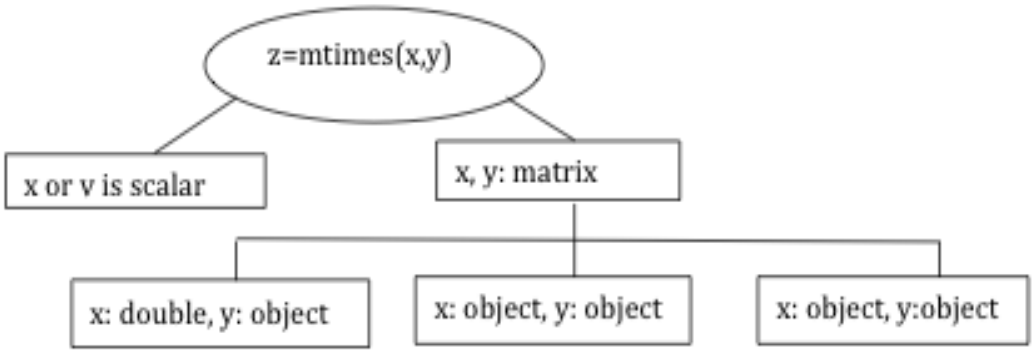

Figure 3-9. Different possibilities of inputs for function “mtimes”

If x is a double precision variable and y is an object of class then the first derivative of the matrix multiplication is $z' = x * y'$. Suppose x is a matrix with the dimensions of $m \times n$ and y is the matrix with the dimensions of $n \times p, y \in R^{n \times p}$, where q is the number of independent variables. So the dimension of y’s first derivative is $np \times q$. In order to be compatible, the derivative of y should be reshaped into the dimensions $n \times pq$. The first derivative of y is a matrix that contains $n \times p$ blocks. Each block is a row vector with dimension of $1 \times q$, which has only one nonzero entry. In order to multiply x and y’, the first derivative of y is converted into another matrix, whose number of rows is the same as the number of columns of x. The second derivative of matrix multiplication is $z'' = x * y''$. Each element in x is multiplied with the corresponding element in the second derivative of y.

If x is an object of class “hes” and y is a double precision variable then the value component of the output, z, is determined by the product of the value components of x and y. The first derivative of z is z.derivative=x.derivative*y. Suppose x and y have the same dimensions as in case one. For the first derivative of x, its dimensions are $m \times n \times q$. However, the dimensions of y are $n \times p$. The dimensions of the two matrices do not fulfill

the requirement of matrix multiplication. The function utilizes the “regroup” function to reshape x’ into a matrix with the dimensions of $\mathrm{mq} \times \mathrm{n}$. The function “regroup” divides different variable’s derivative into different groups, and concatenate those groups into one matrix.

If the variables x and y are both class objects, then both x and y have the class properties: value, derivative, s_nz, s_derivative, nderivs, s_nderivs. The output, z’s value comes from x’s value component and y’s value component. The first and second derivative of output z can be written as,

$$z' = x' * y + x * y' \tag{3-10}$$

$$z = x'' * y + 2 * x * y + x * y'' \tag{3-11}$$

In this case, when computing the first derivative, the function utilizes the results from case 1 and case 2 to compute the first and second term by applying the relevant rule, such as the chain rule and product rule. When computing the second derivative, the way to calculate the first and third term comes from the first and second case. When calculating the second term, the dimensions of x and y are increased by n, where n is the number of independent variables. First, the dimensions of the first derivatives of x and y are converted into compatible sizes. After getting the product of x’ and y’, the nonzero elements are extracted from the product with their row and column index. Then a sparse function is used to create a matrix with the same dimensions as the first or third term.

If the matrix has p stacks of a $\mathrm{m} \times \mathrm{n}$ matrix, the regroup function is to reshape the function z_sderivative=sterm1+2*sterm2+sterm3; this code is used to add the three terms together. For example, if $\mathrm{x} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, y is the class object with independent variable x, the following graph shows the result of the multiplication of these two matrices. It can be

revealed that after multiplication, the dimensions for the first and second derivative will be $R^{2\times2\times4}$ and $R^{2\times2\times4\times4}$.

```
>> sin(y)*cos(y)

HES Object

Value(s) of HES Object:

  -0.4455  -0.9445
   0.8255   0.4360

first Derivative

   2    8



Derivative(s) of HES Object:

sizeDerivative =

   2    2    4

  (1,1)    -0.4161
  (2,1)    -0.1187
  (1,2)     0.4120
  (1,3)    -0.1283
  (2,3)    -0.4281
  (2,4)     0.6471
  (1,5)    -0.2248
```

Figure 3-10. Results of the matrix operation

Logic Functions

There are multiple logic functions in the AD program, such as ge.m, eq.m, and gt.m. These functions are used to compare each element in two inputs, and return logic value of 1 or 0. Whether the two input variables are scalars or matrices, the output will be a variable with the same size as the inputs. Each element in the output will be the comparison result of corresponding element in two inputs. However, the function needs to know whether the inputs are objects of class “hes” or double precision variables. If the input is an object of class “hes,” dot reference is required to get the value component of the input.

Constant Functions

The constant function is used to create a constant value for the output from a given dimension. They can take in multiple inputs and return a class object. The inputs indicate the final dimensions of the output. There are some constant functions in AD, such as zeros.m, ones.m, and hescontant.m.

Functions, such as “ones” and “zeros” have different numbers of inputs. When the input is a scalar, it creates a square matrix with the size provided by the input. When the input is a vector the constant function returns a multi-dimensional matrix. The size of each dimension is stored in the input vector and the first element in the vector indicates the size of first dimension. For those types of functions, the input can have variable data types. However, the function can have only one input which can be a scalar or a vector, or can have multiple inputs. All of these cases are shown in Figure 3-11. In Figure 3-11, the input can be one or more than one, it can be a vector or scalar.

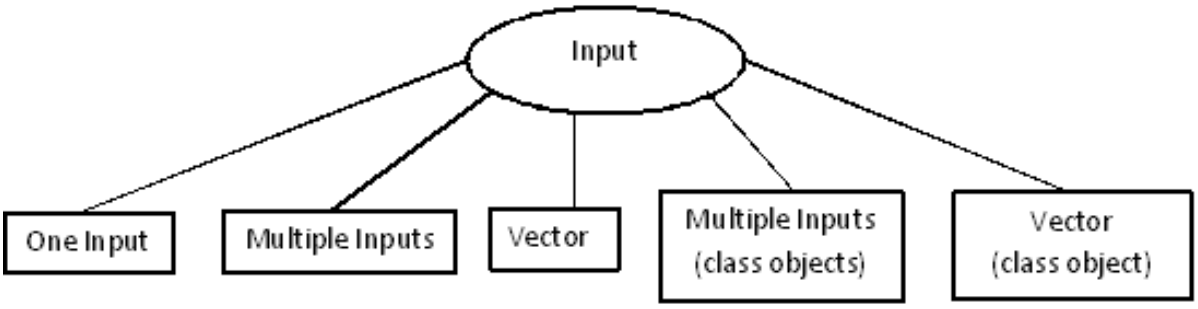

Figure 3-11. Different input possibilities for the constant function

The constant function uses a MATLAB build-in function “nargin” to return the number of inputs for the function. When there is only one input, nargin=1. The function varargin contains all the optional inputs from the function “ones”. For example, the input in ones.m is the size indicator, which is different from other functions.

```
function z=ones(varargin)
```

Therefore, z.value=ones (input.value), where input is an object of class. For multiple input situations, varargin cannot be directly used as the size indicator. The function ones cannot extract the value stored in varargin directly since it is a cell array. In order to get the value stored in varargin, a loop is used to assign every value to a variable vector. Suppose z=ones (d), where d is a vector containing the dimensions [m, n, o, p...]. The output matrix, z, is a constant matrix in which all the elements are one. The derivative of this ones matrix is zero. The size of the first derivative is the product value of the size of the input and the number of the independent variables. The size of second derivative matrix is the same as the first derivative.

Shaping Function

The shaping function is used to change the size of the matrix, and returns the final matrix after reshaping, which is an object of class hes. There are some shaping functions

defined in AD, such as ctranspose.m, horzcat.m, repmat.m, reshape.m, vertcat.m, transpose.m.

For example, the “reshape” function is used to return a matrix with a desired size. In the reshape function, there must be at least two input variables: one is the original matrix and the other one is the size of the matrix after reshaping.

We use the general form to express the function in MATLAB: z=reshape(x, varargin); Varargin indicates the size vector and cannot be zero. Since each element’s value in x does not change after reshaping, the first derivative of the output, z, equals the first derivative of x. However, the location of each value in the matrix changes after reshaping. The dimensions of first and second derivative of output z are the same as the dimensions of the first and second derivative of x.

Array Referencing and Assignment Functions

In addition to the functions above, the AD tool “Hes” needs to deal with array referencing and assignment. Array referencing and assignment functions are used to take in three inputs, and return one output, which is an object of class hes. The overloaded versions of these functions are subsasgn.m and subsref.m. Subsasgn is the function that uses a script to assign the value of an input to an output.

$$y = \mathrm{x(i)} \tag{3-12}$$

The equation above explains how subsasgn works. In MATLAB, the function will be written as follows,

```
function x = subsasgn(x,s,y)
```

Usually, function subsasgn has three inputs: the variable itself, the indicator, and y. X assigns its value to y. The variable s is a structure that stores the indices of the value and the type of referencing. For automatic differentiation, s should be a 1 by 1 structure so

that there are no multiple indexes. At the same time, there is only one type of value assignment, which is '()'. MATLAB has its own intrinsic string tester, strcmp. Strcmp is used to compare two strings to check whether the type of the index is '()' or not. If it is of the type of '()', the function can continue evaluating. It is very important to prevent indices from exceeding the size of the provider x. The output of function "subsagn" is x, where x is an object of class. The function defines all the properties of x, including value, derivative, s_nz, s_derivative, nderivs, s_nderivs. Function "subsasgn" assigns only the referenced part of x's value to y. For example, if the index is i, then the value component of y describes the value component of x (i). The first and second derivative of the output y is equivalent to the derivative of the value of x(i). The value of x does not change because of the assignment.

Composite Function

The composite function aims to combine the outer and inner derivatives and returns the final result of the overall derivative by applying the chain rule, like a compiler. The form of the function in MATLAB is written as:

```
function y =compositeDerivative(x,y,outerDerivative1,outerDerivative2)
```

The first two inputs of this function are class objects x and y. The last two inputs stand for the first and second outer derivative of the function, respectively. The unary function provides the first and second outer derivatives for the composite function.

For y(x), the first derivative is$\frac{\mathrm{dy}}{\mathrm{dx}}\frac{\mathrm{dx}}{\mathrm{dx}}$. The second derivative of y(x) is

$$\frac{\mathrm{d}}{\mathrm{dx}}\left(\frac{\mathrm{dy}}{\mathrm{dx}}\frac{\mathrm{dx}}{\mathrm{dx}}\right)=\frac{\mathrm{d}^2\mathrm{y}}{\mathrm{dx}^2}\left(\frac{\mathrm{dx}}{\mathrm{dx}}\right)^2+\frac{\mathrm{dy}}{\mathrm{dx}}\frac{\mathrm{d}^2\mathrm{x}}{\mathrm{dx}^2}. \tag{3-13}$$

So the composite function utilizes this relationship between outer derivative and inner derivative to propagate the derivative of the function. In the composite function, y is the

output, and the type of y, which is returned by the composite function, is an object of the class. So y's six properties are needed to be defined in the composite function.

The outer derivative is passed to composite function, which is $\frac{\mathrm{dy}}{\mathrm{dx}}$ or $\frac{d^2y}{dx^2}$, the dimension of outer derivative would be the product of the dimensions of x and y. For $\frac{\mathrm{dx}}{\mathrm{dx}}$, the dimension of the inner derivative would be the square of dimension of the independent variable x. As discussed before, the first derivative is a diagonal matrix, and the second derivative is a triagonal in a cubic. In order to speed up the computation, we only need to take care of the non-zero elements in the chain rule equation. For the sparse matrix with many zero elements, it will occupy a lot of storage space. In order to reduce the storage size and speed calculation, all the derivatives returned should be stored sparsely.

A sparse matrix is the two-dimensional matrix removes the zero elements from a matrix in which most of the elements are zero. In order to store the non-zero value in a sparse matrix, one should first extract the non-zero elements as well as their indices. Then the matrix is reconstructed using the sparse function.

CHAPTER 4
EXAMPLES OF OPTIMAL CONTROL PROBLEM

This chapter begins with a brief introduction about nonlinear programming problem and the NLP solver used in the following example is presented. Next, the main concentration of this chapter is on using automatic differentiation for four examples. The first example is a test example, aims to test whether the result from AD tool is correct or not. The following three examples organized from the simplest one to the most complicated one. Those examples utilize MATLAB build-in function “fmincon” to find the optimization result of the cost function. This work chooses the algorithm to be an interior point algorithm, which accepts the user supplied gradient and hessian. Through the comparison of these three examples’ result, it is apparently to see how AD affects the speed for propagating the result and in what kind of problem AD will show its obvious advantages.

Optimization

Optimization is choosing a best element in an allowed set to minimize or maximize a function. Usually, this function is called objective function.

Optimization Problem

Generally, the optimization problem has the form of

Given a function: f(x) from set A to real numbers

Sought: x_0 in A that minimizes f(x), that is $f(x_0) \leq f(x)$; Or maximize f(x), that is $f(x_0) \geq f(x)$

Where f(x) is the objective function, solution x_0 is called optimal solution

Nonlinear Programming

The optimization problems with the nonlinear object functions or constraints can be transformed into nonlinear programming problems. Nonlinear programming is one method to solve optimization problems through some inequality and equality constraints.

Usually the form of Nonlinear Programming is as follows:

Minimize or Maximize f(x)

Subject to $\mathrm{E}(\mathrm{x}) = 0$

$$\mathrm{G}(\mathrm{x}) \geq 0$$

$$\mathrm{L}(\mathrm{x}) \leq 0$$

F(x) is the cost function. It is the target function that we need to get the optimal result. Or, we can call it as goal function. E(x) is the equality constraint. G(x) is the upper bound constraint; L(x) is the lower bound constraint. With these constraints we can get the condition for minimization or maximization of the cost function. Then the Lagrangian function could be written as

$$\mathrm{L}(\mathrm{x}) = \mathrm{f}(\mathrm{x}) + \sum_{i=1}^{m} \lambda_i C_i(x) \qquad (4\text{-}1)$$

Where f(x) is the cost function, C(x) is the constraint function, λ is a Lagrange multiplier. The second derivative of Lagrangian function is shown as below

$$\nabla\mathrm{L}(\mathrm{x}) = \nabla^2\mathrm{f}(\mathrm{x}) + \sum_{i=1}^{m} \nabla^2 \lambda_i C_i(x) \qquad (4\text{-}2)$$

In the following examples, the MATLAB's optimization toolbox: fmincon is used as a NLP solver. In fmincon, interior point algorithm is used. For interior point algorithm, Hessian Matrix of the Lagrangian function and the first derivatives of the objective function and constraint function. And interior point algorithm can accept user-supplied Hessian value.

Otherwise, the hessian value is provided by either dense quasi-Newton approximation, or limited-memory, large-scale quasi-Newton approximation or finite differences.

Example 1

Problem Statement

This problem is used to test the Jacobean and Hessian generated by automatic differentiation, which is called arrowhead function. The function can be written as

$$f_1(\mathrm{x}) = 2x_1^2 + \sum_{i=1}^{n} x_i^2,\ f_j(x) = x_1^2 + x_j^2, (j = 2, \dots, n) \qquad (4\text{-}3)$$

Where f maps from $R^n \to R^n$ and x is the variable with n entries. The Jacobean of this function has a special form, which can be visualized as

$$\mathrm{Jac}\big(\mathrm{f(x)}\big) = \begin{pmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & \\ \bullet & & \bullet & \\ \bullet & & & \bullet \end{pmatrix}$$

In this example, automatic differentiation provides the first derivative value of f(x). Consider a randomized matrix x as an input to the vector function above, who has the size of 2×3. And the number n is 6.

Result

From the result showed in Figure 5-1, the dimension of the Jacobian matrix is a 6x6 sparse matrix. The full matrix of the first derivative of the function is shown at the end. In this matrix, most of the elements are zero except for the arrowhead direction, which stores the nonzero first derivative values. It fulfills the characteristic of the arrowhead function. That means the first derivative generated from AD tool is correct.

```
>> x=rand(2,3)

x =

    0.8143    0.9293    0.1966
    0.2435    0.3500    0.2511
>> z=arrow(m)

HES Object

Value(s) of HES Object:
    3.1362
    0.7224
    1.5266
    0.7855
    0.7017
    0.7261

first Derivative
     6     6


Derivative(s) of HES Object:

sizeDerivative =

     6     1     6

  (1,1)       4.8857
  (2,1)       1.6286
  (3,1)       1.6286
  (4,1)       1.6286
  (5,1)       1.6286
  (6,1)       1.6286
  (1,2)       0.4870
  (2,2)       0.4870
  (1,3)       1.8585
  (3,3)       1.8585
  (1,4)       0.7000
  (4,4)       0.7000
  (1,5)       0.3932
  (5,5)       0.3932
  (1,6)       0.5022
  (6,6)       0.5022
```

Figure 4-1. Result of the arrowhead function

```
Number of Derivatives:
     6

second Derivative

size_sDerivative =

     6     1     6     6

   (1,1)         6
   (2,1)         2
   (3,1)         2
   (4,1)         2
   (5,1)         2
   (6,1)         2
   (1,2)         2
   (2,2)         2
   (1,3)         2
   (3,3)         2
   (1,4)         2
   (4,4)         2
   (1,5)         2
   (5,5)         2
   (1,6)         2
   (6,6)         2


Number of Derivatives:
     6

>> full(z.derivative)

ans =

    4.8857    0.4870    1.8585    0.7000    0.3932    0.5022
    1.6286    0.4870         0         0         0         0
    1.6286         0    1.8585         0         0         0
    1.6286         0         0    0.7000         0         0
    1.6286         0         0         0    0.3932         0
    1.6286         0         0         0         0    0.5022
```

Figure 4-1. Continued

Example 2

Problem Statement

Consider the problem

Minimize $\frac{1}{2}(x_1^2 + x_2^2 + x_3^2)$

Subject to$x_1 + x_2 + x_3 = 3$

This example is a minimization problem, where $\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3$ are three decision variables, $F = \frac{1}{2}(x_1^2 + x_2^2 + x_3^2)$ is the objective function and $g = x_1 + x_2 + x_3 - 3 = 0$ is the equality constraint function. In this problem, the objective function is needed to be minimized through finding the suitable decision variable values. In order to get the optimization result, the first derivative of objective function and constraint functions are needed to compute with respect to all the variables. Also, we need to compute the second derivative of the Lagrangian function. There are three variables,x_1, x_2, x_3 in this problem, and only one output. Suppose the output is y, the Jacobean matrix would be

$$\left[\frac{dy}{dx_1} \quad \frac{dy}{dx_2} \quad \frac{dy}{dx_3}\right] \tag{4-4}$$

Hessian matrix for this problem would be

$$\begin{bmatrix} \frac{d^2y}{dx_1^2} & \frac{d^2y}{dx_1x_2} & \frac{d^2y}{dx_1dx_3} \\ \frac{d^2y}{dx_2dx_1} & \frac{d^2y}{dx_2^2} & \frac{d^2y}{dx_2dx_3} \\ \frac{d^2y}{dx_3dx_1} & \frac{d^2y}{dx_3dx_2} & \frac{d^2y}{dx_3^2} \end{bmatrix} \tag{4-5}$$

As introduced in the previous chapter, the Lagrangian function is

$$\mathrm{L} = \mathrm{F}(\mathrm{x}) + \lambda^T g(x) \tag{4-6}$$

Substituting F as$F = \frac{1}{2}(x_1^2 + x_2^2 + x_3^2),\ \mathrm{g}(\mathrm{x}) = \mathrm{x}_1 + \mathrm{x}_2 + \mathrm{x}_3 - 3 = 0$. Then the Lagrangian Equation will become

$$\mathrm{L} = \frac{1}{2}(x_1^2 + x_2^2 + x_3^2) + \lambda^T(x_1 + x_2 + x_3 - 3) \quad (4\text{-}7)$$

The gradient value is the differentiation of cost function with respect to the variables

$$\nabla \mathrm{g} = \left[\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \frac{\partial F}{\partial x_3}\right] \quad (4\text{-}8)$$

The hessian value user or fmincon need to supply is the second derivative of Lagrangian equation with respect to the state

$$\nabla^2 L = \nabla^2 F + \nabla^2 \lambda^T C(x) \quad (4\text{-}9)$$

Usually, λ has an equality and inequality part of values. Where in this problem, since only the equality constraint function is presented, λ only has the equality nonlinear value.

Result

The initial value for the three variables: $x_1, x_2, x_3,$ are given as [100; 100; 100]. In this example, MATLAB function “fmincon” has been used as a NLP solver The algorithm “interior point” is used that user can provide derivative values of the objective function, the constraint function and the Lagrangian formula. If user does not provide the derivative values, function “fmincon” will generate the derivative through different differentiation methods. Here, finite difference used as the differentiation method for “fmincon”. Figure 5-2 shows the result propagated from MATLAB. This problem is a simple problem, with only three states. And it can easily find the state value and the lambda value from computation of the Lagrange formula even by hand. That’s why MATLAB does not need to give the iterations. The values of three states are [1,1,1], and the equal nonlinear value for the Lagrange multiplier is 1. The function value at this point is 1.5.

```
>> x

x =


    1.0000

    1.0000

    1.0000

>> lambda

lambda =


          lower: [3x1 double]

          upper: [3x1 double]

          eqlin: [1x0 double]

       eqnonlin: 1

         ineqlin: [1x0 double]

     ineqnonlin: [1x0 double]

>> fval


fval =
```

Figure 4-2. Result of the example two

Figure 4-3 is the comparison of three different derivative methods used in NLP solver to propagate the overall results. Those three different derivative methods are:

1) Finite differencing provides the first derivative value of the objective function and constraint function. It also provides the second derivative value of the Lagrangian function.

2) Finite differencing provides the first derivative value of the objective function. AD tool provides the second derivative value of the Lagrangian function.

3) AD tool provides the first derivative value of the objective function and the constraint function. It also provides the second derivative value of the Lagrangian function.

Here, in Figure 4-3, the overall time of the first two methods CPU takes to get the final optimization is similar to each other. However, the third method, AD provide both the first and second derivatives to the relative functions, is much faster.

	Finite Difference	1st derivative(AD)	1st & 2nd derivative(AD)
CPU time	1.464509s	1.4525348s	1.4158044s
State value	1 1 1	1 1 1	1 1 1
Function value	1.5000	1.5000	1.5000
Eqnonlin	1	1	1

Figure 4-3. Comparison of three different derivative methods

Example 3

Problem Statement

Min $f = x_1^2 + x_2^2$

Subject to the constraints

$$0.5 \le x_1$$
$$-x_1 - x_2 + 1 \le 0$$
$$-x_1^2 - x_2^2 + 1 \le 0$$
$$-9x_1^2 - x_2^2 + 9 \le 0$$
$$-x_1^2 + x_2 \le 0$$
$$-x_2^2 + x_1 \le 0$$

Where equations one and two are linear constraints, equation three, four, five and six are nonlinear constraints. So in the “fmincon” function, we need to set up A= [-1,-1], b=-0.5; lb=-1. In this problem, the Lagrange formula is

$$\mathrm{L} = \mathrm{F} + \sum_{i=1}^{m} \lambda_i C(x)$$

$$= x_1^2 + x_2^2 + \lambda_1(-x_1^2 - x_2^2 + 1) + \lambda_2(-9x_1^2 - x_2^2 + 9) + \lambda_3(-x_1^2 + x_2) + \lambda_4(-x_2^2 + x_1) \quad (4\text{-}7)$$

In this problem, we use three different derivative methods.

1) Finite differencing provides the first derivative value of the objective function and constraint function. It also provides the second derivative value of the Lagrangian function.

2) Finite differencing provides the first derivative value of the objective function. AD tool provides the second derivative value of the Lagrangian function.

3) AD tool provides the first derivative value of the objective function and the constraint function. It also provides the second derivative value of the Lagrangian function.

Result

The final value of the function is 2, and the final value of the states are [1;1]. The following graph, Figure 4-3, shows the results iterated from the fmincon of MATLAB when AD provides both the first and the second derivative values. Fmincon in this problem has went through 14 steps to converge to the correct answer. Taking a look at the first order optimality, it shows the feasible solution is very close to the optimal point.

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	1	1.000000e+001	2.000e+000	2.118e+000	
1	2	4.901035e+000	0.000e+000	7.782e-001	1.319e+000
2	3	2.852319e+000	0.000e+000	2.838e-001	5.612e-001
3	4	2.298838e+000	0.000e+000	1.901e-001	1.747e-001
4	5	2.038929e+000	0.000e+000	1.887e-001	8.831e-002
5	6	2.051112e+000	0.000e+000	1.987e-002	6.150e-003
6	7	2.001699e+000	0.000e+000	2.021e-002	1.745e-002
7	8	2.000367e+000	0.000e+000	1.544e-003	4.773e-004
8	9	2.000401e+000	0.000e+000	2.000e-004	1.221e-005
9	10	2.000080e+000	0.000e+000	2.000e-004	1.134e-004
10	11	2.000080e+000	0.000e+000	4.000e-005	1.147e-007
11	12	2.000001e+000	0.000e+000	4.000e-005	2.802e-005
12	13	2.000001e+000	0.000e+000	4.000e-007	7.226e-009
13	14	2.000000e+000	0.000e+000	4.000e-007	2.800e-007
14	15	2.000000e+000	0.000e+000	4.000e-009	7.216e-013

Figure 4-4. Result of example three

In Table 4-1 below, it compares the results of all the three cases. Even though the case, that when AD tool only provide first derivative, uses least steps, but this case consumes more CPU time to get the final result. Comparing the first derivative method to the third derivative method, they consume similar amount of CPU time to get the final results. The first method, that finite differencing provides all the first and second derivatives, is 0.03 second faster than the case that AD tool provides all the first and second derivatives. Since this difference is very tiny, we can almost ignore it. In the table below, the values of function, variable and Lagrange multiplier are the same. That means using AD tool in this optimization problem can get the correct result.

	Finite Differencing	1st derivative(AD)	1st &2nd derivative(AD)
Iteration steps	13	8	14
CPU time	0.9966104	1.2071536	1.0311676
Function value	2.0000	2.0000	2.0000
State value	1.0000 1.0000	1.0000 1.0000	1.0000 1.0000
Costate value	lneqlin=4.0000e-007 lneqnonlin=4.0000e-07 4.0000e-07 2.0000 2.0000	lneqlin=4.0000e-007 lneqnonlin=4.0000e-07 4.0000e-07 2.0000 2.0000	lneqlin=4.0000e-007 lneqnonlin=4.0000e-07 4.0000e-07 2.0000 2.0000

Figure 4-5. Comparison of three different derivative modes

Example 4

Problem Statement

Consider the following NLP problem [21]

Minimize $f(x) = (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4$

Subject to $x_3 + x_4 + x_5 \geq 3$

$$x_1 + x_2^2 + x_3^3 = 3$$

$$x_1 + x_2^2 + x_3^3 = 3$$

$$x_1 + x_2 \geq 1$$

$$x_2 - x_3^2 + x_4 = 1$$

$$x_1 x_5 = 1$$

This problem has five different variables: x_1, x_2, x_3, x_4, x_5; one objective function, which is $f(x) = (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4$; two equality constraints: $x_1 + x_2^2 + x_3^3 = 3$and $x_1 x_5 = 1$; two inequality constraints: $x_3 + x_4 + x_5 \geq 3$and$x_1 + x_2 \geq 1$. In this problem, substituting objective function and all the constraint functions in the Lagrangian function, the function will be

$L = f(x) + \lambda^T C(x)$

$= (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 + \lambda_{e1}^T (x_1 + x_2^2 + x_3^3 - 3) + \lambda_{e2}^T (x_2 - x_3^2 + x_4 - 1) + \lambda_{e3}^T (x_1 x_5 - 1) + \lambda_{i1}^T (-x_3 - x_4 - x_5 + 3) + \lambda_{i2}^T (-x_1 - x_2 + 1)$ (4-8)

Result

The first result is what the AD tool supply those two values. The second result is what the program supply the hessian and gradient of the related values using the finite difference. In the first result, the total number of iteration steps is thirty-five, compared to the second result, whose total number of iteration steps is forty-five. This comparison

obviously reveals that, by using the automatic differentiation, optimality process reduces about twenty-two percent of the total steps of iterates.

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	1	0.000000e+000	1.010e+006	0.000e+000	
1	6	4.581553e+001	9.833e+005	6.522e+001	2.236e+000
2	9	8.117326e+001	9.816e+005	7.027e+001	2.236e+000
3	14	2.248833e+002	9.295e+005	7.225e+000	4.384e+000
4	18	2.527792e+003	9.235e+005	5.088e+002	7.673e+000
5	23	1.047905e+004	7.604e+005	2.499e+001	1.385e+001
6	26	3.821667e+005	7.232e+005	1.546e+004	4.848e+001
7	30	3.876604e+006	6.455e+005	4.834e+004	8.099e+001
8	32	3.904007e+007	4.335e+005	3.050e+005	1.669e+002
9	33	2.438881e+007	1.491e+005	2.199e+005	9.974e+001
10	34	4.010425e+006	4.076e+004	8.891e+004	1.497e+002
11	35	5.054035e+005	1.067e+004	1.379e+004	8.864e+001
12	36	5.335062e+004	2.768e+003	1.264e+004	4.762e+001
13	37	4.761797e+003	7.212e+002	7.519e+002	2.447e+001
14	38	3.504224e+002	1.901e+002	3.242e+002	1.268e+001
15	39	2.937267e+001	5.074e+001	2.468e+001	6.423e+000
16	40	4.055219e+000	1.344e+001	8.977e+000	3.355e+000
17	41	4.458761e-001	3.275e+000	9.507e-001	1.589e+000
18	42	-2.868476e-002	5.921e-001	7.964e-001	7.532e-001
19	43	-2.788698e-002	5.683e-002	2.075e-001	3.137e-001
20	44	-2.599089e-002	2.450e-003	9.838e-002	9.182e-002
21	45	-2.660813e-002	1.001e-004	2.106e-002	1.805e-002
22	46	-2.669505e-002	5.702e-005	3.198e-003	1.804e-002
23	47	-2.671143e-002	1.318e-005	1.586e-003	7.416e-003
24	48	-2.671387e-002	1.208e-006	4.659e-004	2.165e-003
25	49	-2.671413e-002	1.117e-007	2.000e-004	6.574e-004
26	50	-2.671417e-002	3.250e-008	8.965e-005	3.834e-004
27	51	-2.671418e-002	3.002e-009	4.000e-005	1.084e-004
28	52	-2.671418e-002	1.309e-009	1.861e-005	7.843e-005
29	53	-2.671418e-002	1.197e-010	4.788e-006	2.167e-005
30	54	-2.671418e-002	1.111e-011	1.440e-006	6.562e-006
31	55	-2.671418e-002	1.033e-012	4.387e-007	1.999e-006
32	56	-2.671418e-002	2.184e-013	2.260e-007	9.762e-007
33	57	-2.671418e-002	2.010e-014	8.000e-008	2.799e-007
34	58	-2.671418e-002	6.772e-015	4.149e-008	1.761e-007
35	59	-2.671418e-002	8.882e-016	1.086e-008	4.919e-008

Figure 4-6. Result of the example with giving the hessian and gradient by AD tool

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	6	0.000000e+000	1.010e+006	7.433e-007	
1	16	1.585144e+001	9.574e+005	1.593e+001	1.789e+000
2	23	3.447365e+003	9.477e+005	6.985e+002	1.252e+001
3	33	3.529229e+005	4.827e+005	4.598e+004	2.004e+001
4	40	3.422257e+005	4.653e+005	3.474e+004	2.504e+001
5	51	1.028112e+006	1.986e+005	3.413e+004	2.504e+001
6	58	3.650616e+006	1.477e+005	1.258e+005	5.009e+001
7	64	5.464924e+006	5.806e+004	7.911e+004	6.913e+001
8	70	8.727138e+005	1.593e+004	6.924e+004	9.646e+001
9	76	1.187633e+005	4.192e+003	1.223e+004	5.637e+001
10	82	1.385468e+004	1.095e+003	4.837e+003	3.047e+001
11	88	2.107280e+003	2.880e+002	2.091e+003	1.557e+001
12	94	4.311983e+002	7.667e+001	5.722e+002	7.926e+000
13	100	8.361581e+001	2.084e+001	2.136e+002	3.909e+000
14	106	3.784913e+001	6.219e+000	1.043e+002	1.922e+000
15	112	3.763251e+001	6.164e+000	8.628e+001	9.495e-003
16	118	3.677925e+001	6.047e+000	8.504e+001	3.580e-002
17	124	2.979797e+001	1.454e+000	6.879e+001	2.078e+000
18	130	1.429966e+001	3.677e-001	4.616e+001	6.707e-001
19	136	1.610260e+001	3.737e-002	4.814e+001	2.464e-001
20	142	1.142300e+001	2.867e-002	3.910e+001	2.110e-001
21	148	5.656649e+000	7.132e-002	1.333e+001	3.656e-001
22	154	1.839848e+000	7.843e-002	1.185e+001	4.599e-001
23	160	9.287030e-001	2.047e-002	7.053e+000	2.566e-001
24	166	4.663291e-001	1.992e-002	1.615e+000	2.489e-001
25	172	3.154187e-001	8.628e-003	1.523e+000	1.636e-001
26	178	2.972372e-001	3.291e-004	1.511e+000	3.013e-002
27	184	2.908867e-001	7.708e-006	1.503e+000	5.074e-003
28	190	2.714596e-001	7.882e-005	1.473e+000	1.417e-002
29	197	1.921215e-001	4.297e-005	1.402e+000	6.650e-002
30	204	2.385653e-002	1.085e-003	2.719e-001	2.504e-001
31	214	1.609120e-002	1.617e-003	2.389e-001	4.750e-002
32	226	-4.424569e-003	1.480e-004	8.171e-002	1.240e-001
33	232	-1.235370e-002	8.274e-003	1.203e-001	1.601e-001
34	238	-1.838067e-002	8.676e-003	2.746e-001	1.871e-001
35	244	-2.488038e-002	8.787e-003	3.881e-001	2.170e-001
36	250	-2.657897e-002	3.374e-004	1.734e-001	3.847e-002
37	256	-2.661143e-002	7.096e-005	2.417e-002	1.537e-002
38	262	-2.670325e-002	4.333e-005	1.735e-002	1.322e-002
39	268	-2.670971e-002	1.576e-006	4.001e-003	2.787e-003
40	274	-2.671353e-002	6.347e-006	5.115e-004	5.686e-003
41	280	-2.671418e-002	6.600e-009	4.001e-005	1.713e-004
42	286	-2.671418e-002	6.800e-010	1.156e-005	5.950e-005
43	292	-2.671418e-002	2.389e-013	4.000e-007	8.212e-007
44	298	-2.671418e-002	7.105e-014	2.991e-008	6.046e-007
45	304	-2.671418e-002	4.441e-016	6.430e-009	8.476e-009

Figure 4-7. Result with the hessian and gradient value generated by the quasi-Newton approximation

This table below shows all the results of three different derivative methods. It is apparently seen that, with providing the first and second derivatives by AD tool, fmincon uses less iteration steps than the other derivative methods. The CPU time used to find the optimal result is 1.370038s, 0.9611894s and 1.378906s respectively. Therefore, if AD provides the first and second derivative values to the NLP solver, it will save more than 25 percent time as compared to the normal way. Other parameters of these three derivative methods are similar to each other. This shows the efficiency of AD tool in solving the complicated problems.

	1st derivative by AD	1st and 2nd by AD	Finite Differencing
Iteration Steps	20	9	45
CPU time	1.3700388 s	0.9611894 s	1.378906 s
Function value	-2.671418e-002	-2.671418e-002	-2.671418e-002
State value	0.677004396770634 0.726089474240478 1.21549121328710 1.75132941533765 1.47709528146346	0.677004394962072 0.726089472732101 1.21549121418933 1.75132941903936 1.47709528540952	0.677004445954108 0.726089516809437 1.21549118824313 1.75132931188726 1.47709517415457
Costate value	Eqnonlin=[-0.0818;-0.6979;0.1219] Ineqlin=[2.770074638202289e-09;9.924092778942344e-09]	Eqnonlin= (1,1) -0.0818 (2,1) -0.6979 (3,1) 0.1219 Ineqlin= 1.0e-009 * (1,1) 0.1385 (2,1) 0.4962	Eqnonlin= -0.0818 -0.6979 0.1219 Ineqlin= 1.0e-008 * 0.2770 0.9923

Figure 4-8. Comparison of Methods using AD and using Finite Differencing

CHAPTER 6
CONCLUSION AND FUTURE WORK

This work presents a new automatic differentiation called “Hes”. This tool is forward operation overloading tool. In this work, we employ different algorithms of AD. Compared to hand computation or infinite differencing, automatic differentiation is more efficient and less error prone. In Chapter 5, we have implemented the automatic differentiation into some specific problems. The gradient and hessian generated by the automatic differentiation were found to be more accurate and time saving when compared to that generated by finite differencing or quasi-Newton approximation method. This comparison proves that automatic differentiation can not only provide accurate gradient value, but also works well for high-order derivatives.

This work only uses operator overloading method to generate the derivative of a formula. And also, it uses object-oriented MATLAB to realize the code. In the future, we can combine forward mode with backward mode or combine operator overloading with source transformation to make the program run more fluently.

LIST OF REFERENCES

[1] Wolfe, P., "Checking the Calculation of Gradients," *ACM Trans.Math.Softw.ACM Transactions on Mathematical Software,* Vol. 8, No. 4, 1982, pp. 337-343.

[2] Andreas Griewank, "On Automatic Differentiation," 1994,

[3] Christian Bischof, Alan Carle, George Corliss, "ADIFOR: Automatic differentiation in a source translator environment,"

[4] Rosemblun, M.L., and Rosemblun, M.L., "Automatic differentiation: Overview and application to systems of parameterized nonlinear equations," 2009,

[5] Magaña Jimenez, Q., "Nonlinear control via automatic differentiation," 2002,

[6] Fateh, H., "Automatic differentiation for large-scale nonlinear programming problems," 1995,

[7] Ernesto G. Birgin, and Yuri G. Evtushenko, "Automatic Differentiation for Optimal Control Problems," 1998,

[8] Griewank, A., Juedes, D., and Srinivasan, J., "ADOL-C : a package for the automatic differentiation of algorithms written in C/C++," Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991,

[9] Jee, K.-., McShan, D.L., and Fraass, B.A., "Implementation of Automatic Differentiation Tools for Multicriteria IMRT Optimization," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 225-234.

[10] Juedes, D., and Griewank, A., "Implementing automatic differentiation efficiently," 2008,

[11] Kowarz, A., "Advanced concepts for automatic differentiation based on operator overloading," 2008,

[12] Arun Verma, "ADMAT: Automatic differentiation in MATLAB using object oriented methods," 1998,

[13] Thomas F. Coleman, A.V., "ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox," 1999,

[14] Bischof, C.H., Bucker, H.M., and Vehreschild, A., "A Macro Language for Derivative Definition in ADiMat," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 181-188.

[15] Gay, D.M., "Semiautomatic Differentiation for Efficient Gradient Computations," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 147-158.

[16] Utke, J., "Flattening Basic Blocks," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 121-134.

[17] Phipps, E., Casey, R., and Guckenheimer, J., "Periodic Orbits of Hybrid Systems and Parameter Estimation via AD," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 211-224.

[18] Snopok, P., Johnstone, C., and Berz, M., "Simulation and Optimization of the Tevatron Accelerator," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 199-210.

[19] Tadjouddine, M., Bodman, F., Pryce, J.D., "Improving the Performance of the Vertex Elimination Algorithm for Derivative Calculation," *LECTURE NOTES IN COMPUTATIONAL SCIENCE AND ENGINEERING,* Vol. 50, 2006, pp. 111-120.

[20] Kharche, R.V., and Forth, S.A., "Source Transformation for MATLAB Automatic Differentiation," *Lecture Notes in Computer Science.,* No. 3994, 2006, pp. 558-565.

[21] Gill, P.E., Murray, W., and Saunders, M.A., "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM J.Optim.SIAM Journal on Optimization,* Vol. 12, No. 4, 2002,

BIOGRAPHICAL SKETCH

Ying Lin was born in Wen Lin, Zhejiang, China in 1987. She received her bachelor's degree in mechanical engineering from China Agricultural University, Beijing, China in June 2009. She was awarded 3rd prize of Excellent Study of university for three Consecutive Years from China Agricultural University. She received her Master of Science degree with thesis in the Department of Mechanical and Aerospace Engineering at University of Florida in the summer of 2011, under the supervision of Dr. Anil V. Rao and co-advised by Dr. Warren E. Dixon. Her interests lie in the field of optimal control and control systems.