A Structural and Efficient Method for Calculating Gradient and Hessian with CVaR Portfolio Optimization Application

by

Yu Feng

An essay
presented to the University of Waterloo
in fulfillment of the
essay requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2013

Abstract

Calculating or approximating the derivatives for large-scale multi-dimensional functions is an active research area in modern mathematics. The rationale behind this popularity is its wide applications in statistics, financial mathematics and portfolio optimization. For example, in statistics, maximum likelihood estimation seeks the value of parameter vector that maximize the likelihood function, which requires to find the points where derivative of likelihood function is zero; in finance, taking derivatives can be applied to sensitivity analysis for equity valuation, with respect to single or multiple variables; numerical optimization problems calculate the derivatives at each iteration, such as trust-region method. The most straightforward way to calculate a derivative is hand-coding, but it is only applicable to sufficiently simple functions, and it's error-prone. Finite-difference method is easy to implement with a programming language, but the accuracy depends on the choice of discretization steps, and hence can not be guaranteed.

Throughout this essay, we study the methodology of automatic differentiation (AD), and introduce a structural automatic differentiation method (Structural-AD) for calculating gradient and hessian. The implementation of Structural-AD uses the software ADMAT 2.0 installed on MATLAB by Cayuga Research Associates [2009]. Structura-AD exploits the 'natural structure' of some functions and it makes use of the regular reverse mode of AD to achieve more efficient computing time and less memory usage requirement. In the paper by Xu and Coleman [2013], they applied structural-AD to two extreme cases, generalized partially separable problem and dynamic system computations. They showed that computing time and memory requirement using Structural-AD is significantly reduced compared to the regular reverse mode. We also applied structural-AD on a Conditional Value-at-Risk (CVaR) optimization problem, specifically with the underlying function describing the loss function of a stock portfolio.

Acknowledgements

First and foremost I owe my deepest gratitude to my supervisor Dr. Thomas Coleman, who has been always supporting and guiding my study and research. Dr. Coleman is also a knowledgeable mentor for my personal life; when I was confused about how to choose my future career, he taught me to follow my heart; when I had an opportunity to work as an intern for Ontario Teachers' Pension Plan, he gave me supportive recommendations. I will always remember his guidance and encouragement.

Secondly, I would like thank Dr. Stephen Vavasis for his useful suggestions on my essay. Dr. Vavasis supervised me when I was an undergraduate research assistant during my fourth-year study in the Unversity of Waterloo. He taught me a very interesting image segmentation problem, which combined the knowledge of graph theory and convex optimization. I learned from him not only the fundamental theories, but also, more importantly, how to conduct academic research.

Moreover, it's a pleasure to thank all my friends throughout my undergraduate and graduate studies at the University of Waterloo. Our friendship keeps me strong and shares my happiness.

Finally, my family always has my deepest love.

Table of Contents

List of Tables

Chapter 1

Introduction

Calculating or approximating the derivatives for large-scale multi-dimensional functions is an active research area in modern mathematics. The rationale behind this popularity is its wide applications in statistics, financial mathematics and portfolio optimization. For example, in statistics, maximum likelihood estimation seeks the value of parameter vector that maximize the likelihood function, which requires to find the points where derivative of likelihood function is zero; in finance, taking derivatives can be applied to sensitivity analysis for equity valuation, with respect to single or multiple variables; numerical optimization problems calculate the derivatives at each iteration, such as trust-region method.

Some of the common ways to calculate derivatives are: hand-coding, finite-difference, symbolic differentiation, and automatic differentiation. The first three methods are subject to their own limitations. Hand-coding is straight-forward, but only applicable to sufficiently simple functions, and it's error-prone. Finite-difference method is easy to implement with a programming language, but the accuracy depends on the choice of discretization steps, and hence can not be guaranteed. The disadvantage of symbolic differentiation is the inefficiency of computing time with the growth of the independent variables.

The idea behind automatic differentiation (AD) is basically the chain rule. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations and elementary functions. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurate to working precision. This technique enables AD to avoid the round-off error incurred by the finite difference method.

AD has two modes in general - forward mode and reverse mode. In particular, if

$$f(x) = g(h(x)),$$

then chain rule gives,

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx}.$$

The forward mode applies chain rule to the above equation from the right to left, while reverse model proceeds from left to right.

The analysis of the computational cost of these two modes are presented in Chapter 2. Briefly speaking, if $f : R^n \rightarrow R^m$ and the cost of evaluating the function value of f at a single point is denoted as $cost(f)$, then ,

$$\text{Cost of Forward Mode: } n \times cost(f)$$
$$\text{Cost of Reverse Mode: } m \times cost(f)$$

Hence, as a rule of thumb, reverse mode can be used when $n \gg m$, such as most of the optimization problems; forward mode can be used when $m \gg n$, such as sensitivity analysis.

Throughout this research paper, we study a structural automatic differentiation method (Structural-AD). This method exploits the 'natural structure' of some functions and it makes use of the regular AD to achieve more efficient computing time and less memory usage requirement. In the paper by Xu and Coleman [2013], they applied Structural-AD to two extreme cases, generalized partially separable problem and dynamic system computations. They showed that computing time and memory requirement using Structural-AD is significantly reduced compared to the regular reverse mode. The Structural-AD is first raised by Jonsson and Coleman [1999], who illustrated how natural structure can be used to enable efficient gradient computation. In a paper by Coleman et al. [2012], they used a direct edge separator method to find the structure in the computational "tape", which supported their previous work. Similarly, fast Hessian calculation canalso be recovered when there's natural structure in the function.

As a major contribution of this essay, we apply Structural-AD on some problems in different fields. The structure of this essay is outlined as follows: In Chapter 2, we introduce the theory of automatic differentiation, and the two computation methods - forward mode and reverse mode. The comparison of the computational costs for these two methods is also provided. In Chapter 3, we provide the detailed methodology and derivation of gradient and Hessian calculation in the Structured-AD context, following which, the computational costs of calculating gradient and Hessian are also presented compared to the finite-difference method. In Chapter 4, we applied Structural-AD to two extreme cases, generalized partially separable problem and dynamic system computations and showed that computing time and memory requirement using Structural-AD is significantly reduced compared to the regular reverse mode. In Chapter 5, we test structural-AD on a Conditional Value-at-Risk (CVaR) optimization problem, specifically with the underlying function describing the loss function of a stock portfolio. Chapter 6 summarizes the main idea and findings of this thesis.

All the implementation work is performed on the software ADMAT 2.0 - an Automatic Differentiation Toolbox for MATLAB developed by Coleman and Xu.

Chapter 2

Theory of Automatic Differentiation

Automatic differentiation relies on the fact that every function is executed on a computer as a sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos. By repeated application of chain rule on those elementary operations, one can compute derivatives in a completely mechanical fashion. For illustration purpose, we use the following simple example to show forward mode and reverse mode.

$$[y_1, y_2] = f(x_1, x_2, x_3, a, b)$$

with computation order following below:

- $\omega_1 = \log(x_1 \cdot x_2)$
- $\omega_2 = x_2 \cdot x_3^2 - a$
- $\omega_3 = b \cdot \omega_1 + \frac{x_2}{x_3}$
- $y_1 = \omega_1^2 + \omega_2 - x_2$
- $y_2 = \sqrt{\omega_3} - \omega_2$

Therefore, we have

- independent variables: $\mathbf{x} = (x_1, x_2, x_3)$
- dependent variables: $\mathbf{y} = (y_1, y_2)$

- intermediate variables: $\omega = (\omega_1, \omega_2, \omega_3)$
- active variables: $\mathbf{x}, \mathbf{y}, \omega$
- constants: a, b

We want to calculate the Jacobian $\mathbf{Jf}$,

$$\mathbf{Jf} = \begin{bmatrix} \nabla y_1 \\ \nabla y_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

For the simplicity of notations, let's unify all the variables first,

- $u_1 = x_1, u_2 = x_2, u_3 = x_3$
- $u_4 = \Phi_4(u_1, u_2) = \log(u_1 \cdot u_2)$
- $u_5 = \Phi_5(u_2, u_3) = u_2 \cdot u_3^2 - a$
- $u_6 = \Phi_6(u_2, u_3, u_4) = b \cdot u_4 + \frac{u_2}{u_3}$
- $u_7 = \Phi_7(u_2, u_4, u_5) = u_4^2 + u_5 - u_2$
- $u_8 = \Phi_8(u_5, u_6) = \sqrt{u_6} - u_5$

Note that we will use the following notations for the rest of forward mode and reverse mode section.

- n is the number of original independent variables
- N is the total number of variables including the independent, intermediate and dependent variables
- m is the number of dependent variables
- p is the number of intermediate variables

Hence, $N = m + n + p$.

2.1 Forward Mode

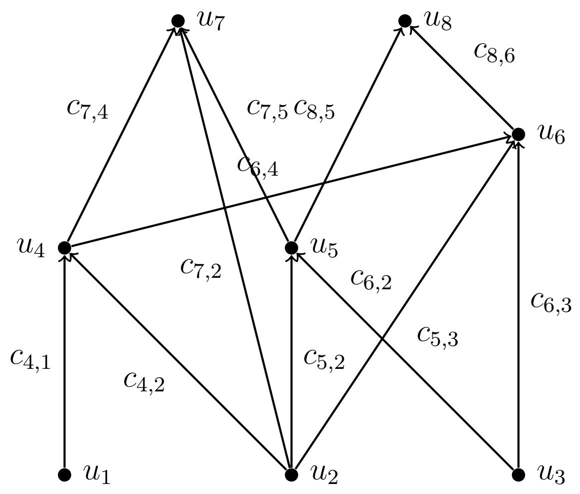
Forward mode computes the gradient of each variable $u_i, i = 1 \dots N$, and use the chain rule to pass the gradient to the next variable u_{i+1}. In each computation, it computes the vectors with input size n. We can proceed the function evaluation and the gradient computation at the same time.
Function evaluation:

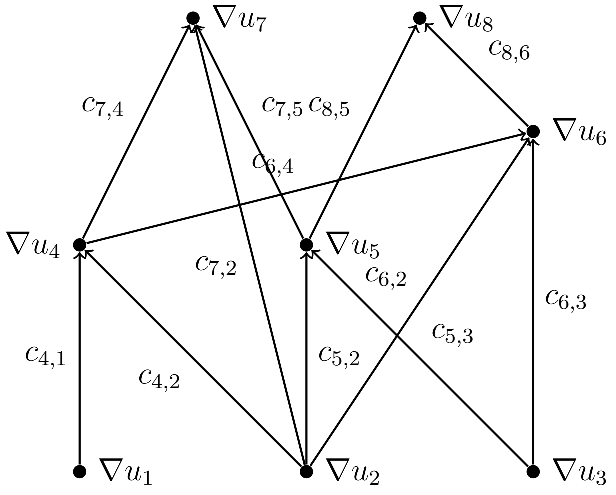
$$u_i = x_i, i = 1 \dots n,$$
$$u_i = \Phi(\{u_j\}_{j<i}), i = n+1 \dots N$$

Differentiation:

$$\nabla u_i = e_i, i = 1 \dots n,$$
$$\nabla u_i = \sum_{j<i} c_{i,j} \cdot \nabla u_j, i = n+1 \dots N$$

where $c_{i,j}$ is the partial derivative of u_i with respect to u_j.
In the following two figures, the left one shows the computational order of the above example, and the right one is the order of differentiation in forward mode AD.





2.2 Reverse Mode

Reverse mode computes the adjoint of each variable $u_i, i = 1 \dots N$, and pass the adjoint to the next variable. In each computation, it computes the vectors with output size m, hence

reverse mode is widely used in the optimization application, since the size of the output of optimization problems is usually one.
Compute the adjoint of the variables:

$$\bar{u_j} = \frac{\partial y}{\partial u_j} = \frac{\partial(y_1, y_2, \ldots, y_m)}{\partial u_j}$$

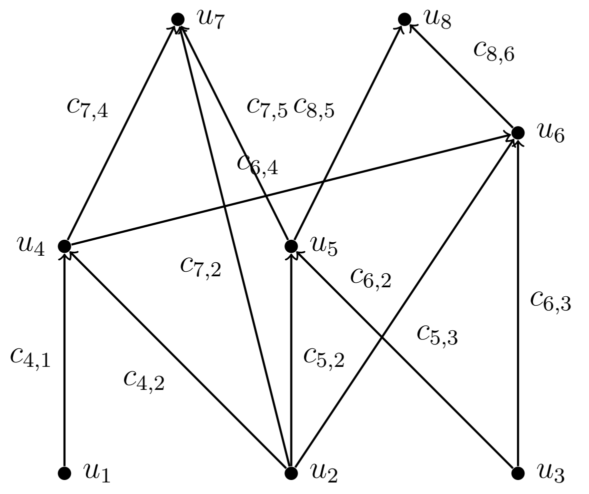
Compute for dependent variables:

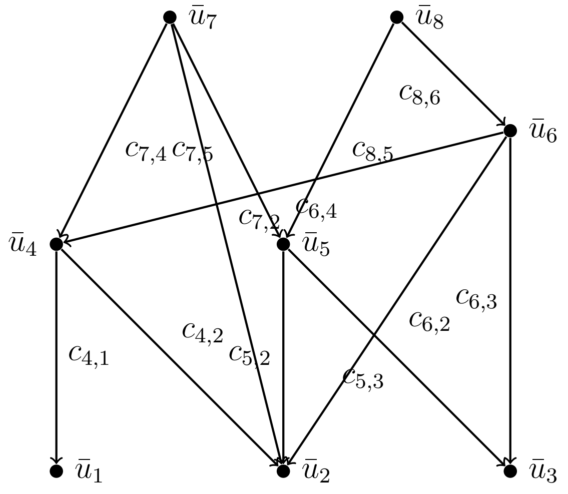
$$\bar{u}_{n+p+j} = \frac{\partial(y_1, y_2, \ldots, y_m)}{\partial u_j} = e_j, j = 1, \ldots, m$$

Compute for intermediates and independents $u_j, j = n+p, \ldots, 1$,

$$\bar{u_j} = \frac{\partial y}{\partial u_j} = \sum_{i>j} \bar{u}_i c_{i,j}$$

Reverse mode traverses through the computational graph reversely and gets the parents of each variable so as to compute the adjoint.
In the following two figures, the left one shows the computational order of the above example, and the right one is the order of differentiation in reverse mode AD and we can tell that reverse mode traverses the computational graph reversely.





2.3 Computational Cost for Forward and Reverse Modes

Before computing the cost of forward mode and reverse mode, let's define the following:

$$\begin{cases} N_u & : \text{number of unit local derivatives, } c_{i,j} = \pm 1 \\ N_{\bar{u}} & : \text{number of non-unit local derivatives, } c_{i,j} \neq 0, \pm 1 \end{cases}$$

In forward mode, we need to solve for derivatives in forward order, $\nabla u_{n+1}, \nabla u_{n+2}, \ldots, \nabla u_N$,

$$\nabla u_i = \sum_{j<i} c_{i,j} \cdot \nabla u_j, i = n+1, \ldots, N$$

with each $\nabla u_i = (\frac{\partial u_i}{\partial x_1}, \ldots, \frac{\partial u_i}{\partial x_n})$, a length n vector.

Hence, the flops in forward mode is given by,

$$flops(forward) = \begin{array}{ll} n \cdot N_{\bar{u}} & (\text{multiplications } c_{i,j} \nabla u_j, c_{i,j} \neq 0, \pm 1) \\ +n(N_u + N_{\bar{u}}) & (\text{additions } +c_{i,j} \nabla u_j) \end{array}$$

$\Rightarrow$

$$flops(forward) = n(2N_{\bar{u}} + N_u)$$

Therefore, the cost of forward mode is proportional to the size of the input variables.

In reverse mode, we need to solve for adjoints in reverse order $\bar{u}_{n+p}, \bar{u}_{n+p-1}, \ldots, \bar{u}_1$,

$$\bar{u}_j = \sum_{i>j} \bar{u}_i c_{i,j}$$

with $\bar{u}_j = \frac{\partial}{\partial u_j}(y_1, y_2, \ldots, y_m)$ is a length m vector.

Hence, the flops in reverse mode is given by,

$$flops(reverse) = \begin{array}{ll} mN_{\bar{u}} & (\text{multiplications } \bar{u}_i \cdot c_{i,j}, c_{i,j} \neq \pm 1, 0) \\ +m(N_u + N_{\bar{u}}) & (\text{additions } +\bar{u}_i \cdot c_{i,j}) \end{array}$$

$\Rightarrow$

$$flops(reverse) = m(2N_{\bar{u}} + N_u)$$

Therefore, the cost of reverse mode is proportional to the size of the output.

2.4 Differentiation Arithmetic and High-Order Derivatives

The differentiation arithmetic is based on chain rule and it can be implemented in Matlab as an structured object with format $(f(x_0), f^{'}(x_0))$. Let u denote the value of the function $u : \mathbb{R} \to \mathbb{R}$ evaluated at the point x_0, and where $u^{'}$denotes the value $u^{'}(x_0)$. Then, we can proceed the function evaluation and differentiation at the same time. Let's denote the object as $\vec{u} = (u, u^{'})$.

$$\vec{u} + \vec{v} = (u + v, u^{'} + v^{'})$$
$$\vec{u} - \vec{v} = (u - v, u^{'} - v^{'})$$
$$\vec{u} \times \vec{v} = (uv, uv^{'} + u^{'}v)$$
$$\vec{u}/\vec{v} = (u/v, u^{'} - (u/v)v^{'}/v))$$
$$\vec{x} = (x, 1)$$
$$\vec{c} = (c, 0)$$

Same structure can be applied to high-order derivative. For example, $\vec{u} = (u, u^{'}, u^{''})$.

$$\vec{u} + \vec{v} = (u + v, u^{'} + v^{'}, u^{''} + v^{''})$$
$$\vec{u} - \vec{v} = (u - v, u^{'} - v^{'}, u^{''} - v^{''})$$
$$\vec{u} \times \vec{v} = (uv, uv^{'} + u^{'}v, uv^{''} + 2u^{'}v^{'} + u^{''}v')$$
$$\vec{u} \div \vec{v} = (u/v, u^{'} - (u/v)v^{'}/v, (u^{''} - 2(u/v)^{'}v^{'} - (u/v)v^{''})/v)$$

Chapter 3

Structural-AD for Gradient and Hessian Calculations

Automatic differentiation has become a popular research area because of its avoidance of truncation error. In theory, for scalar-value function, the gradient can be computed using reverse mode of AD with the same complexity as evaluating the function itself. However, practical results on large-scale problem show poor performance because of huge memory usage. In this chapter, we introduce an improvement over traditional AD when the function itself has a "separation structure".

3.1 Definitions and Notations

- $f^{\lambda}(x) : R^n \to R$, a scalar-valued function, with λ being an uncertain parameter of length p
- $\Lambda = (\lambda_1, \ldots, \lambda_k) \in R^{p \times k}$: a sample of parameters with size k, forming a matrix
- $G(x) = \left[\nabla_x f^{\lambda_1}(x), \ldots, \nabla_x f^{\lambda_k}(x)\right]$: gradient of f^{λ_i} at x for $i = 1, \cdots, k$
- $D^2_{ij} f(x)$: second-order partial derivative of f^{λ} at x

- $H_f(x) := \begin{bmatrix} D^2_{11}f(x) & D^2_{21}f(x) & \dots & D^2_{n1}f(x) \\ D^2_{12}f(x) & D^2_{22}f(x) & \dots & D^2_{n2}f(x) \\ \vdots & \vdots & & \vdots \\ D^2_{1n}f(x) & D^2_{2n}f(x) & \dots & D^2_{nn}f(x) \end{bmatrix}$: Hessian of f at x

3.2 Method for Gradient Calculation

Suppose that we wish to determine, at point x, the gradient of f with respect to x for each λ_i, where $i = 1 \dots k$, that is,

$$G(x) = \left[\nabla_x f^{\lambda_1}(x), \dots, \nabla_x f^{\lambda_k}(x)\right] \tag{3.1}$$

Based on the work of Verma and Coleman [1998], many problems can be written as a structured computation, namely, we can view the evaluation of f^λ(x) at any x as a two-step process:

1. solve for an intermediate vector y: $F(x, y) = 0$;
2. compute $z = \tilde{f}^\lambda(x, y)$.

for some functions F and $\tilde{f}^\lambda$.
Note:

- In the first step, the function F does not have uncertain parameters λ. Uncertain parameters only appear in the second step;
- In the second step, the function $\tilde{f}^\lambda$ is a scalar-value function;
- In many important cases, the first step represents the dominant work.

So for a given λ, differentiation with respect to the original variables x as well as the intermediate variables y gives an "extended" Jacobian matrix of $\tilde{f}^\lambda(x, y)$ is:

$$J^\lambda = \begin{bmatrix} F_x & F_y \\ (\nabla_x \tilde{f}^\lambda)^T & (\nabla_y \tilde{f}^\lambda)^T \end{bmatrix}. \tag{3.2}$$

where F_x and F_y are the Jacobian matrix of F with respect to x and y respectively; $(\nabla_x \tilde{f}^\lambda)^T$ and $(\nabla_y \tilde{f}^\lambda)^T$ are the transpose of the gradients of $\tilde{f}^\lambda$ with respect to x and y.

Note that the Jacobian matrix (F_x, F_y) is often sparse, and F_y is lower-triangular and non-singular.
To get the gradient of $f^\lambda(x)$,

$$(\nabla_x f^\lambda) = (\nabla_x \tilde{f}^\lambda) - F_x^T F_y^{-T} (\nabla_y \tilde{f}^\lambda). \tag{3.3}$$

The **Key Observation** is that F_x, F_y do not depend on λ, and therefore can be computed just once for all λ_i, $i = 1, ..., k$. Therefore, having computed F_x, F_y, and $\{\nabla_x \tilde{f}^{\lambda_i}, \nabla_y \tilde{f}^{\lambda_i}, i = 1, ..., k\}$, column i of (3.1) can be computed by applying (3.3).

We can generalize the scalar-valued function case to vector-valued functions case; let's consider the following. Suppose $F : R^n \to R^m$ and to evaluate $z = F(x)$:

$$\begin{aligned}
\text{Solve for } y_1 &: F_1(x) - y_1 = 0 \\
\text{Solve for } y_2 &: F_2(x, y_1) - y_2 = 0 \\
&\vdots \\
\text{Solve for } y_p &: F_p(x, y_1, y_2, \cdots, y_{p-1}) - y_p = 0 \\
\text{Solve for } z &: \bar{F}(x, y_1, y_2, \cdots, y_p) - z = 0
\end{aligned}$$

The corresponding extended Jacobian can be written:

$$J^E = \begin{bmatrix} J_x^1 & -I & & & \\ J_x^2 & J_{y_1}^2 & -I & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & \vdots & J_{y_{p-1}}^p & -I \\ \bar{J}_x & \bar{J}_{y_1} & \cdots & \cdots & \bar{J}_{y_p} \end{bmatrix}. \tag{3.4}$$

If we partition J^E as

$$\left[\begin{array}{c|ccccc} J_x^1 & -I & & & \\ J_x^2 & J_{y_1}^2 & -I & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & \vdots & J_{y_{p-1}}^p & -I \\ \hline \bar{J}_x & \bar{J}_{y_1} & \cdots & \cdots & \bar{J}_{y_p} \end{array}\right]$$

$=$

$$\begin{bmatrix} A & L \\ B & M \end{bmatrix}$$

Then the Jacobian of F satisfies

$$J = B - ML^{-1}A.$$

In the paper by Xu and Coleman [2013], they demonstrate that it is not necessary to compute the matrix (A, L) explicitly, since the off-diagonal submatrices in (A, L) involved in the calculation of above equation occur only in a product form, which can be obtained by the reverse mode of AD.

3.3 Method for Hessian Calculation

Consider a scalar-valued function $f^\lambda(x) : \Re^n \to \Re$, where λ is a p-vector of (uncertain) parameters. Suppose we wish to determine the Hessian matrix of f, at point c,

$$H_f(c) := \begin{bmatrix} D^2_{11}f(c) & D^2_{21}f(c) & \dots & D^2_{n1}f(c) \\ D^2_{12}f(c) & D^2_{22}f(c) & \dots & D^2_{n2}f(c) \\ \vdots & \vdots & & \vdots \\ D^2_{1n}f(c) & D^2_{2n}f(c) & \dots & D^2_{nn}f(c) \end{bmatrix}. \tag{3.5}$$

We want to make use of the special structure of $f^\lambda(x)$, that is:

$$f^\lambda(x) = g(F(x), \lambda), \tag{3.6}$$

where $F : \Re^n \to \Re^m$ is a vector-valued function, and $g : \Re^m \times \Re^p \to \Re$ is scalar-valued function. One common example of this structure is the weighted average function with the weight vector λ.

Now, let $Z = F(x)$, and $y = g(Z, \lambda)$. For the $ij-$th entry of the Hessian matrix, we have:

$$
\begin{aligned}
\frac{\partial^2 f}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right) \\
&= \frac{\partial}{\partial x_j}\left[\sum_{k=1}^{m} \frac{\partial g}{\partial Z_k}\cdot\frac{\partial F_k}{\partial x_i}\right] \\
&= \sum_{k=1}^{m} \frac{\partial}{\partial x_j}\left[\frac{\partial g}{\partial Z_k}\cdot\frac{\partial F_k}{\partial x_i}\right] \\
&= \sum_{k=1}^{m}\left[\frac{\partial^2 g}{\partial x_j \partial Z_k}\cdot\frac{\partial F_k}{\partial x_i} + \frac{\partial^2 F_k}{\partial x_j \partial x_i}\cdot\frac{\partial g}{\partial Z_k}\right] \\
&= \sum_{k=1}^{m}\left[\left(\sum_{l=1}^{m} \underbrace{\frac{\partial^2 g}{\partial Z_k \partial Z_l}}_{(1)}\cdot\underbrace{\frac{\partial F_l}{\partial x_j}}_{(2)}\right)\cdot\underbrace{\frac{\partial F_k}{\partial x_i}}_{(3)} + \underbrace{\frac{\partial^2 F_k}{\partial x_j \partial x_i}}_{(4)}\cdot\underbrace{\frac{\partial g}{\partial Z_k}}_{(5)}\right].
\end{aligned}
\tag{3.7}
$$

In the above equation,

- (1) can be acquired from the Hessian of g with respect to Z;
- $(2), (3)$ can be acquired from the Jacobian of F with respect to x;
- (4) can be acquired from the Hessian of F with respect to x;
- (5) can be acquired from the gradient of g with respect to Z.

In this case, for a fixed x, $(2), (3), (4)$ are only calculated once for all λ_i, since they are only related the function $F(x)$. (4) and (5) must be recalculated for each λ_i. In situations where the main computational cost lies in evaluating the function $F(x)$, calculating $(2), (3), (4)$ only once for all λ_i would save us a lot of computation.

In matrix format, in which we use the notation of Kronecker product,

$$
H_f(c) = D_F(c)^T \cdot [H_g(b)] \cdot D_F(c) + [D_g(b) \otimes I_n] \cdot H_F(c), \tag{3.8}
$$

where $b = F(c)$, D_F is the Jacobian of F, D_g is the gradient of g, H_F is the Hessian of F, H_g is the Hessian of g.

$$D_F(c) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1}(c) & \frac{\partial F_1}{\partial x_2}(c) & \dots & \frac{\partial F_1}{\partial x_n}(c) \\ \frac{\partial F_2}{\partial x_1}(c) & \frac{\partial F_2}{\partial x_2}(c) & \dots & \frac{\partial F_2}{\partial x_n}(c) \\ \vdots & \vdots & & \vdots \\ \frac{\partial F_m}{\partial x_1}(c) & \frac{\partial F_m}{\partial x_2}(c) & \dots & \frac{\partial F_m}{\partial x_n}(c) \end{bmatrix},$$

$$H_F(c) = \begin{bmatrix} H_{F_1}(c) \\ H_{F_2}(c) \\ \vdots \\ H_{F_m}(c) \end{bmatrix},$$

$$D_g(b) = \left[\frac{\partial g}{\partial F_1}(b), \frac{\partial g}{\partial F_2}(b), \dots, \frac{\partial g}{\partial F_m}(b)\right],$$

$$H_g(b) = \begin{bmatrix} D^2_{11}g(b) & D^2_{21}g(b) & \dots & D^2_{m1}g(b) \\ D^2_{12}g(b) & D^2_{22}g(b) & \dots & D^2_{m2}g(b) \\ \vdots & \vdots & & \vdots \\ D^2_{1m}g(b) & D^2_{2m}g(b) & \dots & D^2_{mm}g(b) \end{bmatrix}.$$

Again, $D_F(c), H_F(c)$ are only calculated once for all λ_i; whereas $D_g(b), H_g(b)$ must be recalculated for each λ_i.

3.4 Comparison of the Computational Costs

Now we want to analyze the computational costs of the structural-AD method and the finite-difference method. Before doing analysis, let's define:

$\omega(F)$: the computational cost of evaluating a single entry of function F

$\omega(g)$: the computational cost of evaluating function g

3.4.1 Compare Computational Costs for Evaluating Gradients

For the function $f^\lambda(x) = g(F(x), \lambda)$, where $F : \Re^n \to \Re^m$ is a vector-valued function, and $g : \Re^m \times \Re^p \to \Re$ is scalar-valued function, let's make the assumption that $m < n$. In our method, the gradient will be calculated as:

$$G(x) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_2}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_1} \\ \frac{\partial F_1}{\partial x_2} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial F_1}{\partial x_n} & \frac{\partial F_2}{\partial x_n} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \frac{\partial g}{\partial Z_1} \\ \frac{\partial g}{\partial Z_2} \\ \vdots \\ \frac{\partial g}{\partial Z_m} \end{bmatrix}$$

To calculate the left matrix, each of the $\frac{\partial F_i}{\partial x_j}$ needs two function evaluations $\omega(F)$(here, we assume each entry are calculated using finite-difference method), so in total, the left matrix will need $2mn$ function evaluations $\omega(F)$. For the right matrix, each of the $\frac{\partial g}{\partial Z_i}$ needs two function evaluations $\omega(g)$, so the right matrix will need $2m$ function evaluations $\omega(g)$. In summary, the computational cost of calculating the gradient using the procedural method is $2[mn \cdot \omega(F) + m \cdot \omega(g)]$.

Using finite-difference method to calculate the gradient, each of $\frac{\partial f}{\partial x_i}$ needs two function evaluations of f, and f needs $m \cdot \omega(F) + \omega(g)$ (from the structure of f in (3.6)). Hence, the computational cost of calculating the gradient using finite-difference method is $2n[m \cdot \omega(F) + \omega(g)]$.

Comparing these two methods, for one gradient evaluation:

$$\begin{aligned} \text{The structural-AD method: } & 2[mn \cdot \omega(F) + m \cdot \omega(g)], \\ \text{Finite-difference method: } & 2[mn \cdot \omega(F) + n \cdot \omega(g)], \end{aligned}$$

When x is fixed, and λ are drawn from a sample of size k, we are interested in the savings in our method to the finite-difference method. Now, since everything related to F will only need to be calculated once, our method costs $2[mn \cdot \omega(F) + km \cdot \omega(g)]$, whereas the finite-difference method costs $2[kmn \cdot \omega(F) + kn \cdot \omega(g)]$, hence, the difference of structural-AD method and finite-difference method is $k(m-n) \cdot \omega(g) - (k-1)mn \cdot \omega(F)$. Since we make the assumption at the beginning that $m < n$, as k is getting large, the structrual-AD method will beat the finite-difference method.

3.4.2 Compare Computational Costs for Evaluating Hessians

Calculating the Hessian using finite-difference method usually takes 3 evaluations of f for each entry $\frac{\partial^2 f}{\partial x_i \partial x_j}$ in the Hessian matrix, so the finite-difference method will cost $3n^2[m \cdot \omega(F) + \omega(g)]$.

For our method, the Hessian can be calculated from the following formula (see (3.8)):

$$H_f(c) = D_F(c)^T \cdot [H_g(b)] \cdot D_F(c) + [D_g(b) \otimes I_n] \cdot H_F(c),$$

where,

$$\begin{aligned} D_F &\text{ costs } 2mn \cdot \omega(F), \\ H_g &\text{ costs } 3m^2 \cdot \omega(g), \\ D_g &\text{ costs } 2m \cdot \omega(g), \\ H_F &\text{ costs } 3mn^2 \cdot \omega(F). \end{aligned}$$

Hence, in total, our method will cost $3m^2 \cdot \omega(g) + 2mn \cdot \omega(F) + 2m \cdot \omega(g) + 3mn^2 \cdot \omega(F)$.

In summary, for one Hessian evaluation, the costs are the following:

$$\begin{aligned} \text{The procedural method: } & 3m^2 \cdot \omega(g) + 2mn \cdot \omega(F) + 2m \cdot \omega(g) + 3mn^2 \cdot \omega(F), \\ \text{Finite-difference method: } & 3n^2[m \cdot \omega(F) + \omega(g)] \end{aligned}$$

When x is fixed, and λ are drawn from a sample of size k, we are interested in the saving in our method to the finite-difference method. Now, since everything related to F will only be calculated once, our method costs:

$$(3m^2k + 2mk) \cdot \omega(g) + (2mn + 3mn^2) \cdot \omega(F),$$

Whereas the finite-difference method will cost:

$$(3n^2k) \cdot \omega(g) + (3n^2mk) \cdot \omega(F),$$

If m and n are of the same order of magnitude, the first terms in both methods will cancel with each other, and the procedural method will eventually beat the finite-difference method as k getting larger, from the second terms we can tell.

Chapter 4

Two Extreme Cases for Structural-AD

Xu and Coleman [2013] exploit the structured AD and apply it to two extreme cases for gradient computation. These problems are generalized partially separable (GPS) problem and dynamic system computations.
A generalized partially separable problem has the following form:

```
for $i = 1, \cdots, p$
    Solve for $y_i : F_i(x) - y_i = 0$.
and then
    Solve for $z : \bar{F}(x, y_1, \cdots, y_p) - z = 0$.
```

Where the final function value is z, and it can be either a scalar or a vector.
Hence, the corresponding extended Jacobian matrix becomes,

$$J^E = \begin{bmatrix} J_x^1 & -I & & & \\ J_x^2 & 0 & -I & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & 0 & \vdots & 0 & -I \\ \bar{J}_x & \bar{J}_{y_1} & \cdots & \cdots & \bar{J}_{y_p} \end{bmatrix}. \tag{4.1}$$

As we can tell, in the GPS problem, the intermediate variable y_i only depends on the independent variable x, not other intermediate variables.

The dynamic system problem is the other extreme cases. The intermediate variables y_i only depends on the previous intermediate variable y_{i-1}. It takes the following form :

```
for $i = 1, \cdots, p$
    Solve for $y_i : F_i(y_{i-1}) - y_i = 0.$
and then
    Solve for $z : \bar{F}(x, y_1, \cdots, y_p) - z = 0.$
```

where $y_0 = x$, and hence the corresponding extended Jacobian matrix becomes,

$$J^E = \begin{bmatrix} J_x^1 & -I & & & \\ 0 & J_{y_1}^2 & -I & & \\ \vdots & \vdots & \ddots & \ddots & \\ 0 & 0 & \vdots & J_{y_{p-1}}^p & -I \\ \bar{J}_x & \bar{J}_{y_1} & \cdots & \cdots & \bar{J}_{y_p} \end{bmatrix}. \tag{4.2}$$

Xu and Coleman [2013] performed comparison for computational time and memory usage between structured AD and regular

	Reverse Mode		Structured Gradient		memory	time
p	Memory (MB)	Times (s)	Memory (MB)	Times (s)	ratio	speedup
10	29.27	3.75	3.56	3.15	8.22	1.19
30	87.71	10.27	3.62	8.96	24.23	1.15
50	146.17	16.98	3.68	15.05	39.72	1.13
100	292.30	33.76	3.85	29.73	75.92	1.14
150	438.43	51.00	3.99	45.25	109.88	1.13

Table 4.1: Memory usage and running times of the gradient computation based on the structure and plain reverse-mode AD for the dynamic system.

	Reverse Mode		Structured Gradient		memory	time
p	Memory (MB)	Times (s)	Memory (MB)	Times (s)	ratio	speedup
10	48.55	4.86	13.55	3.08	3.58	1.58
30	214.73	14.66	13.58	9.96	15.81	1.47
50	473.07	24.54	13.60	14.95	34.78	1.64
100	1232.21	52.18	13.68	29.95	90.00	1.74
150	2547.30	837.77	13.72	57.56	185.66	16.06

Table 4.2: Memory usage and running times of the gradient computation based on the structure and plain reverse-mode AD for the generalized partial separability problem.

Chapter 5

CVaR Optimization with the Structural Gradient/Hessian Calculation

We will show in this section how the structural gradient/Hessian calculation method reduces the computational cost in the CVaR optimization.

Consider the minimization problem

$$\min_{x \in \Omega} f(x, \mu) \tag{5.1}$$

where f is a smooth (twice continuously differentiable) function of (x, μ), x is the vector of n real design variables, μ is a vector of p parameters, and Ω is a connected region in $\Re^n$. The problem we address is concerned with the situation where μ is uncertain. In particular, we suppose the distribution of μ is known (or can be estimated). In order to manage "near worst case" outcomes, one approach is a CVaR-minimization approach. In particular, consider the CVaR minimization problem:

$$\min_{x \in \Omega} CVaR_\beta(f(x, \mu)) \tag{5.2}$$

The computational solution of (5.2) will be discussed below. However, the evaluation of $CVaR_\beta(f(x, \mu))$ at a given x requires the evaluation of $\{f(x, \mu_i), i = 1 : m\}$ and this can be very expensive. One approach to (5.2) is to use the $\epsilon - method$ by Coleman et al. [2009]. Specifically, given a resolution parameter $\epsilon > 0$, define

$$f_\beta(x,\alpha) = \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} p_\epsilon(f(x,\mu_i) - \alpha) \tag{5.3}$$

where $\alpha \in \Re$, $p_\epsilon(z)$ is a continuously differentiable piecewise quadratic function:

$$p_\epsilon(z) = \begin{cases} z & \text{if } z \geq \epsilon \\ \frac{z^2}{4\epsilon} + \frac{z}{2} + \frac{\epsilon}{4} & \text{if } -\epsilon \leq z \leq \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

Problem (5.2) can then be approximated by

$$\min_{\alpha, x \in \Omega} f_\beta(x,\alpha) \tag{5.5}$$

Note that (5.3) is differentiable and

$$\nabla_{x,\alpha} f_\beta(x,\alpha) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} \nabla_{x,\alpha} p_\epsilon(f(x,\mu_i) - \alpha) \tag{5.6}$$

The last term in (5.6) can be summed over three parts. Specifically,

$$\nabla_{x,\alpha} p_\epsilon(f(x,\mu_i) - \alpha) = \begin{cases} \begin{bmatrix} \nabla_x f(x,\mu_i) \\ -1 \end{bmatrix} & \text{if } f(x,\mu_i) - \alpha \geq \epsilon \\ \left(\frac{f(x,\mu_i)-\alpha}{2\epsilon} + \frac{1}{2} \right) \begin{bmatrix} \nabla_x f(x,\mu_i) \\ -1 \end{bmatrix} & \text{if } -\epsilon \leq f(x,\mu_i) - \alpha \leq \epsilon \\ 0 & \text{if } f(x,\mu_i) - \alpha \leq -\epsilon \end{cases} \tag{5.7}$$

Note: although the gradient is written in piecewise format, it is actually continuous at the critical points.
To make use of Newton's Method (or other methods) to solve the optimization problem in (5.5), we also need to examine the Hessian of $f_\beta(x,\alpha)$,

$$\nabla^2_{(x,\alpha)} f_\beta(x,\alpha) = \frac{1}{m(1-\beta)} \sum_{i=1}^{m} \nabla^2_{(x,\alpha)} p_\epsilon(f(x,\mu_i) - \alpha) \tag{5.8}$$

The last term in (5.8) can be summed over three parts. Specifically,

if $f(x,\mu_i) - \alpha \geq \epsilon$,

$$\nabla^2_{(x,\alpha)} p_\epsilon(f(x,\mu_i) - \alpha) = \begin{bmatrix} \nabla^2_{xx} f(x,\mu_i) & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} \tag{5.9}$$

if $-\epsilon \leq f(x,\mu_i) - \alpha \leq \epsilon$,

$$\nabla^2_{(x,\alpha)} p_\epsilon(f(x,\mu_i) - \alpha) = \begin{bmatrix} \frac{\nabla_x f(x,\mu_i) \nabla_x f(x,\mu_i)^T}{2\epsilon} + \left(\frac{f(x,\mu_i)-\alpha}{2\epsilon} + \frac{1}{2}\right) \nabla^2_{xx} f(x,\mu_i) & , -\frac{\nabla_x f(x,\mu_i)}{2\epsilon} \\ -\frac{\nabla_x f(x,\mu_i)^T}{2\epsilon} & , \frac{1}{2\epsilon} \end{bmatrix} \tag{5.10}$$

if $f(x,\mu_i) - \alpha \leq -\epsilon$,

$$\nabla^2_{(x,\alpha)} p_\epsilon(f(x,\mu_i) - \alpha) = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} \tag{5.11}$$

Note: The Hessian above, very often, is not continuous at the critical points.
It is often not practical to solve (5.5) directly due to the expense of evaluating $\{f(x,\mu_i), i = 1:m\}$ and related gradients, possibly Hessians. But since we have set the foundation using our previously discussed structural-AD method, we can benefit from it with the following local approximations:

$$f(x,\mu_i) \cong f(x,\bar{\mu}) + \nabla_\mu f(x,\bar{\mu})^T (\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T [\nabla^2_{\mu\mu} f(x,\bar{\mu})](\mu_i - \bar{\mu}) := \tilde{f}(x,\mu_i) \tag{5.12}$$

Hence, the objective function now becomes,

$$\tilde{f}_\beta(x,\alpha) = \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} p_\epsilon(\tilde{f}(x,\mu_i) - \alpha) \tag{5.13}$$

Note that use of (5.12) only requires the evaluation of $f, \nabla f$ at a single pair $(x,\bar{\mu})$ for any point (but does require m matrix multiplications). However, the use of the resulting approximate values for $\nabla \tilde{f}_\beta$ must be investigated.

To find the gradient of $\tilde{f}_\beta$, from (5.7), we can see that we need to examine $\nabla_x \tilde{f}(x,\mu_i)$, (written in component-wise)

$$\frac{\partial \tilde{f}(x,\mu_i)}{\partial x_j} = \frac{\partial f(x,\bar{\mu})}{\partial x_j} + \frac{\partial \nabla_\mu f(x,\bar{\mu})}{\partial x_j}(\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T \frac{\partial [\nabla^2_{\mu\mu} f(x,\bar{\mu})]}{\partial x_j}(\mu_i - \bar{\mu}) \tag{5.14}$$

(Note: Another view is to define a vector valued function $F(x,\mu)$ where component i is $f(x,\mu_i), i = 1:m$. It is often possible to design the code for F so that $cost(F) \ll m * cost(f)$.)

5.1 Special Case: Product Form

Generally the approximations (5.12) may be difficult to use since the quadratic changes with x. However, in some speical cases, such as weighted average function, the following *product form* holds:

$$f(x, \mu) = g(x) \cdot h(\mu) + v(x)$$

where $g(x), h(\mu)$ and $v(x)$ are real-valued functions.

And in this case, the situation is more tractable. For ease of explanation, and without loss of generality, assume here that $v(x) = 0$. Clearly,

$$\{f(x, \mu_1), ..., f(x, \mu_m)\} = g(x) \cdot \{h(\mu_1), ..., h(\mu_m)\}$$

and

$$\{\nabla_x f(x, \mu_1), ... \nabla_x f(x, \mu_m)\} = \nabla_x g(x) \cdot \{h(\mu_1), ..., h(\mu_m)\}$$

Obviously then, the local approximation will become,

$$f(x, \mu_i) \cong g(x) \cdot h(\bar{\mu}) + g(x)[\nabla_\mu h(\bar{\mu})](\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T g(x)[\nabla^2_{\mu\mu} h(\bar{\mu})](\mu_i - \bar{\mu}) := \tilde{f}(x, \mu_i)$$

Hence, the objective function now becomes,

$$\tilde{f}_\beta(x, \alpha) = \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} p_\epsilon(\tilde{f}(x, \mu_i) - \alpha)$$

From the equation (7), we can see that, to find the gradient of $\tilde{f}_\beta(x, \mu_i)$, we need to find out $\nabla_x \tilde{f}(x, \mu_i)$, (written in component-wise)

$$\frac{\partial \tilde{f}(x, \mu_i)}{\partial x_j} = \frac{\partial g(x)}{\partial x_j} \cdot \left[h(\bar{\mu}) + [\nabla_\mu h(\bar{\mu})^T](\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T [\nabla^2_{\mu\mu} h(\bar{\mu})](\mu_i - \bar{\mu}) \right]$$

We can see that in the above formula, the braketed expression does not depend on x, hence we can write down the gradient and Hessian immediately:

$$\nabla_x \tilde{f}(x, \mu_i) = \left[h(\bar{\mu}) + [\nabla_\mu h(\bar{\mu})^T](\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T [\nabla^2_{\mu\mu} h(\bar{\mu})](\mu_i - \bar{\mu}) \right] \cdot \bigtriangledown_x g(x)$$

$$\nabla^2_{xx} \tilde{f}(x, \mu_i) = \left[h(\bar{\mu}) + [\nabla_\mu h(\bar{\mu})^T](\mu_i - \bar{\mu}) + \frac{1}{2}(\mu_i - \bar{\mu})^T [\nabla^2_{\mu\mu} h(\bar{\mu})](\mu_i - \bar{\mu}) \right] \cdot \bigtriangledown^2_{xx} g(x)$$

Hence, to implement this problem in Matlab, we only need to record $h(\tilde{\mu}), \nabla_\mu h(\tilde{\mu}), \nabla^2_{\mu\mu} h(\tilde{\mu})$ for all points $(x, \tilde{\mu})$.

5.2 Stock Portfolio Application

The problem is described as follows. Suppose our portfolio contains 10 stocks from five different industries, each industry has two stocks. The 10 stocks we selected in our problem and their corresponding industries are:

- Credit Services: Visa Inc. (V) and MasterCard Incorporated (MA)
- Home Improvement Stores: The Home Depot, Inc. (HD) and Lowes Companies Inc. (LOW)
- Money Center Banks: Royal Bank of Canada (RY) and The Toronto-Dominion Bank (TD)
- Oil and Gas: Exxon Mobil Corporation (XOM) and Chevron Corporation (CVX)
- Drug Manufactures: Pfizer Inc. (PFE) and Bristol-Myers Squibb Company (BMY)

We assume that stock prices are correlated within an industry, but have no correlations between different industries. We are interested in assigning weights to stocks such that the CVaR of the loss variable, for one day horizon, at 95% level, is minimized; we assume that we have a fixed amount of money to invest. Now, let's introduce all the notations,

- $S_{t,i}$: the price for stock i at time t, $i = 1, 2, \ldots, 10$
- $Z_{t,i} := \log S_{t,i}$ the logarithmic prices for stock i at time t, $i = 1, 2, \ldots, 10$
- $a_{t,i}$: the number of shares invested in stock i at time t
- $X_{t,i} := Z_{t+1,i} - Z_{t,i}$: the log-returns over one day horizon from t to $t+1$ for stock i, $i = 1, 2, \ldots, 10$
- V_t : the value of the portfolio at time t, we could fix it to be 1.
- $w_{t,i}$: denotes the relative weight of stock i at time t
- $L_{t+1} := -V_t \sum_{i=1}^{10} w_{t,i}(e^{X_{t+1,i}} - 1)$: denotes the loss from t to $t+1$.

Mathematically, our problem can be formed by a minimization problem as the following:

$$\min_{w_{t,i}} \ \mathrm{CVaR}_{0.95}(L_{t+1})$$

$$\text{s.t.} \ \sum_i w_{t,i} = 1$$

Note that we don't have restrictions on $w_{t,i}$, since we allow short-selling. And the optimized weights do not depend on the portfolio value V_t at time t. Hence, for our portolio optimization problem mentioned before,

$$\min_{w_{t,i}} \ \mathrm{CVaR}_{0.95}(L_{t+1})$$

$$\text{s.t.} \ \sum_i w_{t,i} = 1$$

We can use the following piecewise linear function to realize the above optimization problem

$$\bar{F}_\beta(x,\alpha) = \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} [L-\alpha]^+$$

and

$$\min_x \mathrm{CVaR}_{0.95}(L_{t+1}) = \min_{x,\alpha}(\bar{F}_\beta(x,\alpha))$$

The advantages of using a piecewise linear function is that it can be formulated as a convex programming problem, and many optimization softwares solve this question in polynomial time. However, since we haven't add the variance term into the objective function, which can be expressed as $w^T \Sigma w$, and Σ is the covariance matrix for stocks, the objective function can be quadratic or even more complicated. Notice that the derivative at the critical point of the piecewise linear function is not continuous; this can cause computational inefficiency when we add a quadratic term to the objective function. Hence, we can use the smoothing technique by Coleman, Li, and Zhu [2009]. Instead of using the a piecewise linear function, Coleman, Li, and Zhu [2009] suggests a piecewise quadratic function

$$\widetilde{F}_\beta(x,\alpha) = \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} \rho_\epsilon(L-\alpha)$$

Where $\rho_\epsilon(z)$ is defined as:

$$\rho_\epsilon(z) = \begin{cases} z & \text{if } z \geq \epsilon \\ \frac{z^2}{4\epsilon} + \frac{1}{2}z + \frac{1}{4}\epsilon & \text{if } -\epsilon \leq z \leq \epsilon \\ 0 & \text{otherwise} \end{cases}$$

Our portfolio CVaR formulation based on Alexander *et al* (2006) will becomes:

$$\min_{\alpha,w} \bar{F}_\beta(x,\alpha) = \min_{\alpha,w} \alpha + \frac{1}{m(1-\beta)} \sum_{i=1}^{m} \rho_\epsilon(-w^T(e^{x_i}-1)-\alpha)$$

$$s.t.\ \ w^T e = 1$$

Note that the above objective function is the special case product form introduced in 5.1, hence we use the gradient formula in 5.1 to calculate the gradient of the objective function and feed it into the optimization process. We compare the running time using the structural method in 5.1 with the finite-difference gradient method and present the results in next section.

5.3 Result Analysis

The implementation of previous smoothing CVaR optimization is done on Matlab. We summarize our result here, the last column is the optimal weights for each stock. As we can see, the annualized volatility of MA is the largest, and the weight assigned to MA is smallest among all the stocks; the annualized volatility of PFE and BMY are the smallest, and the weights assigned to them are the largest two weights. These results are consistent with the risk-aversion convention. Moreover, we can tell from the last column that the weights are quite diversified, where the classical Markowitz mean-variance portfolio optimization does not have this feature. Many research papers have shown the the classical mean-variance portfolio optimization faces the problem of highly concentration of the weights, when short-selling is allowed, which is one of the main reason that industry did not widely use the classical mean-variance portfolio optimization. CVaR optimization overcomes this shortcoming in the classical theory.

	mean	Volatility (daily)	Volatility (annually)	weight
V	0.0009297925	0.02326158	0.444412	0.1056
MA	0.0007880421	0.02530042	0.483364	0.0220
HD	0.0005244788	0.01969459	0.376265	0.1276
LOW	0.0002576364	0.02219893	0.42411	0.0316
RY	0.0003038427	0.02065314	0.394578	0.0583
TD	0.0003510037	0.01979393	0.378163	0.1231
XOM	0.0002296607	0.01767983	0.337773	0.0655
CVX	0.0004691025	0.01911334	0.36516	0.1475
PFE	0.0002349748	0.01580621	0.301977	0.1592
BMY	0.0005091557	0.01587122	0.303219	0.1596

From the running time comparison table, we can see that as the number of sample m getting larger, running time for solving the above optimization problem with finite-difference gradient is increasing dramatically, while the running time using structrual gradient method is growing slowly. Hence, it supports our previous findings.

m	structural method (s)	finite-difference method (s)
100	0.134	0.281
500	0.388	0.913
1000	0.546	2.445
2000	0.858	6.567
5000	1.387	14.772
10000	2.013	30.573

Table 5.1: Running time comparison between structural gradient method and finite-difference method.

Chapter 6

Conclusion

Throughout this research paper, we studied a structural automatic differentiation method (Structural-AD). This method exploits the 'natural structure' of some functions and it makes use of the regular AD to achieve more efficient computing time and less memory usage requirement. In the paper by Xu and Coleman [2013], they applied Structural-AD to two extreme cases, generalized partially separable problem and dynamic system computations. They showed that computing time and memory requirement using Structural-AD is significantly reduced compared to the regular reverse mode. The Structural-AD is first raised by Jonsson and Coleman [1999], who illustrated how natural structure can be used to enable efficient gradient computation. In a paper by Coleman et al. [2012], they used a direct edge separator method to find the structure in the computational "tape", which supported their previous work. Similarly, fast Hessian calculation canalso be recovered when there's natural structure in the function.

As a major contribution of this essay, we apply Structural-AD on some problems in different fields. In Chapter 2, we introduce the theory of automatic differentiation, and the two computation methods - forward mode and reverse mode. The comparison of the computational costs for these two methods is also provided. In Chapter 3, we provide the detailed methodology and derivation of gradient and Hessian calculation in the Structured-AD context, following which, the computational costs of calculating gradient and Hessian are also presented compared to the finite-difference method. We showed that the computational cost is significantly reduced using Structural-AD method compared to the finite-difference method. In Chapter 4, we applied Structural-AD to two extreme cases, generalized partially separable problem and dynamic system computations and showed that computing time and memory requirement using Structural-AD is significantly reduced compared to the regular reverse mode. In Chapter 5, we test structural-AD on a Conditional Value-at-Risk (CVaR) optimization problem, specifically with the underlying function describing the loss function of a stock portfolio.

References

LLC. Cayuga Research Associates. Admat-2.0 users guide, 2009. URL `http://www.cayugaresearch.com/`.

Thomas F. Coleman, Yuying Li, and Lei Zhu. Min-max robust and cvar robust mean-variance portfolios. *The Journal of Risk*, 11(3), 2009.

Thomas F. Coleman, Xin Xiong, and Wei Xu. Recent advances in algorithmic differentiation. 87:209–219, 2012.

Gudbjorn Jonsson and Thomas F. Coleman. The efficient computation of structured gradients using automatic differentiation. *SIAM Journal of Scientific Computing*, 20(4): 1430–1437, 1999.

Arun Verma and T. F. Coleman. The efficient computation of sparse jacobian matrices using automatic differentiation. *SIAM Journal of Scientific Computing*, (4):1210–1233, 1998.

Wei Xu and Thomas F. Coleman. The efficient evaluation of structured gradients (and underdetermined jacobian matrices) by automatic differentiation. 2013.