An Object-Oriented, Maximum-Likelihood
Parameter Estimation Program for GARCH\((p,q)\)

by

Joseph B. Irineo

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 4, 1999

© Copyright 1999 Joseph B. Irineo. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

May 17, 1999

Certified by

Daniel Jackson

Thesis Supervisor

Accepted by

Arthur C. Smith

Chairman, Department Committee on Graduate Theses
An Object-Oriented, Maximum-Likelihood Parameter Estimation Program for GARCH\((p,q)\) 

by 

Joseph B. Irineo 

Submitted to the 

Department of Electrical Engineering and Computer Science 

In Partial Fulfillment of the Requirements for the Degree of 

Master of Engineering in Electrical Engineering and Computer Science 

Abstract 

This thesis studies the approach of object modeling when applied to the compute-intensive problem of maximum-likelihood estimation, a problem that has generally been tackled using functional approaches. Specifically, this research takes the problem of finding maximum-likelihood parameter estimates for an econometric model called Generalized Autoregressive Conditional Heteroskedasticity, and develops an object model for it. What results is an object model that clearly lays out the nature of the information in the problem, showing that there is complexity in its structure. This complexity consequently expresses a deficiency in a functional approach that assumes that the capture and expression of information can be easily solved, and that effort is therefore better spent focusing on the process. This thesis argues the opposing view and states that most of the complexity in a program is in its data, and that the best way to manage this complexity is through the design and implementation of an object model.

Thesis Supervisor: Daniel Jackson 
Title: Associate Professor, Electrical Engineering and Computer Science
Acknowledgments

Much of the research for this thesis was done while spending my summer and fall 1998 academic terms working in the Corporate Portfolio Management group (formerly the Corporate Risk Management group) at Bankers Trust Company in New York. This work was organized through the VI-A Internship program run by the Electrical Engineering and Computer Science Department at MIT.

All these people played a crucial part in bringing my Masters of Engineering thesis work to life, and I would like to acknowledge them and thank them:

First, I would like to thank my sponsors at Bankers Trust, Thomas Daula and Michael Morell. I first met them way back in August 1996, when I was about to finish my first VI-A assignment at Bankers Trust and was looking for an area at BT in which to do my undergraduate project, and hopefully, my M.Eng thesis work. I found their work in risk management immensely interesting, specifically in how they were using computer systems to meld finance and mathematics and applying their approach to quantify market risks taken by BT on a daily basis.

They agreed to take me onto their group in August 1996, and I want to thank them for doing so. This culminated in my working for them during the subsequent summer of 1997 and during the summer and academic fall term of 1998. This was pretty generous of them, considering that I had no prior experience in this field; they always had a confidence in me that I would pick things up and learn along the way, and I want to thank them for that. And lastly, I want to thank them for assigning me to such a wide variety of projects within risk management over the past two years. I obtained a good survey of the field, and eventually one of these projects became the case study for this thesis on object modeling: writing a maximum-likelihood estimation program for GARCH(p,q).

I would also like to thank Barbara Reeves at Bankers Trust and Jennifer Lemaigre, who was with BT until the end of the summer of 1997. They both took responsibility for the well-being of the VI-A program and the VI-A interns at Bankers Trust, and my enjoyable VI-A experience was in no small part due to them. In addition, I would also like to thank Barbara Reeves for having BT fund my VI-A Fellowship while I was working on my thesis there during the fall term of 1998.

My thesis supervisor at MIT, Professor Daniel Jackson, was of immense help to me
throughout the whole year. First of all, I would like to thank him just for agreeing to advise me on my thesis back in July of 1998. It was quite a risk in my opinion, as I had not done any previous research work for him. In addition, my thesis work at that point was my own independent work, and it did not immediately present any apparent benefit for his own research objectives. Nevertheless, he took me on as an advisee, and I would like to thank him for that.

One thing that Professor Jackson did was give my thesis direction. What I had in July 1998 was a whole bunch of ideas, but not much unifying direction. I knew I was going somewhere with my research, but where exactly was unknown to me. Fortunately, Professor Jackson was able to take my slew of ideas and incoherent diagrams and clarify the direction of my thesis work.

Specifically, I credit to him the notions of the History object and the Slot object given in Chapter 3. He reintroduced to me the idea of the Environment object, analogies of which I learned four years ago in a freshman introductory course to computer science. He gave me rigorous building blocks with which to build my object models, and what resulted is the complex and interesting object model given in Chapter 3. He was crucial in making what started off as a task in programming into a pretty respectable thesis exploring the applicability of object modeling to a problem that most programmers intuitively tackle with a functional approach.

Finally, I would like to thank my parents for instilling in me a work ethic that I had to call upon numerous times over my years at MIT, especially during the weeks leading up to the completion of this thesis. This work ethic made the last five years of my life here at MIT productive and enjoyable ones.
# Contents

1 Introduction ........................................................................................................................................ 11
  1.1 Motivation for Object Modeling ................................................................................................. 12
    1.1.1 Focus on Essential Structure ............................................................................................. 12
    1.1.2 Implementation Independence ............................................................................................ 13
    1.1.3 Scalability .......................................................................................................................... 14
    1.1.4 Polymorphism .................................................................................................................... 15
    1.1.5 Disadvantages and Trade-offs ............................................................................................ 15
  1.2 Case Study: Maximum-Likelihood Estimation ............................................................................. 16

2 The Client Problem ............................................................................................................................ 18
  2.1 Motivation for GARCH\((p,q)\) .................................................................................................... 18
  2.2 Econometric Basis ....................................................................................................................... 19
    2.2.1 Classical Linear Regression .............................................................................................. 19
    2.2.2 Value-At-Risk ................................................................................................................ 21
  2.3 Parameter Estimation .................................................................................................................. 22
    2.3.1 Maximum-Likelihood Estimation ..................................................................................... 22
    2.3.2 Berndt, Hall, Hall, and Hausman ..................................................................................... 25
  2.4 Program Requirements ............................................................................................................... 28
    2.4.1 Platform ........................................................................................................................... 28
    2.4.2 Interface ........................................................................................................................... 29
    2.4.3 Input File .......................................................................................................................... 31
    2.4.4 Parameters ....................................................................................................................... 31
    2.4.5 Output ............................................................................................................................. 32
List of Figures

3-1 Object Model for GARCH\((p,q)\) parameter estimation ........................................ 34
3-2 Function hierarchy in GARCH\((p,q)\) parameter estimation ........................................ 35
3-3 Additional constraints for object model, Part I of V ........................................ 36
3-4 Additional constraints for object model, Part II of V ........................................ 37
3-5 Additional constraints for object model, Part III of V ........................................ 38
3-6 Additional constraints for object model, Part IV of V ........................................ 39
3-7 Additional constraints for object model, Part V of V ........................................ 40
3-8 The Name object model .................................................................................................. 40
3-9 The Environment object model ....................................................................................... 42
3-10 Invalidly co-existing Environment objects .................................................................... 42
3-11 Environment to Value projections ............................................................................... 43
3-12 The Series object model .............................................................................................. 44
3-13 Valid Series object A and invalid Series object B ..................................................... 44
3-14 A Series to Value projection ......................................................................................... 45
3-15 The History object model ............................................................................................ 45
3-16 Valid History object A and invalid History object B .................................................. 46
3-17 Projecting a History object to a HistorySeries object and a HistoryEnvironment object .............................................................................................................................. 46
3-18 Valid EpsilonFunction object A and invalid EpsilonFunction object B ..................... 51
3-19 The Function object model ........................................................................................... 53
3-20 System of equations captured by an EpsilonSlopeFunction object ......................... 54
3-21 Valid GarchFunction object A and invalid GarchFunction object B ......................... 56
3-22 System of equations captured by a GarchSlopeFunction object ................................. 57
3-23 System of equations captured by a LikelihoodSlopeFunction object ......................... 59
B-27 The LikelihoodFunction class specification ....................... 117
B-28 The first part of the LikelihoodSlopeFunction class specification .... 118
B-29 The second part of the LikelihoodSlopeFunction class specification ... 119
List of Tables

3.1  The Function objects used in the model and their equivalents in GARCH(p,q) 52
3.2  The implementation classes for object model 62
Chapter 1

Introduction

A difficult stage in software design is decomposition, where the developer attempts to reduce client requirements into simpler modules. If done correctly, this stage will separate the software developing task into modules that can be tackled independently, then combined later to form the total solution. One robust approach to decomposition is object modeling, a software design methodology that reduces problem complexity through the use of data abstractions.

This chapter introduces the goal of object modeling and the advantages it offers over designing software using a procedural approach. It presents a data-intensive problem that has traditionally been approached from a procedural point of view and proposes solving it using object modeling techniques. The goal is to determine whether or not this approach offers a better way to organize and solve this problem.

Chapter 2 gives an overview of the problem—creating a program to efficiently calculate maximum-likelihood parameter estimates for an econometric model called Generalized Autoregressive Conditional Heteroskedasticity, abbreviated hereafter as GARCH\((p,q)\). This model is frequently used in the finance industry, primarily by risk managers and portfolio managers. Chapter 2 will then conclude with a presentation of the program requirements of these users.

Chapter 3 gives an object model for a program to solve the GARCH\((p,q)\) parameter estimation problem. This design is presented in three phases: the design object model, the snapshot, and the code object model.

Chapter 4 presents the implementation of the design given in Chapter 3. The chapter
first presents the *module dependency diagram* of the code modules developed for the final
program. It then gives the specification for each of the major classes used in the program.
Finally, the methods used to test the implementation are given.

Chapter 5 evaluates the design and implementation of the program, comparing design
and implementation choices made over other alternatives. The chapter concludes the thesis,
stating that object modeling gives the software developer a power over program state and
information that procedural approaches lack. Although there may be some sacrifices in
performance, they are small compared to the immense benefit of organization and program
correctness that object modeling provides.

### 1.1 Motivation for Object Modeling

A problem can be viewed as either an issue involving processes or an issue involving things.
One can probably argue that it is best to take a middle road and argue that problems should
be approached by looking at both processes and things. However, in order to understand an
especially complex problem, decomposition is a necessary first step. Decomposition forces
one to, at the minimum, favor one point of view over the other.

This section argues that by favoring object modeling, one captures the essential nature
of a problem, its *semantics*. During the design phase of software construction, constructing
an object model for the client problem translates into a software blueprint that focuses on
the problem’s essential structure and is scalable and polymorphic in the face of changes.

#### 1.1.1 Focus on Essential Structure

The argument for favoring an object focus is that objects drive a program’s complexity.
It is the interaction among objects and the state maintenance within them that give rise
to the complexity of a problem and the higher level behavior of systems. One can argue
that writing a computer program involves constructing procedures, so it is the process that
matters. But generally, the difficult, and therefore interesting, procedures are those that
manipulate and target complex data types.

In other words, decomposing a problem to its constituent objects in the design phase
allows one to see the problem’s essential structure. While processes come and go, objects in
a problem are generally the ones that exist the longest throughout the problem’s lifetime.
The persistent core of a problem are its objects, their interrelations, and how these objects are maintained. Processes are still crucial to the problem, but in the end they are considered in terms that describe how they affect a problem's object or its relationship with another object.

Thus, a practical benefit of an object model is that it lends itself very easily to division of labor among developers. By establishing a framework and terminology of processes based upon common objects, there can be semantic consistency among separately coded modules. A developer can more easily understand the behavior of another developer's module just by understanding a common base of objects. Communication between modules is more fluid; for example, vector objects can be passed among different developer code bases since there is an agreement on how a vector should behave.

Another advantage of this focus is that the object model does not force one during the design stage to specify and commit to the methods of each object in the model. The only items to be found in a design object model are the sets of objects in a problem, any constraints placed on object migration among these sets, the relations among these objects, and any mutability constraints imposed upon these relations. Nowhere does one see a specification for a method of an object. The point to be made here is that the designer must first place the effort on the hard part of design, which is organizing and characterizing the information within the client problem. This is not to say that specifying the method interface for an object is easy; this says that its specification should naturally follow during implementation if the purpose of an object and its relation to other objects are firmly established.

1.1.2 Implementation Independence

The object model allows one to characterize abstractions for data that the target program must ultimately handle. If done correctly, these abstractions can be expressed free from limitations of an implementation language or platform. This results in a design that can be implemented in almost any language, as the design should give a structure for extending the data types of the target language, if such an extension is necessary.

However, this is not as easily done with a procedural approach since it is does not provide semantics for characterizing data. A recourse for one using the procedural approach is to loosely specify the nature of the data, or one can make an assumption about how the data
will be represented in the context of an implementation language. Either way, the promise of the design being implementation independent is ruined. In the first case, the specification might not provide enough direction on how to implement abstractions in code. In the second case, assumptions made might not hold across languages.

Pushing even further, object modeling does not even make the opposing assumption that data abstraction must be used to implement every single object in the model. Although one generally favors the use of classes in code to implement objects in an object model, there might be times when it is just simpler to implement some objects using primitive data types such as characters, integers, doubles, etc. This reiterates the advantage outlined in the previous section above; the object model only models the information that characterize the target problem; it does not commit the developer in the design stage to any implementation approach.

1.1.3 Scalability

Problems in the world tend to scale in complexity by layering abstraction upon abstraction. Since the object model expresses abstractions as objects, it can include similar layering to handle further complexity. For example, one might begin with a piece of software designed to control taxiing, landings and take-offs from an airport. This same software, if designed using an object model, can be scaled to control air traffic within a collection of airports by augmenting the design to have an airport control the ability to schedule not only airplanes but other airport controllers. One can then subject local airport air traffic controllers to the control of one global controller.

The progression from a local controller to a global controller is made because object models can be augmented to mimic real-world behavior. A procedural approach might become too focused on a process and miss the behavior that caused that process. Rather than focusing on how a controller relates to airport terminals, planes, and runways, the procedural approach might have focused on the actual process of scheduling. This can yield a perfectly viable design, but such a design becomes strained when one has to augment the duties of the controller to include scheduling the activities of other controllers.
1.1.4 Polymorphism

Object modeling can also yield designs that naturally accommodate editions to the client problem. For example, the developer might encounter later in the life cycle of a program the unexpected situation of having to extend the data types handled by an object. For example, one might have to extend an integer hash table object to also support floating point values. If the object model built for the hash table is correctly constructed, extending the model to accommodate the additional floating point type should be natural. For example, the object model for a hash table might only care about keys and values, but does not specify how those keys and values are specified in code. Extending the hash table to accommodate both integers and floating point values would only require defining these types as two subsets of the set of values.

1.1.5 Disadvantages and Trade-offs

Nevertheless, object modeling is difficult. It requires one to intimately understand the problem at hand and make decisions about what are its essential kernels. A good thing about a procedural approach is that the design stage can basically boil down to knowledge acquisition. One can focus on getting an expert to give a set of step-by-step instructions on how to get the required work done. Implementation becomes a rote operation of encoding these instructions in a program. Taking a procedural approach can make one's job straightforward and quick, and is indeed a natural approach for implementing routines and algorithms.

Furthermore, an implementation of an object model might not be as fast in practice as a program written from a purely functional standpoint. As will be seen later in Chapters 3 and 4, an object model imposes many constraints on its objects, and adhering to those constraints in code generally requires the implementation of data abstractions, each with a representation invariant. Each invariant of each abstraction must then be maintained throughout the program's use of the abstraction, and ensuring these invariants requires clock cycles that could be used for other things. In addition, the introduction of abstractions and invariants hide data that might have been exposed publicly to all areas of code in a procedural design. Access to such data must now occur through calls to observer and mutator methods that objects expose. Such access layers add further overhead to the
implementation.

The question comes down to whether or not object modeling's potential to organize and simplify complexity outweighs the performance gains in a purely procedural approach. To explore this issue further, the next section presents the problem of maximum-likelihood estimation where computational speed and data processing are both crucial features. The question presented in this case study and analyzed throughout this thesis is whether or not object modeling will offer a better solution than a procedural approach, whether the gains in program organization and guarantees of correctness far outweigh any loss in performance.

1.2 Case Study: Maximum-Likelihood Estimation

Maximum-likelihood estimation is a class of problems that involves taking a parameterized model and finding parameter values that best fit the model to observed data. It relies on many samples of data to satisfy statistical assumptions and requires iterative evaluations of a fitness function to determine goodness of parameter estimates. It is a class of problems that can be both data and compute intensive.

Clients of this problem have generally placed a priority on speed, so this problem has been traditionally solved from a procedural standpoint, and such approaches have been very successful. However, these clients have generally been econometricians and risk managers who use maximum-likelihood estimation in isolated statistical studies of data. Basically, these clients have turned to scripts and programs written by others within the field or within the econometric software industry and used these programs for custom, *ad hoc* jobs, where getting the job done as quickly as possible was the overriding goal.

Many of these people work in large financial firms that have back-end production systems printing systematic reports based upon their research. Good examples of such systems are the risk management systems in Wall Street firms that value the riskiness of traded positions using complicated econometric models. The job of these people is to come up with better performing models to replace the current ones in production. A problem is encountered when these clients determine through their research efforts that maximum-likelihood estimation is a good thing and should be integrated in a larger system.

How does one scale programs targetted for small research jobs to roles serving the needs of large production systems? One recourse is to start from the beginning and design a
new program that solves the problem of maximum-likelihood estimation while at the same time has the ability to fit within the larger scheme of an industrial-strength system. The rest of this thesis will take this course, presenting how to do that using an object model approach. It will explore whether this approach does indeed offer a good way to design for complexity and intensive data needs. At the same time, it will explore whether this approach can be molded to accommodate purely computational parts of a problem. Although maximum-likelihood estimation does deal with lots of data, it is also very algorithmic in nature. As such, the thesis will also explore a way for placing an algorithmic process within a broader, object-driven framework.
Chapter 2

The Client Problem

This chapter presents an instance of a maximum-likelihood estimation problem as this thesis’s case study in object modeling. The problem involves generating best-fit parameter estimates for an econometric model called Generalized Autoregressive Conditional Heteroskedasticity. The main purpose of this chapter is to give background information on this problem, detailing the economic and statistical basis for this model. This chapter presents why this problem is interesting, and presents the requirements of those who would want to have a software solution for it.

2.1 Motivation for GARCH\((p,q)\)

Any effective financial risk management scheme will involve capturing and quantifying the risks associated with taking monetary positions in financial markets. Only through robust, tangible measures of how much loss (or profit) a position can incur can one reliably decide whether or not risks taken are worth the expected rewards.

Quantifying financial risk has, in general, become a computation-heavy process. In their drive for quantifying risk, corporate departments chartered with managing the financial risks of their companies have have turned increasingly towards statistical models. A main benefit of using statistical models is their foundation in mathematics, which provides a scientific basis for their results. In addition, statistical models in the end produce numbers, and the whole mission here is to place a value on risk.

An important task in risk management is quantifying the volatility in an investment product’s value. An instrument’s risk is directly dependent on its volatility. A highly
volatile instrument implies large value fluctuations over time, implying a greater chance of losing value than another, less volatile instrument. So, modeling volatility allows one to model how much risk one is taking investing in one product versus other products, in turn allowing one to manage risk.

One model for volatility is GARCH($p,q$), a recursive specification for how volatility moves through time. Portfolio and risk managers find this model attractive since it asserts that the current volatility of an instrument's value is explained by past prevailing volatility and movements in value. Thus, volatile periods are followed by further volatile ones and calm ones followed by further calm ones. Empirical evidence supports such a specification since volatility clumping can be observed when studying the volatility of interest rates and option prices.

Such a specification can seem reasonable in an economic sense as highly volatile periods in a market are most probably indications of disequilibrium. So, if one assumes that it takes time for markets to reach their equilibria, one can assert that it is more likely that highly volatile periods will be followed by further periods of volatility rather than calm ones. And on the other hand, one can also assert that it would take time for a market in equilibrium to move into disequilibrium. Thus, calm periods generally follow calm periods.

### 2.2 Econometric Basis

GARCH($p,q$) was developed in 1986 by Tim Bollerslev within the field of econometrics, the statistical study of economics [5]. This section gives the econometric derivation of and justification for GARCH($p,q$), showing how it extends a traditional mathematical specification used throughout econometrics called classical linear regression and its place within the risk management framework of Value-At-Risk.

#### 2.2.1 Classical Linear Regression

Classical linear regression, at least when applied to time varying random variables, begins by stating that a linear specification for a temporal random variable $y$ observed over a
length of time $T$ is given by:

$$y_t = b_0 + \sum_{i=1}^{k} b_i x_{it} + \varepsilon_t, \quad k \geq 0, \quad 0 < t \leq T \quad (2.1)$$

where the $b$'s are constants. The $x$'s are temporal, non-stochastic variables observed over the same period of time, and one assumes that no two $x$'s are linearly related to each other. The non-stochastic nature of the $x$'s implies that the uncertainty in the behavior of $y$ is captured by the stochastic behavior of $\varepsilon$, the residual. This further implies that the variance of $y$ at time $t$ is simply the variance of the residual at time $t$:

$$\text{Var}(y_t) = \text{Var}(b_0 + \sum_{i=1}^{k} b_i x_{it} + \varepsilon_t)$$

$$= \text{Var}(\varepsilon_t) \quad (2.2)$$

To derive the conditional variance of $\varepsilon$, and thus the conditional variance of $y$, assume the probability density function of $\varepsilon$ at time $t$ is distributed normally with a mean of 0 and a volatility, or standard deviation, of $\sigma$, conditional on the information set $\psi$ known at time $t$:

$$\varepsilon_t | \psi_{t-1} \sim \mathcal{N}(0, \sigma_t) \quad (2.3)$$

where $\psi$ is the set of all volatilities previously observed for $\varepsilon$. The conditional variance, then, of $\varepsilon$ at time $t$ is the square of $\sigma$ at time $t$.

GARCH($p,q$) states that this conditional variance through time is driven by the past $p$ squared residuals for $y$ (ARCH terms) and by the past $q$ forecasts of variance for $y$ (GARCH terms)$^1$:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{p} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2 \quad (2.4)$$

$^1$Thus, the ($p,q$) modifiers. A GARCH(1,1) model would drive current variance by the last observations for $\varepsilon$ and $\sigma$, i.e., $\sigma_t^2 = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2$. 

20
where

\[ p > 0 \]
\[ q \geq 0 \]
\[ \alpha_0 > 0 \]
\[ \alpha_i \geq 0 \quad i = 1, \ldots, p \]
\[ \beta_i \geq 0 \quad i = 1, \ldots, q \]

### 2.2.2 Value-At-Risk

The GARCH\((p, q)\) specification can be used to derive forward expacements of \(\sigma\), making it useful within the framework of Value-At-Risk, or VaR for short. VaR is widely used in corporate risk management departments as a measure of the worst possible loss due to a position taken in a market, given a certain level of confidence (a measure of probability).

A simple example using VaR is as follows. Assume an investor has $100 million invested in stock issued by International Business Machines Corporation (IBM). Examining the per-share price of IBM equity between July 7, 1997, and July 7, 1998\(^2\), the average daily return to a share of IBM stock is 0.09%. The variance of daily returns is 0.0004327, implying that its standard deviation is 2.08%.

Now assume that returns to a share of IBM can be modeled by a random variable that is distributed normally. According to our sample then, daily returns to IBM is normally distributed with a standard deviation of 2.08% around a mean of 0.09%. Given this distribution, one can now calculate the worst possible loss to our portfolio of IBM stock in a VaR framework.

Given a probability \(p\) called the confidence level, a VaR calculation will determine the loss to the portfolio such that probability of losses exceeding this value is \(1 - p\). Thus, if we desire a confidence level of 99%, then our VaR calculation will give us the “worst” possible loss such that any greater loss can only occur 1% of the time. For a random variable that is normally distributed about its mean \(m\) with standard deviation \(\sigma\), any value that is less than \(m - 2.33 \times \sigma\) will occur 1% of the time. Thus, if we set a confidence level of 99%, daily returns below \(0.09\% - 2.33 \times 2.08\% = -4.76\%\) will only occur 1% of the time. Translated to a VaR framework, we say: Under normal market conditions, the most our portfolio of

\(^2\)Data retrieved from Bloomberg L.P., a company that specializes in supplying financial data.
IBM equity can lose over a day is $4.76 million.

What GARCH(p,q) can do in such a framework is replace the 2.08% number, which was an unconditional volatility, with a volatility conditional upon information from the past, i.e., past returns and volatilities in IBM stock. One can argue that this number would have a richer meaning, appealing to the observation that volatility tends to group into periods of high levels and low levels. It also appeals to the argument that one should differently weight past periods, maybe giving more weight to events that occurred in recent memory. And as mentioned before, GARCH(p,q) provides a plausible model for option volatility, allowing one to assess the risk of optionality in a VaR framework.

2.3 Parameter Estimation

However attractive a model GARCH(p,q) is from a conceptual standpoint, those who would choose it as a risk management tool become burdened with the problem of obtaining estimates for the \( \alpha \) and \( \beta \) weighting parameters. This specification is recursive and nonlinear, and there does not exist a closed form solution for it\(^3\) The approach generally accepted among econometric circles is to obtain parameter values through maximum-likelihood estimation. Most econometricians specifically find the Berndt, Hall, Hall, and Hausman method of maximum-likelihood estimation desirable for this problem.

2.3.1 Maximum-Likelihood Estimation

In general, if we have observed values \( g_1, g_2, \ldots, g_n \) for random variable \( g \), maximum likelihood estimation (MLE) maximizes the probability density function for observing such values from the sample space for \( g \):

\[
f(g_1) \cdot f(g_2) \cdot \cdots f(g_n)
\]

where \( f_g \) is the probability density function for \( g \). An equivalent and generally more convenient expression of this problem is to instead maximize the log-likelihood function of

\(^3\)Although, there does exist a closed form solution for GARCH(1,1), given certain restrictions on the \( \alpha \) and \( \beta \) weights.
observing these values, namely:

\[ \ln f_g(g_1) + \ln f_g(g_2) + \cdots + \ln f_g(g_n) \]

This method is equivalent to the former because the logarithmic function is monotonic and increasing.

The idea behind using MLE for GARCH\((p,q)\) is to choose estimates for the GARCH\((p,q)\) parameters that yield a series of \( \sigma \)'s, which in turn yield conditionally normal distributions that would most likely generate the observed \( \varepsilon \)'s. We apply maximum-likelihood to estimating Eqn. (2.4) by first translating Eqn. (2.3), the conditional normality assumption for the \( \varepsilon \) term in Eqn. (2.1), into a probability distribution function for \( \varepsilon \):

\[ f_{\varepsilon|\psi}(\varepsilon_t | \psi_{t-1}) = \frac{1}{\sqrt{2\pi \sigma_t^2}} e^{-\frac{\varepsilon_t^2}{2\sigma_t^2}} \quad (2.5) \]

where

\[ \sigma_t^2 = \omega' z_t \quad (2.6) \]

and

\[
z_t = \begin{pmatrix} 1 \\ \varepsilon_{t-1}^2 \\ \vdots \\ \varepsilon_{t-p}^2 \\ \sigma_{t-1}^2 \\ \vdots \\ \sigma_{t-q}^2 \end{pmatrix}, \quad \omega = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_p \\ \beta_1 \\ \vdots \\ \beta_q \end{pmatrix}
\]

and \( \omega' \) is the transpose of \( \omega \).

Thus, we have expressed \( \sigma^2 \) as a function of \( \alpha \)'s, \( \beta \)'s, and past values of \( \varepsilon \). Additionally, we have expressed it as a recursive function, depending on past values of itself. However, by (2.1), it is possible that we do not observe the values of \( \varepsilon \); given our observations of \( x \) and \( y \), \( \varepsilon \) is determined by the—as yet unknown—parameters \( b_0, \ldots, b_k \). To express \( \sigma^2 \)'s
dependence on these unknown parameters, we redefine (2.1) using matrix notation. Let

\[ X = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{k1} & x_{k2} & \cdots & x_{kT} \end{pmatrix} \]

and

\[ Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{kT} \end{pmatrix}, \quad b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{pmatrix}, \quad \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_{kT} \end{pmatrix} \]

This allows us to express (2.1) as:

\[ Y = Xb + \varepsilon \quad (2.7) \]

Let us now define a vector of parameters, \( \theta \):

\[ \theta = \begin{pmatrix} b \\ \omega \end{pmatrix} \]

We now have a compact way to express our unknown parameters. Using such notation, we transform (2.5) into a log-linear function of \( \theta \):

\[ l_t(\theta) = \ln f_{\varepsilon_t|\psi}(\varepsilon_t | \psi_{t-1}) = \frac{1}{2} \ln 2\pi \sigma_t^2 - \frac{\varepsilon_t^2}{2\sigma_t^2} = \frac{1}{2} \ln 2\pi - \frac{1}{2} \ln \sigma_t^2 - \frac{\varepsilon_t^2}{2\sigma_t^2} \]

As will be seen shortly, what we are interested in are in the derivatives of \( l \), so we redefine \( l \) by dropping the constant term:

\[ l_t(\theta) = -\frac{1}{2} \ln \sigma_t^2 - \frac{\varepsilon_t^2}{2\sigma_t^2} \quad (2.8) \]
We then maximize the probability density of observing the sequence $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_T$:

$$f_{e|\psi}(\varepsilon_1 \mid \psi_0) \cdot f_{e|\psi}(\varepsilon_2 \mid \psi_1) \cdots f_{e|\psi}(\varepsilon_T \mid \psi_{T-1})$$

This implies that the log-likelihood function that we wish to maximize is the sum of (2.8) over the observed time period for $y$:

$$\mathcal{L} = \sum_{t=1}^{T} l_t(\theta)$$

The goal then is to compute estimates for $\theta$ that yield the largest values for $\mathcal{L}$.

### 2.3.2 Berndt, Hall, Hall, and Hausman

It can be observed that at its maximum, the partial derivative of $\mathcal{L}$ with respect to any one parameter will be zero. Thus, a way to determine maximizing values for $\theta$ would be to calculate the first order derivatives of $\mathcal{L}$ with respect to each parameter in $\theta$ and then set the resulting equations all to zero. Yet, the first derivatives of GARCH($p,q$) are even more complex than the original specification, so finding a closed form solution to that system is impossible.

However, there exist iterative techniques that allow one to find a solution vector for $\theta$ without a closed form solution. Some of these iterative techniques are greedy algorithms. When applied to this problem, they would simply adjust parameter estimates such that every adjustment increases $\mathcal{L}$. When the iterative adjustments no longer increase $\mathcal{L}$ to a certain precision, convergence is declared and the last solution vector for $\theta$ is returned.

These greedy algorithms are a variant of Newton's method in one form or another. The idea behind Newton's method is to look at the second order derivatives of the target function (in this case $\mathcal{L}$) and make parameter adjustments that increase the absolute value of these derivatives. The idea is that the second order derivatives measure the concavity of $\mathcal{L}$ and that the part of $\mathcal{L}$ where it reaches a maximum with respect to $\theta$ is negatively concave. Therefore, moving towards increasing negative concavity will move towards a point where $\mathcal{L}$ reaches a maximum.

Given $\theta^{(i)}$ as the current estimate of $\theta$, Newton's method yields the adjusted estimate
\( \theta^{(i+1)} \) as so:

\[
\theta^{(i+1)} = \theta^{(i)} + \frac{1}{T} \mathbf{H}^{-1}(\theta^{(i)}) \mathbf{g}(\theta^{(i)})
\]  

(2.10)

Here \( \mathbf{g} \) is the gradient, or vector of first order derivatives of \( \mathcal{L} \) with respect to \( \theta \):

\[
\mathbf{g} = \frac{\partial \mathcal{L}}{\partial \theta}
\]

(2.11)

\[
\mathbf{g} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \theta}
\]

The Hessian matrix \( \mathbf{H} \) encapsulates the concavity of \( \mathcal{L} \).

The computation of the Hessian is very prohibitive, and there exists a variant of Newton's method that offers a simpler proxy. Berndt, Hall, Hall, and Hausman in their 1974 paper propose using the outer-product-of-the-gradient as an estimate for the Hessian [6]:

\[
\mathbf{H} = \frac{1}{T} \mathbf{G}'\mathbf{G}
\]

(2.12)

where \( T \) is the length of time used in Eqn. (2.1). \( \mathbf{G} \) is the matrix of contributions to the gradient of \( \mathcal{L} \), and it is given by:

\[
\mathbf{G} \equiv \begin{pmatrix}
\frac{\partial l_1}{\partial \theta} \\
\frac{\partial l_2}{\partial \theta} \\
\vdots \\
\frac{\partial l_T}{\partial \theta}
\end{pmatrix}
\]

(2.13)

Calculating \( \mathbf{G} \) will require calculating a vector of partial derivatives for \( l \) with respect to \( \theta \) at each time \( t \):

\[
\frac{\partial l_t}{\partial \theta} = -\frac{\varepsilon_t}{\sigma_t^2} \cdot \frac{\partial \varepsilon_t}{\partial \theta} + \frac{1}{2\sigma_t^2} \left[ \frac{\varepsilon_t^2}{\sigma_t^2} - 1 \right] \cdot \frac{\partial \sigma_t^2}{\partial \theta}
\]

(2.14)
where:

$$\frac{\partial \varepsilon_t}{\partial \theta} = \begin{pmatrix}
\frac{\partial \varepsilon_t}{\partial \theta_0} \\
\frac{\partial \varepsilon_t}{\partial \theta_1} \\
\vdots \\
\frac{\partial \varepsilon_t}{\partial \theta_k} \\
\frac{\partial \varepsilon_t}{\partial \theta_p} \\
\frac{\partial \varepsilon_t}{\partial \beta_1} \\
\vdots \\
\frac{\partial \varepsilon_t}{\partial \beta_q}
\end{pmatrix} = \begin{pmatrix}
-1 \\
-x_{1t} \\
\vdots \\
-x_{kt} \\
0 \\
0 \\
\vdots \\
0
\end{pmatrix} \quad (2.15)$$

and:

$$\frac{\partial \sigma^2_t}{\partial \theta} = \begin{pmatrix}
\frac{\partial \sigma^2_t}{\partial \theta_0} \\
\frac{\partial \sigma^2_t}{\partial \theta_1} \\
\vdots \\
\frac{\partial \sigma^2_t}{\partial \theta_k} \\
\frac{\partial \sigma^2_t}{\partial \theta_p} \\
\frac{\partial \sigma^2_t}{\partial \beta_1} \\
\vdots \\
\frac{\partial \sigma^2_t}{\partial \beta_q}
\end{pmatrix} = 2 \sum_{i=1}^{p} \alpha_i \varepsilon_{t-i} \frac{\partial \varepsilon_{t-i}}{\partial \theta} + \begin{pmatrix}
0 \\
0 \\
\vdots \\
0 \\
1 \\
\varepsilon^2_{t-1} \\
\vdots \\
\varepsilon^2_{t-p} \\
\sigma^2_{t-1} \\
\vdots \\
\sigma^2_{t-q}
\end{pmatrix} + \sum_{j=1}^{q} \beta_j \frac{\partial \sigma^2_{t-j}}{\partial \theta} \quad (2.16)$$

Appendix A steps through a more in-depth derivation of Eqns. (2.14), (2.15), and (2.16).

The BHHH algorithm also adds a step size factor $\lambda$, adjusted every iteration step to prevent the process from overshooting a maximum. Combined with the proxy for $H$, Eqn.
(2.10) becomes:

$$\theta^{(i+1)} = \theta^{(i)} + \lambda^{(i)} \left[ G'(\theta^{(i)}) G(\theta^{(i)}) \right]^{-1} g(\theta^{(i)})$$  \hspace{1cm} (2.17)

This algorithm is then iterating over Eqn. (2.17), at each step evaluating $G$ at the current settings for $\theta$ to get the adjusted settings. These adjusted settings become the current settings in the subsequent step, and so on until the change in $L$ from the current setting to the adjusted setting is less than a small predetermined number called the convergence criterion. To get the iteration going, a starting values vector $\theta^{(0)}$ is needed.

The choice of starting values for $\theta$ is important since a major flaw in greedy algorithms is that they cannot guarantee that they yield the global solution for all cases. For example, it is possible for Newton’s method to return a set of parameters that correspond only to a local maximum of $L$. The problem is that if the target function has many areas where the first order conditions all evaluate to zero, i.e., there are many “hills and valleys”, the greediness of Newton’s method will direct the process towards the nearest area where the gradient is zero. The nearest area is not necessarily the global maximum. Thus, starting values are important since they determine to which maximum the process will converge, but unfortunately there is not a hard and fast rule on how to choose them as to guarantee a global maximum.

### 2.4 Program Requirements

This section lays out program requirements for one who would who use GARCH($p,q$) in the context of risk management. At the most basic level, what the user of GARCH($p,q$) would like to have is a program that inputs large sets of historical data and outputs estimates for the $\alpha$ and $\beta$ parameters.

#### 2.4.1 Platform

The program will be written using a Microsoft version of C++, Visual C++ version 5.0. C++ is the language of choice because it is the predominant programming language used in the risk management departments of Wall Street firms\(^4\). In addition, the platform of

\(^4\text{other than Microsoft’s Visual Basic.} \)
choice for most Wall Street risk managers is the Windows NT machine, and Visual C++ is the most natural compiler for programs written for this platform.

There are benefits and disadvantages to using Microsoft’s Visual C++. One benefit is that its libraries offer implementations of standard data structures such as strings, hash tables, and lists. One can then link these libraries in programs and save a lot of time from having to ‘re-invent the wheel’ and write such structures oneself. However, the main disadvantage in writing the program in Visual C++ is that if one does end up using its libraries, the code implementation is not portable to other C++ compilers, as they will not have those libraries. In addition, a Unix flavor of C++ is probably best if one were targeting the program to run on back-end production level systems, which are most probably implemented with machines running Unix.

Regardless of these benefits and disadvantages, the main goal of this thesis is to present a design for GARCH\((p,q)\) estimation that is language and platform independent. The choice of platform should not add to or take away from the overall goodness of the design to be presented later. Therefore, the main criteria for choosing the target platform should then be the platform required by the client, who is a Wall Street risk manager most probably running Windows NT on an Intel x86 machine.

2.4.2 Interface

The most basic interface any program can offer is a command line interface. This is the interface of choice for this program, since the process’s interaction with the user is not very complex. In terms of user interface, GARCH\((p,q)\) estimation basically comes down to inputting then outputting data. Again, the focus of this thesis is how to design and implement a system for computing GARCH\((p,q)\) estimates and having a simple interface allows more focus on this issue. In addition, a system successfully designed and implemented using the most basic interface has the flexibility to support more feature-rich interfaces later on.

As described in section 2.1, GARCH\((p,q)\) is an extension to the linear regression specification given in Eqn. (2.1). This equation features one dependent variable \(y\) and zero or more independent variables \(x\). Therefore, the program must input a time series or a set of time series that capture the linear specification given by Eqn. (2.1) and yield a set of parameter estimates for Eqn. (2.4), given values for \(p\) and \(q\). For example, assume that we
have a set of time series that define the linear relation Eqn. (2.1) and assignments for \( p \) and \( q \) such as:

\[
y_1, y_2, y_3 \equiv 3.4, 2.3, 2.3
\]
\[
x_{11}, x_{12}, x_{13} \equiv 1.0, 0.9, 0.2
\]
\[
x_{21}, x_{22}, x_{23} \equiv 1.4, 7.3, 2.3
\]
\[
p = 1
\]
\[
q = 2
\]

Then the associated instance of Eqn. (2.1) would be:

\[
\begin{pmatrix}
  y_1 \\
  y_2 \\
  y_3
\end{pmatrix} = \begin{pmatrix}
  1 & x_{11} & x_{21} \\
  1 & x_{12} & x_{22} \\
  1 & x_{13} & x_{23}
\end{pmatrix} \begin{pmatrix}
  b_0 \\
  b_1 \\
  b_2
\end{pmatrix} + \begin{pmatrix}
  \epsilon_1 \\
  \epsilon_2 \\
  \epsilon_3
\end{pmatrix}
\]

and the associated instance of Eqn. (2.4) would be:

\[
\begin{pmatrix}
  \sigma_1^2 \\
  \sigma_2^2 \\
  \sigma_3^2
\end{pmatrix} = \begin{pmatrix}
  1 & \epsilon_0^2 \\
  1 & \epsilon_1^2 \\
  1 & \epsilon_2^2
\end{pmatrix} \begin{pmatrix}
  \alpha_0 \\
  \alpha_1
\end{pmatrix} + \begin{pmatrix}
  \sigma_0^2 & \sigma_1^2 \\
  \sigma_1^2 & \sigma_2^2 \\
  \sigma_2^2 & \sigma_3^2
\end{pmatrix} \begin{pmatrix}
  \beta_0 \\
  \beta_1
\end{pmatrix}
\]

The program must output parameter estimates for Eqn. (2.1) and (2.4) as so:

\[
\begin{array}{c}
  y_1, y_2, y_3 \\
  x_{11}, x_{12}, x_{13} \\
  x_{21}, x_{22}, x_{23} \\
  p = 1 \\
  q = 2
\end{array}
\]

GARCH(\( p, q \)) estimation program

\[
a_0, a_1, b_1, b_2, b_0, b_1, b_2
\]

where \( a_0 \), \( a_1 \), \( b_1 \), \( b_2 \), \( b_0 \), \( b_1 \), and \( b_2 \) are the maximum likelihood estimates for parameters \( \alpha_0 \), \( \alpha_1 \), \( \beta_1 \), \( \beta_2 \), \( b_0 \), and \( b_2 \).
Assuming the executable file is named garch, the program is run at the command prompt as so:

```bash
>garch -p<i> -q<j> <endo> [exo1] [exo2] ... [exoN]
```

where:

- `<i>` the number of lagged squared residual terms
- `<j>` the number of lagged conditional variance terms
- `<dep>` the filename containing the endogenous time series
- `[exo1]` optional: filename of the first exogenous time series
- `[exo2]` optional: filename of the second exogenous time series
- `:`
- `[exoN]` optional: filename of the Nth exogenous time series

### 2.4.3 Input File

As alluded to above, a series of observations must be inputted for each of variable in Eqn. (2.1) and Eqn. (2.4). Each series must of the same length. In addition, observations across series are assumed to occur contemporaneously. In other words, the ith value in every series is assumed to be observed at the same time i.

The simplest and therefore most manageable form of input would be a text file. The series for each of the variables in the GARCH(p,q) specification are arranged in comma-delimited columns, with the series for the dependent variable occupying the first column, the series for the first independent variable occupying the second, the series for the second independent variable occupying the third, and so on. Names labeling the series occupy the first row.

### 2.4.4 Parameters

Parameters arise in various parts of the estimation process. For one, the GARCH(p,q) model itself is a parameterized as it allows one to specify through the p and q terms the nature of the process’s memory. The p term defines how far back the process incorporates past residual terms, and the q term defines how far back the process incorporates past volatility terms. The setting of p and q is left to the user, and the estimation program must allow a user to pass these settings as input on the command line.
Setting of parameters is also required in the BHHH estimation algorithm. As mentioned in the previous section, beginning an iteration over Eqn. (2.17) requires a vector of starting values $\theta^{(0)}$. And as mentioned above, there really is no rule for determining optimum starting values, so the best recourse is to let the user specify these values. Thus, the program must also allow the user to pass starting values as input on the command line.

2.4.5 Output

The program will output estimates for each parameter that is a member of the $\theta$ vector. Again, to keep things simple, outputting to the standard console output should suffice, as this also allows one to write to a file easily using a pipe command. An error message is written to the standard output rather than parameter estimates if convergence for the estimates has not been achieved.
Chapter 3

Design

One purpose of this thesis is to design an instance of maximum-likelihood estimation from the approach of object modeling. A key feature of object modeling is that it gives one robust tools to specify the states and data of a problem. This section presents the object model (OM) created for the problem of GARCH(p,q) parameter estimation described in Chapter 2. The object model is presented in its three parts, the design object model, its snapshots, and the code object model.

3.1 Design Object Model

This section presents the design decisions made in how to characterize the data, system states and mutability constraints in the problem of GARCH(p,q) parameter estimation. There are seven domains that this section argues to be the most general types of data used in this problem:

- Name
- Value
- Slot
- Environment
- Series
- History
- Function

These types are presented, including their interrelations. Put together, these derive the design object model shown in Figures 3-1 and 3-2. Any constraints that couldn’t be
expressed notationally in these figures are written down in Figures 3-3, 3-4, 3-5, 3-6, and 3-7.

3.1.1 Names and Values

The equations given in Chapter 2 are expressed in terms whose assigned values change over time. These terms have different purposes in the context of maximum-likelihood estimation. For example, there are terms such as the $x_{it}$ term in Eqn. (2.1) that represent observations
for random variables at specific times. Then there are terms such as the $\beta_j$ term in Eqn. (2.4) that represent assignments for parameters that are invariant to time.

Two basic data types then in the problem of estimation are variables and parameters. In terms of the data model, these are simply names, each of which is either a parameter name or a variable name. The object model for them is very simple, and it is shown in Figure 3-8. The only constraint within this part of the overall object model is already expressed notationally in Figure 3-8, that is, parameters and variables completely partition the set of names, and that a name can only be a variable or parameter and not both.

3.1.2 Environment

In the domain of estimation, variables and parameters are assigned values, where a value within is simply a signed floating point number such as 2.34 or $-0.45634$. When a value is assigned to a particular name, then a binding is said to have occurred for that name. Frequently during estimation, however, the same variable takes on different values. The simplest example is when one thinks of the variable $x_i$ in Eqn. (2.1) as representing a series of values to be observed over time. At times, one would need to be able to look at the $j$th
• Slot

1. A Slot object is always in one or both of the Env::map[Name] and the Ser::assoc[Index] relations.

• Environment

1. Existence of Slot Each Name object in the Env::binds relation maps to one and only one Slot object.
2. Uniqueness of Slot No two Name objects in the Env::binds relation map to the same Slot object.
3. Permanence of Slot Each Name object in the Env::binds relation always maps to the same Slot object.

• ParameterEnvironment

1. Only Parameter name objects are in the Env::binds relation, i.e., only Parameter name objects map to Slot objects.

• VariableEnvironment

1. Only Variable name objects are in the Env::binds relation, i.e., only Variable name objects map to Slot objects.
2. A VariableEnvironment object that is not a HistoryEnvironment object can never become one.

• Series

1. The Index objects in the Ser::associates relation are integers that go from 1 to the number of Slot objects contained in the relation Ser::assoc.
2. Indices in the Ser::associates relation are contiguous and do not repeat.
3. Existence of Slot Each index in the Ser::associates relation is associated with one and only one Slot object.
4. Uniqueness of Slot No two indices in the Ser::associates relation are associated with the same Slot object.
5. A Series object that is not a HistorySeries object can never become one.

Figure 3-3: Additional constraints for object model, Part I of V
• History

1. The Index objects in the His::associates relation are integers that go from 1 to the number of HistoryEnvironment objects in the relation His::assoc.
2. Indices in the His::associates relation are contiguous and do not repeat.
3. The HistorySeries objects contained in the His::map relation have the same length.
4. The number of HistoryEnvironment objects in the relation His::assoc is the same as the length of the HistorySeries objects contained in the His::map relation.
5. Existence of Series Each Variable name object in the His::binds relation maps to one and only one HistorySeries object.
6. Uniqueness of Series No two Variable name objects in the His::binds relation map to the same HistorySeries object.
7. Permanence of Series Each Name object in the His::binds relation always maps to the same HistorySeries object.
8. Existence of Environment Each index in the His::associates relation is associated with one and only one HistoryEnvironment object.
9. Uniqueness of Environment No two indices in the His::associates relation are associated with the same HistoryEnvironment object.
10. For each HistoryEnvironment object x in the relation His::assoc, the Variable names mapped in x are precisely the same set of names in the relation His::binds.
11. Consistency of Slot within History For all pairs of index i in the relation His::associates and Variable name j in the relation His::binds, His::assoc[i].Env::map[j] == His::map[j].Ser::assoc[i].
12. Independence of Slot across Histories No two Slot objects accessible from different History objects can ever be the same.

Figure 3-4: Additional constraints for object model, Part II of V
• Function

1. There are no Variable name objects that are in both the invars relation and in the outvars relation.

2. If a ParameterEnvironment object is in the relation applies, then this object maps each Parameter name object found in the relation params to a Slot object.

3. If a History object is in the relation inputs, then this object maps each Variable name object found in the relation invars to a Series object.

4. Constraints 2 and 3 are necessary conditions for a History object to be in the relation outputs.

5. If a History object is in the relation outputs, then this object maps each Variable name object found in the relation outvars to a Series object.

6. If a History object is in the relation outputs, then the Series objects bound within this History object are of the same length, i.e., associate the same Index objects, as those Series objects bound within the History object in the inputs relation.

• EpsilonFunction

1. Inherits the constraints of the Function object.

2. An EpsilonFunction object must have zero or more independent, non-stochastic variables, i.e., the $x$ terms of Eqn. (2.1).

3. If $k$ is the number of independent, non-stochastic variables in an EpsilonFunction object, then Variable name objects in the relation invars are $y, x_1, x_2, \ldots, x_k$.

4. If $k$ is the number of independent, non-stochastic variables in an EpsilonFunction object, then Parameter name objects in the relation params are $b_0, b_1, \ldots, b_k$.

5. A Variable name object in the relation outvars is epsilon.

Figure 3-5: Additional constraints for object model, Part III of V
• EpsilonSlopeFunction

1. Inherits the constraints of the EpsilonFunction object.

2. An EpsilonSlopeFunction object must have one or more ARCH terms (the \( p \) parameter in GARCH(\( p,q \))) and zero or more GARCH terms (the \( q \) parameter in GARCH(\( p,q \))).

3. If \( k \) is the number of independent, non-stochastic variables, \( p \) the number of ARCH terms, and \( q \) the number of GARCH terms in an EpsilonSlopeFunction object, then Variable name objects in the relation outvars are \( \text{de/db0}, \text{de/db1}, \ldots, \text{de/dbk}, \text{de/dalpha0}, \text{de/dalphi1}, \ldots, \text{de/dalphap}, \text{and de/dbeta1}, \ldots, \text{de/dbetaq} \).

• GarchFunction

1. Inherits the constraints of the Function object.

2. A GarchFunction object must have one or more ARCH terms and zero or more GARCH terms.

3. A Variable name object in the relation invars is \( \text{epsilon} \).

4. If \( p \) is the number ARCH terms and \( q \) is the number of GARCH terms in a GarchFunction object, then Parameter name objects in the relation params are \( \text{alpha0}, \text{alphal}, \ldots, \text{alphap}, \text{and beta1}, \ldots, \text{betaq} \).

5. A Variable name object in the relation outvars is \( \text{variance} \).

6. A Series object bound to the Variable name object \( \text{variance} \) in the output History object of a GarchFunction object must never bind a zero, negative, or infinite value.

• GarchSlopeFunction

1. Inherits the constraints of the GarchFunction object.

2. A GarchSlopeFunction object must have zero or more independent, non-stochastic variables.

3. If \( k \) is the number of independent, non-stochastic variables, \( p \) is the number ARCH terms, and \( q \) is the number of GARCH terms in a GarchSlopeFunction object, then Variable name objects in the relation invars is \( \text{de/db0}, \text{de/db1}, \ldots, \text{de/dbk}, \text{de/dalpha0}, \text{de/dalphi1}, \ldots, \text{de/dalphap}, \text{and de/dbeta1}, \ldots, \text{de/dbetaq} \).

4. Variable name objects in the relation outvars is \( \text{dv/db0}, \text{dv/db1}, \ldots, \text{dv/dbk}, \text{dv/dalpha0}, \text{dv/dalphi1}, \ldots, \text{dv/dalphap}, \text{and dv/dbeta1}, \ldots, \text{dv/dbetaq} \).

Figure 3-6: Additional constraints for object model, Part IV of V
• **LikelihoodFunction**

1. Inherits the constraints of the Function object.
2. **Variable** name objects in the relation invars are **epsilon** and variance.
3. A **Variable** name object in the relation outvars is **likelihood**.

• **LikelihoodSlopeFunction**

1. Inherits the constraints of the LikelihoodFunction object.
2. A LikelihoodSlopeFunction object must have zero or more independent, non-stochastic variables, one or more ARCH terms, and zero or more GARCH terms.
3. If $k$ is the number of independent, non-stochastic variables, $p$ is the number ARCH terms, and $q$ is the number of GARCH terms in a LikelihoodSlopeFunction object, then Variable name objects in the relation invars is $de/db_0, de/db_1, ..., de/db_k, de/dalpha_0, de/dalpha_1, ..., de/dalpha_p, de/dbeta_0, de/dbeta_1, ..., de/dbeta_q, dv/db_0, dv/db_1, ..., dv/db_k, dv/dalpha_0, dv/dalpha_1, ..., dv/dalpha_q, and dv/dbeta_1, ..., dv/dbeta_q$.
4. Variable name objects in the relation outvars are $dl/db_0, dl/db_1, ..., dl/db_k, dl/dalpha_0, dl/dalpha_1, ..., dl/dalpha_p, and dl/dbeta_1, ..., dl/dbeta_q$.

Figure 3-7: Additional constraints for object model, Part V of V

![Diagram](image)

Figure 3-8: The Name object model
and $j + 1$th values for $x_i$ at the same time, such as when approximating the Hessian matrix as in Eqn. (2.12).

Although parameter values tend to be globally set in the problem domain of estimation, one might run into situations where it is useful to have simultaneous assignments to the same parameter. For example, one might want to extend the program to work with multiple data sets and run maximum-likelihood estimation on each set of data. One must then internally associate with each set of data a separate set of parameter estimates.

Therefore, to allow multiple values be bound to the same name, it is useful to introduce the notion of the *environment*. An environment is a set of bindings, where no two bounded names are equivalent. Although a value-to-name binding is effective only in one environment, one can have multiple environments. This construct, for example, allows multiple values to be bound to the same parameter name by having each binding occur in a different environment.

Since maximum-likelihood estimation only binds values to variables and parameters, there are only two subsets of environments in the data model, a *variable environment* and a *parameter environment*. As the descriptions indicate, a `VariableEnvironment` only contains variable name bindings whereas a `ParameterEnvironment` only contains parameter name bindings. This partition is depicted in the object model for environments given in Figure 3-9.

As seen in Figure 3-9, each name has one slot. This is to specifically illustrate how a binding occurs in an environment. When a value is bound to a name, that value is assigned to that name’s slot. Slots can only have one value, so at least under one environment, a name can only have one bound value. The motivation for slots may not be apparent, but for now it suffices to say that they are there to allow multiple views to a value. The motivation for slots will be made clear in a subsequent section, after the notions of series and histories are introduced.

And as is also seen in 3-9, there is one subset of `VariableEnvironment` objects called `HistoryEnvironment` objects. These latter environments behave the same as variable environments, with one crucial exception that they expose their representation to histories. The purpose of this type of variable environment will be detailed in a subsequent section, again after the notions of series and histories are introduced.
Environment constraints and operations

Maximum-likelihood estimation conceptually separates variables and parameters, so these two subsets are disjoint in the data model, and they completely partition the set of environments. In addition, a parameter environment cannot become a variable environment, and vice versa. Finally, a name bound within a parameter environment cannot be bound within a variable environment, and vice versa (Fig. 3-10).

To obtain the value assigned to a specific variable or parameter name under an environment, one projects the environment onto the name. For example in Figure 3-11, the name \( x \) maps a \( \text{VariableEnvironment} \) to its bounded value 3.234, and the name \( \text{alpha} \) maps a \( \text{ParameterEnvironment} \) to its bounded value 7.003. As will be seen in the following
sections, the projection relation is common among many object sets in the model.

![Diagram showing Environment to Value projections]

**Figure 3-11: Environment to Value projections**

### 3.1.3 Series

As alluded to before in section 3.1.2, the difference between a variable and a parameter in maximum-likelihood estimation is that a variable is used to represent a *series*, which is an ordered multi-set of values. In a series, each value is associated with a unique integer, and the ordering of the integers determines the ordering of the values. The object model for a series is given in Figure(3-12).

As seen in Figure 3-12, the notion of the slot again arises, in this case being the medium through which an index in a series is associated with a value. The idea is that each index of a series has a slot, and the value associated with that index is stored in that index’s slot. The ordering of the indices orders their respective slots, which in turn orders the associated values. Again, the purpose of slots will be detailed in a later section.

As is also seen in 3-12, there is one subset of *Series* objects called *HistorySeries* objects. In a conceptual fashion similar to *HistoryEnvironment* objects, these history series behave the same as series, with the one exception being that they expose their representation to histories. The purpose of this type of series will be detailed in a subsequent section, again after the notions of series and histories are introduced.

**Series constraints and operations**

Beyond this, there are two notes to be said about series. First, series are constrained to be contiguous with respect to its indices. The *Series* object in Figure 3-13 is an example of a
series that fails to meet this constraint. Second, a value in a Series object can be obtained by projecting the Series object on an index. Therefore, to obtain the 2.54 value in the example given in Figure 3-14, one projects that Series object onto the index 2.

Figure 3-12: The Series object model

Figure 3-13: Valid Series object A and invalid Series object B

3.1.4 History

The data model uses the history construct to contain variable name-to-series associations. By projecting a History object on a variable name it contains, one obtains the HistorySeries object associated with that name. The history construct also contains index-to-variable environment associations, allowing one to project a History object on an index it contains. This operation would yield a HistoryEnvironment object. Figure 3-15 shows how History objects are captured in the data model.
Figure 3-14: A Series to Value projection

Figure 3-15: The History object model
History constraints and operations

The model imposes a constraint that HistorySeries objects contained within the same History object have the same indices and the same number of indices. Figure 3-16 shows an example of a History object that satisfies this constraint and another that does not. Examples of valid projections that obtain a HistorySeries object and a HistoryEnvironment object are shown in Figure 3-17.

3.1.5 Slot

Having now introduced the concept of the history and its relation to series and environments, the motivation for having slots in the design model can now be presented. The motivation
for slots comes from a design decision made that says a history provides multiple views through which one can observe and change a value in that history. This design decision, in turn, was motivated by an observation that maximum-likelihood estimation features a need to adjust values within series bound to variables.

For example, the goal of the target program is to iteratively estimate values for the $\alpha$ and $\beta$ parameters in the GARCH($p,q$) specification given in Eqn. (2.4). The quality of a set of parameter estimates is driven by how they affect the series of log-likelihood observations for $l$ captured by Eqn. (2.8). As stated in Chapter 2, better estimates yield a greater sum of series values. Thus, the process of choosing different parameter estimates requires being able to change values within the series of values produced by Eqn. (2.8). And since this series is bound to a variable, namely, $l$, and since series are bound to variables only within a history, the change in this series can occur only if values in a history can be changed.

But, in order to feature changes in a history, one must assure that any view to a value is updated if and when that value is changed. For example, in Figure 3-17, one can obtain the 7.6 value for variable $y$ at index 2 in one of two ways. The first is by projecting the input History object on index 2 and projecting the resulting VariableEnvironment object on the variable name $y$. The other way is to project the History object on variable name $y$ and projecting the resulting Series on index 2.

The idea is to have the 7.6 value returned by both access approaches be the same. So, if one were to use one access approach and change the value yielded to, say, 3.2, the other access approach should yield the value 3.2. The goal is to have values be consistent across all series and environments that are obtainable from the same history, and this requires having values that can be shared between environments and series. This is achieved by having values being internally stored within Slot objects, and having these objects be bound to indices within Series and bound to names within Environment objects. When a value needs to be the same between a Series and a VariableEnvironment object, the slots holding that value within the Series and VariableEnvironment is made the same.

However, it is also important to note that it is the intention of the design to hide the Slot object from clients of the Series and VariableEnvironment objects. The idea is to keep the interfaces of these objects simple and have clients only think of values as simple doubles\(^1\). In addition, hiding Slot objects also serves to protect the internal representation

---

\(^1\)A double is a 32-bit floating point number
of `Series` and `VariableEnvironment` objects. For example, if a `Slot` object were to be
returned from a projection operation on a `Series` object, the `Slot` object could be changed
in such a way that might invalidate the representation invariants of the `Series` object. For
one, the `Slot` object could be deleted from memory. And even if this scenario was to be
mitigated by returning an independent copy of the `Slot` object, a change to the value stored
within the copied slot would never be reflected in the `Series` object.

**History Environments and History Series**

The discussion about slots finally leads to the motivation for history environments and
history series in the object model. Having consistency of values as described above re-
quires the `History` object having the ability to modify the representation of any `Series` or
`VariableEnvironment` object that it outputs to a client. The problem is that the internal
`Value` objects stored in the series accessible from a history must be the same `Value` objects
stored in the environments that are also accessible from that same history. If the `History`
object is to only be a strict client of the `Series` and `VariableEnvironment` objects, the
`History` is limited to mutator methods that change only the `doubles` stored by the internal
`Value` objects. The `History` object has no way of ensuring that a `Series` object it yields
have the same values accessible using the appropriate `VariableEnvironment` objects.

However, it is also desired by the design to have `Series`, `VariableEnvironment`, and
`History` objects implement separate domains. The motivation here is allowing for the reuse
of objects. For example, one might want to design an object model that has no notion
of the `History` object whatsoever, but finds utility in using the `Series` objects and the
`VariableEnvironment` objects by themselves. For example, the `Series` object in the later
implementation of the program aids in the parsing of series data inputted as text files. A
`VariableEnvironment` object on its own could be used in another object model as a holder
of transient state variables. But exposing the representation of `VariableEnvironment`
objects and `Series` objects to histories defeats this goal, since it makes the specification
of the `VariableEnvironment` and `Series` object dependent upon, at the minimum, the
existence of the `History` object. A compromise is to have for the `History` object something
analogous to `inner class` versions of `Series` and `VariableEnvironment` objects that only
a `History` object can instantiate.

These versions are shown as the `HistorySeries` object and the `HistoryEnvironment`
object in the design OM. The specifications for these objects are simple. The HistorySeries object inherits the functionality of the Series object, and the HistoryEnvironment inherits the functionality of the VariableEnvironment object. Internally, then, a History object stores an array of HistoryEnvironments and a table of HistorySeries, but the existence of these objects within the internal representation is hidden from clients of History objects.

Since these objects expose their representation to the History object, the object can preserve the consistency of stored Value objects. In addition, their inheritance of functionality from the Series and VariableEnvironment objects allows the History object to cast them to these objects upon a client request. One can think of a History object providing Series and VariableEnvironment objects as *interfaces*, implementing these interfaces using HistorySeries and HistoryEnvironment objects.

For example, when a History object is projected on a name, the mapped HistorySeries is cast to a Series object and is outputted as a Series object. Similarly, whenever the History object is projected on an index, the appropriate HistoryEnvironment corresponding to that index is cast to a VariableEnvironment object and is outputted as a VariableEnvironment object. Thus, to the outside world, one does not need to care about the existence of HistorySeries and HistoryEnvironment objects.

Yet, this compromise still allows the ugly scenario of a History object recasting to its representation-exposing form any Series and VariableEnvironment object outputted by another History. Clearly, if this can occur, then a History object has the ability to modify the internal representation of another History object. However, this scenario is mitigated by the fact that a History object, as will be seen later in Chapter 4, copies any Series or VariableEnvironment object that it inputs. Such copying of inputs prevents the above scenario from ever happening.

### 3.1.6 Function

The purpose of maximum-likelihood estimation is to maximize the value of a likelihood function, such as Eqn. (2.9). As seen in this equation, this value is a sum of a series of values. This series of values is itself dependent on other series of values bound to variables, which are weighted by parameters. In this problem domain then, a function maps a set of series and a set of parameter weights to another series. For example, one can express eqn.
(2.1) notationally as:

\[ y = f(b_0, b_1 x_1, b_2 x_2, \ldots, b_k x_k) \]

where \( y, x_1, \ldots, x_k \) represent variables, each of which has a bound series. \( b_0, \ldots, b_k \) represent parameters, each of which has a bound variable.

However, the mathematics of maximum-likelihood estimation also features functions that have multiple output variables. For example, one can consider taking the derivative of the above equation with respect to a vector of parameters:

\[
\begin{pmatrix}
    b_0 \\
    b_1 \\
    \vdots \\
    b_k
\end{pmatrix}
\]

which yields:

\[
\frac{\partial y}{\partial b} = \begin{pmatrix}
    \frac{\partial y}{\partial b_0} \\
    \frac{\partial y}{\partial b_1} \\
    \vdots \\
    \frac{\partial y}{\partial b_k}
\end{pmatrix}
\]

In the above equation, each partial derivative represents a variable which has a bound series. Thus, the output of \( \frac{\partial y}{\partial b} \) is not one equation as in the previous example, but a system of equations. In terms of the object model, this example is of a function that outputs a set of series bound to multiple variables.

What is common to the above two examples is that a history and a parameter environment together yield another history. The model uses the function construct to capture how a history and parameter environment can yield another history. In terms of the object model, the Function object applies a ParameterEnvironment object to an input History object to obtain another History object. The History object contains the variable-to-series bindings required by the Function object whereas the ParameterEnvironment object contains the necessary parameter-to-value bindings.
Constraints and operations

The constraint on any history input to a function is that the history must bind series to variables needed to evaluate the function. Similarly, parameter environments applied to a function must bind values to parameters required by the function. This constraint does not exclude other bindings in the input history or the applied parameter environment. So for example, if a History object binds series to both variable names x and y and a Function object only requires a binding for variable name x, then this History object can be input to this Function object (Figure 3-18).

![Diagram showing the relationship between ParameterEnvironment, History, and EpsilonFunction objects.](image)

**Figure 3-18:** Valid EpsilonFunction object A and invalid EpsilonFunction object B

When evaluating a Function object, the object calculates a target mathematical func-
tion, using the bindings for function’s variables found in the associated History object and the bindings for the function’s parameters found in the associated ParameterEnvironment object. The result is an output History object that binds Series objects to the output Variable names of the function. These Series objects must reflect the correct mathematical evaluation for the output variable Name objects to which they are bound. Each function generally handles only one mathematical function, thus partitioning the set of functions in the data model. For the purposes of maximum-likelihood estimation, the set of functions are limited to those that handle equations given in Table 3.1. The resulting partitioning of the set of functions in the object model is given in Figure 3-19.

<table>
<thead>
<tr>
<th>Function object</th>
<th>GARCH(p,q) equivalent</th>
<th>Eqn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EpsilonFunction</td>
<td>$\varepsilon$</td>
<td>2.1</td>
</tr>
<tr>
<td>GarchFunction</td>
<td>$\sigma^2$</td>
<td>2.4</td>
</tr>
<tr>
<td>LikelihoodFunction</td>
<td>$l$</td>
<td>2.8</td>
</tr>
<tr>
<td>EpsilonSlopeFunction</td>
<td>$\frac{\partial \varepsilon}{\partial \theta}$</td>
<td>2.15</td>
</tr>
<tr>
<td>GarchSlopeFunction</td>
<td>$\frac{\partial \sigma^2}{\partial \theta}$</td>
<td>2.16</td>
</tr>
<tr>
<td>LikelihoodSlopeFunction</td>
<td>$\frac{\partial l}{\partial \theta}$</td>
<td>2.14</td>
</tr>
</tbody>
</table>

Table 3.1: The Function objects used in the model and their equivalents in GARCH(p,q)

The specific subsets of Function objects given in the object model have more stringent constraints, as is seen in the constraints listing of Figures 3-5, 3-6, and 3-7. Thus, the next sub-sections below will characterize these function subsets and detail their constraints.

**EpsilonFunction**

An EpsilonFunction object evaluates the linear specification given in Eqn. (2.1) of Chapter 2. The equation it captures is:

$$\varepsilon_t = y_t - b_0 - b_1 x_{1t} - b_2 x_{2t} \ldots - b_k x_{kt}$$

Eqn. (3.1) is just Eqn. (2.1) with the terms rearranged to isolate $\varepsilon_t$ on the left. With this rearrangement, the input History object of a EpsilonFunction object must bind Series objects to the variable names $y$, $x_1$, $x_2$, ..., $x_k$, where $k$ is the number of independent variable names in the EpsilonFunction object. In addition, the EpsilonFunction object’s applied ParameterEnvironment object must bind values to parameter names $b_0$, $b_1$, ...,
Figure 3-19: The Function object model
bk. The output History object of a EpsilonFunction object must bind a Series object to the variable name \textit{epsilon}. The length of this Series object must be the same length as those Series objects bound in the input History object.

\textbf{EpsilonSlopeFunction}

An EpsilonSlopeFunction object is an EpsilonFunction object that not only evaluates the linear specification given in Eqn. (2.1), but also the derivative of that specification given in Eqn. (2.15). The system of equations it captures is given in Figure 3-20. Since

\begin{align*}
\varepsilon_t &= y_t - b_0 - b_1 x_{1t} - b_2 x_{2t} \ldots - b_k x_{kt} \\
\frac{\partial \varepsilon_t}{\partial b_0} &= -1 \\
\frac{\partial \varepsilon_t}{\partial b_1} &= -x_{1t} \\
\vdots \\
\frac{\partial \varepsilon_t}{\partial b_k} &= -x_{kt} \\
\frac{\partial \varepsilon_t}{\partial \alpha_0} &= 0 \\
\frac{\partial \varepsilon_t}{\partial \alpha_1} &= 0 \\
\vdots \\
\frac{\partial \varepsilon_t}{\partial \alpha_p} &= 0 \\
\frac{\partial \varepsilon_t}{\partial \beta_1} &= 0 \\
\vdots \\
\frac{\partial \varepsilon_t}{\partial \beta_q} &= 0
\end{align*}

Figure 3-20: System of equations captured by an EpsilonSlopeFunction object

it is an EpsilonFunction object, the input History object of an EpsilonSlopeFunction object must bind series to the variable names \textit{y}, \textit{x1}, \textit{x2}, \ldots, \textit{xk}, where \textit{k} is the number of independent variable names in the EpsilonSlopeFunction object. And as with the EpsilonFunction object, the applied ParameterEnvironment object must bind values to parameter names b0, b1, \ldots, bk. The output History object of an EpsilonSlopeFunction object
The input History object of a GarchFunction object must bind a Series object to the variable name epsilon. The applied ParameterEnvironment object of a GarchFunction object must bind values to the following parameter names: alpha0, alpha1, ..., alphas, and betas, ..., betas. p and q are the number of ARCH parameter names and GARCH parameter names, respectively, in the GarchFunction object. The output History object of a GarchFunction object must bind a Series object to the variable name variance. In addition to being the same length as the Series object in the GarchFunction object's input History object, there must be no zero or negative values in the Series object bound to the variable name variance. Figure (3-21) gives an example of a valid GarchFunction object that correctly enforces this constraint and one that incorrectly does not.

GarchSlopeFunction

A GarchSlopeFunction object is a GarchFunction object that not only evaluates the linear specification given in Eqn. (2.4), but also the derivative of that specification given in Eqn. (2.16). The system of equations it captures is given in Figure 3-22. The input History object of a GarchSlopeFunction object must bind Series objects to the following variable names: epsilon, de/db0, de/db1, ..., de/dbk, de/dalpha0, de/dalpha1, ..., de/dalphas, de/dbetas, and de/dbetas. k is the number of independent variable names in the GarchSlopeFunction object. p and q are the number of ARCH parameter names and GARCH parameter names, respectively, in the GarchSlopeFunction object.
Figure 3.21: Valid GarchFunction object A and invalid GarchFunction object B.
\[
\begin{align*}
\sigma_t^2 &= \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \alpha_2 \varepsilon_{t-2}^2 + \ldots + \alpha_p \varepsilon_{t-p}^2 + \beta_1 \sigma_{t-1}^2 + \ldots + \beta_q \sigma_{t-q}^2 \\
\frac{\partial \sigma_t^2}{\partial b_0} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial b_0} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial b_0} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial b_0} \\
\frac{\partial \sigma_t^2}{\partial b_1} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial b_1} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial b_1} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial b_1} \\
&\vdots \\
\frac{\partial \sigma_t^2}{\partial b_k} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial b_k} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial b_k} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial b_k} \\
\frac{\partial \sigma_t^2}{\partial \alpha_0} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial \alpha_0} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + 1 + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial \alpha_0} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial \alpha_0} \\
\frac{\partial \sigma_t^2}{\partial \alpha_1} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial \alpha_1} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \varepsilon_{t-1}^2 + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial \alpha_1} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial \alpha_1} \\
&\vdots \\
\frac{\partial \sigma_t^2}{\partial \alpha_p} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial \alpha_p} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \varepsilon_{t-p}^2 + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial \alpha_p} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial \alpha_p} \\
\frac{\partial \sigma_t^2}{\partial \beta_1} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial \beta_1} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \sigma_{t-1}^2 + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial \beta_1} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial \beta_1} \\
&\vdots \\
\frac{\partial \sigma_t^2}{\partial \beta_q} &= 2 \left( \frac{\partial \varepsilon_{t-1}}{\partial \beta_q} + \ldots + \alpha_p \varepsilon_{t-p}^2 \right) + \sigma_{t-q}^2 + \beta_1 \frac{\partial \sigma_{t-1}^2}{\partial \beta_q} + \ldots + \beta_q \frac{\partial \sigma_{t-q}^2}{\partial \beta_q}
\end{align*}
\]
The applied ParameterEnvironment object of a GarchFunction object must bind values to the parameter names alpha0, alpha1, alpha2, ..., alphap, and betal, ..., betaq. The output History object of a GarchSlopeFunction object must bind Series objects to the following variable names: dv/db0, dv/db1, ..., dv/dbk, dv/dalpha0, dv/dalpha1, ..., dv/dalphap, and dv/dbetal, ..., dv/dbetaq. Each of these Series objects must be of the same length as the Series objects bound in the input History object of the GarchSlopeFunction.

LikelihoodFunction

A LikelihoodFunction object evaluates the linear specification given in Eqn. (2.8) of Chapter 2. The equation it captures is:

$$l_t = -\frac{1}{2} \ln \sigma_t^2 - \frac{\varepsilon_t^2}{2\sigma_t^2}$$  \hspace{1cm} (3.5)

The input History object of a LikelihoodFunction object must bind Series objects to the variable names epsilon and variance. The applied ParameterEnvironment object of a LikelihoodFunction object does not have to bind values to any parameter name. The output History object of a LikelihoodFunction object must bind a Series object to the variable name likelihood. This Series object must have the same length as the Series objects bound in the input History object of the LikelihoodFunction object.

LikelihoodSlopeFunction

A LikelihoodSlopeFunction object is a LikelihoodFunction object that not only evaluates the linear specification given in Eqn. (2.8), but also the derivative of that specification given in Eqn. (2.14). The system of equations it captures is given in Figure 3-23. The input History object of a LikelihoodFunction object must bind Series objects to the following variable names: epsilon, de/db0, de/db1, ..., de/dbk, de/dalpha0, de/dalpha1, ..., de/dalphap, de/dbetal1, ..., de/dbetaq, variance, dv/db0, dv/db1, ..., dv/dbk, dv/dalpha0, dv/dalpha1, ..., dv/dalphap, and dv/dbetal, ..., dv/dbetaq. k is the number of independent variable names in the LikelihoodSlopeFunction object. p and q are the number of ARCH parameter names and GARCH parameter names, respectively, in the LikelihoodSlopeFunction object. The applied ParameterEnvironment object of
\begin{align*}
  l_t &= -\frac{1}{2} \ln \sigma_i^2 - \frac{e_i^2}{2\sigma_i^2} \\
  \frac{\partial l_t}{\partial \theta_0} &= -e_t \cdot \frac{\partial \theta_0}{\partial \theta_0} + \frac{1}{2\sigma_i^2} \left[ \frac{e_i^2}{\sigma_i^2} - 1 \right] \cdot \frac{\partial \sigma_i^2}{\partial \theta_0} \\
  \frac{\partial l_t}{\partial \theta_1} &= -e_t \cdot \frac{\partial \theta_1}{\partial \theta_1} + \frac{1}{2\sigma_i^2} \left[ \frac{e_i^2}{\sigma_i^2} - 1 \right] \cdot \frac{\partial \sigma_i^2}{\partial \theta_1} \\
  \vdots \\
  \frac{\partial l_t}{\partial \theta_p} &= -e_t \cdot \frac{\partial \theta_p}{\partial \theta_p} + \frac{1}{2\sigma_i^2} \left[ \frac{e_i^2}{\sigma_i^2} - 1 \right] \cdot \frac{\partial \sigma_i^2}{\partial \theta_p} \\
  \frac{\partial l_t}{\partial \beta_1} &= -e_t \cdot \frac{\partial \beta_1}{\partial \beta_1} + \frac{1}{2\sigma_i^2} \left[ \frac{e_i^2}{\sigma_i^2} - 1 \right] \cdot \frac{\partial \sigma_i^2}{\partial \beta_1} \\
  \vdots \\
  \frac{\partial l_t}{\partial \beta_q} &= -e_t \cdot \frac{\partial \beta_q}{\partial \beta_q} + \frac{1}{2\sigma_i^2} \left[ \frac{e_i^2}{\sigma_i^2} - 1 \right] \cdot \frac{\partial \sigma_i^2}{\partial \beta_q}
\end{align*}

Figure 3-23: System of equations captured by a \texttt{LikelihoodSlopeFunction} object
a LikelihoodSlopeFunction object does not have to bind values to any parameter name. The output History object of a LikelihoodSlopeFunction object must bind a Series object to the each of the following variable names: \( \frac{dl}{db_0}, \frac{dl}{db_1}, \ldots, \frac{dl}{db_k}, \frac{dl}{dalpha_0}, \frac{dl}{dalphap}, \ldots, \frac{dl}{dbeta_1}, \ldots, \frac{dl}{dbeta_q} \). Each of these Series objects must have the same length as the Series objects bound in the input History object of the LikelihoodSlopeFunction object.

### 3.2 Snapshots

To make the design model a little more concrete, this section presents a sample snapshot in Figure 3-24. The snapshot is of an EpsilonFunction object that inputs a History object and applies a ParameterEnvironment object to obtain another History object. Within the framework of GARCH\((p,q)\) estimation, the idea is to heuristically iterate over objects such as this EpsilonFunction object, determining parameter values for the applied ParameterEnvironment that maximize an objective likelihood function.

### 3.3 Code Object Model

The code object model given in Figure 3-25 illustrates how to represent in code the sets and relationships given in the design object model. In the code model, each set in the design OM is represented as a class, listed in Table 3.2. The C\texttt{String} class used to represent the \texttt{Name} objects comes from Microsoft’s Visual C++ class library. As seen in the code model, other Microsoft classes used are the C\texttt{Map} class for hashing duties and the C\texttt{Array} class for vector duties.
EpsilonFunction applies
\[ \varepsilon = y - b_0 - b_1 x_1 \]

Figure 3-24: A Sample Snapshot
<table>
<thead>
<tr>
<th>Set name</th>
<th>Implementation class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>CString</td>
</tr>
<tr>
<td>Parameter</td>
<td>CString</td>
</tr>
<tr>
<td>Variable</td>
<td>TString</td>
</tr>
<tr>
<td>Slot</td>
<td>Slot</td>
</tr>
<tr>
<td>Value</td>
<td>double (primitive)</td>
</tr>
<tr>
<td>Index</td>
<td>integer (primitive)</td>
</tr>
<tr>
<td>Environment</td>
<td>Environment</td>
</tr>
<tr>
<td>ParameterEnvironment</td>
<td>ParameterEnvironment</td>
</tr>
<tr>
<td>VariableEnvironment</td>
<td>VariableEnvironment</td>
</tr>
<tr>
<td>HistoryEnvironment</td>
<td>HistoryEnvironment</td>
</tr>
<tr>
<td>Series</td>
<td>Series</td>
</tr>
<tr>
<td>HistorySeries</td>
<td>HistorySeries</td>
</tr>
<tr>
<td>History</td>
<td>History</td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
</tr>
<tr>
<td>EpsilonFunction</td>
<td>EpsilonFunction</td>
</tr>
<tr>
<td>EpsilonSlopeFunction</td>
<td>EpsilonSlopeFunction</td>
</tr>
<tr>
<td>GarchFunction</td>
<td>GarchFunction</td>
</tr>
<tr>
<td>GarchSlopeFunction</td>
<td>GarchSlopeFunction</td>
</tr>
<tr>
<td>LikelihoodFunction</td>
<td>LikelihoodFunction</td>
</tr>
<tr>
<td>LikelihoodSlopeFunction</td>
<td>LikelihoodSlopeFunction</td>
</tr>
</tbody>
</table>

Table 3.2: The implementation classes for object model
Figure 3-25: Code Model for GARCH\((p,q)\) parameter estimation
Chapter 4

Implementation

This chapter presents an implementation for the program design presented in Chapter 3. The code modules that are created in the implementation are presented, and module dependency diagrams are used to illustrate their interdependencies. The representation for each class is then shown, detailing the abstraction function and representation invariants that characterize it.

4.1 Module Dependencies

The module dependencies of the implementation is shown in Figure 4-1. The large number of arcs in the diagram depict the integrated nature of the whole design; not too many objects can be considered independently of other objects. Nevertheless, the design OM defines tight relations among these objects, so the expectation is that the modules of the program can be implemented independently and later integrated to form the whole. The next section details the specifications and any notable implementation details for each of the modules given in Figure 4-1.

4.2 Specification

The code model given in section 3.3 suggests that each domain of the problem can be represented as a class. For the Name domain, Microsoft’s CString class provides adequate implementation. This section, then, is dedicated towards giving a class specification for each of the other five domains, namely the Slot, Environment, Series, History, and
Figure 4-1: Module Dependencies for design model
Function domains. Each class specification includes a representation, abstraction function, and representation invariant. The methods of each class are also specified.

One thing that will be noticed throughout the specifications given below is that each class extends the specification given for the `RefCount` class. Another thing that will be noticed is that class representations storing pointers to class instances implement these pointers using the `GCPointer` class. A `RefCount` is simply a wrapper object that keeps count of how many pointers are pointing to it. A `GCPointer` object only stores zero or one pointers to a `RefCount` object. It deletes the `RefCount` object it is pointing to when that object's pointer count goes to zero. Combined, these two classes implement a garbage collection mechanism for each of the five classes below. The implementation of these classes is not a real necessity of the design; it only addresses the lack of garbage collection in the Microsoft Visual C++ compiler. The object model and specifications for these classes are given in Appendix B.

4.2.1 Slot

The `Slot` object is implemented as a polymorphic container class that stores a pointer to an instance of another class. The type of this target class is specified upon instantiation of the `Slot` class. The representation of the class, then, is a pointer to a class instance:

```c
// The representation is:
T *val;
```

The abstraction function for this class is:

```c
// The abstraction function is:
// AF(c) = { c->val }
```

The representation invariant is:

```c
// The representation invariant is:
// I(c) = c->val != NULL
```

The complete specification for the member functions is given in Figures B-4 and B-5 of Appendix B. These functions are simply observer and mutator methods that provide avenues for changing the class instance pointed to by a `Slot` object.
4.2.2 Environment

The Environment object is implemented as a class that maintains a table associating strings to primitives of type double. Internally, the Environment class does this by using Microsoft’s CMap hash table class to hash CString objects to Slot objects that store pointers to primitives of type double. The representation of the class is:

// The representation is:
CMap<CString, LPCTSTR, GCPointer< Slot<double> >, GCPointer< Slot<double> > > *bindings;
int m_cardinality; // the number of names in bindings.
friend class EnvironmentNameGen; // a name generator for this.

The abstraction function for this class is:

// The abstraction function is:
// AF(c) = \{ <name, c->bindings->Lookup(name)->val> | name is in bindings \}

The representation invariant is:

// The representation invariant is:
// I(c) =
// 1.) c->bindings != NULL
// 2.) c->bindings->GetCount() == c->m_cardinality
// 3.) for each name in c->bindings, c->bindings->Lookup(name) != NULL
// 4.) there are no two names m,n in c->bindings such that
// c->bindings->Lookup(m) == c->bindings->Lookup(n)

The specification for the methods of an Environment class is given in Figures B-6 and B-7 of Appendix B. These methods allow one to bind values to names, to look-up bound values, and to change a name’s bound value to another value.

VariableEnvironment, ParameterEnvironment

To enforce the partition of environments in code, the constructors of the Environment class are protected, preventing the explicit instantiation of an Environment object. The only Environment objects then that can be instantiated in code are its subsets given in the object model of Chapter 3, namely VariableEnvironment objects and ParameterEnvironment objects. These subsets of environments are implemented as classes using the same representation, abstraction function, and representation invariant of the Environment class they extend. As such, they are not shown.

However, because VariableEnvironment objects and ParameterEnvironment objects
extend the same class, any constructors or copying methods must be designed to prevent potential mixing of environments. The specifications for these classes listed in Figures B-8 and B-9 of Appendix B show how this is done. For example, the copy constructors only allow environments to be copied to those of the same subtype. Similarly, the assignment operators only allow environments to be set equal to those of the same subtype. Finally, concatenation can only occur among the same type of environments.

**HistoryEnvironment**

As is mentioned in Chapter 3, the purpose of the HistoryEnvironment object is to expose the representation of the VariableEnvironment object to the History object. The simple way of doing this in C++ is to make the History object a friend class of the HistoryEnvironment. This adds a line to the representation of the HistoryEnvironment:

```cpp
// The representation is:
friend class History; // expose the rep to history to allow direct updates.
```

The other major item of note beyond the representation of the HistoryEnvironment class is that it does not extend the method suite of the VariableEnvironment class. It only declares required constructors and destructors, and these are declared private. The motivation behind this is that the HistoryEnvironment object exists only to help implement the functionality of History objects. As is stated in Chapter 3, it is through these objects that History objects have access to the representation of VariableEnvironment objects. The implementation presented here aims to make HistoryEnvironment objects completely internal to History objects, as such only History objects can instantiate these objects and delete these objects.

Other than this change in representation, a HistoryEnvironment object is identical in every other way to a VariableEnvironment object. This can be seen in the specifications of the HistoryEnvironment class given in Figure B-10 of Appendix B. And can also be seen in the specification, the constraint that VariableEnvironment objects can never become HistoryEnvironment objects is obeyed. The copy constructor method that inputs a VariableEnvironment object is not inconsistent with this, since it instantiates a new HistoryEnvironment object that is independent of the input. The inputted VariableEnvironment object therefore never becomes a HistoryEnvironment object.
4.2.3 Series

The **Series** object is implemented as a class that maintains an array storing and indexing primitives of type `double`. Internally, the **Series** class does this by using Microsoft’s **CArray** data structure to store and index **Slot** objects storing pointers to primitives of type `double`. As such, the representation is:

```cpp
// The representation is:
CArray<GCPointer<Slot<double>>, GCPointer<Slot<double>>> *slotArray;
int m_length; // the number of elements in this.
```

The abstraction function is:

```cpp
// The abstraction function is:
// AF(c) = { c->slotArray[1]->val, c->slotArray[2]->val, ..., c->slotArray[c->m_length]->val }
```

The representation invariant is:

```cpp
// The representation invariant is:
// I(c) =
// 1.) c->slotArray != NULL
// 2.) c->slotArray->GetSize() == c->m_length
// 3.) for i = 1 to c->m_length
//     { c->slotArray[i] != NULL }
// 4.) there is no i,j such that
//     i != j && c->slotArray[i] == c->slotArray[j]
```

The specification for the methods of a **Series** class is given in Figures B-11 and B-12 of Appendix B. These methods allow one to set values to indices, to look-up set values, and to change the set value of an index to another value.

**HistorySeries**

In the same spirit of the **HistoryEnvironment** class, the **HistorySeries** class extends the specification for the **Series** class to make the **History** class a **friend** class:

```cpp
// The representation is:
friend class History; // expose the rep to history to allow direct updates.
```

And like the specification for the **HistoryEnvironment** class, the **HistorySeries** class does not specify any `public` methods beyond those of the **Series** class. All it does is specify required constructor and destructor methods and defines an additional constructor that copy constructs a **Series** object to a **HistorySeries** object. In any case, these constructors
and destructors are all declared *private*, so the only objects that can instantiate and destroy
`HistorySeries` objects are `History` objects, which makes sense in terms of the object model
as `HistorySeries` objects only exist to aid in the implementation of `History` objects.

Just as is the case between `VariableEnvironment` objects and `HistoryEnvironment` objects,
there isn’t much of a difference between `Series` and `HistorySeries` objects. This can
be seen in the specification of the `HistorySeries` class given in Figure B-13 of Appendix B.
This specification satisfies the constraint given in the design OM that a `Series` object cannot
become a `HistorySeries` object. The copy constructor ensurea that a `HistorySeries`
object instantiated from a `Series` object is independent of that `Series` object.

### 4.2.4 History

The `History` object is implemented as a class that maintains both a table that maps names
to series and an array that stores and indexes variable environments. Internally, this is
accomplished by using Microsoft’s `CMap` class to hash `CString` objects to `HistorySeries`
objects and its `CArray` class to store and index pointers to `HistoryEnvironment` objects.

The representation is:

```c
// The representation is:
CMap<CString, LPCTSTR, GCP certo<HistorySeries>, GCP certo<HistorySeries>> *series;
CArray<GCP certo<HistoryEnvironment>, GCP certo<HistoryEnvironment>> *environments;
int m_length;        // the length of environments.
int m_cardinality;   // the number of names in series.
friend class HistoryNameGen; // a name generator for this.
```

The abstraction function is:

```c
// The abstraction function is:
// AF(c) =
// { <name, c->series->Lookup(name)> | name is in c->series } U
// { c->environments[1], c->environments[2], ..., c->environments[c->m_length] }
```

The representation invariant is:

```c
// The representation invariant is:
// I(c) =
// 1.) c->series != NULL
// 2.) c->environments != NULL
// 3.) c->series->GetCount() == c->m_cardinality
// 4.) c->environments->GetSize() == c->m_length
// 5.) for each name in c->series, c->series->Lookup(name) != NULL
// 6.) for i = 1 to c.m_length, c->environments[i] != NULL
// 7.) there are no two names m,n in c->series such that
```
The specification for the public methods of the History class is given in Figures B-14, B-15, and B-16 of Appendix B. These methods allow one to bind series to names, look up the series bound to a name, associate an index with a variable environment, and look up the variable environment associated with an index. Other methods include concatenation, copying, and assignment.

A major item of note is that these methods are expressed only in terms of Series objects and VariableEnvironment objects. As such, the History object hides the details of its internal representation from outside clients. Clients should never know that any VariableEnvironment object that they bind to a name is copied to a HistoryEnvironment, or that any Series object that they associate to an index is copied to a HistorySeries object. In addition, they should never know that the environments and series that they obtain from a History object are in fact HistoryEnvironment objects and the HistorySeries objects.

Another item of note is that a History object will always copy an inputted Series object or VariableEnvironment object. This mitigates the possibility of a History object invalidating the internal representation of another History object. Without this copying of inputs, a History object can input a HistoryEnvironment object retrieved from some other History object, obtaining access to its internal representation.

As the figures listing its specification might imply, the History object is not the most lightweight of objects. This is basically due to the complex representation invariant that the History object must maintain. Much of the complication lies in ensuring the Slot object consistency required by the design object model.

The specification uses two private methods, updateSeries and updateEnvironments to implement slot consistency across stored HistoryEnvironment objects and HistorySeries objects. Every time a new series-to-variable name binding is added to a History object, the object internally calls updateEnvironments to make sure that the Series object returned on subsequent lookups on the bound variable name is slot-consistent with each of the history’s VariableEnvironment objects. Similarly, this function ensures that any subsequent
retrieval of a **VariableEnvironment** object returns an environment that reflects the addition the **Series** object and is slot-consistent with that **Series** object. When a variable environment-to-index association is added to a **History** object, **updateSeries** is called and it does a similar function, ensuring that the **VariableEnvironment** object returned by lookups on that index are slot-consistent with all the **Series** objects stored within the **History** object.

### 4.2.5 Function

The **Function** object is implemented by a class that stores one **ParameterEnvironment** object, one input **History** object, and one output **History** object. The **Function** object uses a list to store the parameter names for which it needs values and a list to store the variable names for which it needs series. The representation is:

```c
// The representation is:
GCPointer< History > theHistory; // the input history.
GCPointer< ParameterEnvironment > theParams; // the applied parameter environment.
GCPointer< History > outHistory; // the output history.
CList<CString, LPCTSTR> *varNames; // the set of variable names in this.
CList<CString, LPCTSTR> *paramNames; // the set of parameter names in this.
BOOL valid-output; // flag indicating validity of outHistory.
```

The abstraction function is:

```c
// The abstraction function is:
// $\\AF(c) = \{ F(H_i, P) = H_o \mid c\rightarrow valid\_output = \text{TRUE}, H_i = c\rightarrow theHistory,$
// $P = c\rightarrow theParams, H_o = c\rightarrow outHistory \} U$
// $\{ F(H_i, P) \mid c\rightarrow valid\_output = \text{FALSE}, H_i = c\rightarrow theHistory, P = c\rightarrow theParams \}$
```

The representation invariant is:

```c
// The representation invariant is:
// $I(c) =$
// $\{ 1.) c\rightarrow theHistory != NULL.$
// $2.) c\rightarrow theParams != NULL.$
// $3.) c\rightarrow outHistory != NULL.$
// $4.) c\rightarrow varNames != NULL.$
// $5.) c\rightarrow paramNames != NULL.$
// $6.) each name in c\rightarrow varNames has a bound series in c\rightarrow theHistory.$
// $7.) each name in c\rightarrow paramNames has a bound value in c\rightarrow theParams.$
// $8.) there is no name in c\rightarrow varNames that has a bound value in c\rightarrow theParams.$
// $9.) there is no name in c\rightarrow paramNames that has a bound series in c\rightarrow theHistory.$
// $10.) each name in c\rightarrow varNames is unique.$
// $11.) each name in c\rightarrow paramNames is unique.$
// $12.) if c\rightarrow valid\_output = \text{TRUE}, no name in c\rightarrow outHistory has a bound series
// in c\rightarrow theHistory or a bound value in c\rightarrow theParams$
```

72
The specification for the `Function` class is given in Figures B-17, B-18, and B-19 of Appendix B. The methods here allow one to set and retrieve the input `History` object containing series bound to the dependent variables of the function. There are also methods for applying a `ParameterEnvironment` object to the function and for retrieving the environment after it has been applied.

The `Function` object has to maintain a somewhat complicated representation invariant, and the specification ensures its preservation in two ways. First, the specification provides two `private` methods, `validEnvironment` and `validHistory` that give the `Function` object the tools to make sure applied `ParameterEnvironment` objects and set `History` objects satisfy the conditions of the invariant.

Second, the specification only has quasi-reader methods, as it needs to ensure the correctness of the output history of the `Function` object. Any time a client of a `Function` object retrieves its applied `ParameterEnvironment` or its set `History`, there is nothing preventing that client from modifying bound values within those objects. Such modifications will invalidate the output history of the `Function` object. To mitigate this, the reader functions flag the output history of the `Function` object invalid whenever they are called. Invalidating the output history will cause any subsequent clients calls to the output method to throw an `InvalidHistoryException`. The output method itself is not exactly a reader function itself, as it will always return a copy of the output `History` object.

As might be noticed in the method specification given in Figures B-17, B-18, and B-19, the constructors are all declared `protected`. In the same spirit of the `Environment` object, `Function` objects are not meant to be explicitly instantiated. The aim is to have the `Function` object exist through its subsets given in the design object model. Each class that extends the `Function` class overloads the constructor of the `Function` object and its `refresh` method. A subclass's overload of `refresh` gives the method the mathematical functionality represented by this subclass. The subsections below give the specification for each class that extends the `Function` class.
**EpsilonFunction**

The `EpsilonFunction` class overloads the constructor of the `Function` class so that when instantiating an `EpsilonFunction` object, one must specify \( k \), the number of independent variable names in that object. A call to an `EpsilonFunction` object’s overload of the `refresh` method results in a `History` object that binds `Series` objects in the fashion described in Section 3.1.6 of Chapter 3. The representation of the `EpsilonFunction` class is:

```cpp
// The representation is:
int m.k; // the number of independent variables in this.
```

The abstraction function is:

```cpp
// The abstraction function is:
// AF(c) = Function::AF(c)
```

The representation invariant is:

```cpp
// The representation invariant is:
// I(c) =
// 1.) Function::I(c)
// 2.) c->m.k >= 0
// 3.) if k = c->m.k, then names in c->varNames are
// y, x1, ..., xk.
// 4.) if k = c->m.k, then names in c->paramNames are
// b0, b1, ..., bk.
// 5.) if c->valid_output == TRUE, then a name in c->outHistory is
// epsilon.
```

The complete specification for this class is given in Figure B-20 of Appendix B.

**EpsilonSlopeFunction**

The `EpsilonSlopeFunction` class extends the `EpsilonFunction` class. But since its purpose is to also evaluate Eqn. (2.15), it must also be constructed with \( p \) ARCH terms and \( q \) GARCH terms, in addition to being constructed with \( k \) independent variable names. A call to an `EpsilonSlopeFunction` object’s overload of the `refresh` method results in a `History` object that binds `Series` objects in the fashion described in Section 3.1.6 of Chapter 3. The representation of the `EpsilonFunction` class is:

```cpp
// The representation is:
int m.p; // the number of ARCH terms in this.
```
int m_p; // the number of ARCH terms in this.
int m_q; // the number of GARCH terms in this.

The abstraction function is:

// The abstraction function is:
// AF(c) = EpsilonFunction::AF(c)

The representation invariant is:

// The representation invariant is:
// I(c) =
// 1.) EpsilonFunction::I(c)
// 2.) c→m.p ≥ 1
// 3.) c→m.q ≥ 0
// 4.) If c→valid.output == TRUE, k = c→m.k, p = c→m.p, and q = c→m.q,
// then names in c→outHistory are de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalpha1,
// ..., de/dalphap, de/dbeta1, ..., de/dbetaq.

The complete specification for this class is given in Figures B-21 and B-22 of Appendix B.

GarchFunction

The GarchFunction class overloads the constructor of the Function class so that when instantiating a GarchFunction object, one must specify p and q, which are the number of ARCH terms and GARCH terms, respectively, in that object. A call to a GarchFunction object's overload of the refresh method results in a History object that binds Series objects in the fashion described in Section 3.1.6 of Chapter 3. The representation for the GarchFunction class is:

// The representation is:
int m_p; // the number of ARCH terms in this.
int m_q; // the number of GARCH terms in this.

The abstraction function is:

// The abstraction function is:
// AF(c) = Function::AF(c)

The representation invariant is:

// The representation invariant is:
// I(c) =
// 1.) Function::I(c)
// 2.) c→m.p ≥ 1
The complete specification for this class is given in Figures B-23 and B-24 of Appendix B.

GarchSlopeFunction

The GarchSlopeFunction class extends the GarchFunction class. But since its purpose is to also evaluate Eqn. (2.16), it must also be constructed with k independent variable names, in addition to being constructed with p ARCH terms and q GARCH terms. A call to a GarchSlopeFunction object’s overload of the refresh method results in a History object that binds Series objects in the fashion described in Section 3.1.6 of Chapter 3.

The representation of the GarchSlopeFunction class is:

```
// The representation is:
int m.k;  // the number of independent variables in this
```

The abstraction function is:

```
// The abstraction function is:
// AF(c) = GarchFunction::AF(c)
```

The representation invariant is:

```
// The representation invariant is:
// I(c) =
// 1.) GarchFunction::I(c)
// 2.) c->m.k >= 0
// 3.) if k = c->m.k, p = c->m.p, q = c->m.q, then names in c->varNames are
de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalphal,  
... , de/dalphap, de/dbetal, ..., de/dbetaq.
// 4.) if c->valid.output == TRUE, k = c->m.k, p = c->m.p, and q = c->m.q,
then names in c->outHistory are
dv/b0, dv/b1, ..., dv/dbk, dv/dalpha0, dv/dalphal,  
... , dv/dalphap, dv/dbetal, ..., dv/dbetaq.
```

The complete specification for this class is given in Figures B-25 and B-26 of Appendix B.
LikelihoodFunction

The LikelihoodFunction class overloads the constructor of the Function class, but does not require any specification of term structure like the other Function classes described in this section. A call to a LikelihoodFunction object’s overload of the refresh method results in a History object that binds Series objects in the fashion described in Section 3.1.6 of Chapter 3. The representation of the LikelihoodFunction class is just the representation of the Function class, so nothing is shown. The abstraction function is:

```cpp
// The abstraction function is:
// AF(c) = Function::AF(c)
```

The representation invariant is:

```cpp
// The representation invariant is:
// I(c) =
// 1.) Function::I(c)
// 2.) epsilon and variance are names in c->varNames.
// 3.) The series mapped to variance in c->theHistory does not have
//     any zero, negative, or infinite values.
// 4.) if c->valid.output == TRUE, then likelihood is a name in c->outHistory.
```

The complete specification for this class is given in Figure B-27 of Appendix B.

LikelihoodSlopeFunction

The LikelihoodSlopeFunction class extends the LikelihoodFunction class. But since its purpose is to also evaluate Eqn. (2.14), it must also be constructed with k independent variable names, p ARCH terms, and q GARCH terms. A call to a LikelihoodSlopeFunction object’s overload of the refresh method results in a History object that binds Series objects in the fashion described in Section 3.1.6 of Chapter 3. The representation of the LikelihoodSlopeFunction class is:

```cpp
// The representation is:
int m.k; // the number of independent variables in this.
int m.p; // the number of ARCH terms in this.
int m.q; // the number of GARCH terms in this.
```

The abstraction function is:

```cpp
// The abstraction function is:
// AF(c) = LikelihoodFunction::AF(c)
```
The representation invariant is:

```c
// The representation invariant is:
// I(c) =
// 1.) LikelihoodFunction::I(c)
// 2.) c->m.k >= 0
// 3.) c->m.p >= 1
// 4.) c->m.q >= 0
// 5.) if k = c->m.k, p = c->m.p, q = c->m.q, then names in c->varNames are
de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalphal,
\ldots\ldots\ldots\ldots
// dv/b0, dv/b1, ..., dv/dbk, dv/dalpha0, dv/dalphal,
\ldots\ldots\ldots\ldots
// 6.) if c->valid_output == TRUE, then names in c->outHistory are
dl/b0, dl/b1, ..., dl/dbk, dl/dalpha0, dl/dalphal,
\ldots\ldots\ldots\ldots
```

The complete specification for this class is given in Figures B-28 and B-29 of Appendix B.

### 4.3 Testing

Testing for the program was pretty much done in the traditional unit then integration testing format. Each class specified in section 4.2 had a suite of test drivers to verify the implementation for each of its public methods. Each class specified in section 4.2 also provides a repOk method to outside clients, which verifies that an instance of that class is maintaining the class’s representation invariant. Unit testing an implementation for a class, then, involved stressing the public methods of an instance of that class using the test drivers written for it. Each call to a test driver was interspersed with a call to the object’s repOk method.

Due to the high degree of interdependency among the modules, integration testing was acheived as the more complicated implementations for the History class and the Function class and the Function subclasses were unit tested. Unit testing the implementation for the History class necessarily integration-tests the implementations for the Series class and its subclass, the Environment class and its subclasses, and the Slot class. Unit testing the the Function class and the Function subclasses necessarily integration-tests the implementations for every other class given in section 4.2.
Chapter 5

Evaluation

This chapter evaluates the design and implementation presented in the previous two chapters. It does this by presenting alternatives to the design and implementation that were not pursued and analyzes whether or not such decisions were correct. Finally, this chapter ends the thesis, summarizing the major points learned from this research.

5.1 Design Alternatives

There are three good features to the design presented in Chapter 3. First, the design does the important duty of capturing objects relevant to the domain of maximum-likelihood estimation. The History object introduced in Chapter 3 is a very powerful concept, and it represents how data is generally viewed in this domain—series of data bound to variables, where cross-sections of this data are indexed on integers. The Function object then captures how such data within this domain is manipulated to generate new data—applying a parameter environment to a history begets another history. The third good feature of the design is its explicit separation of variables and parameters, paralleling their independent treatment in the problem of maximum-likelihood estimation.

Yet, there are two questions concerning the appropriateness of the design. The first raises the issue of whether or not the Series object is an adequate representation and whether the notion of a time series is a better alternative. The second question considers the Function object in the design object model, raising two issues of necessity and adequacy. The next two subsections address these two questions and argue why the design OM does not ultimately incorporate these design alternatives.
5.1.1 Series

The first question about the design is whether or not the Series object correctly models the series of data actually used by clients of maximum-likelihood estimation. What one observes in the financial risk management or portfolio management area is an everyday use of time series. These series are arrays of values indexed not on integers, but on the dates on which these values were observed. For example, a common series tracked by any firm having to do with the markets is the closing level of the S&P 500:

<table>
<thead>
<tr>
<th>Date</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5/11/99</td>
<td>1355.61</td>
</tr>
<tr>
<td>5/12/99</td>
<td>1364.00</td>
</tr>
<tr>
<td>5/13/99</td>
<td>1367.56</td>
</tr>
<tr>
<td>5/14/99</td>
<td>1337.80</td>
</tr>
</tbody>
</table>

However, the object model given Chapter 3 gives a Series object that is indexed on integers, potentially opening the design OM to the argument that it is inadequate. However, it is argued here that indexing series on integers is a better approach, as it is a superset of the approach of indexing series on dates. The S&P 500 series above is still compatible with the design OM; all one has to do is assign to each value an integer defining that value's order in the array. But if dates are absolutely required as indices, then what one can do is extend the object model to include the notion of the time series. Such an extension is given in Figure 5-1.

If the design OM only considered series as date-indexed values, this extension would be less natural. Such an extension would say that all series are date-indexed and that only a subset are indexed on integers. Clearly, extending the current design OM’s notion of an integer-indexed series to a date-indexed time series is more natural. Time series are not essential to the problem of maximum-likelihood estimation, but if they are needed, the design OM can accommodate their existence as a subset of the current Series object.

5.1.2 Function

The second question brings up two concerns about the Function object. The question is about whether or not the Function object is necessary to the design object model, and if it is, then whether or not the current Function object model is adequate. The necessity of the Function object is questionable if one argues that it is purely a functional construct. In other words, one might argue that having it in the object model is like having an object
Yet, the design OM includes the Function object with the belief that it helps motivate the existence of the other objects in the model. Environments, series, and histories can exist on their own right, but their inclusion in an object model for maximum-likelihood estimation makes better sense in light of their relations to the Function object. In addition, the Function is the only object in the model that makes apparent the difference between a variable and a parameter. But put simply, the ultimate goal of maximum-likelihood estimation is to optimize the parameters of a function. An object model for this problem must somehow feature functions; as such, the design OM of Chapter 3 features a Function object.

But is the object model currently given for the Function object adequate? Originally, the design was supposed to have functions associate histories and parameter environments with equation constructs. Under this idea, if a function is projected on a history, the function yields an equation that associates a parameter environment with another history. Similarly, if a function is projected on a parameter environment, the function yields an equation that associates a history with another history. The purpose of these equation constructs is to illustrate that a function is the Cartesian product of all possible histories and parameter environments that are applicable to that function. An evaluation of a function is choosing a
particular \{\text{History}, \text{ParameterEnvironment}\} object tuple of this product, applying the \text{ParameterEnvironment} object to the \text{History} object to yield another \text{History}.

The one attraction of the above representation is that it explicitly shows that functions evaluate to different histories, depending on the chosen tuple in the Cartesian product. Yet, the design OM does not model \text{Function} objects in such a fashion. First, the above representation for a function is a little obtuse. In addition, the idea of having intermediate \text{Equation} objects is cumbersome, and their existence cannot be really justified.

It is simpler and more straightforward conceptually to have a \text{Function} object be associated with only one tuple of the Cartesian product of all possible histories and parameter environments. A \text{Function} object is associated, then, with only one \{\text{History}, \text{ParameterEnvironment}\} tuple, and the application of its one \text{ParameterEnvironment} to its one \text{History} yields another \text{History} object. If one needs a particular \text{Function} object to evaluate to different \text{History} object, then one can just reassign the associated tuple and reevaluate the \text{Function} object.

5.2 Implementation Alternatives

The implementation approach employed during the thesis work was to assign a class to each object in the design object model. This approach resulted in an implementation that was natural to carry out, as the design object model is clear about where objects are in reference to each other. However, there is one implementation alternative that might have been pursued but wasn’t.

\textbf{Function}

The current implementation uses the \text{Function} class to implement the \text{Function} object. An alternative that could have been pursued is to use procedural abstraction and have the \text{Function} object be implemented as a class of static methods. For example, instead of implementing the \text{GarchFunction} class, the code instead could just feature a function below:

\begin{verbatim}
History garch_function(int p, int q, const History& input, const ParameterEnvironment& env);
\end{verbatim}

This procedure could then be subsumed in some class of static methods:
class Function
{
public:

    static History epsilon_function(int k, const History& input, const ParameterEnvironment& env);

    static History garch_function(int p, int q, const History& input, const ParameterEnvironment& env);

private:

    Function();
    ~Function();

}; // end Function.

There really is nothing wrong with the above approach. One might even argue that procedural abstraction is the best approach, since the simplest and most natural representation for a function in code is a procedure.

And yet, the implementation chooses to represent the Function object as a class. The motivation here is that there are areas in the program where it would be convenient to have a Function object. An example is where a procedure will pass in as input the output history of a GarchFunction object. However, in order to retrieve the series bound within this history, the procedure will also need to know the \( p \) and \( q \) parameters for this object. Fortunately, a GarchFunction object contains such information, so a convenient solution would be to pass to the procedure a reference to a GarchFunction object rather than its output history.

This same argument is applicable to the return value of a procedure. For example, one might have a procedure in code that returns the output history of an EpsilonFunction object. Returning this history to the procedure’s caller is insufficient, as the caller might need to know the number of independent variables that were used to calculate the series bound to the name epsilon in that History object. Again, a convenient solution would be to return a copy of the EpsilonFunction that outputs that History object.
5.3 Lessons Learned

This thesis studies the approach of object modeling when applied to the compute-intensive problem of maximum-likelihood estimation, a problem that has generally been tackled using functional approaches. What results is an object model that clearly lays out the nature of the information in the problem, showing that there is complexity in its structure. The complexity of this object model consequently expresses a deficiency in a functional approach. Such an approach assumes that the modeling of information can be easily solved, e.g., “just malloc a bunch of arrays”, and does not prescribe such rigor in the consideration of data. Rather, a functional approach makes it too easy to start from the point of view of the main() function, putting all design and development effort in the processing of the data.

What also results through the creation of an object model is a contract that tells the implementer of the design what one can and cannot do in code. If diligently followed, what then results are code modules that strictly follow invariants throughout their lifetimes, and having such guarantees is crucial when designing and implementing large, complex systems. Functional approaches, with their focus on process, do not place in the design process such focus on representation invariance.

This is not to say that invariance cannot be incorporated in such approaches; it is just common for programmers, especially in the domain of risk management and portfolio management, to just allocate data structures and not think about what assumptions they are making about those structures. And if they do consider their assumptions, their approach lacks ways for them to guarantee the validity of their assumptions throughout the lifetimes of their data structures. It is nearly impossible, for example, to hide the internal representation of a C-style struct from anything that can get a handle to that struct.

However, an ultimate question that any customer of a program for maximum-likelihood estimation is bound to ask is, “What is the performance penalty?”. As seen in Chapter 4, one really gets to see and takes advantage of the power of the abstractions in an object model when one utilizes classes in code. The use of classes do not necessarily impose a performance hit in comparison to a program that does not use classes, just because the latter program will most probably be using data structures that are analogous to a class.

But one cannot argue that the object model prescribe invariants and constraints for each class that must be maintained by each instance of that class. Imposing such requirements
adds some implementation overhead. For example, the Function object has a seemingly innocuous clause in its representation invariant that states that its output History object must be valid whenever its internal valid flag is true. However, having this representation invariant introduces quite a few inefficiencies in the Function object.

First, any retrieval of the output History of a Function object must be a copy, rather than a relatively cheaper reference, to the actual history stored by the Function object. This protects the stored output History object from any outside changes that might invalidate it. Second, any reads of the History object set as input to the Function object or any reads of the applied ParameterEnvironment automatically invalidate the output history. The conservative assumption is that changes might occur to these objects that might invalidate the output history. Consequently, any call to the parameters or history method of a Function object will require a subsequent call to the computationally expensive refresh function, if the output history will be needed anytime later.

One can of course get around these issues of efficiency by coding procedures as described in the previous section, or by removing the clause from the Function class's representation invariant that guarantees the correctness of its output history. However, doing so places responsibility of correctness in the hands of the developer, and in the hands of anyone else who is working on the system. For relatively simple tasks, this is perfectly fine. However, this thesis deals with complicated data structures that have quite a degree of interdependencies, exposing objects to myriad ways to modify them. It is worth the extra clock cycles to ensure that these interdependencies can never affect the correctness of objects in use.

And in the end, the numbers output by an estimation program must be correct beyond any doubt. The mathematics of maximum-likelihood estimation are complex, so ensuring the correctness of code that implements it is already hard. Much of the burden can be managed through abstractions that guarantee a certain behavior. For example, it helps immensely to just have a simple array that holds a bunch of numbers. A standard hope for such an abstraction is that it helps one avoid mistakes, such as accessing a value out of bounds.

However, a functional approach might just provide a contiguous block of memory to serve as an array. Again, why not just malloc a bunch of arrays? A bad scenario is a program using such a data structure accessing an element out of bounds, causing a memory fault. An even more subtle and disastrous scenario that is seen all too often in programs without strict
Abstracts is the program accessing an element out of bounds and successfully retrieving a value. The program might easily run to completion, where the only error in execution is in the number that is returned. No software developer desires such subtleness of errors.

To conclude, there are three things to be learned from this thesis:

1. Object modeling features an organizational power not to be found in functional approaches.
   - provides invariants that organizes implementation; makes clear what one can and cannot do in an object.
   - mutability constraints state where objects must be and where there are degrees of flexibility.
   - establishes a contract among developers. Although thesis work was done alone, having that blueprint kept work on track.

2. The design object model only features constraints unique to the problem at hand.
   - makes no assumptions concerning how the design will be ultimately implemented.
   - does not concern itself with a target language or platform.

3. There is a tradeoff in performance when pursuing an object modeling approach.
   - Realizing the power of the abstractions in the design object model generally requires the use of classes in the actual code. Classes themselves abstract their internal representations, adding a layer of indirection.
   - The constraints of design object model translate to a representation invariant that each class must maintain.
   - However, the tradeoff is worth making for the organization and program correctness that abstraction and representation invariance provide.
Appendix A

Additional Derivations

This section is dedicated to giving a more in-depth derivation for the gradient of $\mathcal{L}$. Just to reiterate what was defined in Chapter 2, the gradient of $\mathcal{L}$ is defined as:

$$
g = \frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \theta}
$$

(A.1)

where

$$
\frac{\partial l_t}{\partial \theta} = \begin{pmatrix} \frac{\partial l_t}{\partial b} \\ \frac{\partial l_t}{\partial \omega} \end{pmatrix}
$$

(A.2)

Taking the derivative of $l_t$ with respect to $b$ yields:

$$
\frac{\partial l_t}{\partial b} = \frac{\partial}{\partial b} \left[ -\frac{1}{2} \ln \sigma_t^2 - \frac{\varepsilon_t^2}{2\sigma_t^2} \right] = -\frac{1}{2\sigma_t^2} \cdot \frac{\partial \sigma_t^2}{\partial b} - \frac{\varepsilon_t^2}{2\sigma_t^4} \cdot \frac{\partial \sigma_t^2}{\partial b} = -\frac{\varepsilon_t}{\sigma_t^2} \cdot \frac{\partial \varepsilon_t}{\partial b} + \frac{1}{2\sigma_t^2} \left[ \frac{\varepsilon_t^2}{\sigma_t^2} - 1 \right] \cdot \frac{\partial \sigma_t^2}{\partial b}
$$

(A.3)

(A.4)

(A.5)
Letting \( \eta = -\eta^2 / \sigma_i^2 \) and \( \zeta = 1 / 2 \sigma_i^2 \left( \frac{\varepsilon^2_i}{\sigma_i^2} - 1 \right) \), one can expand (A.5) as so:

\[
\begin{align*}
\frac{\partial l_t}{\partial \mathbf{b}} &= \begin{pmatrix}
\eta + \zeta \frac{\partial \sigma_i^2}{\partial b_0} \\
\eta x_{1t} + \zeta \frac{\partial \sigma_i^2}{\partial b_1} \\
\vdots \\
\eta x_{kt} + \zeta \frac{\partial \sigma_i^2}{\partial b_k}
\end{pmatrix} \\
&= \eta \mathbf{X}'_t + \zeta \begin{pmatrix}
\frac{\partial \sigma_i^2}{\partial b_0} \\
\frac{\partial \sigma_i^2}{\partial b_1} \\
\vdots \\
\frac{\partial \sigma_i^2}{\partial b_k}
\end{pmatrix}
\end{align*}
\]

where \( \mathbf{X}_t \) is the \( t \)th row of \( \mathbf{X} \) and

\[
\frac{\partial \sigma_i^2}{\partial b_n} = \begin{cases}
-2 \sum_{i=1}^{p} \alpha_i \varepsilon_{t-i} + \sum_{j=1}^{q} \beta_j \frac{\partial \sigma_{t-i-j}^2}{\partial b_n} & n = 0 \\
-2 \sum_{i=1}^{p} \alpha_i \varepsilon_{t-i} x_n(t-i) + \sum_{j=1}^{q} \beta_j \frac{\partial \sigma_{t-i-j}^2}{\partial b_n} & n > 0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

This leads to the expression of (A.5) as:

\[
\frac{\partial l_t}{\partial \mathbf{b}} = \eta \mathbf{X}'_t + \zeta \left[ -2 \sum_{i=1}^{p} \alpha_i \varepsilon_{t-i} \mathbf{X}'_{t-i} + \sum_{j=1}^{q} \beta_j \frac{\partial \sigma_{t-i-j}^2}{\partial \mathbf{b}} \right] 
\]

(A.6)

In a similar fashion, one can take the derivative of \( l_t \) with respect to \( \omega \), and noting that \( \frac{\partial \varepsilon_i}{\partial \omega} = 0 \), this yields:

\[
\frac{\partial l_t}{\partial \omega} = \frac{\partial}{\partial \omega} \left[ -\frac{1}{2} \ln \sigma_i^2 - \frac{\varepsilon_i^2}{2 \sigma_i^2} \right] 
\]

(A.7)

\[
= -\frac{1}{2 \sigma_i^2} \frac{\partial \sigma_i^2}{\partial \omega} - \left[ \frac{\varepsilon_i}{\sigma_i^2} \cdot \frac{\partial \varepsilon_i}{\partial \omega} - \frac{\varepsilon_i^2}{2 \sigma_i^2} \cdot \frac{\partial \sigma_i^2}{\partial \omega} \right] 
\]

(A.8)

\[
= -\frac{1}{2 \sigma_i^2} \frac{\partial \sigma_i^2}{\partial \omega} + \frac{\varepsilon_i^2}{2 \sigma_i^4} \cdot \frac{\partial \sigma_i^2}{\partial \omega} 
\]

(A.9)

\[
= \frac{1}{2 \sigma_i^2} \left( \frac{\varepsilon_i^2}{\sigma_i^2} - 1 \right) \cdot \frac{\partial \sigma_i^2}{\partial \omega} 
\]

(A.10)
And similarly, one can expand (A.10) as so:

\[
\frac{\partial l_t}{\partial \omega} = \zeta \left( \begin{array}{c}
\frac{\partial \sigma_{t-1}^{2}}{\partial \sigma_{0}} \\
\frac{\partial \sigma_{t}^{2}}{\partial \sigma_{0}} \\
\frac{\partial \sigma_{t}^{2}}{\partial \sigma_{1}} \\
\vdots \\
\frac{\partial \sigma_{t}^{2}}{\partial \beta_{r}} \\
\frac{\partial \sigma_{t}^{2}}{\partial \beta_{r}} \\
\end{array} \right)
\]

where

\[
\frac{\partial \sigma_{t}^{2}}{\partial \alpha_{n}} = \begin{cases}
1 + \sum_{j=1}^{q} \beta_{j} \frac{\partial \sigma_{t-j}^{2}}{\partial \alpha_{n}} & n = 0 \\
\varepsilon_{t-n}^{2} + \sum_{j=1}^{q} \beta_{j} \frac{\partial \sigma_{t-j}^{2}}{\partial \alpha_{n}} & n > 0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

\[
\frac{\partial \sigma_{t}^{2}}{\partial \beta_{n}} = \begin{cases}
\sigma_{t-n}^{2} + \sum_{j=1}^{q} \beta_{j} \frac{\partial \sigma_{t-j}^{2}}{\partial \beta_{n}} & n > 0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

This leads (A.10) to become

\[
\frac{\partial l_t}{\partial \omega} = \zeta \left[ x_t' + \sum_{j=1}^{q} \beta_{j} \frac{\partial \sigma_{t-j}^{2}}{\partial \omega} \right] \tag{A.11}
\]

Substituting (A.2), (A.6), and (A.11) into (A.1), produces the gradient of \( \mathcal{L} \).
Appendix B

Class Specifications

This section presents the class specifications for the objects presented in Section 4.2:

- RefCount
- GCPPointer
- Slot
- Environment
- VariableEnvironment
- ParameterEnvironment
- HistoryEnvironment
- Series
- HistorySeries
- History
- Function
- EpsilonFunction
- EpsilonSlopeFunction
- GarchFunction
- GarchSlopeFunction
- LikelihoodFunction
- LikelihoodSlopeFunction

The specifications for the class templates feature implementation code, as the Microsoft Visual C++ linker could not resolve the types for the methods of these classes if done otherwise.
class RefCount
{
  // overview: A RefCount is an object that keeps track
  //           of how many pointer references there are to it.

private:
  // The representation is:
  int crefs;

  // The abstraction function is:
  // AF(c) = { c→crefs }

  // The representation invariant is:
  // I(c) = c→crefs ≥ 0

public:
  RefCount(void) {
    // modifies:  this
    // effects:   Instantiates this as an instance of the RefCount class
                // with reference count = 0.
    crefs = 0;
  }

  virtual ~RefCount() {
    // modifies:  this
    // effects:   Removes this from memory.
  }

  void upcount(void) {
    // modifies:  this
    // effects:   Increments the reference counter of this by one.
    ++crefs;
  }

  void downcount(void) {
    // modifies:  this
    // effects:   If the reference counter of this is 1, removes this
                // from memory.
                // Otherwise, decrements the reference counter of this
                // by one.
    if (--crefs == 0) delete this;
  }

}; // end RefCount

Figure B-1: The RefCount class specification
template <class T>
class GCPointer
{
    // overview: A GCPointer is an object that stores a pointer to another
    // object whose type was specified when the GCPointer object
    // was first instantiated.
    // The type pointed to by this MUST extend the RefCount class.

    protected:
    // The representation is:
    T* p;

    // The abstraction function is:
    // AF(c) = { c->p }

    // The representation invariant is:
    // I(c) = True

    public:
    GCPointer() {
        // modifies: this
        // effects: Instantiates this as an instance of the GCPointer<T>
        // class pointing to null.
        p = NULL;
    }

    GCPointer(const GCPointer& GCPtr) {
        // modifies: this
        // effects: Instantiates this as a copy of 'GCPtr'.
        p = NULL;
        *this = GCPtr;
    }

    GCPointer(T* p_) : p(p_) {
        // requires: p_ is an instance of the RefCount class.
        // modifies: this
        // effects: Sets this to point to p_, incrementing the reference
        // in p_ if p_ is not null.
        if (p_ != NULL)
            p->upcount();
    }

    "GCPointer(void) {
        // modifies: this
        // effects: If this is pointing to an object, decrements its
        // reference count. Removes this from memory.
        if (p != NULL)
            p->downcount();
    }

    operator T*(void) {
        // effects: Returns a pointer to the object pointed to by this.
        return p;
    }


Figure B-2: The first part of the GCPointer class specification
T& operator*(void) {
    // effects: Returns a reference to the object pointed to by this.
    return *p;
}

T* operator->(void) {
    // effects: Returns a pointer to the object pointed to by this.
    return p;
}

operator T*(void) const {
    // effects: Returns a constant pointer to the object pointed to by this.
    return p;
}

T* operator->(void) const {
    // effects: Returns a constant pointer to the object pointed to by this.
    return p;
}

const GCPointer& operator=(T* p) {
    // modifies: this
    // effects: If the object pointed to by this is not null, decrements
    // the reference count of that object. Sets the object
    // pointed to by this to the object pointed to by p.
    // If p is not null, then its reference counter is incremented.
    if (p != NULL)
        p->downcount();
    p = p;
    if (p != NULL)
        p->upcount();
    return *this;
}

const GCPointer& operator=(const GCPointer& GCPtr) {
    // modifies: this
    // effects: If the object pointed to by this is not null, decrements
    // the reference count of that object. Sets the object
    // pointed to by this to the object pointed to by 'GCPtr'.
    // If the object pointed to by 'GCPtr' is not null, then
    // its reference counter is incremented.
    if (p != NULL)
        p->downcount();
    p = GCPtr.p;
    if (p != NULL)
        p->upcount();
    return *this;
}

}; // end GCPointer

Figure B-3: The second part of the GCPointer class specification
template <class T>
class Slot : public RefCount
{
   // overview: A slot is a polymorphic container that stores a
   // pointer to a value of the type that was specified when the
   // slot was first instantiated.

private:
   // The representation is:
   T *val;
   
   // The abstraction function is:
   // AF(c) = \{ c→val \}
   
   // The representation invariant is:
   // I(c) = c→val != NULL

public:
   Slot() {
      // modifies: this
      // effects: Instantiates this as an instance of the Slot<T> class.
      val = new T();
   }
   
   Slot(const T& v) {
      // modifies: this
      // effects: Instantiates this as an instance of the Slot<T> class
      // and initializes it to point to a copy of 'v'.
      val = new T(v);
   }
   
   Slot (const Slot& slot) {
      // modifies: this
      // effects: Instantiates this as a copy of 'slot'.
      val = NULL;
      *this = slot;
   }
   
   virtual ~Slot() {
      // modifies: this
      // effects: Removes this from memory.
      delete val;
   }
   
   void setSlot(const T& v) {
      // modifies: this
      // effects: Deletes whatever this is pointing to and sets this to
      // point to a copy of 'v'.
      delete val;
      val = new T(v);
   }
   
   T getSlot() const {
      // effects: Returns a copy of the value pointed to by this.
      return *val;
   }

Figure B-4: The first part of the Slot class specification
operator T () const {
    // effects:  Casts this to an immutable form of the value pointed to by this.
    return *val;
}

operator T& () {
    // modifies:  this
    // effects:  Casts this to a mutable form of the value pointed to by this.
    // requires:  The destructor of the returned value is never explicitly called.
    return *val;
}

const Slot& operator= (const T& v) {
    // modifies:  this
    // effects:  Sets this to point to an independent copy of 'v'.
    delete val;
    val = new T(v);
    return *this;
}

const Slot& operator= (const Slot& slot) {
    // modifies:  this
    // effects:  Sets this to be an independent copy of 'slot'.
    delete val;
    val = new T(*(slot.val));
    return *this;
}

inline virtual CString toString() const;
    // effects:  Returns the string representation of this.

inline virtual BOOL repOk() const {
    // effects:  Returns True if this preserves the representation invariant.
    // False, otherwise.
    return (val != NULL);
}
}; // end Slot<T>

CString Slot<double>::toString () const {
    CString returnStr;
    returnStr.Format("%10.6f", (double)*val);
    return returnStr;
}

CString Slot<float>::toString () const {
    CString returnStr;
    returnStr.Format("%10.6f", (double)*val);
    return returnStr;
}

CString Slot<int>::toString () const {
    CString returnStr;
    returnStr.Format("%10d", (int)*val);
    return returnStr;
}

Figure B-5: The second part of the Slot class specification
class Environment : public RefCount
{
    // overview: An environment is a set of value-to-name bindings
    // of the form <name,value> where there are no two names
    // that are the same.

protected:
    // The representation is:
    CMap<CString, LPCTSTR, GCPointer< Slot<double> >, GCPointer< Slot<double> > > *bindings;
    int m_cardinality;  // the number of names in bindings.
    friend class EnvironmentNameGen;  // a name generator for this.

    // The abstraction function is:
    // AF(c) = { <name, c/bindings/Lookup(name)→val> | name is in bindings }

    // The representation invariant is:
    // I(c) =
    // 1.) c/bindings != NULL
    // 2.) c/bindings/GetCount() == c/m/cardinality
    // 3.) for each name in c/bindings, c/bindings/Lookup(name) != NULL
    // 4.) there are no two names m,n in c/bindings such that
    //     c/bindings/Lookup(m) == c/bindings/Lookup(n)

protected:
    Environment();
    // requires: this is a derived subclass.
    // modifies: this
    // effects: Instantiates this as an instance of the Environment class.

    Environment(const Environment& origEnvironment);
    // requires: this is a derived subclass and 'origEnvironment' is an instance of that subclass.
    // modifies: this
    // effects: Instantiates this as a copy of 'origEnvironment'.

    const Environment& operator= (const Environment &origEnvironment);
    // requires: this is a derived subclass and origEnvironment is an
    // instance of that subclass.
    // modifies: this
    // effects: Sets this to be an independent copy of 'origEnvironment'.

    Environment operator+ (const Environment &appendEnvironment) const;
    // requires: this is a derived subclass and appendEnvironment is an
    // instance of that subclass.
    // effects: Throws DuplicateNameException if there is any name in
    // 'appendEnvironment' that is also in this. Otherwise,
    // returns a new Environment that combines the bindings of
    // this with 'appendEnvironment'.

public:
    virtual ~Environment();
    // modifies: this
    // effects: Removes this from memory.

Figure B-6: The first part of the Environment class specification
void bind(LPCTSTR name, double value);
   // modifies: this
   // effects: Throws DuplicateNameException if 'name' already has a bound
   // 'value'. Otherwise, binds 'value' to 'name', and adds
   // binding to this.

void rebind(LPCTSTR name, double value) const;
   // effects: Throws NameNotFoundException if 'name' is not found in this.
   // Otherwise, changes the value bound to 'name' to 'value'.

double lookup(LPCTSTR name) const;
   // effects: Throws NotFoundException if 'name' is not found in this.
   // Otherwise, returns the value bound to 'name'.

double& operator[](LPCTSTR name) const;
   // modifies: this
   // effects: If used on the left-hand side of an '==' operator, same effect as
   // this-rebind(name, right-hand side of '==' operator).
   // Otherwise, same effect as this-lookup(name).

int cardinality() const;
   // effects: Returns the number of names in this.

Enumeration<LPCTSTR>* names() const;
   // effects: Returns a generator that will produce each name stored in this
   // exactly once, in arbitrary order.
   // requires: this not be modified while the returned generator is in use.

virtual CString toString() const;
   // effects: Returns the string representation of this.

virtual BOOL repOk() const;
   // effects: Returns True if this preserves the representation invariant.
   // False, otherwise.

}; // end Environment

Figure B-7: The second part of the Environment class specification
class VariableEnvironment : public Environment
{
  // overview: A variable environment is a set of value-to-variable name
  // bindings of the form <variable name, value> where there are
  // no two variable names that are the same.

  // The abstraction function is:
  // AF(c) = Environment::AF(c)

  // The representation invariant is:
  // I(c) = Environment::I(c)

public:
  VariableEnvironment();
  // modifies: this
  // effects: Instantiates this as an instance of the VariableEnvironment class.

  VariableEnvironment(const VariableEnvironment& origEnvironment);
  // modifies: this
  // effects: Instantiates this as a copy of 'origEnvironment'.

protected:
  VariableEnvironment(const Environment& origEnvironment);
  // requires: origEnvironment is an instance of VariableEnvironment.
  // modifies: this
  // effects: Instantiates this as a subclassed copy of 'origEnvironment'.

public:
  virtual ~VariableEnvironment();
  // modifies: this
  // effects: Removes this from memory.

  const VariableEnvironment& operator=(const VariableEnvironment &origEnvironment);
  // modifies: this
  // effects: Sets this to be an independent copy of 'origEnvironment'.

  VariableEnvironment operator+(const VariableEnvironment &appendEnvironment) const;
  // effects: Throws DuplicateNameException if there is any name in
  // 'appendEnvironment' that is also in this. Otherwise,
  // returns a new VariableEnvironment that combines the bindings of
  // this with 'appendEnvironment'.

}; // end VariableEnvironment

Figure B-8: The VariableEnvironment class specification
class ParameterEnvironment : public Environment
{
    // overview: A parameter environment is a set of value-to-parameter name
    // bindings of the form <parameter name, value> where there are
    // no two parameter names that are the same.

    // The abstraction function is:
    // \( AF(c) = \text{Environment}::AF(c) \)

    // The representation invariant is:
    // \( I(c) = \text{Environment}::I(c) \)

public:
    ParameterEnvironment();
    // modifies: this
    // effects: Instantiates this as an instance of the
    // ParameterEnvironment class.

    ParameterEnvironment(const ParameterEnvironment& origEnvironment);
    // modifies: this
    // effects: Instantiates this as a copy of 'origEnvironment'.

protected:
    ParameterEnvironment(const Environment& origEnvironment);
    // requires: origEnvironment is an instance of ParameterEnvironment.
    // modifies: this
    // effects: Instantiates this as a subclassed copy of 'origEnvironment'.

public:
    virtual ~ParameterEnvironment();
    // modifies: this
    // effects: Removes this from memory.

    const ParameterEnvironment& operator= (const ParameterEnvironment &origEnvironment);
    // modifies: this
    // effects: Sets this to be an independent copy of 'origEnvironment'.

    ParameterEnvironment operator+ (const ParameterEnvironment &appendEnvironment) const;
    // effects: Throws DuplicateNameException if there is any name in
    // 'appendEnvironment' that is also in this. Otherwise,
    // returns a new ParameterEnvironment that combines the bindings of
    // this with 'appendEnvironment'.

}; // end ParameterEnvironment

Figure B-9: The ParameterEnvironment class specification
class HistoryEnvironment : public VariableEnvironment
{
    // overview: A history environment is simply a variable environment that
             // exposes its representation to histories.

    private:
    // The representation is:
    friend class History; // expose the rep to history to allow direct updates.
    // The abstraction function is:
    //  AF(c) = VariableEnvironment::AF(c)
    // The representation invariant is:
    //  I(c) = VariableEnvironment::I(c)

    HistoryEnvironment();
    // modifies:  this
    // effects:   Instantiates this as an instance of the HistoryEnvironment class.

    HistoryEnvironment(const HistoryEnvironment& origEnvironment);
    // modifies:  this
    // effects:   Instantiates this as a copy of 'origEnvironment'.

    HistoryEnvironment(const VariableEnvironment& origEnvironment);
    // modifies:  this
    // effects:   Instantiates this as a subclassed copy of 'origEnvironment'.

    virtual ~HistoryEnvironment();
    // modifies:  this
    // effects:   Removes this from memory.

};  // end HistoryEnvironment

Figure B-10: The HistoryEnvironment class specification
class Series : public RefCount
{
  // overview: A series is an ordered multiset of values of type double,
  //          where the index of a value is its order in the set, beginning
  //          with one.

protected:
  // The representation is:
  CArray<GCPolygon<Slot<double>>, GCPolygon<Slot<double>>>* slotArray;
  int m_length; // the number of elements in this.

  // The abstraction function is:
  // AF(c) = { c->slotArray[1]->val, c->slotArray[2]->val, ..., c->slotArray[m_length]->val }

  // The representation invariant is:
  // I(c) =
  // 1.) c->slotArray != NULL
  // 2.) c->slotArray->GetSize() == c->m_length
  // 3.) for i = 1 to c->m_length
  //     { c->slotArray[i] != NULL }
  // 4.) there is no i,j such that
  //      i != j && c->slotArray[i] == c->slotArray[j]

public:
  Series();
  // modifies: this
  // effects: Instantiates this as an instance of the Environment class.

  Series(const Series& origSeries);
  // modifies: this
  // effects: Instantiates this as a copy of 'origSeries'.

  virtual ~Series();
  // modifies: this
  // effects: Removes this from memory.

  void set(int i, double value) const;
  // modifies: this
  // effects: Throws OutOfBoundsException if 'i' <= 0 or if 'i' > this->length().
  //          Otherwise, sets the value at index 'i' to 'value'.

  double get(int i) const;
  // modifies: this
  // effects: Throws OutOfBoundsException if 'i' <= 0 or if 'i' > this->length().
  //          Otherwise, returns the value of this set at index 'i'.

  double& operator[](int i) const;
  // modifies: this
  // effects: If used on the left-hand side of an '=' operator, same effect as
  //          this->set[i], right-hand side of '=' operator.
  //          Otherwise, same effect as this->get(i).

  void add(double value);
  // modifies: this
  // effects: Appends 'value' to the end of this.

Figure B-11: The first part of the Series class specification
void insert(int i, double value);
  // modifies: this
  // effects: Throws OutOfBoundsException if i ≤ 0 or if i > this.length().
  // Otherwise, inserts 'value' at the 'i'\text{th} index of this, shifting
  // the element at that index and those above up one notch.

void remove(int i);
  // modifies: this
  // effects: Throws OutOfBoundsException if i ≤ 0 or if i > this.length().
  // Otherwise, removes that value stored at the 'i'\text{th} index of this.

const Series& operator= (const Series &origSeries);
  // effects: Sets this to be an independent copy of 'origSeries'.

int length() const;
  // effects: Returns the number of values in this.

virtual CString toString() const;
  // effects: Returns the string representation of this.

virtual BOOL repOk() const;
  // effects: Returns True if this preserves the representation invariant.
  // False, otherwise.

}; // end Series

Figure B-12: The second part of the Series class specification
class HistorySeries : public Series
{
    // overview: A history series is simply a series that
    // exposes its representation to histories.

private:
    // The representation is:
    friend class History; // expose the rep to history to allow direct updates.

    // The abstraction function is:
    // AF(c) = Series::AF(c)

    // The representation invariant is:
    // I(c) = Series::I(c)

    HistorySeries( );
        // modifies: this
        // effects: Instantiates this as an instance of the HistorySeries class.

    HistorySeries(const HistorySeries& origSeries);
        // modifies: this
        // effects: Instantiates this as a copy of 'origSeries'.

    HistorySeries(const Series& origSeries);
        // modifies: this
        // effects: Instantiates this as a subclassed copy of 'origSeries'.

    virtual ~HistorySeries( );
        // modifies: this
        // effects: Removes this from memory.

}; // end HistorySeries

Figure B-13: The HistorySeries class specification
class History : public RefCount
{
  /// overview: A history contains a set of series-to-variable name bindings and
  /// a set of indexed variable environments. The bindings are of
  /// the form \langle name, variable environment\rangle, where no two names are
  /// the same. An index specifies the order of one variable environment
  /// within the set, starting with zero.

private:
  /// The representation is:
  CMap<CString, LPCTSTR, GCPointer<HistorySeries>, GCPointer<HistorySeries>> *series;
  CArray<GCPointer<HistoryEnvironment>, GCPointer<HistoryEnvironment>> *environments;
  int m_length;  /// the length of environments.
  int m_cardinality;  /// the number of names in series.
  friend class HistoryNameGen;  /// a name generator for this.

  /// The abstraction function is:
  /// \( \text{AF}(c) = \{ \langle \text{name}, c\rightarrow\text{series}\rightarrow\text{Lookup}(\text{name}) \rangle | \text{name is in } c\rightarrow\text{series} \} U \{ c\rightarrow\text{environments}[1], c\rightarrow\text{environments}[2], ..., c\rightarrow\text{environments}[c\rightarrow m\_length] \} \)

  /// The representation invariant is:
  /// If(c) =
  /// 1.) c\rightarrow\text{series} != NULL
  /// 2.) c\rightarrow\text{environments} != NULL
  /// 3.) c\rightarrow\text{series}\rightarrow\text{GetCount()} == c\rightarrow m\_cardinality
  /// 4.) c\rightarrow\text{environments}\rightarrow\text{GetSize()} == c\rightarrow m\_length
  /// 5.) for each name in c\rightarrow\text{series}, c\rightarrow\text{series}\rightarrow\text{Lookup}(\text{name}) != NULL
  /// 6.) for i = 1 to c.m.length, c\rightarrow\text{environments}[i] != NULL
  /// 7.) there are no two names m, n in c\rightarrow\text{series} such that
  ///     c\rightarrow\text{series}\rightarrow\text{Lookup}(m) == c\rightarrow\text{series}\rightarrow\text{Lookup}(n)
  /// 8.) there are no indices i, j such that
  ///      i != j && c\rightarrow\text{environments}[i] == c\rightarrow\text{environments}[j]
  /// 9.) for each name in c\rightarrow\text{series}
  ///     for i = 1 to c.m.length
  ///      c\rightarrow\text{series}\rightarrow\text{Lookup}(\text{name})\rightarrow\text{slotArray}[i] == c\rightarrow\text{environments}[i]→bindings→\text{Lookup}(\text{name})

public:
  History();
  /// modifies: this
  /// effects: Instantiates this as an instance of the History class.

  History(const History& origHistory);
  /// modifies: this
  /// effects: Instantiates this as a copy of 'origHistory'.

  virtual ~History();
  /// modifies: this
  /// effects: Removes this from memory.

Figure B-14: The first part of the History class specification
void bind(LPCTSTR name, const Series& series);
// modifies: this
// effects:  Throws InvalidSeriesException if there exist series in this
// and this->length() != 'series'.length().
// Throws DuplicateNameException if 'name' is already associated
// with another series.
// Otherwise, binds 'series' to 'name' and adds binding to this.
// It also adds a value to 'name' binding in each variable environment
// in this, according to the ordering of the values in 'series'.

const Series& lookup(LPCTSTR name) const;
// modifies: this
// effects:  Throws NotFoundException if 'name' is not in this. Else
// returns the series mapped to 'name'.
// requires: The destructor of the returned series is never explicitly called.

const Series& operator[](LPCTSTR name) const;
// modifies: this
// effects:  The same as this->lookup(name) on either side of the '==' operator.

void add(const VariableEnvironment& env);
// modifies: this
// effects:  Throws InvalidEnvironmentException if this has names and the
// set of names in 'env' does not equal it.
// Otherwise, appends 'env' to the end of the ordered set of
// variable environments in this. It also appends to the end of
// each series in this its value according to the bindings in 'env'.

const VariableEnvironment& get(int index) const;
// modifies: this
// effects:  Throws OutOfBoundsException if 'index' < 1 or 'index' >
// this->length(). Otherwise, the 'i'th variable environment
// in the ordered set of variable environments is returned.
// requires: The destructor of the returned environment is never explicitly called.

const VariableEnvironment& operator[](int index) const;
// modifies: this
// effects:  The same as this->get(index) on either side of the '==' operator.

int length() const;
// effects:  Returns the length of the series in this.

int cardinality() const;
// effects:  Returns the number of names in this.

const History& operator=(const History&origHistory);
// modifies: this
// effects:  Sets this to be an independent copy of 'origHistory'.

Figure B-15: The second part of the History class specification
History operator+ (const History& rhsHistory) const;
  // effects:  Throws DuplicateNameException if there is any name in 'rhsHistory'
  // that is also in this.
  // Throws InvalidHistoryException if this.length() != rhsHistory.length().
  // Otherwise, returns a new History that combines the variable name
  // bindings and indexed variable environments of this with those of
  // 'rhsHistory'.

Enumeration<LPCTSTR>* names() const;
  // effects:  Returns a generator that will produce each name stored in this
  // exactly once, in arbitrary order.
  // requires: this not be modified while the returned generator is in use.

virtual CString toString() const;
  // effects:  Returns the string representation of this.

virtual BOOL repOk() const;
  // effects:  Returns True if this preserves the representation invariant.
  // False, otherwise.

private:
  BOOL validEnvironment(const VariableEnvironment& env) const;
  // effects:  Returns true if the set of names in 'env' is the same as the set
  // of names in this->names(), false otherwise.

  BOOL validSeries(const Series& series) const;
  // effects:  Returns true if series.length() = this->length(), false otherwise.

void updateSeries(int index, const HistoryEnvironment& env);
  // requires:  1 ≤ 'index' ≤ this->length() + 1, the set of names in 'env'
  // is same as the set of names in this;
  // modifies:  this
  // effects:  Adds to each series in this at 'index' the
  // appropriate value stored in 'env'[name of series].

void updateEnvironments(LPCTSTR name, const HistorySeries& series);
  // requires:  'name' does not exists in this, series->length() = this->length()
  // modifies:  this
  // effects:  Binds to 'name' in each environment in this the
  // appropriate value stored in 'series'[index of environment].

}; // end History

Figure B-16: The rest of the History class specification
class Function : public RefCount
{
    // overview: A function applies an environment of parameters to
    // a transformation of a history into another history,
    // i.e., a function is F(H_i,P) = H_o, where H_i, H_o
    // are histories and P is a parameter environment.

protected:
    // The representation is:
    GCPointer< History > theHistory;  // the input history.
    GCPointer< ParameterEnvironment > theParams;  // the applied parameter environment.
    GCPointer< History > outHistory;  // the output history.
    CList<CString, LPCTSTR> *varNames;  // the set of variable names in this.
    CList<CString, LPCTSTR> *paramNames;  // the set of parameter names in this.
    BOOL valid_output;  // flag indicating validity of outHistory.

    // The abstraction function is:
    AF(c) = { F(H_i,P) = H_o | c->valid_output == TRUE, H_i = c->theHistory,
             P = c->theParams, H_o = c->outHistory } U
             { F(H_i,P) | c->valid_output == FALSE, H_i = c->theHistory, P = c->theParams }

    // The representation invariant is:
    I(c) =
    //   1.) c->theHistory != NULL.
    //   2.) c->theParams != NULL.
    //   3.) c->outHistory != NULL.
    //   4.) c->varNames != NULL.
    //   5.) c->paramNames != NULL.
    //   6.) each name in c->varNames has a bound series in c->theHistory.
    //   7.) each name in c->paramNames has a bound value in c->theParams.
    //   8.) there is no name in c->varNames that has a bound value in c->theParams.
    //   9.) there is no name in c->paramNames that has a bound series in c->theHistory.
    //  10.) each name in c->varNames is unique.
    //  11.) each name in c->paramNames is unique.
    //  12.) if c->valid_output == TRUE, no name in c->outHistory has a bound series
            in c->theHistory or a bound value in c->theParams
    //  13.) if c->valid_output == TRUE, c->outHistory is the correct evaluation
            of this function, given the current input history and applied parameter
            environment.
    //  14.) if c->valid_output == TRUE, c->outHistory->length() == c->theHistory->length().

protected:
    Function();
    // requires: this is a derived subclass.
    // modifies: this
    // effects: Instantiates this as an instance of the Function class.
    //          The output history of this is valid.

Figure B-17: The first part of the Function class specification
Function(const History& his, const ParameterEnvironment& env);
    // requires: this is a derived subclass.
    // modifies: this
    // effects: Throws InvalidHistoryException if 'his' does not bind
    // a series to each variable name in this or if 'his' has a name
    // that has a bound value in the applied parameter environment of this.
    // Throws InvalidEnvironmentException if 'env' does not bind
    // a value to each parameter name in this or if 'env' has a name
    // that has a bound series in the input history of this.
    // Otherwise, instantiates this as an instance of the Function class
    // that inputs history 'his' and applies parameter environment 'env',
    // the output history of this is made invalid.

Function(const Function& origFunction);
    // requires: this is a derived subclass and 'origFunction' is an instance of that subclass.
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

public:
    virtual ~Function();
        // modifies: this
        // effects: Removes this from memory.

void input(const History& his);
        // modifies: this
        // effects: Throws InvalidHistoryException if 'his' does not bind
        // a series to each variable name in this or if 'his' has a name
        // that has a bound value in the applied parameter environment of this.
        // Otherwise, replaces the current input history of this with
        // 'his' and the output history of this is made invalid.

void apply(const ParameterEnvironment& env);
        // modifies: this
        // effects: Throws InvalidEnvironmentException if 'env' does not bind
        // a value to each parameter name in this or if 'env' has a name
        // that has a bound series in the input history of this.
        // Otherwise, replaces the currently applied parameter environment
        // of this with 'env' and the output history of this is made invalid.

const History& history() const;
    // modifies: this
    // effects: Returns the history environment that this currently inputs.
    // The output history of this is made invalid.

const ParameterEnvironment& parameters() const;
    // modifies: this
    // effects: Returns the parameter environment that this currently applies.
    // The output history of this is made invalid.

History output() const;
    // effects: Throws InvalidHistoryException if the output history of this
    // is invalid. Otherwise, returns the a copy of the output history
    // of this.

Figure B-18: The second part of the Function class specification
BOOL valid() const;
  // effects: Returns True if the output history of this is valid.
  //         False, otherwise.

virtual void refresh();
  // modifies: this
  // effects: Evaluates this assuming the currently set input history and
  //           applied environment. The resulting series are written to the
  //           output history and the output history is made valid.

const Function& operator=(const Function& origFunction);
  // modifies: this
  // effects: Sets this to be an independent copy of 'origFunction'.

CString toString() const;
  // effects: Returns the string representation of this.

virtual BOOL repOk() const;
  // effects: Returns True if this preserves the representation invariant.
  //          False, otherwise.

private:
  BOOL validEnvironment(const ParameterEnvironment& env) const;
    // effects: Returns True if 'env' can be validly applied by this, where
    //          the criteria is given by the representation invariant above.
    //          False, otherwise.

  BOOL validHistory(const History& his) const;
    // effects: Returns True if 'his' can be validly input by this, where
    //          the criteria is given by the representation invariant above.
    //          False, otherwise.

}; // end Function

Figure B-19: The rest of the Function class specification
class EpsilonFunction : public Function
{
    // overview: An epsilon function evaluates the following equation:
    //     epsilon = y - b0 - b1*x1 - b2*x2 ... - bk*xk
    //     where k is an integer ≥ 0.

protected:
    // The representation is:
    int m.k; // the number of independent variables in this.

    // The abstraction function is:
    // AF(c) = Function::AF(c)

    // The representation invariant is:
    // I(c) =
    // 1.) Function::I(c)
    // 2.) c->m.k ≥ 0
    // 3.) if k = c->m.k, then names in c->varNames are
    //     y, x1, ..., xk.
    // 4.) if k = c->m.k, then names in c->paramNames are
    //     b0, b1, ..., bk.
    // 5.) if c->validOutput == TRUE, then a name in c->outHistory is
    //     epsilon.

public:
    EpsilonFunction(int k = 0);
    // modifies: this
    // effects: Validates EpsilonTermStructureException if k < 0.
    //          Otherwise, instantiates this as an instance of the EpsilonFunction class
    //          with k variable names x1, x2, ..., xk and k + 1 parameter names
    //          b0, b1, ..., bk.
    //          The output history of this is valid.

    EpsilonFunction(const EpsilonFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

    virtual ~EpsilonFunction();
    // modifies: this
    // effects: Removes this from memory.

    int k() const;
    // effects: Returns the number of independent variables in this.

    void refresh();
    // modifies: this
    // effects: Same as inherited specification for Function::refresh(), except that
    //          the resulting series is bound to epsilon in the output history of this.
    //          The output history is made valid.

    const EpsilonFunction& operator= (const EpsilonFunction &origFunction);
    // modifies: this
    // effects: Sets this to be an independent copy of 'origFunction'.

}; // end EpsilonFunction

Figure B-20: The EpsilonFunction class specification
class EpsilonSlopeFunction : public EpsilonFunction
{
    // overview: An epsilon slope function evaluates the following set of equations:
    // 
    // epsilon = y - b0 - b1*x1 - b2*x2 ... - bk*xk
    // de/db0 = -1
    // de/db1 = -x1
    // ...
    // de/dbk = -xk
    // de/dalpha0 = 0
    // ...
    // de/dalphap = 0
    // de/dbeta1 = 0
    // ...
    // de/dbetaq = 0
    // where k is an integer ≥ 0, p is an integer ≥ 1, and
    // q is an integer ≥ 0.

protected:
    // The representation is:
    int m_p; // the number of ARCH terms in this.
    int m_q; // the number of GARCH terms in this.

    // The abstraction function is:
    // AF(c) = EpsilonFunction::AF(c)

    // The representation invariant is:
    // I(c) =
    // 1.) EpsilonFunction::I(c)
    // 2.) c->m.p ≥ 1
    // 3.) c->m.q ≥ 0
    // 4.) If c->valid.output == TRUE, k = c->m.k, p = c->m.p, and q = c->m.q,
    // then names in c->outHistory are
    // de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalphap, de/dbeta1,
    // ..., de/dalphap, de/dbeta1, ..., de/dbetaq.

public:
    EpsilonSlopeFunction(int k = 0, int p = 1, int q = 0);
    // modifies: this
    // effects: Same as inherited EpsilonFunction::EpsilonFunction(), except
    // also throws InvalidGARCHTermStructureException if p < 1 or q < 0.
    // Otherwise, instantiates this as an instance of the EpsilonSlopeFunction
    // class.

    EpsilonSlopeFunction(const EpsilonSlopeFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

    virtual ~EpsilonSlopeFunction();
    // modifies: this
    // effects: Removes this from memory.

Figure B-21: The first part of the EpsilonSlopeFunction class specification.
void refresh();
   // modifies: this
   // effects: Same as inherited specification for EpsilonFunction::refresh(), except that
   // additional series are bound to de/db0, de/db1, ..., de/dbk,
   // de/dalpha0, de/dalphal, ..., de/dalphap, de/dbeta1, ..., de/dbetaq
   // in the output history of this. The output history is made valid.

const EpsilonSlopeFunction& operator= (const EpsilonSlopeFunction &origFunction);
   // modifies: this
   // effects: Sets this to be an independent copy of 'origFunction'.

}; // end EpsilonSlopeFunction

Figure B-22: The second part of the EpsilonSlopeFunction class specification
class GarchFunction : public Function
{
    // overview: An epsilon function evaluates the following equation:
    // variance = alpha0 + alpha1*epsilon(-1)^2 + ... + alphap*epsilon(-p)^2
    // + beta1*variance(-1)^2 + ... + betaq*variance(-1)^2
    // where p is an integer ≥ 1, and q is an integer ≥ 0.

protected:
    // The representation is:
    int m_p; // the number of ARCH terms in this.
    int m_q; // the number of GARCH terms in this.

    // The abstraction function is:
    // AF(c) = Function::AF(c)

    // The representation invariant is:
    // I(c) =
    // 1.) Function::I(c)
    // 2.) c->m_p ≥ 1
    // 3.) c->m_q ≥ 0
    // 4.) a name in c->varNames is epsilon.
    // 5.) if p = c->m_p and q = c->m_q, then names in c->paramNames are
    //     alpha0, alpha1, ..., alphap, betal, ..., betaq.
    // 6.) if c->valid_output == TRUE, then a name in c->outHistory is
    //     variance and the series bound to variance does not have a
    //     zero, negative, or infinite value.

public:
    GarchFunction(int p = 1, int q = 0);
    // modifies: this
    // effects: Throws InvalidGARCHTermStructureException if p < 1 or q < 0.
    //          Otherwise, instantiates this as an instance of the GarchFunction class
    //          with the variable names epsilon and p + q + 1 parameter names
    //          alpha0, alpha1, ..., alphap, betal, ..., betaq.
    //          The output history of this is valid.

    GarchFunction(const GarchFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

    virtual ~GarchFunction();
    // modifies: this
    // effects: Removes this from memory.

    int p() const;
    // effects: Returns the number of ARCH terms in this.

    int q() const;
    // effects: Returns the number of GARCH terms in this.

Figure B-23: The first part of the GarchFunction class specification
void refresh(void);

// modifies: this
// effects: Same as inherited specification for Function::refresh(), except that
// a NonPositiveVarianceException is thrown if the series calculated for
// variance ever has a negative or zero value at an index. Throws
// InfiniteVarianceException if the series ever has an infinite value
// at an index. The output history is made invalid in either case.
// Otherwise, the resulting series is bound to variance in the output
// history of this and the output history is made valid.

const GarchFunction& operator= (const GarchFunction &origFunction);

// modifies: this
// effects: Sets this to be an independent copy of 'origFunction'.

private:

double alphaSum() const;
// effects: Return the sum of values bound to parameters alpha1, alpha2, ..., alpha(m.p)

double betaSum() const;
// effects: Return the sum of values bound to parameters beta1, beta2, ..., beta(m.q)
}; // end GarchFunction

Figure B-24: The second part of the GarchFunction class specification
class GarchSlopeFunction : public GarchFunction
{
    // overview: A garch slope function evaluates the following set of equations:
    //
    // \[ \begin{align*}
    //     \text{variance} & = \alpha_0 + \alpha_1 \epsilon(-1)^2 + \ldots + \alpha_p \epsilon(-p)^2 \\
    //                     & + \beta_1 \text{variance}(-1) + \ldots + \beta_q \text{variance}(-1) \\
    //     \frac{d\text{v}}{db_0} & = 2\alpha_1 \frac{d\epsilon(-1)}{db_0} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_0} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_0} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_0} \\
    //     \frac{d\text{v}}{db_1} & = 2\alpha_1 \frac{d\epsilon(-1)}{db_1} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_1} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_1} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_1} \\
    //     \frac{d\text{v}}{db_k} & = 2\alpha_1 \frac{d\epsilon(-1)}{db_k} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_k} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_k} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_k} \\
    //     \frac{d\text{v}}{d\alpha_0} & = 1 + 2\alpha_1 \frac{d\epsilon(-1)}{d\alpha_0} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{d\alpha_0} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{d\alpha_0} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{d\alpha_0} \\
    //     \frac{d\text{v}}{d\alpha_1} & = \epsilon(-p)^2 + 2\alpha_1 \frac{d\epsilon(-1)}{d\alpha_1} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{d\alpha_1} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{d\alpha_1} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{d\alpha_1} \\
    //     \frac{d\text{v}}{db_0} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_0} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_0} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_0} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_0} \\
    //     \frac{d\text{v}}{db_1} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_1} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_1} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_1} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_1} \\
    //     \frac{d\text{v}}{db_k} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_k} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_k} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_k} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_k} \\
    //     \frac{d\text{v}}{d\alpha_0} & = 1 + 2\alpha_1 \frac{d\epsilon(-1)}{d\alpha_0} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{d\alpha_0} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{d\alpha_0} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{d\alpha_0} \\
    //     \frac{d\text{v}}{d\alpha_1} & = \epsilon(-p)^2 + 2\alpha_1 \frac{d\epsilon(-1)}{d\alpha_1} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{d\alpha_1} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{d\alpha_1} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{d\alpha_1} \\
    //     \frac{d\text{v}}{db_0} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_0} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_0} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_0} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_0} \\
    //     \frac{d\text{v}}{db_1} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_1} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_1} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_1} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_1} \\
    //     \frac{d\text{v}}{db_k} & = \text{variance}(-1) + 2\alpha_1 \frac{d\epsilon(-1)}{db_k} + \ldots + 2\alpha_p \frac{d\epsilon(-p)}{db_k} \\
    //                     & + \beta_1 \frac{d\text{v}(\text{variance})(-1)}{db_k} + \ldots + \beta_q \frac{d\text{v}(\text{variance})(-q)}{db_k} \\
    //     \end{align*} \]
    // where k is an integer \( \geq 0 \), p is an integer \( \geq 1 \), and q is an integer \( \geq 0 \).
    
    protected:
    // The representation is:
    int m_k; // the number of independent variables in this
    
    // The abstraction function is:
    // \[ \text{AF}(c) = \text{GarchFunction::AF}(c) \]
    
    // The representation invariant is:
    // If(c) = 
    // 1.) GarchFunction::I(c)
    // 2.) c -> m_k \( \geq 0 \)
    // 3.) if k = c -> m_k, p = c -> m_p, q = c -> m_q, then names in c -> varNames are
    //     de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalpha1, ...
    //     de/dalphap, de/dbetal, ...
    //     de/dbetaq.
    // 4.) If c -> valid_output == TRUE, k = c -> m_k, p = c -> m_p, and q = c -> m_q,
    //     then names in c -> outHistory are
    //     dv/b0, dv/b1, ..., dv/dbk, dv/dalpha0, dv/dalpha1, ...
    //     dv/dalphap, dv/dbetal, ...
    // public:
    GarchSlopeFunction(int k = 0, int p = 1, int q = 0);
    // modifies: this
    // effects: Inherits GarchFunction::GarchFunction(). Extends constructor,
    //         throwing InvalidEpsilonTermStructureException if k < 0.
    //         Otherwise, instantiates this as an instance of the GarchSlopeFunction class
    //         with the additional variable names de/b0, de/b1, ..., de/bk,
    //         de/dalpha0, de/dalpha1, ..., de/dalphap, de/dbetal, ..., de/dbetaq.
    //         The output history of this is valid.

Figure B-25: The first part of the GarchSlopeFunction class specification
GarchSlopeFunction(const GarchSlopeFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

virtual GarchSlopeFunction();
    // modifies: this
    // effects: Removes this from memory.

int k() const;
    // effects: Returns the number of independent variables in this.

const GarchSlopeFunction& operator=(const GarchSlopeFunction &origFunction);
    // modifies: this
    // effects: Sets this to be an independent copy of 'origFunction'.

void refresh();
    // modifies: this
    // effects: Same as inherited specification for GarchFunction::refresh(), except that
    // additional series are bound to dv/db0, dv/db1, ..., dv/dbk,
    // dv/dalpha0, dv/dalpha1, ..., dv/dalphap, dv/dbeta1, ..., dv/dbetaq
    // The output history is made valid.

}; // end GarchSlopeFunction

Figure B-26: The second part of the GarchSlopeFunction class specification
class LikelihoodFunction : public Function
{
    // overview: A likelihood function evaluates the following equation:
    // likelihood = −1/2 * ln variance − epsilon^2 / (2 * variance)

    // The abstraction function is:
    // AF(c) = Function::AF(c)

    // The representation invariant is:
    // I(c) =
    // 1.) Function::I(c)
    // 2.) epsilon and variance are names in c→varNames.
    // 3.) The series mapped to variance in c→theHistory does not have
    // any zero, negative, or infinite values.
    // 4.) if c→validOutput == TRUE, then likelihood is a name in c→outHistory.

public:
    LikelihoodFunction();
    // modifies: this
    // effects: Instantiates this as an instance of the LikelihoodFunction
    // class with the variable names epsilon and variance.
    // The output history of this is valid.

    LikelihoodFunction(const LikelihoodFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.

    virtual ~LikelihoodFunction();
    // modifies: this
    // effects: Removes this from memory.

    void refresh();
    // modifies: this
    // effects: Same as inherited specification for Function::refresh(), except that
    // a NonPositiveVarianceException is thrown if the series calculated for
    // variance ever has a negative or zero value at an index. Throws
    // InfiniteVarianceException if the series ever has an infinite value
    // at an index.
    // Otherwise, a series is bound to likelihood in the output history
    // of this. The output history is made valid.

    const LikelihoodFunction& operator= (const LikelihoodFunction &origFunction);
    // modifies: this
    // effects: Sets this to be an independent copy of 'origFunction'.

}; // end LikelihoodFunction

Figure B-27: The LikelihoodFunction class specification
class LikelihoodSlopeFunction : public LikelihoodFunction
{
    // overview: A garch slope function evaluates the following set of equations
    // likelihood = \(-\frac{1}{2} \ln \text{variance} - \epsilon^2 / (2 \times \text{variance})\)
    // dl/db0 = \(\eta \times de/db0 + \zeta \times dv/db0\)
    // dl/db1 = \(\eta \times de/db1 + \zeta \times dv/db1\)
    // ...
    // dl/d1 = \(\eta \times de/d1 + \zeta \times dv/d1\)
    // where \(k\) is an integer \(\geq 0\), \(p\) is an integer \(\geq 1\), and \(q\) is an integer \(\geq 0\)
    // and:
    // \(\eta = -\epsilon / \text{variance}\)
    // \(\zeta = 1 / (2 \times \text{variance}) \times (\epsilon^2 / \text{variance} - 1)\)

protected:
    // The representation is:
    int m_k; // the number of independent variables in this.
    int m_p; // the number of ARCH terms in this.
    int m_q; // the number of GARCH terms in this.

    // The abstraction function is:
    // \(AF(c) = \text{LikelihoodFunction::AF(c)}\)

    // The representation invariant is:
    // I(c) =
    // 1.) LikelihoodFunction::I(c)
    // 2.) c->m_k \geq 0
    // 3.) c->m_p \geq 1
    // 4.) c->m_q \geq 0
    // 5.) if k = c->m_k, p = c->m_p, q = c->m_q, then names in c->varNames are
    // de/b0, de/b1, ..., de/dbk, de/dalpha0, de/dalpha1, ...
    // de/dalphap, de/dbeta0, de/dbeta1, ...
    // 6.) if c->valid_output == TRUE, then names in c->outHistory are
    // dl/b0, dl/b1, ..., dl/dbk, dl/dalpha0, dl/dalpha1, ...
    // ...

Figure B-28: The first part of the LikelihoodSlopeFunction class specification
public:
    LikelihoodSlopeFunction(int k = 0, int p = 1, int q = 0);
    // modifies: this
    // effects: Throws InvalidEpsilonTermStructureException if k < 0.
    //          Throws InvalidGARCHTermStructureException if p < 1 or q < 0.
    // Otherwise, instantiates this as an instance of the
    // LikelihoodSlopeFunction class with the additional variable
    // names de/b0, de/b1, ..., de/bk, de/dalpha0,
    // de/dalphal, ..., de/dalphap, de/dbetal, ...,
    // de/dbetaq,
    // dv/b0, dv/b1, ..., dv/dbk, dv/dalpha0, dv/dalpha1,
    // ..., dv/dalphap, dv/dbeta1, ..., dv/dbetaq.
    // The output history of this is valid.
    LikelihoodSlopeFunction(const LikelihoodSlopeFunction& origFunction);
    // modifies: this
    // effects: Instantiates this as a copy of 'origFunction'.
    virtual ~LikelihoodSlopeFunction();
    // modifies: this
    // effects: Removes this from memory.
    int k() const;
    // effects: Returns the number of independent variables in this.
    int p() const;
    // effects: Returns the number of ARCH terms in this.
    int q() const;
    // effects: Returns the number of GARCH terms in this.
    void refresh();
    // modifies: this
    // effects: Same as inherited specification for LikelihoodFunction::refresh(), except that
    // additional series are bound to dl/db0, dl/db1, ..., dl/dbk,
    // dl/dalpha0, dl/dalpha1, ..., dl/dalphap, dl/dbeta1, ..., dl/dbetaq.
    // The output history is made valid.
    const LikelihoodSlopeFunction& operator=(const LikelihoodSlopeFunction &origFunction);
    // modifies: this
    // effects: Sets this to be an independent copy of 'origFunction'.
}; // end LikelihoodSlopeFunction

Figure B-29: The second part of the LikelihoodSlopeFunction class specification
Bibliography


