

**A COMPUTATIONAL APPROACH TO AFFINE MODELS OF THE
TERM STRUCTURE**

By

Barton Baker

Submitted to the

Faculty of the College of Arts and Sciences

of American University

in Partial Fulfillment of

the Requirements for the Degree

of Doctor of Philosophy

In

Economics

Chair:



Professor Robin L. Lumsdaine



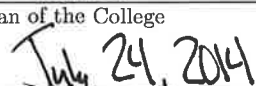
Professor Alan G. Isaac



Professor Colleen M. Callahan



Dean of the College



Date

© COPYRIGHT

by

Barton Baker

2014

ALL RIGHTS RESERVED

A COMPUTATIONAL APPROACH TO AFFINE MODELS OF THE TERM STRUCTURE

by
Barton Baker

ABSTRACT

This dissertation makes contributions to the term structure modeling literature by examining the role of observed macroeconomic data in the pricing kernel and providing a single computational framework for building and estimating a range of affine term structure models. Chapter 2 attempts to replicate and extend the model of Bernanke et al. (2005), finding that proxies for uncertainty, particularly for practitioner disagreement and stock volatility, lower the pricing error of models estimated only with observed macroeconomic information. The term premia generated by models including the proxies produce term premia that are higher during recessions, suggesting that these proxies for uncertainty represent information that is of particular value to bond market agents during crisis periods. Chapter 3 finds that a real-time data specified pricing kernel produces lower average pricing errors compared to analogous models estimated using final release data. Comparisons between final release and real-time data driven models are performed by estimating observed factor models with two, three, and four factors. The real-time data driven models generate more volatile term premia for shorter maturity yields, a result not found in final data driven models. This suggests that the use of real-time over final release data has implications for model performance and term premia estimation. Chapter 4 presents a unified computational framework written in the Python programming language for estimating discrete-time affine term structure models, supporting the major canonical approaches. The chapter also documents the use of the package, the solution methods and approaches directly supported, and development issues encountered when writing C-language extensions for Python packages. The package gives researchers a flexible interface that admits a wide variety of affine term structure specifications.

ACKNOWLEDGEMENTS

I would like to thank my dissertation committee chair, Professor Robin L. Lumsdaine, for her invaluable guidance, comments, and encouragement in the completion of this dissertation. I have learned an immense amount, not only about the field of affine term structure modeling, but also about contributing to the field of economics in general from her. I would also like to thank my other committee members, Professor Alan G. Isaac and Professor Colleen Callahan, for their support and review at key times during this process.

I would also like to thank the American University Department of Economics for an education that has proved immensely valuable in my formation as a researcher. Of particular value to me was the opportunity to study many different fields within economics.

I would also like to thank the NumPy development community. When issues were encountered in development of the package, they were extremely knowledgeable and helpful.

Most of all, I would like to give special thanks to my wife, Jenny, for all of her support during the research and writing of this dissertation. The completion of this dissertation would not have been possible without her constant encouragement.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	viii
CHAPTER	
1. INTRODUCTION	1
1.1 Approach	3
1.2 Contributions and Structure	4
2. AN EXTENSION AND REPLICATION OF BERNANKE ET AL. (2005)	6
2.1 Data	8
2.2 Replication	15
2.3 Extension into the Great Recession of 2007-2009	25
2.4 Conclusion	38
3. REAL-TIME DATA AND INFORMING AFFINE MODELS OF THE TERM STRUCTURE	40
3.1 Model	43
3.2 Data	49
3.2.1 Yields	54
3.3 Results	56
3.4 Conclusion	69
4. AN INTRODUCTION TO <i>AFFINE</i> , A PYTHON SOLVER CLASS FOR AFFINE MODELS OF THE TERM STRUCTURE	71
4.1 A Python Framework for Affine Models of the Term Structure	73
4.1.1 Why Python?	77

4.1.2	Package Logic	78
4.2	Assumptions of the Package	82
4.2.1	Data/Model Assumptions	82
4.2.2	Solution Assumptions	84
4.3	API	86
4.3.1	Parameter Specification by Masked Arrays	88
4.3.2	Estimation	92
4.4	Development	101
4.4.1	Testing	111
4.4.2	Issues	114
4.5	Building Models	115
4.5.1	Method of Bernanke et al. (2005)	116
4.5.2	Method of Ang and Piazzesi (2003)	118
4.5.3	Method of Orphanides and Wei (2012)	122
4.6	Conclusion	125
5.	CONCLUSION	127

APPENDIX

A.	Data for Chapter 2	130
B.	Data for Chapter 3	131
C.	Additional figures and table for Chapter 2	133
D.	Source Code for affine	136
E.	Sample scripts for executing models and viewing results	184

REFERENCES	202
----------------------	-----

LIST OF TABLES

Table	Page
2.1 Descriptive Statistics of Difference in Percentage between Constant Maturity Government Bond Yields and Fama-Bliss Implied Zero-coupon Bond Yields for One, Three, and Five year Maturities 1982-2012.	11
2.2 Page 47 from Bernanke et al. (2005)	19
2.3 Standard Deviation of Pricing Error in Basis Points.	21
2.4 Standard Deviation of Pricing Error by Parameter Difference Convergence Criterion.	23
2.5 Difference in Standard Deviation of Pricing Error between Model with and without Eurodollar Factor.	24
2.6 Root Mean Squared Pricing Error in Basis Points	25
2.7 Model Classifications	29
2.8 RMSE for Estimated Models.	30
2.9 Model Comparisons for T-Test.	30
2.10 RMSE for Model Using Fama-Bliss Zero-coupon Bonds.	32
2.11 Mean Five Year Term Premium by Date Range and Model.	38
3.1 Quarterly Releases of Real GDP Growth and Civilian Unemployment for Q3 1996	45
3.2 Descriptive Statistics for Output Growth and Inflation as Measured by the Median Survey of Professional Forecasters within Quarter Statistic, the Greenbook Current Quarter Statistic, and the Final Release Statistic.	52
3.3 Sample of Real-time Data Set for Macroeconomists Real GNP	53
3.4 Descriptive Statistics of Real-time and Final Data, Quarterly Data, 1969-2012.	53
3.5 RMSE for Models using Final (F) and Real-time (RT) Data.	59
3.6 AR Models of Pricing Errors Taken from the Four Factor Models.	63

4.1	Algebraic Model Parameters Mapped to Affine Class Instantiation Arguments . .	88
4.2	Profiling Output of Pure Python Solve Function.	107
4.3	Profiling Output of Hybrid Python/C Solve Function.	107
4.4	Affine Term Structure Modeling Papers Matched with Degree of Support from Pack- age	116
C.1	Maximum Five Year Term Premium by Date Range and Model.	133
C.2	Minimum Five Year Term Premium by Date Range and Model.	133

LIST OF FIGURES

Figure	Page
2.1 One, Three, and Five-year Yield Plots of Constant Maturity Government Bonds vs. Fama-Bliss Implied Zero-coupon Bonds	10
2.2 Blue Chip Financial Forecasts Next Year Output Growth Disagreement.	13
2.3 Distribution of Disagreement for Next Year by Month.	14
2.4 Distribution of Disagreement for Current Year by Month.	15
2.5 Page 46 from Bernanke et al. (2005)	18
2.6 Author's Estimation Results Showing Actual, Predicted, and Risk-neutral Percentage Yields for 2-year Treasury Constant Maturity.	20
2.7 Author's Estimation Results Showing Actual, Predicted, and Risk-neutral Percentage Yields for 10-year Treasury Constant Maturity.	20
2.8 Blue Chip Disagreement and VIX, 1990-2012	28
2.9 Plots of Pricing Error for Five Year Yield for Select Models.	34
2.10 Plots of Time-varying Term Premium for Five Year Yield for Select Models.	37
3.1 SPF and Greenbook Output Growth Statistics.	51
3.2 SPF, Greenbook, and Final Output Growth Statistics.	51
3.3 Residuals of Univariate Regression of Final Output on Real-time Output. NBER (2013) recessions.	54
3.4 Time Series of F-statistics Used to Test for Structural Breaks in the Observed One, Two, Three, Four, and Five Year Yields.	56
3.5 Time Series of F-statistics Used to Test for Structural Breaks in the Final Values of Output Growth, Inflation, Residential Investment, and Unemployment.	57
3.6 Time Series of F-statistics Used to Test for Structural Breaks in the Real-time Values of Output growth, Inflation, Residential Investment, and Unemployment.	58

3.7	Plots of Residuals for Final (4) and Real-time (4) Models by Maturity.	62
3.8	Plots of Implied Term Premium for Final (4) and Real-time (4) Models by Maturity on Left and Right Hand Side, Respectively.	65
3.9	Autocorrelation Plots of Implied Time-varying Term Premium for Final (4) and Real-time (4) Models by Maturity on the Left and Right Hand Side, Respectively. . .	68
4.1	Package Logic	80
4.2	Package Logic (continued)	81
4.3	Graphical Output Profiling Pure-Python Solve Function.	108
4.4	Graphical Output Profiling Hybrid Python/C Solve Function.	109
C.1	Plots of Difference between Yields on One, Three, and Five-year Constant Maturity Government Bond Yields and Fama-Bliss Implied Zero Coupon Bond Yields.	134
C.2	Pricing Error Across Estimated Models for One and Five Year Maturity.	135

CHAPTER 1

INTRODUCTION

The relationship between macroeconomic fluctuations and the yields on government bonds has long been a subject of study. Macroeconomic conditions such as output, inflation, and investment affect the market in at least two ways. First, the macroeconomic conditions will partially determine the market environment under which a single bond-market agent is making decisions. Second, the publication of macroeconomic indicators communicates to agents their own financial position relative to the rest of the market and the conditions of the market as a whole. Recognizing specifically how macroeconomic conditions influence government bond markets should be an important component of any term structure modeling approach.

Monetary policy also affects the term structure of government bonds: at shorter maturities through the federal funds rate and open market operations, and at longer maturities through large scale asset purchases such as quantitative easing (Krishnamurthy and Vissing-Jorgensen, 2011) and formal management of expectations of future federal funds rate targets and inflation (Bernanke et al., 2005). The federal funds rate serves as a benchmark not only for bond markets but for many other financial markets. Monetary policy measures are particularly valuable for term structure modeling because the federal funds rate, the shortest maturity yield in the term structure, is the primary instrument of the monetary authority.

In addition to the current macroeconomic condition and monetary policy environment, expectations of both over different horizons will inevitably have an impact on the perceived risk of holding government bonds. With higher perceived macroeconomic risk over the maturity of the bond, bond buying agents will require higher expected yields in order to be compensated for that risk. Expectations of future monetary policy also may affect the term structure through the Expectation Hypothesis, where long term rates are the product of expectations of future short term rates. Agents will also integrate the expectations of how the monetary authority could react to the

economic conditions at that time. In particular with longer maturity bonds, expectations of future macroeconomic and monetary conditions may play a prominent role in bond pricing outcomes.

Government bond-market participants use information related to both current and expected macroeconomic conditions and monetary policy to inform their market behavior. As a result, government bonds offer a link between macroeconomic policy and financial markets and decomposing what specifically drives the yields on these bonds can help in determining how macroeconomic policy may alter the yield curve. A term structure modeling framework should utilize macroeconomic and monetary policy information in a data generating process that influences the yields on government bonds.

Affine term structure models offer a framework through which the information driving government bond markets can be linked to government bond yields. These models are a convenient tool for both modeling a process governing agents' beliefs of future economic conditions and using this process to predict yields all along the yield curve. Macroeconomic conditions, the data generating process for these conditions, and other factors are linked to a spread of yields through the assumption that a single pricing kernel can be used to explain all of the yields¹. It is assumed that the macroeconomic variables and other factors included in the kernel encapsulate the primary information driving bond market pricing decisions. It is often necessary to add unobserved latent factors to the set of observed factors or replace the observed factors completely in order to capture all relevant moments of yields over time. Models with multiple latent factors were introduced in Duffie and Kan (1996). By defining the data generating process governing this pricing kernel, the yields on bonds all along the yield curve can be decomposed into a predicted component and a risky component. After the yields have been decomposed in this manner, a time-varying estimate of the term premium is obtained, which is the additional yield required by agents who have tied up liquidity in the bond over the maturity of the bond.

This term premium estimate can be used to demonstrate how perceived risk as reflected in bond yields responds to specific historical events. Term premia have been shown to react to macroeconomic expansions and recessions (Rudebusch et al., 2007), to specific monetary policy announcements and policy changes (Kim and Wright, 2005), and to changes in expected and unexpected inflation (Piazzesi and Schneider (2007) and Wright (2011)). In many of these cases, latent factors are used in the pricing kernel to maximize the fit of the model and generate the

¹The pricing kernel is defined in Equation 2.2.1.

time-varying term premia. In addition to examining responses to events, these models can also be used to measure what information is useful for generating a high performing pricing kernel and what information generates changes in the time-varying term premium.

Observed information added to the pricing kernel can change the performance of the model (Bernanke et al., 2005) and lead to changes in the moments of the time-varying term premia. The literature around how modifications and additions to the observed information included in the pricing kernel is less well-developed. This dissertation contributes to the observed factor approach, showing how specific observed factors included in the pricing kernel can alter the performance of the model and can lead to different measures of the term premia.

1.1 Approach

The trend in the affine term structure model literature over the past fifteen years has been to supplement or supplant observed information driving the bond markets with unobserved latent factors. These latent factors are derived in the estimation process through assumptions about the structure of bond markets and, depending on the calculation of the likelihood, are calculated by assuming that certain yields are priced without error. Dai and Singleton (2002), Ang and Piazzesi (2003), Kim and Orphanides (2005), and Orphanides and Wei (2012) each estimate models that use a combination of observed and unobserved factors to inform bond pricing decisions. Kim and Wright (2005), Diebold et al. (2006), and Rudebusch and Wu (2008) rely purely on unobserved latent factors. The addition of even a single latent factor increases the performance of these models at multiple maturities as measured by the pricing error (the difference between the actual and predicted yield). Adding latent factors is a popular choice when the intent of developing the model is to build a high-performing model and develop an estimate of the time-varying term premium. Even though these latent factors can often be related back to moments of the yield curve, they are not as useful when part of the research effort is to break down the information entering bond market pricing decisions into what information is valuable to agents and how it is valuable. For example, adding even a single latent factor could mask the individual subtleties of different types of observed information included in the pricing kernel.

Rather than maximizing the fit of the model through the addition of latent factors, this dissertation takes an approach of adding and modifying observed macroeconomic factors to the yield curve to gain a better understanding of what drives bond market pricing behavior. These macroeconomic factors can include output, inflation, investment, expected output, and practitioner

forecast disagreement. This is more in line with the approach of Bernanke et al. (2005) and Joslin et al. (2011), where latent factors are avoided to gain a better understanding of what observed information drives government bond markets. While the models estimated in this dissertation do not fit the term structure as closely as models with latent factors, they do reveal important information about how different types of observed information become valuable at different times of the business cycle when pricing the term structure. The first two chapters both investigate the impact that modifications and additions to this observed information set have on model performance and the time-varying term premium, with a latent factor model included in the second chapter for comparison and illustration of the value of the observed information.

1.2 Contributions and Structure

Chapter 2, *An Extension and Replication of Bernanke et al. (2005)*, attempts to replicate the original model of the referenced paper and extend it into the recent financial crisis. The chapter also examines how an affine term structure model driven solely by the observed macroeconomic factors used in Bernanke et al. (2005) could benefit from the addition of observed factors that attempt to capture economic uncertainty, namely, practitioner forecast disagreement and stock market volatility. These additions become especially useful when recessions are included in the observation period and lead to higher estimated term premia during recessions. By pricing uncertainty explicitly, better fitting models with lower pricing error as measured by root-mean-square error are estimated.

Chapter 3, *Real-time Data and Informing Affine Models of the Term Structure*, focuses on accurately reflecting the information used by the bond market to contemporaneously price the yield curve. This refinement of the information set is accomplished through the use of a real-time data-driven process governing agents' beliefs about the macroeconomic information driving bond market decisions. This chapter is inspired by the real-time modeling approach of Orphanides (2001) and Orphanides and Wei (2012), but focuses entirely on the potential role of real-time data in affine term structure models. A real-time process is compared to an affine process governed by final release data to show the advantage of using real-time data through model performance measures and the characteristics of the resulting term premia. A real-time data derived pricing kernel is shown to both perform better and offer a wider variety of time-varying term premia time series across the yield curve. These results suggest that term premia may also be driven by different factors, or changes in weights of factors, at different ends of the yield curve. The potential role of

latent factors in smoothing differences between real-time and final data-driven models is also briefly examined.

Construction and estimation of affine term structure models can be a time consuming process. The transformation of the information entering pricing decisions into the yields of government bonds spread across the yield curve involves the construction of a non-linear model. A closed-form solution for the parameters of the model given a data generating process does not exist, so the model parameters must be estimated using numerical approximation methods coupled with an objective function. In the process of researching the affine term structure model literature, I discovered that there was a dearth of software built explicitly for building and estimating these models. Chapter 4, *An Introduction to affine, a Python Solver Class for Affine Models of the Term Structure*, presents a package written by the author to begin to fill this void and a broad framework through which affine term structure models can be understood. This package represents a unique addition to the field, not only in its ability to solve a broad class of affine models of the term structure, but by also providing a way of understanding different models as permutations of the same structure modified by a selection of parameters. The chapter presents information on how the package can be used, issues encountered during development of the package, and lessons learned on developing computational C language extensions for Python. The package also provides a general approach to building affine models of the term structure that allows models built for specific purposes in other papers to be compared using a single framework, aligning their similarities and pinpointing their differences. It is the intention of the author that this package will lower the costs involved in developing affine models of the term structure and will lead to a wider variety of papers in the field.

CHAPTER 2

AN EXTENSION AND REPLICATION OF BERNANKE ET AL. (2005)

In a 2005 Brookings Paper, Bernanke et al. (2005) (BRS) investigate the effects of alternative monetary policy at a binding zero-lower bound (ZLB) for the federal funds rate. One of the main conclusions of their study is the importance of including policy expectations when pricing zero-coupon yields through an affine model of the term structure¹. Their model uses a collection of observed macroeconomic variables or “factors” modeled using a vector autoregression (VAR) to price zero-coupon bond yields along the term structure. While it is common practice in affine term structure literature to use a combination of observed and unobserved factors to inform the pricing kernel (see Ang and Piazzesi (2003) and Kim and Wright (2005)), BRS are able to price a large amount of the variation in observed yields using information derived only from observed macroeconomic factors. As a specific test of the importance of policy expectations, BRS add an additional macroeconomic measure (year-ahead Eurodollar futures) to the information set entering the model, and they adduce the resulting lowered pricing error as evidence of the importance of policy expectations in bond markets.

BRS’s period of study, 1982 to 2004, lies almost entirely within the period commonly known as the “Great Moderation” (see Stock and Watson (2003)). This period was characterized by low inflation and consistent output growth, where expectations of future economic activity stabilized to a degree not previously seen in American economic history. Because of this stability in both current and expected economic activity, the inclusion of year-ahead Eurodollar futures as an additional factor may have been appropriate, given how predictable the economic environment was during

¹These models are “affine” through the transformation performed to relate the pricing kernel to the observed yields. This transformation allows for mathematical tractability when relating the factors driving the pricing kernel to the observed term structure yields.

this period. In contrast, the period following their observation period has been characterized by economic instability and uncertainty, primarily because of the housing boom and bust and associated stock market crash and the Great Recession. BRS emphasize the ability of their model to fit yields across multiple maturities without the need for unobserved factors, but given the volatility following their original observation period, stability in their chosen macroeconomic factors could have driven lower pricing errors, and not necessarily the ability of those factors to price term structure volatility in diverse circumstances.

This chapter considers BRS's model in the context of the recent financial crisis. To do this, this chapter will attempt to replicate BRS's results, then extend BRS's observation period into 2012, past the "Great Moderation" into the Financial Crisis of 2007-2008 and into the Great Recession and slow recovery of 2009-2012. If BRS's choice of factors are suitable for all periods and not just 1982-2004, then the pricing error should not deteriorate with an observation extension into the modern economic era.

This chapter will be divided into the following 3 sections:

1) Data

I start by addressing the bond yields and macroeconomic factors used in their model. This section will also consider alternatives to their original data. The use of a different yield set will be addressed. It will also address possible issues with using Blue Chip Financial Forecast data, unadjusted, in a time series model.

2) Replication

In this section, I estimate the model using BRS's original factors and yields and use their exact observation period of 1982 to 2004. Time series plots of fitted and risk-neutral yields from this estimation will be shown alongside with BRS's original results. The pricing errors of the estimated yields will also be displayed, with and without the Eurodollar factors, and will also be compared to BRS's original results. There is also a discussion of the importance of convergence criteria of the numerical approximation methods used in estimating affine models.

3) Extension

The model will then be re-estimated using a shifted observation period of 1986-2012 in order to maintain the same number of observations and introduce new factors. The fitted plot and average pricing error of this model will be compared to the estimated model results from the original observation period. Given the occurrence of the "Great Recession" during this period, the author's hypothesis is that the model will miss pricing kernel information beyond BRS's original model

estimation end date in 2004. Macroeconomic uncertainty increased significantly during the lead-up to and during the “Great Recession” and likely played a major role in influencing bond pricing agents’ decision process. BRS’s five factor model without any measure of aggregate uncertainty may not fully account for all major contributing factors to the pricing kernel when the observation period includes a time of high uncertainty. This could lead to inaccurate forecasting and misleading measures of the term premium. Measures of forecast disagreement from the Blue Chip Financial Forecasts and contemporaneous stock market volatility (VIX) will be used to attempt to proxy for economic uncertainty. In order to make the case for including these measures in an affine model, this section will estimate additional models in order to show that factors that control for short- and medium-term uncertainty are important to any affine model of the term structure where pricing in both stable and unstable economic environments is important. The results suggest that, while BRS’s original model with Eurodollar futures is able to price a large amount of the variation in bond prices, adding these additional factors of disagreement and volatility improve the performance of the model and lead to term premium measures that are more sensitive to recessions.

2.1 Data

This section presents the data used in the model estimation and discusses the potential problems with using unadjusted Blue Chip Financial Forecast data. The yields to estimate the model (the details of which are discussed in the next section) are Treasury Bill and Treasury Constant Maturity yields from the Federal Reserve Bank of St. Louis (2013)². Fama-Bliss zero-coupon bonds were available at one, two, three, four, and five year maturities (CRSP, 2013). These latter yields are the industry standard for term structure modeling, used in countless time series studies, and are based on the implied zero-coupon yield estimation strategy from Fama and Bliss (1987). Figures 2.1a, 2.1b, and 2.1c show time-series plots of the same maturity yields for both the constant maturity government bond series available from Federal Reserve Economic Data (FRED) (Federal Reserve Bank of St. Louis, 2013) and the Fama-Bliss implied zero-coupon series³. All three plots show that there are only a few months where there is any noticeable difference between the two time series. These months are all concentrated in the time during Paul Volker’s term as

²These yields were used as a replacement for the Fed internal zero-coupon yield set that BRS originally used in their model. The 4 year maturity treasuries are also not included because of unavailability. Fama-Bliss were only available as a subset of the maturities used in BRS.

³This plots of differences between the treasury constant maturity yields and the Fama-Bliss implied zero-coupon yields are included in the appendix in Figures C.1a, C.1b, and C.1c.

Fed chairman when there was a concentrated effort to stamp out the high inflation of the 1970's. It remains outside of the period of observation for both BRS's original model and the extension presented in a later section. Table 2.1 presents descriptive statistics for the difference between these two measures for the five year maturity yields. While differences do exist, the two follow largely the same pattern and are unlikely to drive significant differences in results when estimating models based on either of these values. If we can assume that the implied zero-coupon yield derivation method used for the internal Fed set produces similar results to the Fama-Bliss method, the results using the constant maturity set should compare to BRS's results.

Figure 2.1: One, Three, and Five-year Yield Plots of Constant Maturity Government Bonds vs. Fama-Bliss Implied Zero-coupon Bonds

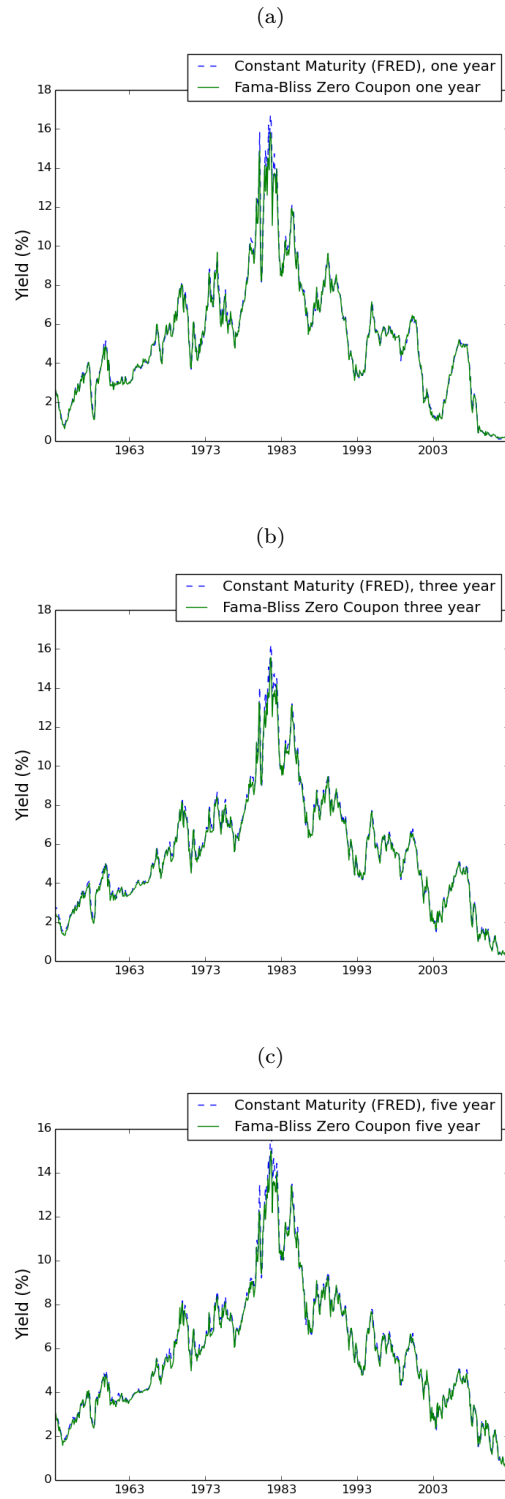


Table 2.1: Descriptive Statistics of Difference in Percentage between Constant Maturity Government Bond Yields and Fama-Bliss Implied Zero-coupon Bond Yields for One, Three, and Five year Maturities 1982-2012.

Statistic	One Year	Three Year	Five Year
mean	0.0549	0.0826	0.0630
std	0.1976	0.2157	0.2246
min	-0.3863	-0.4289	-0.5788
25%	-0.0569	-0.0517	-0.0700
50%	0.0323	0.0754	0.0573
75%	0.1515	0.1867	0.1703
max	0.9966	1.1632	1.1142

The information that BRS use to inform the term structure of yields is an employment gap of total non-farm employment measured as the difference between observed employment and Hodrick-Prescott filtered employment, inflation over the past year measured using the personal consumption expenditures (PCE) price index excluding food and energy, mean expected inflation over the subsequent year from the Blue Chip Financial Forecasts, the effective federal funds rate, and the year-ahead Eurodollar futures rate. Their data are monthly, June 1982 to August 2004. Total non-farm employment is from the Bureau of Labor Statistics (2012). PCE inflation and the effective federal funds were both taken from the Federal Reserve Economic Data (2013), sponsored by the St. Louis Federal Reserve Bank. Year-ahead Eurodollar futures were downloaded from Bloomberg (2012).

Before moving on to the model estimation, it is important to note the peculiar structure of Blue Chip Financial Forecasts (BCFF) and the possible issues with including them, unadjusted, in a time series econometric model. The Blue Chip Financial Forecasts survey has been conducted every month since 1976, polling at least 50 economists for their current-year and year-ahead forecasts for a variety of macroeconomic measures, including GNP/GDP, inflation (as measured by the GNP/GDP deflator), output from key industries, housing, etc. While percentiles of the predictions are not included, means of the top and bottom 10 predictions are included. The survey periodically revises what questions and statistics to include, but the major macroeconomic measures are always included. The BCFF survey recipients are asked about their best guess for each indicator over a given calendar year, no matter the current month of the survey. Beginning in 1980, BCFF began consistently asking in January their forecast for the following year and the current year. Specifically, in January of 1980, economists were asked for their forecast of real GNP growth and

inflation as measured by the GNP deflator for the entire year of 1980 and 1981, separately. The survey is re-administered every month, but the years in question do not change until the following January. Continuing our example, for February through December of 1980, the questions will refer to forecasts for the entire years of 1980 and 1981, separately.

Given that the December year-ahead prediction is only one month away from the first month of the year in question, while the January year-ahead prediction is 12 months away, one might expect that the two are not comparable without adjustment. Specifically, there might be consistently greater disagreement in predictions in earlier months in the year compared to later months, as point predictions converge as practitioners have more information gathered for the same target. This could result in a naturally converging prediction throughout the year towards a certain value, with a jump in the predicted value once January returns. There would thus be a form of seasonality that might be present in both the point-values and dispersion of the values.

Many practitioners, inside and outside the affine model literature, have taken this issue for granted and corrected for it either in the modeling scheme or adjusting the data. Chun (2011) and Grishchenko and Huang (2012) both adjust all forecasts after the first period by using linear interpolation between the forecast for the next period and forecast for two periods ahead. For monthly data, this results in eleven out of every 12 months in a year being the weighted average of two data points. If 11 out of every 12 values are entirely based on a linear interpolation between two values, a lot of potential variation between these values is lost and stability is imposed on values that might otherwise be volatile. Batchelor and Dua (1992) follow the substitution method of McCallum (1976) and Wickens (1982) and correct for this fixed horizon issue by explicitly modeling the rational expectations corrections of the values throughout the year. This method allows the uncertainty pattern to be modeled and adjusted values to be used in the model.

BRS do not explicitly address this issue, implicitly using the unadjusted year-ahead Blue Chip forecasts. While there is a theoretical case for adjustment, let us examine whether the forecasts empirically exhibit trends within the calendar year. As a simple test of whether these values might require adjustment before their inclusion in the VAR determining the pricing kernel, we show the movement of the disagreement over time. Figure 2.2 plots next year disagreement as measured by the average of the top 10 GDP growth predictions minus the bottom 10 GDP growth predictions over time. Each prediction refers to a single economist queried for the survey. The graph reveals downward movements in disagreement over the course of a series of months, but it is not clear whether they are associated with movement within a year. The dark areas are intended to

highlight periods of sustained consecutive downward movement in disagreement. From this cursory view, disagreement seems to decline especially in cases where it is preceded by a previous decrease in disagreement. While this pattern may partially result from a decrease in disagreement over the period of a business cycle expansion, it could also be driven by the fixed horizon issue mentioned above, where uncertainty decreases just by the nature of being a later month in the year.

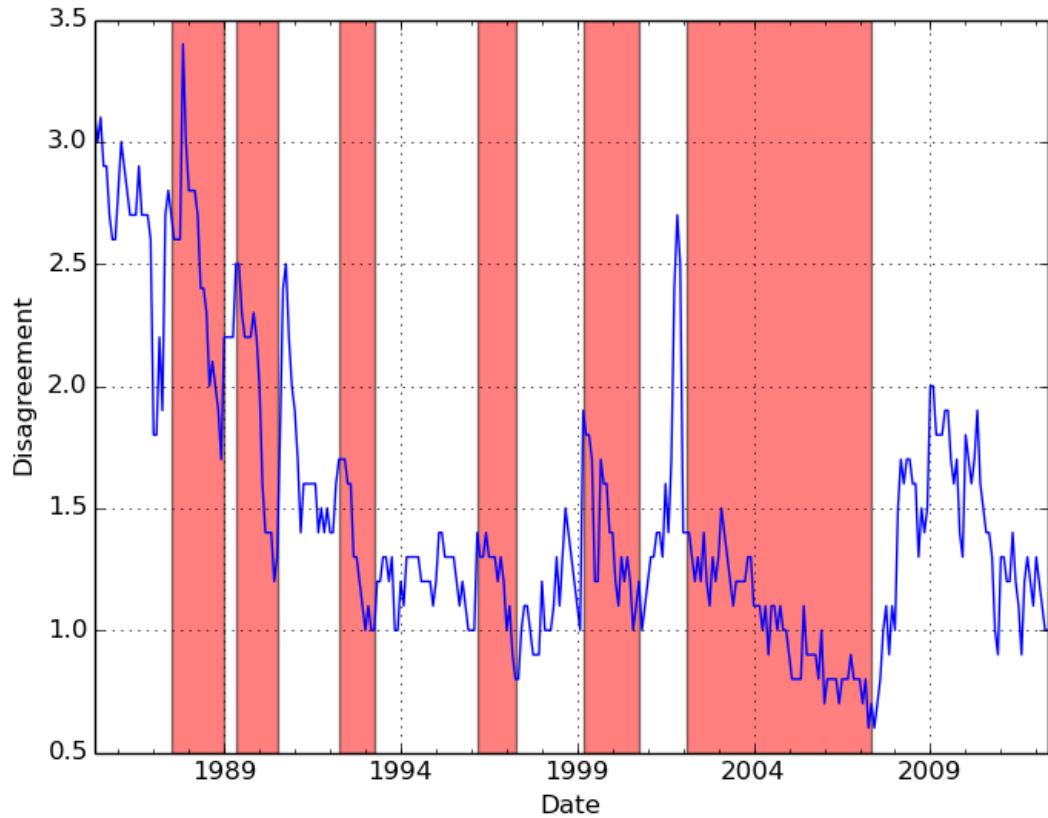


Figure 2.2: Blue Chip Financial Forecasts Next Year Output Growth Disagreement. Highlighted areas could reveal autocorrelated year-ahead disagreement for GDP.

For further investigation, a box-and-whisker graph is presented in Figure 2.3 summarizing the distribution of disagreement by month, across 324 months between 1985 and 2012. The top and bottom wicks represent the maximum and minimum disagreement for that given month. The top and bottom of the box represent the 75% and 25% percentile of the 27 months within that calendar month, respectively. Given this more concise visual representation of disagreement, there does not seem to be any downward pattern to disagreement over the year, as might be expected.

Even though there is more information about the next year in December compared to January, that does not seem to consistently decrease the dispersion of forecasts as the ending of the calendar year approaches.

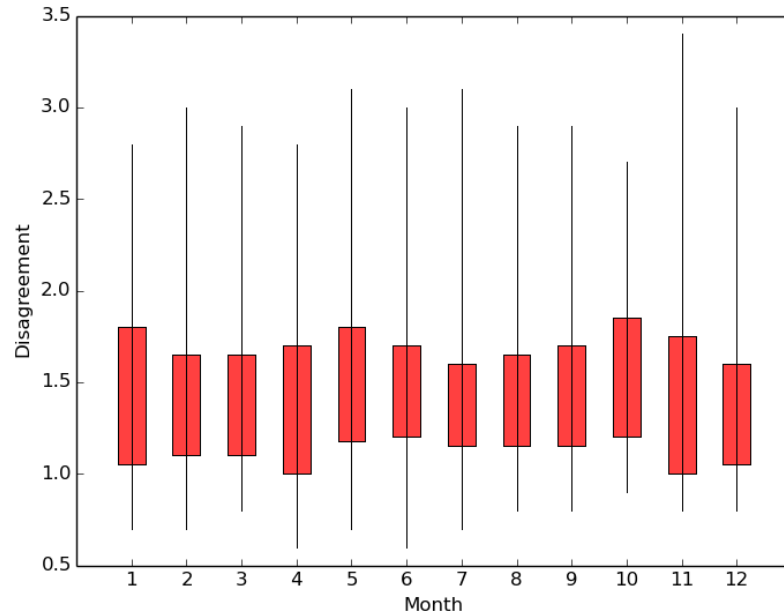


Figure 2.3: Distribution of Disagreement for Next Year by Month. Disagreement measured as average of top 10 predictions minus average of bottom 10 predictions.

On the other hand, for the within year forecasts, there is a clear decline in disagreement over the course of any given year, as shown in Figure 2.4. These are the disagreement in GDP growth for year Y during the months within year Y . As the year passes for these within-year forecasts, the span of possible final values decreases given that a higher fraction of the influencing observations for that year have already been observed, leading to the convergence shown in the figure.

The year-ahead prediction values do not show a within year bias that needs to be corrected for. In the next section, disagreement is measured using the year-ahead prediction measures, so unadjusted Blue Chip data should be appropriate for inclusion in a VAR process. All other macroeconomic measures are included consistent with BRS's original prescribed model.

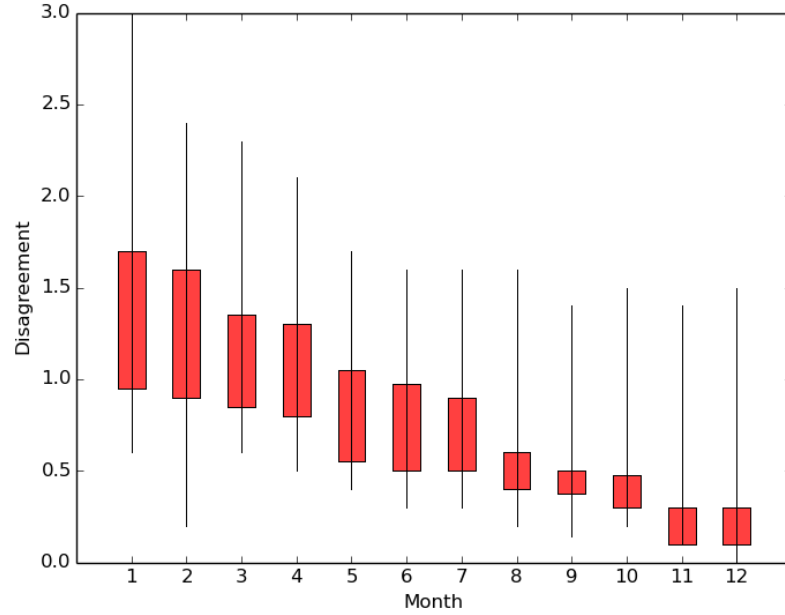


Figure 2.4: Distribution of Disagreement for Current Year by Month. Disagreement measured as average of top 10 predictions minus average of bottom 10 predictions.

2.2 Replication

This section attempts to replicate the results of BRS's estimated affine model of the term structure using the data described in the above section. BRS compare two of these models, with and without Eurodollar futures, to demonstrate the importance of policy expectations in government bond pricing. We begin by addressing the general form of affine term structure models and continue with the specifics outlined by BRS in their model structure and estimation.

The price of any n -period zero-coupon bond in period t can be recursively defined as the expected product of the pricing kernel in period $t + 1$, k_{t+1} , and the price of the same security matured one period in $t + 1$:

$$p_t^n = E_t[k_{t+1}p_{t+1}^{n-1}] \quad (2.2.1)$$

The pricing kernel, k_t , encapsulates all relevant information to bond pricing decisions and is used to price along all relevant maturities. In affine term structure models, as in BRS, zero-coupon bonds are used so that yields all along the yield curve are comparable. Differences in yields must be

solely determined by the perceived risk and expected changes in the pricing kernel. For simplicity, it is assumed that the period-ahead pricing kernel is conditionally log-normal, a function only of the current one-period risk-free rate, i_t and the prices of risk, λ_t :

$$k_{t+1} = \exp\left(-i_t - \frac{1}{2}\lambda_t'\lambda_t - \lambda_t'\varepsilon_{t+1}\right) \quad (2.2.2)$$

where λ_t is $q \times 1$, with $q = f * l$, where f is the number of factors and l is the number of lags. ε of shape $q \times 1$ is assumed $\mathcal{N}(0, 1)$ and are the shocks to the VAR process described below.

Without perfect foresight, agents price risk via a set of macroeconomic factors, X_t . The process governing the evolution of the five factors influencing the pricing kernel is assumed represented as a VAR(1):

$$X_t = \mu + \Phi X_{t-1} + \Sigma \varepsilon_t \quad (2.2.3)$$

where X_t is an $q \times 1$ vector. BRS include five factors and three lags of these factors in X_t , with zeros in μ below the f element and ones and zeros in Φ picking out the appropriate values as a result of lags $l > 1$. BRS's chosen factors are mentioned in the above section. It is assumed that this process fully identifies the time series of information entering bond pricing decisions. μ and Φ are estimated using OLS. Σ summarizes covariance across of the residuals and is assumed an identity matrix.

Agents price risk attributed to each macro factor given a linear (affine) transformation of the current state-space, X_t :

$$\lambda_t = \lambda_0 + \lambda_1 X_t \quad (2.2.4)$$

where λ_0 is $q \times 1$ and λ_1 is $q \times q$.

We can then define the price of any zero-coupon bond of maturity n in period t as a function of the pricing kernel, combining Eqs. 2.2.1–2.2.4, in Equation 2.2.5. This is the relationship that makes these models “affine” and is consistent across the affine term structure model literature.

$$p_t^n = \exp(\bar{A}_n + \bar{B}_n' X_t) \quad (2.2.5)$$

where \bar{A}_n and \bar{B}_n are recursively defined as follows:

$$\begin{aligned}\bar{A}_{n+1} &= \bar{A}_n + \bar{B}'_n(\mu - \Sigma\lambda_0) + \frac{1}{2}\bar{B}'_n\Sigma\Sigma'\bar{B}'_n - \delta_0 \\ \bar{B}'_{n+1} &= \bar{B}'_n(\Phi - \Sigma\lambda_1) - \delta'_1\end{aligned}\tag{2.2.6}$$

where $\bar{A}_1 = \delta_0$ and $\bar{B}_1 = \delta_1$ and δ_0 and δ_1 relate the macro factors to the one-period risk-free rate:

$$i_t = \exp(\delta_0 + \delta_1 X_t)\tag{2.2.7}$$

In the same way, the yield can be expressed as:

$$y_t^n = A_n + B'_n X_t\tag{2.2.8}$$

where $A_n = -\bar{A}_n/n$ and $B_n = -\bar{B}_n/n$.

Equations (2.2.1)–(2.2.4) completely identify a system relating a data-generating process of macroeconomic measures to a pricing kernel and that pricing kernel to assets of similar characteristics along a single yield curve. λ_0 and λ_1 are estimated using non-linear least squares to fit the pricing error of selected yields along the yield curve, in this case: one, two, three, four, five, seven and ten year maturity zero-coupon bonds. The model-predicted yields are generated by feeding the VAR elements, X_t , for each t into Equation 2.2.8, using the estimated λ_0 and λ_1 in Equation 2.2.6. By setting the prices of risk to zero in λ_0 and λ_1 , the implied risk-neutral yields can be generated. To reduce the parameter space, it is assumed that the prices of risk corresponding to lagged elements of X_t are zero, resulting in blocks of zeros below the f element of λ_0 and outside of upper left $f \times f$ elements of λ_1 .

Presented in Figure 2.5 are two graphs presented as they appear in Bernanke et al. (2005, p. 46). Each graph shows three lines: the actual yield, the model-predicted yield, and the risk-neutral yield. The model-predicted and risk-neutral yield are both generated from the estimated parameters where the difference between the two is the implied term premium.

Table 2.2 is also taken from Bernanke et al. (2005, p. 47), presenting the standard deviation of the pricing errors for all of the yields used in the estimation of the two models, with and without Eurodollar futures included as a macro factor influencing the pricing kernel. At each maturity, the pricing error is lower after the inclusion of Eurodollar futures, although the gain in fit is greater

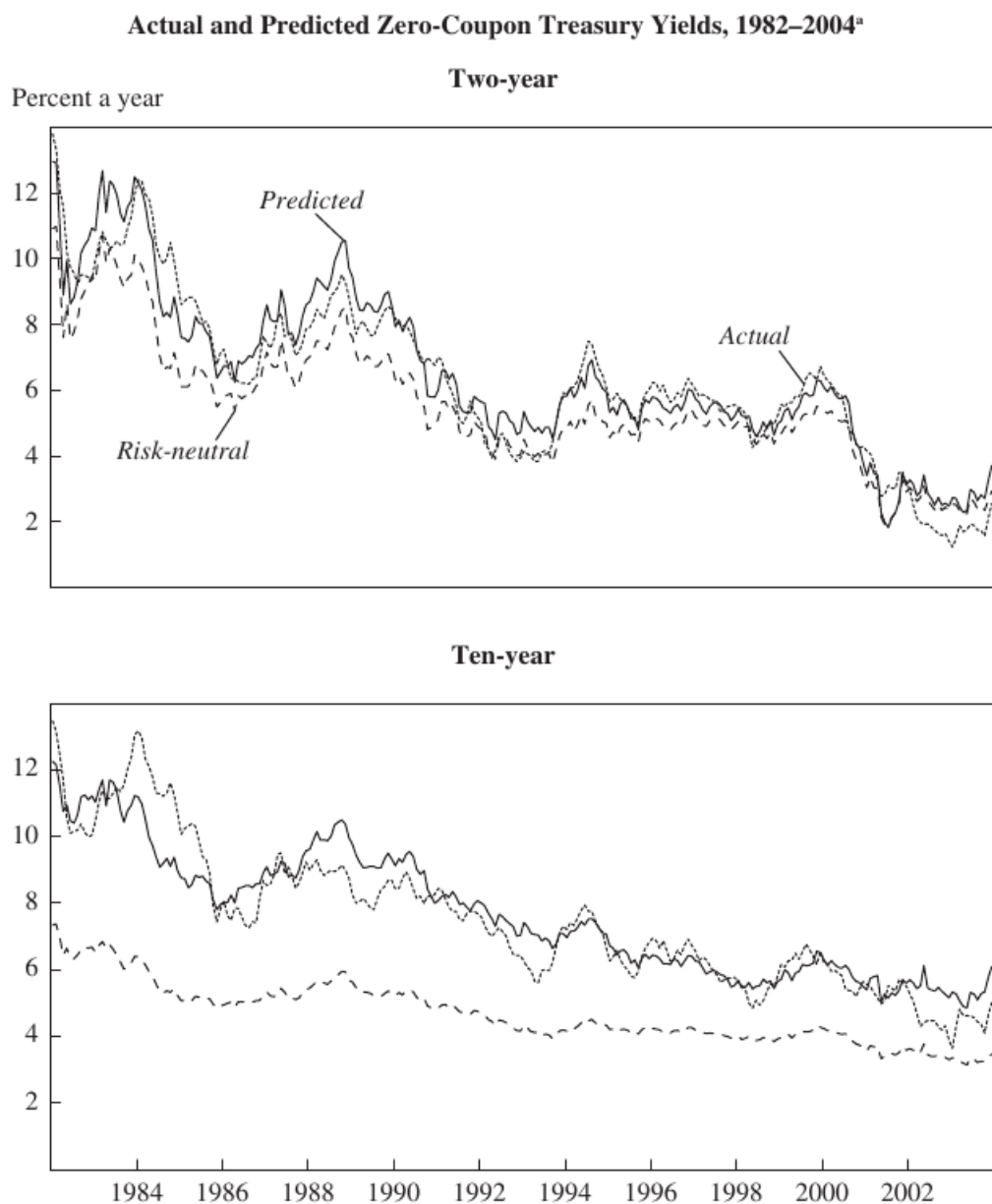


Figure 2.5: Page 46 from Bernanke et al. (2005)

for the shorter maturity yields. BRS take this as evidence that policy expectations play a major role in shaping government bond yields.

Table 2.2: Page 47 from Bernanke et al. (2005)

Prediction Errors for Treasury Yields in the Term Structure Model		
Basis points		
<i>Maturity</i>	<i>Standard deviation of predicted yield</i>	
	<i>VAR with Eurodollar shocks</i>	<i>VAR without Eurodollar shocks</i>
6 months	33.0	62.1
1 year	50.3	78.9
2 years	73.3	97.4
3 years	81.2	100.7
4 years	82.5	98.3
5 years	81.5	95.0
7 years	83.3	93.3
10 years	80.8	87.8

Source: Authors' calculations based on data from the Bureau of Labor Statistics, the Bureau of Economic Analysis, Blue Chip Financial Forecasts, the Chicago Mercantile Exchange, and the Federal Reserve.

We now attempt to replicate the results presented in BRS using a custom-written solution method in Python (see Chapter 4 for details) and using data available outside the Fed⁴. All factors are as they appear in BRS. The two time series of yields, predicated, actual, and risk-neutral are presented graphically in Figures 2.6 and 2.7, echoing their presentation in Figure 2.5 from BRS.

There are a few main results that align between both BRS's original results and the results of this chapter's model runs. First, there is a positive term premium throughout the observation period. In both Figures 2.6 and 2.7, the risk neutral predicted yield is below the actual and predicted yield for the majority of the observation period. Second, for the ten-year yield, the term premium declines throughout the observation period. This reinforces the qualitative observation that stable inflation and growth decreased the perceived liquidity risk of longer maturity bonds over the course of the Great Moderation, reducing the yield to hold these bonds above and beyond that predicted by the modeled risk-neutral expectations.

Table 2.3 presents the standard deviation of the pricing error at each estimated maturity. The pricing errors for the model with Eurodollar futures are similar to the original BRS estimation results

⁴The data was requested but was not available.

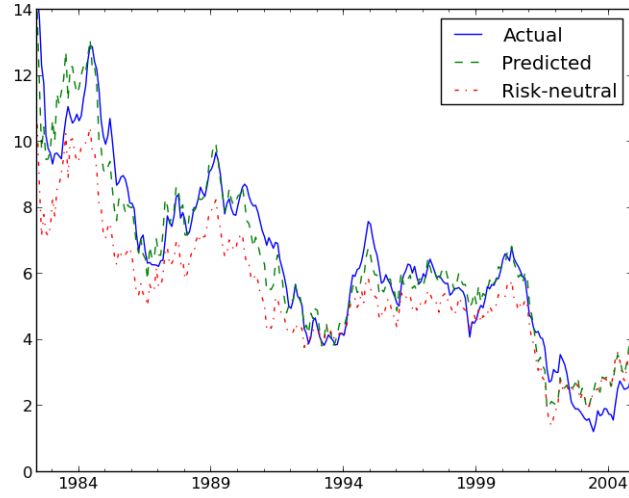


Figure 2.6: Author's Estimation Results Showing Actual, Predicted, and Risk-neutral Percentage Yields for 2-year Treasury Constant Maturity. The risk-neutral yield is calculated by settings the prices of risk in the estimated model equal zero. Actual yield is included for comparison.

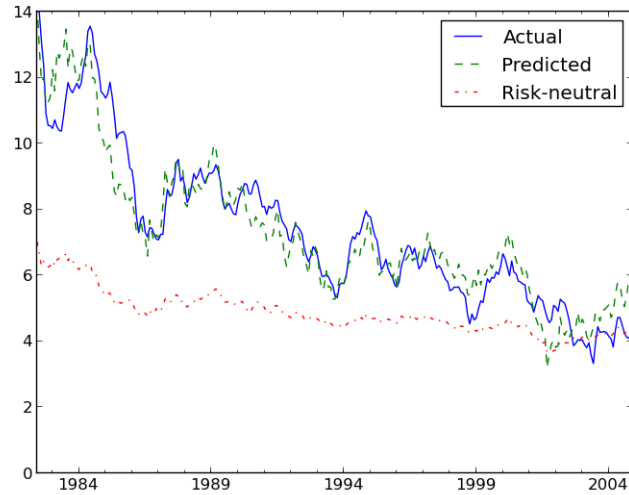


Figure 2.7: Author's Estimation Results Showing Actual, Predicted, and Risk-neutral Percentage Yields for 10-year Treasury Constant Maturity. The risk-neutral yield is calculated by settings the prices of risk in the estimated model equal zero. Actual yield is included for comparison.

shown in Table 2.2, while the pricing errors for the model without Eurodollar futures diverge more substantially. The same convergence tolerance thresholds were used for both of the these models,

with the only difference between them the addition of Eurodollar futures. The improvement in pricing error from the addition of Eurodollar futures is not as large as that presented in 2.2. In fact, the only difference in pricing error of more than five basis points is at the six month maturity.

Table 2.3: Standard Deviation of Pricing Error in Basis Points.

Maturity	VAR with ED shocks	VAR without ED shocks
6 months	34.84	39.27
1 year	53.07	57.22
2 years	74.29	77.30
3 years	79.88	82.22
5 years	79.09	81.06
7 years	77.15	79.36
10 years	72.62	74.80

It is difficult to perceive exactly why the results of replication differ so significantly from BRS's model. While the model run with Eurodollar futures comes quite close to the original results, the author's estimation of the model without Eurodollar futures performs much better than the results presented by BRS. This could be due to a number of factors.

First, BRS use internal Fed zero-coupon yield data while this author's estimation uses a separate set of treasuries as described above. Using the same set of factors as BRS in Eq. 2.2.3 but a slightly different set of predicted yields in Eq. 2.2.8 would result in different results for pricing errors along the yield curve. Given the similarity of the yields used as described above and shown in Figures 2.1a-2.1c, it is unlikely that this large of a difference in model performance could be completely accounted for by small differences in the input yields.

Second, BRS may have set the convergence criteria for both parameter and/or function evaluation differences differently between the models with and without Eurodollar futures. In order to investigate this sensitivity, multiple estimations were performed using the same set of factors but a wide range of convergence criterion for both the sum of squared pricing errors and the parameter estimates. Convergence tolerance thresholds for the sum of squared pricing errors were set to range from 0.1 to 1×10^{-5} with 15 values between the two, with four values at each power of 10. Specifically, the values were 7.5×10^{-n} , 5×10^{-n} , 2.5×10^{-n} , and 1×10^{-n} , with $n \in \{-2, -3, -4, -5\}$ with

the addition of 0.1, making for 17 possible values. The same range was used for the convergence tolerance thresholds for the parameter estimates, making for $17 \times 17 = 289$ estimations for each of the models with and without Eurodollar futures. Convergence thresholds lower than 1×10^{-5} for either the sum of squared pricing errors or the parameter estimates did not make any noticeable changes in the parameter estimates. After examining the results of these models, it seemed that changes in parameter estimates and thus pricing errors were driven primarily by the parameter convergence threshold rather than the sum of squared errors convergence threshold.

Varying the different convergence criterion for the parameter estimates errors results in very different pricing errors. Sets of pricing errors for a few key values of parameter convergence criterion are presented in Table 2.4 for the models without Eurodollar futures first followed by the models with Eurodollar futures. The sum of squared error convergence criterion is 1×10^{-8} across these models. As can be seen, convergence could be reached with even lower thresholds, resulting in better model fit than BRS's presented results across all maturities. BRS do not explicitly mention the convergence criteria that they use. If we compare the pricing error along the same convergence threshold for the models without Eurodollar futures and with Eurodollar futures in Table 2.4, we find that comparing the two models depends very much on the convergence criteria used. Again, BRS emphasize the gain in model performance, as measured by the pricing error, by adding a fifth factor, Eurodollar futures to their model.

For a true comparison between models, it makes sense to compare the two models using the exact same convergence criteria along all dimensions. The differences in pricing error of the two models, with and without Eurodollar, *ceteris paribus*, are presented in Table 2.5. Using the strictest convergence criteria presented (xtol=0.0001), we find that the model with Eurodollar futures outperforms the model without Eurodollar futures comparing all seven key yields. Using looser convergence criteria, we find that the improvement in pricing error occurs only at certain maturities. At the 0.05 and 0.1 levels, the model without Eurodollar futures actually outperforms the model with futures at most key yields. At the 0.01 level and below, the model with Eurodollar futures consistently outperforms the model without Eurodollar futures. It is not until these lower convergence tolerance thresholds are used that the parameter estimates and thus the pricing errors settle down to reliable levels. The levels chosen (0.1, 0.05, 0.01, 0.001, 0.0001) seemed to be key convergence tolerance thresholds to generating more precise parameter estimates. In the context of this modeling exercise, these levels seemed to generate changes in the parameter estimates when

Table 2.4: Standard Deviation of Pricing Error by Parameter Difference Convergence Criterion. Monthly data 1982-2004.

Model without Eurodollar factor					
Maturity	xtol=0.1	xtol=0.05	xtol=0.01	xtol=0.001	xtol=0.0001
6 months	52.78	53.39	44.60	39.64	39.81
1 year	111.07	112.15	84.09	57.79	57.73
2 years	160.35	160.10	112.34	78.68	77.97
3 years	166.23	165.33	116.72	83.30	82.74
5 years	175.48	173.95	114.20	82.21	81.65
7 years	185.80	184.04	118.81	80.54	79.95
10 years	195.21	193.72	141.93	76.10	75.43
Model with Eurodollar factor					
6 months	51.57	51.59	36.88	37.46	35.21
1 year	113.33	112.89	57.98	54.71	53.41
2 years	171.24	168.44	85.97	77.53	74.46
3 years	180.94	177.10	92.63	82.50	79.91
5 years	189.03	184.70	91.80	81.65	79.21
7 years	196.81	192.94	93.81	80.23	77.26
10 years	203.66	201.14	95.12	76.14	72.71

moving to the next lower threshold. For example, moving from 0.075 to 0.05 did not always generate a change in the parameter estimates, but moving from 0.05 to 0.1 consistently produced a change in the parameter estimates.

While these thresholds are characteristic of this specific model and not generally applicable to all affine models, the author recommends that convergence criteria should be lowered until either convergence can no longer be reached or a relevant machine epsilon is hit. This recommendation is based on the observation that model performance comparisons based on pricing error are inconsistent when the convergence tolerance is too high (loose), as shown in Table 2.5. It should also be noted that even if a software claims to support very low convergence tolerances (higher precision), the precision of the datatype is of special consideration with the recursive calculations of A_n and B_n in equation 2.2.8. With each calculation of A_n and B_n based on A_{n-1} and B_{n-1} , any numerical precision difference will be expounded based on how high n goes up to. For example, for any C based language using numbers based on the type **double**, precision below $2^{-52} = 2.22 \times 10^{-16}$ is not reliable for a single calculation and will be even higher once any kind of recursive calculations are considered. Staying far above this machine epsilon for the convergence tolerance threshold is recommended. In case of the C **double**, staying around 1×10^{-9} should result in high precision while still staying away from any datatype precision issues, if that level of precision can be attained. These recommendations are especially important when pricing error results are compared across

models. Only when these levels are pushed low can comparable results be generated. This is primarily a result of the fact that these models are non-linear and results can be highly sensitive to the parameters of the numerical optimization method.

Table 2.5: Difference in Standard Deviation of Pricing Error between Model with and without Eurodollar Factor. Function Difference is 1.49012e-8 for all columns. Monthly Data 1982-2004.

Maturity	xtol=0.1	xtol=0.05	xtol=0.01	xtol=0.001	xtol=0.0001
6 months	-1.21	-1.80	-7.72	-2.18	-4.60
1 year	2.26	0.74	-26.11	-3.08	-4.32
2 years	10.89	8.34	-26.37	-1.15	-3.51
3 years	14.71	11.77	-24.09	-0.80	-2.83
5 years	13.55	10.75	-22.40	-0.56	-2.44
7 years	11.01	8.90	-25.00	-0.31	-2.69
10 years	8.45	7.42	-46.81	0.04	-2.72
Sum	<i>59.66</i>	<i>46.12</i>	<i>-178.50</i>	<i>-8.04</i>	<i>-23.11</i>

Another interesting direction to consider is the statistic that BRS use to judge whether improved model fit takes place. While standard deviation of the errors may be important for higher order convergence, root-mean-square error (RMSE) or mean absolute deviation (MAD) is a much more commonly used statistic used to compare the fit of different models (See Ang and Piazzesi (2003) and Kim and Orphanides (2005)). Given this choice of comparison, this section also presents an analogous table to Table 2.3 using RMSE in Table 2.6. This table largely mirrors the values and patterns in 2.3.

As evidence of the importance of policy expectations to the yield curve, BRS use the improvement in pricing error gained by adding Eurodollar futures as the fifth factor to the VAR determining the pricing kernel. The results of this section confirm the use of Eurodollar futures in improving the performance of BRS's four factor model by lowering the pricing error across all directly estimated maturities, although the improvement in pricing error is not quite as large when lower convergence criteria were used. Using these lower convergence criteria, the models with and without Eurodollar futures both outperformed BRS's original presented results. Overall, this section confirms that Eurodollar futures offer meaningful explanatory value in a term structure model estimated during the "Great Moderation".

Table 2.6: Root Mean Squared Pricing Error in Basis Points

Maturity	VAR with ED shocks	VAR without ED shocks
6 months	34.84	37.82
1 year	53.07	56.95
2 years	74.29	76.39
3 years	79.88	81.52
5 years	79.09	81.25
7 years	77.15	80.07
10 years	72.62	74.80

2.3 Extension into the Great Recession of 2007-2009

BRS indicate that their five observed factor model estimated from 1982 to 2004 does “quite a creditable job of explaining the behavior of the term structure over time” (Bernanke et al., 2005, p. 45). They also justify the use of Eurodollar futures as a fifth factor by noting the decrease in pricing error when the factor is added. This section will consider the robustness of the model’s fit when the observation period is extended to include the recent financial crisis. As noted above, BRS’s period of study is firmly within the “Great Moderation”, a term introduced in Stock and Watson (2003), to refer to the period from the early 1980’s to the mid 2000’s, when inflation was low and growth was stable. Given the predictable economic conditions during this period, the choice of year-ahead Eurodollar futures may have added explanatory value to the model through its correlation with these stable economic conditions rather than via its value as an inter-temporally accurate proxy of policy expectations. If this hypothesis is true, we may expect the explanatory value of the model to deteriorate when estimated in a time period that includes the periods of higher economic uncertainty and volatility not seen during the “Great Moderation”, particularly the recent financial crisis.

In addition to testing the ability of BRS’s model outside of the original sample, this section would also like to propose the addition of measures of economic uncertainty in order to further extend the model and lead to more robust measures of the term premium. Following the housing market collapse of 2007 and accompanying stock market crash and financial crisis, there was a popular perception that aggregate economic uncertainty had increased. The potential of economic uncertainty to affect the real economy has theoretical roots in Keynes (1936) and Minsky (1986), who linked uncertainty to real economic activity through its effect on asset prices and investment. While interest in this topic waned during the “Great Moderation”, the ability of economic uncer-

tainty to drive and exacerbate real economic outcomes received a revival of interest during and following the recent financial crisis. Bloom (2009) found that increases in economic uncertainty built up from firm level data lead to a decrease followed by a rebound in both aggregate output and employment. Baker et al. (2013) also finds that increases in uncertainty as measured using indicators including newspaper references to uncertainty, economist forecast dispersion, and scheduled congressional tax-code expirations lead to decreases in investment and other measures of economic activity when included in a VAR.

If uncertainty has an impact on real economic outcomes, and if real economic outcomes are used to inform the pricing kernel, then economic uncertainty could have an independent effect on pricing the yield curve. Given the potential impact of uncertainty on the yield curve and the increase in uncertainty following the stock market crash of 2008-2009, this section will also propose the addition of proxies for economic uncertainty to BRS's five factor model. Given the limited availability of monthly survey data that includes forecast uncertainty measures, this section will propose the use of two proxies, one for disagreement and one for volatility, in an attempt to price the movements in the term structure associated with short- and medium-term uncertainty. The proxy for disagreement will be the difference between the average of the top 10 predictions and the average of the bottom 10 predictions of the year-ahead output forecasts from the Blue Chip Economic Indicators. More robust measures based on difference between percentiles or dispersion measures from the Blue Chip Financial Forecasts were not available. Even though the difference between upper and lower survey result percentiles has commonly been used as a proxy for uncertainty (Zarnowitz and Lambros (1987), Giordani and Söderlind (2003)), a fairly recent groups of papers, Rich and Tracy (2010) and Rich et al. (2012), show that the relationship between economists' disagreement and uncertainty is inconsistent, challenging the main conclusion of Bomberger (1996). Rich et al. (2012) use a mix of moment-based and inter-quartile range-based (IQR) approaches to show that, in the best cases, disagreement measures can only explain about 20% of the variation in uncertainty measures. With this conclusion, the authors instead use a quarterly measure of prediction uncertainty from the European Central Bank conducted Survey of Professional Forecasters. A similarly detailed monthly survey of U.S economists' predictions is not currently available, so while forecast disagreement may capture some of economic uncertainty, it alone cannot be expected to capture economic uncertainty in general. Even though forecast disagreement may not capture uncertainty alone, it has been shown to independently have a relationship with real output and inflation. For example, Mankiw et al. (2004) show that a New-Keynesian model with sticky information is able

to produce autocorrelated forecast errors seen in forecast disagreement data. Also, Dovern et al. (2012) show that forecast disagreement is strongly correlated with business cycle movements.

The proxy for volatility will be a measure of stock market volatility, the Chicago Board Options Exchange (CBOE) S&P 500 volatility index, commonly known as the VIX. Even though the VIX has only been traded since March 26, 2004, it has been retrospectively calculated going back to 1990. The value of the VIX is calculated to represent the expected 30-day volatility of the S&P 500 and can be thought of as volatility expressed at an annual rate (CBOE, 2009). Volatility can represent movement in any direction, so the measure is not a pure measure of uncertainty, but does represent an indicator of expected movements in the stock market over a fairly short-period, at least in the context of the monthly macroeconomic variables used as other factors in the model. The use of stock market volatility in macroeconomic models and the result that stock market volatility impacts other markets are both well-established. Fleming et al. (1998) specify a basic trading model with transactions between stock, bond, and money markets, showing that the volatility linkages between these markets are strong. It has also been established that volatility does have an impact on macroeconomic growth, at least in some countries (Diebold and Yilmaz, 2008). Adrian et al. (2010) include the VIX in a VAR estimating relationships between monetary, business cycle, and financial markets.

As a partial response to Rich et al. (2012)'s critique, a third model will be estimated which adds practitioner disagreement and stock market volatility together to the basic BRS model. Figure 2.8 plots both the disagreement measure and the VIX, revealing at least some visual correlation between the two measures. While individually these measures may not reflect all economic uncertainty, together in a single model they may come closer to summarizing short- to medium-term uncertainty. These models will all be estimated using monthly data from May, 1990 to May, 2012. During this period, the disagreement and volatility measures have a correlation coefficient of 0.4, suggesting that, while the two are correlated, the correlation is far from perfect and they may individually reveal different information about uncertainty.

For convenience, let us define the set of models by the macro factors that constitute them. The model definitions are summarized in Table 2.7. Inclusion of the employment gap, inflation, expected inflation, and the federal funds rate is defined as 'baseline' or 'b' since these are consistent across all of the models and constitute BRS's original comparison model. 'E' indicates that Eurodollars futures are included in the model. 'D' represents the disagreement proxy and 'V' represents the volatility proxy. Originally, appending data onto the end of the original test period (1982-2004) was

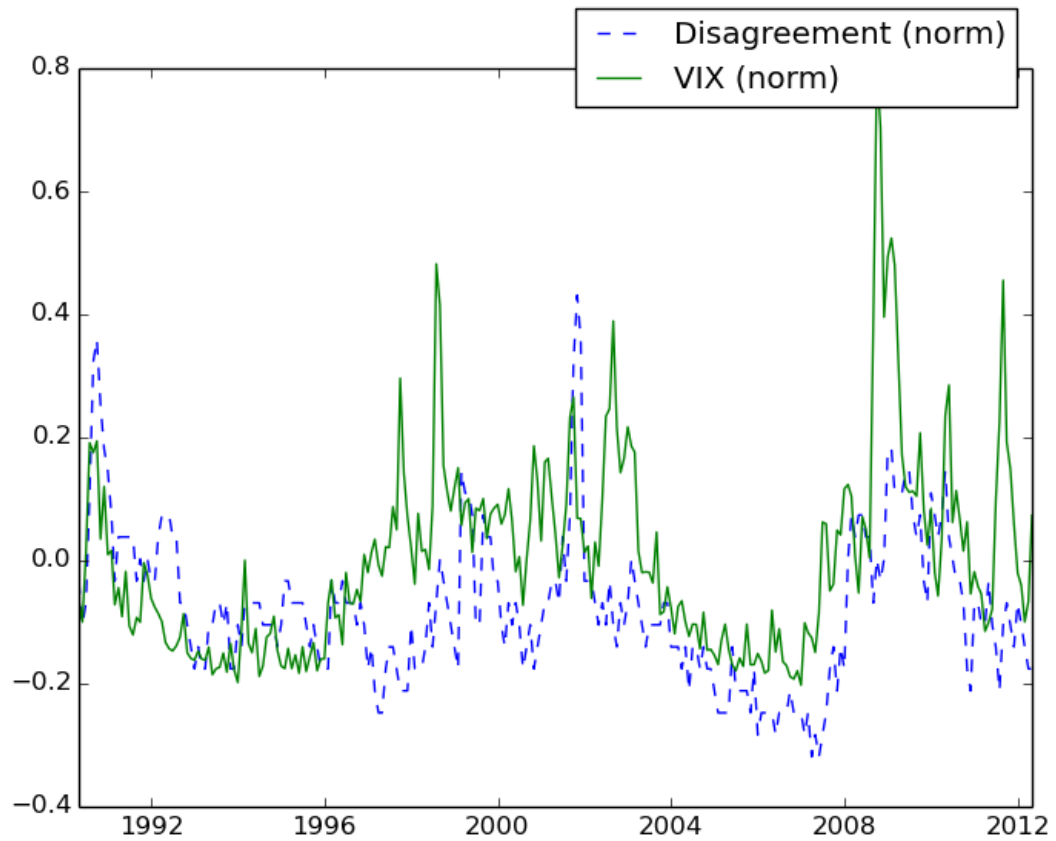


Figure 2.8: Blue Chip Disagreement and VIX, 1990-2012

considered, but this was rejected for two reasons. First, reliable measures of disagreement (D) did not begin until 1986. Second, in order to make reliable comparisons with the BRS model estimated with our yield data, the results of which were presented in Table 2.3, it was important to use an observation period of the same length, namely, 22 years of monthly data. Comparing a model with more observations could result in decreases in the pricing error gained from a better fitting set of observations added to one end of the observation period. This could lead to the conclusion that a model performed better, when performance in the original period of observation has not improved. Restricting to the same number of observations allows any model improvements to be a result of changes in the macroeconomic environment as reflected in the data compared to the original observation period of 1986-2004 and/or changes in the choice of factors in X_t of Equation 2.2.3 and not the result of the addition of more observations.

Table 2.7: Model Classifications

Name	Macro Factors
b	Empl gap, inflation, expected inflation, fed funds = baseline
b + E	baseline, Eurodollar futures
b + D	baseline, Blue Chip expectations disagreement
b + V	baseline, S&P 500 VIX
b + D + V	baseline, Blue Chip expectations disagreement, S&P 500 VIX
b + E + D	baseline, Eurodollar futures, Blue Chip expectations disagreement
b + E + V	baseline, Eurodollar futures, S&P 500 VIX
b + E + D + V	baseline, Eurodollar futures, Blue Chip expectations disagreement, S&P 500 VIX

Again, each one of the models was estimated using the assumptions concerning block zeros in λ_0 and λ_1 in Equation 2.2.3. The unknown parameters in λ_0 and λ_1 were estimated using nonlinear least squares, initializing the guesses of all unknown elements to 0. A parameter convergence tolerance threshold of 0.00001 and sum of squared errors convergence tolerance threshold of 1.49012×10^{-8} were used. These thresholds were set tighter than that implied by BRS's original model as a response to the questions raised about proper thresholds raised in the previous section.

Examining the results in Table 2.8 we see that there is a clear improvement in average pricing error by adding any of the extra factors in models **baseline + E**, **baseline + D**, **baseline + V**, and **baseline + D + V** over **baseline**. While **baseline + D**, **baseline + V**, and **baseline + D + V** all have higher average pricing errors at all estimated maturities than **baseline + E**, each offers additional explanatory value compared to the **baseline** model. This extension reinforces BRS's inclusion of Eurodollar futures as a proxy for expectations, and, when combined with the other **baseline** variables, holds as a macroeconomic model summarizing a good deal of macroeconomic movement and as a reasonable information set of the pricing kernel. While **D**, **V**, and **D + V** do

not seem to replace or offer more explanatory value than **E** in this model, they do seem to offer some explanatory value and may offer explanatory value that is entirely separate from expectations.

Given this hypothesis, three additional models are included: **baseline + E + D**, **baseline + E + V**, and **baseline + E + D + V**. These models are included to test whether measurements of disagreement and volatility lower the pricing error of the model beyond adding Eurodollar futures. Adding each of these measurements individually and together to **baseline + E** results in lower pricing errors. This supports the hypothesis that disagreement and volatility both seem to have information valuable to explaining movement in the term structure not contained in Eurodollar futures. Eurodollar futures also may not fully capture higher moments of expectations that disagreement and/or volatility do.

Table 2.8: RMSE for Estimated Models. Parameter difference is 0.00001 and function difference is 1.49012×10^{-8} for all columns. Monthly data May, 1990 to May, 2012. *=90%, **=95%, and ***=99%, where these refer to confidence levels for a two-sided t-test for a difference in the mean pricing error between the models shown in Table 2.9.

Maturity	b	b+E	b+D	b+V	b+D+V	b+E+D	b+E+V	b+E+D+V
6 months	24.25	18.49***	23.01	23.19	22.89	16.64	15.86**	15.61
1 year	38.84	27.86***	32.86*	33.45*	32.57	20.93***	19.59***	19.00
2 years	56.08	33.38***	45.10***	48.39**	45.20	23.53***	22.58***	22.10
3 years	62.32	35.44***	50.14***	54.50**	50.30	26.60***	26.40***	25.55
5 years	64.73	36.91***	54.76***	59.38	54.92	31.92***	32.12**	31.19
7 years	63.42	38.04***	54.67***	59.55	54.71	33.20***	34.68	33.02
10 years	58.97	35.60***	52.59**	56.39	52.76	34.13	35.41	33.95

Table 2.9: Model Comparisons for T-Test.

Model	Comparison
b + E	b
b + D	b
b + V	b
b + D + V	b + D
b + E + D	b + E
b + E + V	b + E
b + E + D + V	b + E + D

Table 2.9 matches models to the comparison model that was used for running a two sample, two-sided t-test for difference between the RMSE. The associated confidence levels, 90% (*), 95% (**), and 99% (***), from these t-tests are included in Table 2.8 to show whether differences

in pricing error are significant between each pair of models. The statistical significance of the differences in pricing error reinforce the aforementioned conclusions that Eurodollar futures are important to pricing the term structure and disagreement and volatility offer additional explanatory value in helping to price higher moments of expectations. The decrease in pricing error from the inclusion of Eurodollar futures is statistically significant at all maturity levels over the standard **baseline** model. Importantly, the inclusion of Eurodollar futures results in a statistically significant decline in pricing error, compared with the reference four factor model even in an observation period that includes the financial crisis. Confidence levels for the **b+D** and the **b+E+D** models suggest that the impact of adding disagreement is much greater at the longer maturity end of the yield curve, with highest significance at the 2-7 year maturity levels in the **b+D** model and 1-7 years in the **b+E+D** model. Overall, volatility does not have as much of an impact on the pricing error, but there does seem to be some evidence that, when it does have an impact, it is primarily concentrated in the shorter maturity end of the yield curve. Adding volatility produces statistically significant differences in pricing error only in the 1-3 year range for the **b+V** model and in the six-month to five-year range for the **b+E+V** model. The impact of disagreement concentrated more on the medium- to long-term portion of the yield curve and the impact of volatility concentrated more on the short- to medium-term portion of the yield curve suggests that the two measures offer complementary but unique information to the pricing kernel.

For completeness, models are also estimated using Fama-Bliss (CRSP, 2013) implied zero-coupon bonds data that are often used in the affine term structure model literature. While these yields are at different points along the yield curve than the data used in the original BRS model, unlike the yields used in Table 2.8, Fama-Bliss yields are true zero-coupon bonds and are a better match for the theory underlining affine models, unlike the constant maturity yields used for the previous analysis⁵. These results are provided as a validation of the model estimation procedure used. Model results using Fama-Bliss yields are presented in Table 2.10. Results largely confirm the significance of disagreement, with even more added explanatory value in volatility in the **baseline+E+D+V** over **baseline+E+D**. These results reinforce the main conclusions above, primarily that: 1) Eurodollar futures contain information important to the pricing kernel informing the term structure of zero-coupon government bonds, 2) disagreement and volatility measures provide information important to the pricing kernel above and beyond that of Eurodollar futures,

⁵In order for a single pricing kernel to price bonds all along the yield curve, the yields must be easily priced and compared, making zero-coupon yields a good fit. This is addressed at the beginning of the replication section.

and 3) disagreement and volatility themselves provide separate information and together lower the pricing error more than each individually, given how they lower pricing error at different maturities.

Table 2.10: RMSE for Model Using Fama-Bliss Zero-coupon Bonds. Parameter difference is 0.00001 and function difference is 1.49012×10^{-8} for all columns. Monthly data May, 1990 to May, 2012. ***=99%, **=95%, and *=90%.

Maturity	b	b+E	b+D	b+V	b+D+V	b+E+D	b+E+V	b+E+D+V
1 year	40.85	23.53***	35.24**	36.05*	35.69	14.03***	15.44***	13.46
2 years	58.06	27.39***	48.99***	49.92**	48.67	15.81***	17.64***	11.99***
3 years	65.90	30.87***	55.06***	56.55**	54.54	20.54***	22.82***	16.94***
4 years	68.03	32.95***	57.31***	59.37**	56.64	25.47***	27.55***	22.09**
5 years	67.62	32.72***	58.59***	60.88*	58.01	28.27***	29.68*	25.26**

While the RMSE helps to establish the value of Eurodollar futures, disagreement, and volatility measures to informing pricing kernel, plots of the errors and term premia can help to build a story as to why they may matter. Figure 2.9 plots the pricing errors of the five year yield for a few select models from Table 2.8⁶. Comparing the error plots, there is a clear change in the error process moving from the baseline four factor **b** model (the first plot) to the model with Eurodollar futures added **b+E** (the second plot). The error process becomes more concentrated around zero and there seem to be fewer consecutive periods where the error is consecutively positive or consecutively negative, revealing that more variance in the yield process is captured by information explicitly included in the model. A few select periods, 1999-2000, 2001-2002, 2005-2007, and 2008-2009, are highlighted to show the change in the pattern of the pricing errors after adding Eurodollar futures. These date ranges are linked to the tech stock boom of 1990-2000, the recession of 2001, the housing market bubble of 2005-2007, and the Great Recession of 2007-2009. This desirable change in the error process along with the lower pricing error further reinforces BRS's original conclusion that Eurodollar futures offer important information in a pricing kernel. Adding disagreement (the third plot) and volatility (the fourth plot) to the model do not seem to fundamentally change the error process to the degree that adding Eurodollar futures did, although there does seem to be a further concentration of the pricing error around zero. The further concentration around zero can also be seen in the highlighted periods. The error process taken alone seems to indicate that the addition of Eurodollar futures leads to a more well-behaved error process and lower pricing error, while the addition of disagreement and volatility lower the pricing error, but

⁶A more complete set of plots for the pricing error on one and five year yields is included in the Appendix in Figure C.2

do not generate a fundamental change in the error process in the same manner that the addition of Eurodollar futures do.

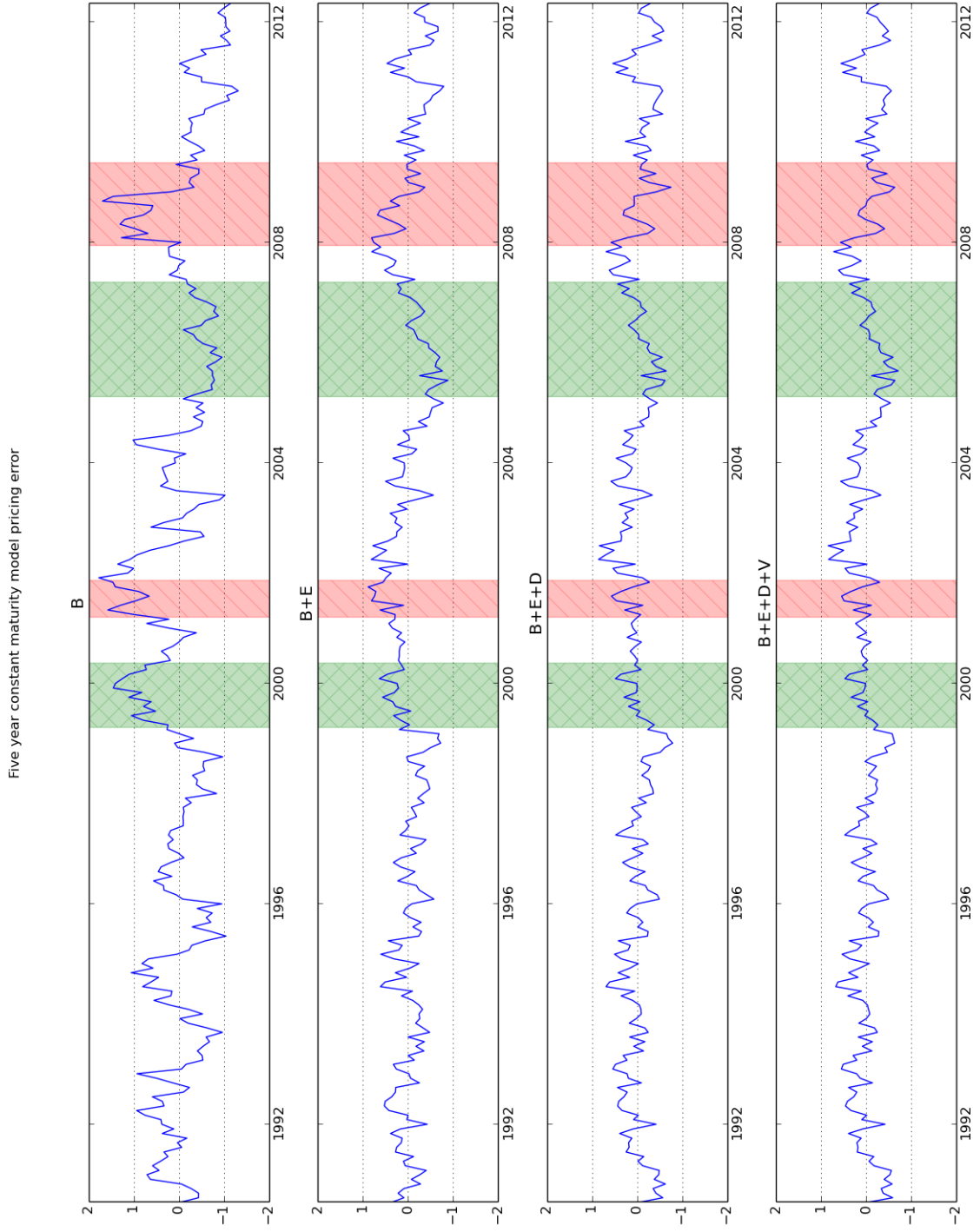


Figure 2.9: Plots of Pricing Error for Five Year Yield for Select Models. Each plot shows the error process for an individual model.

Examining the resulting time-varying term premia offers more information on what value disagreement and volatility add to bond pricing agents' decision processes. Figure 2.10 shows the time series of the implied five year bond term premium. This maturity was shown because it is in the middle of the maturities, but other maturities reflected similar processes with larger term premia at the longer end of the yield curve and smaller term premia at the shorter end of the yield curve. Again, each plot is taken from a single estimated model. The addition of disagreement to the pricing kernel in the third plot and volatility in the fourth plot do not reveal any noticeable changes for most of the observation period. The only exceptions are in the recession periods during 2001 and 2008-2009, highlighted in red.

In the 2001 recession, the change comes in the form of a double dip rather than a single dip with the addition of disagreement. In both the 2001 and 2007-2009 recession, the term premium is overall higher during the recession periods with the addition of the uncertainty proxies. This observation is made more clear in Table 2.11, which shows the mean time-varying term premium by date range, with a single estimated model per column⁷. The first row shows the entire sample period, while each subsequent row shows the mean term premium during an expansion or recession as defined by the Bureau of Labor Statistics. It is clear to see that there is not a large difference in the time-varying term premium with the addition of disagreement or volatility in the entire sample period or the expansion periods. A significant difference in the term premium only arises with the addition of disagreement and volatility in the recession periods. With the addition of these uncertainty proxies, the term premium is on average 34-36 basis points higher in the first recession and 27-32 basis points higher in the second recession compared to the baseline model with Eurodollar futures. In the context of this model, disagreement and volatility offer meaningful information for bond pricing agents beyond that contained in Eurodollar futures, especially during recessionary periods. This observation suggests that not only do bond pricing decisions respond to these uncertainty proxies, but the response is aggravated during recessions. In the context of this study, the impact of uncertainty on term premia in the yield curve is larger during recessionary periods, but does not seem to generate meaningful differences during expansionary periods.

With this observation about the nature of the term premium, BRS's original model could underestimate the term premium during recessionary periods. This may also be true of any affine model attempting to price the yield curve with observed factors with an observation period that

⁷Analogous tables for the maximum and minimum term premium are included in the Appendix in Tables C.1 and C.2. These tables largely mirror the qualitative results presented in Table 2.11

includes a recession, although this study only investigates the impact of two recessions. This inconsistency between recessions and expansions may reflect a general observation that the loadings on macroeconomic factors informing the pricing kernel may be different during expansions and recessions. During expansionary periods, government bond market agents may rely more heavily on what they perceive to be stable economic indicators of real activity such as output and inflation. While this information is not likely to be abandoned completely by these agents during recessions, disagreement and volatility may crop up as particularly influential as most agents' appetites for risk go down during recessions. This leads to the increase in the term premia seen during recessions as reflected in Table 2.11.

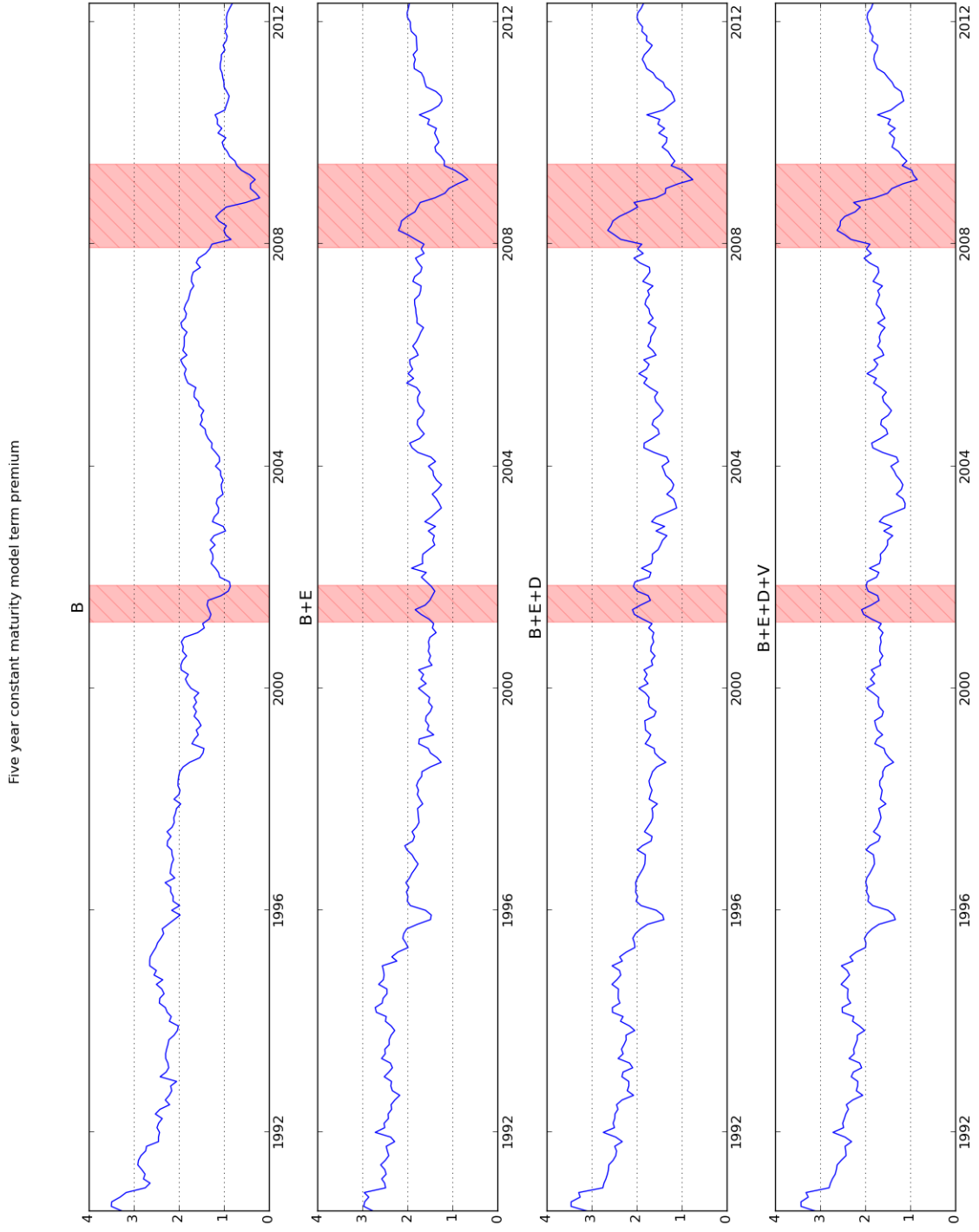


Figure 2.10: Plots of Time-varying Term Premium for Five Year Yield for Select Models. Each plot shows the term premium for an individual model.

Table 2.11: Mean Five Year Term Premium by Date Range and Model. Each row represents a date range within which the mean is calculated and each column represents an individually estimated model.

	BRS factor models		Uncertainty proxy models	
	b	b+E	b+E+D	b+E+D+V
08/90 - 05/12 (Full Sample)	1.70	1.84	1.85	1.84
03/91 - 03/01 (Expansion)	2.14	2.01	2.00	1.98
03/01 - 11/01 (Recession)	1.18	1.56	1.92	1.90
11/01 - 12/07 (Expansion)	1.46	1.67	1.61	1.60
12/07 - 06/09 (Recession)	0.73	1.52	1.79	1.84

Potentially, the most valuable contribution of this extension is drawing out the difference between types of uncertainty embedded in the premia on government bonds. While the proxies for disagreement and volatility should be able to price at least some of the short- and medium-term risk, there seems to be a fundamental increase in other risks of holding long-term bonds during the financial crisis that are not fully captured by BRS's original five factor model. As shown in Table 2.11, there is an increase in the mean time-varying term premium moving from the 2001-2007 expansion to the 2007-2009 recession that is not observed in either of BRS's original models, but is observed with the addition of the two uncertainty proxies. With disagreement and volatility measures capturing some of the movement in short-term uncertainty, the remaining unexpected risk embedded in the premium should primarily be driven by longer-term uncertainty. Moreover, this long-term risk rose when moving into the 2007-2009 recession and was imbedded in the yields on government bonds.

Accurately estimating the time-varying term premium has large implications for monetary policy, especially when a zero lower-bound on the federal funds rate is binding. In order for central bank decisions regarding large scale asset purchases and expectation management to be effective, the term premium on longer-maturity bonds must be accurately measured. It is important to understand the varied impact Federal Reserve Board decisions will have on different forms of risk and whether individual forms of monetary policy have an impact on each form differently.

2.4 Conclusion

This chapter extended BRS's five factor model into the financial crisis and illustrated the value of explicitly including measures of uncertainty in the information set driving government bond yields. Eurodollar futures offer important information for pricing government bonds in a sample including the "Great Recession" of 2007-2009. Disagreement and volatility measures were

added to the model in an attempt to proxy for economic uncertainty. Adding these uncertainty measures to the model further lowered the pricing error and produced an even higher performing model beyond that produced by only including Eurodollar futures. Including uncertainty measures also led to higher measures of the term premium on government bonds during both the 2001 and 2007-2009 recessions in comparing term premia to the four and five factor models proposed by BRS. Higher estimated term premia may result from properly accounting for changes in the loadings on observed factors in recessions, when uncertainty may play a larger role in bond market agents' information set. Properly accounting for different types of uncertainty may also prove valuable when evaluating the impact of monetary policy decisions, especially those targeting the yields on longer maturity bonds.

For future research, this investigation could continue examining the value of measures of disagreement and volatility as they inform the pricing kernel of affine models of the term structure. This information may be priced in other affine models through the use of unobserved latent factors, so correlating these observed uncertainty factors with estimated latent factor values could reveal whether affine models are unnecessarily pricing these factors as unobserved. Explicitly pricing these forms of risk could lead to higher performing models and easier interpretation of the information set driving bond market decisions. Pricing yields using observed factors could also contribute to better out of sample performance of these models.

It could also be interesting to investigate changes in the time series of the term premium after adding measures to proxy for disagreement and volatility. Structural break tests as in Banerjee et al. (1992) could reveal information as to how the data-generating process of the term premium changes or shifts when adding these observed factors. This investigation could also reveal the significance of certain events, such as Federal Reserve Board announcements, in contributing to short- versus long-term risk.

CHAPTER 3

REAL-TIME DATA AND INFORMING

AFFINE MODELS OF THE TERM

STRUCTURE

Economic agents participating in government bond markets respond to both individual and external information when participating in bond transactions. Macroeconomic indicators influence the price an agent is willing to pay for a bond of a given maturity through the effect that these conditions have on current and future bond markets. Even though any given bond buying agent may not plan on holding onto the bond for the entire maturity, they will still form their own expectations of where they think the market will be when they decide to sell the bond. This observation has led to the formal use of macroeconomic measures to inform bond yields in affine term structure models.

Affine models of the term structure are an attempt to price government bonds all along the yield curve over time. These models are estimated using assumptions about the process governing both observed and unobserved information implicit in bond-market pricing behavior. After estimating the parameters of the model, estimates of a time-varying term premium can be derived from the difference between the predicted yield and the risk-neutral yield. This term premium is the additional return required by agents to compensate for the risk of holding the bond for its maturity. The fit of these models can be examined by measuring the difference between the predicted yield and the actual yields, also known as the pricing error. In cases where macroeconomic information such as output and inflation measures are used to inform bond-pricing agents in these models, final published data is often used. Yet, macroeconomic data is often revised quarters after its original publication, so while this information represents movements in core macroeconomic measures, these final data are not the public information that were available to bond-pricing agents at the time they

made their bond buying decision. This may result in pricing errors that result from an information set used to model the yields that was not available to the agents when the yields were determined. Real-time data, the best guesses, and releases of current and recent macroeconomic measures at the time of the market decision may more accurately reflect the information entering bond-pricing decisions than final data. Through the use of the Survey of Professional Forecasters (2013) and the Real-Time Data Set for Macroeconomists (2013b), made available through the Philadelphia Federal Reserve Bank, real-time data can now easily be compiled to gain a more accurate picture of the information driving yields. The use of real-time data to inform an affine model could thus result in lower pricing errors.

This chapter will attempt to more fully address the role of real-time data in affine models of the term structure than has been addressed in the literature so far. The importance of real-time data in monetary macroeconomic models was seminally addressed in Orphanides (2001). In this investigation, Orphanides demonstrates the inability of a Taylor (1993) rule to describe target federal funds rate movement when using fully revised output and inflation rather than real-time output and inflation measures. Orphanides also extends the importance of real-time data to other macroeconomic relationships that depend on agents' perceptions of past, present, and future economic conditions. His main prescription for macroeconomic modeling is that real-time data is more appropriate than final data when modeling any sort of economic behavior that depends on agents' perceptions of economic conditions. Seminal papers in the affine term structure model literature, such as Ang and Piazzesi (2003) and Kim and Wright (2005), use final data when fitting their models to observed yield curves. The implicit assumption in these and most affine models using final macroeconomic data to inform prices and yields is that either government bond market behavior is driven by economic fundamentals and not agents' perceptions of economic fundamentals or that agents' perceptions of economic fundamentals mirror the true, revised final values. Because yields by their nature are real-time, these models are relating real-time observations of yields to movements in final data that were observed with error at the time the yields prevailed. The closest attempt to document the value of using real-time data to inform the term structure of interest rates was in a 2012 paper by Orphanides and Wei. In their paper, Orphanides and Wei attempt to generate a better-fitting term structure model through three key adjustments: 1) using real-time data, 2) modeling the pricing kernel using a VAR with rolling sample of 40 periods, and 3) additionally informing the model using survey data, leading to a better-fitting model and better out-of-sample prediction. These results are very interesting and suggest the value of real-time data

in affine models, but the result of adding real-time data alone is not explicitly addressed. This chapter will focus more explicitly on the value of real-time data alone, rather than the combined value of multiple adjustments to an affine term structure model.

By investigating the value of real-time data alone, this chapter falls into a line of papers attempting to supplement an affine term structure model with as much observed information as possible. A common practice in affine term structure modeling is to combine observed and unobserved information to explain movements in the yield curve. Explaining term structure movements with observed information has some important advantages over using unobserved factors. Observed information follows more naturally from a conception of bond markets driven by rational agents that absorb available information and base their market decisions on this information. While using unobserved information can be attractive for better performing pricing models, this information ends up serving as a catch-all for different types of information not explicitly included in the model, the value of which is difficult to derive based on the model results alone. Identifying observed information that drives bond markets allows practitioners to build a more convincing story around what information is valuable to these agents, rather than relying on unobserved information to fill in that information gap.

In order to gain a theoretical understanding of the value of unobserved factors, some practitioners often relate them back to moments of the term structure known as the “level”, “slope”, and “curvature” as in Diebold et al. (2006), and Rudebusch and Wu (2008). This approach, while leading to high performing models, does not help to build an understanding of agents’ decisions, but rather models the term structure using characteristics of the yield curve. Other practitioners correlate the unobserved information with observed macroeconomic information or information generated by structural models, such as in Ang and Piazzesi (2003) and Doh (2011). Correlating unobserved information with observed macroeconomic variables does help with understanding the decision making process of agents, but leads one to question why this observed information was not explicitly included in the information set to begin with.

In the case of Doh (2011), the author correlates the unobserved factors with the shocks (unexplained movement) generated by a dynamic-stochastic general equilibrium model (DSGE). Even though these shocks are related back to a structural model and can be linked to specific economic relationships, such as a Taylor rule or preferences in a utility specification, the approach is still relating vital term structure pricing information back to random, unexplained shocks from a structural model. If one of the end goals of term structure modeling is to gain a better understanding

of bond pricing agents' decision making process, using shocks to explain unobserved, estimated information still leaves the driving force behind this unobserved information unexplained. While unobserved factors will be introduced briefly later in this chapter, our focus will be on the value of observed, real-time information in affine models of the term structure.

The structure of this chapter is as follows. Section 3.1 will introduce the model and detail how information for a data generating process for real-time data is compiled. Section 3.2 will introduce the data, considerations of real-time data specifically, and the yields priced. Section 3.3 will present the results of estimating affine term structure models driven by both final data and real-time data and compare the performance of these models as measured by root-mean-square error (RMSE) along the relevant bond maturities used. The structure of the errors and implied term premia generated by these models will also be presented using structural break and persistence tests. This section will also make some observations about the nature of information entering bond pricing decisions at different maturities. The last section will conclude.

3.1 Model

A starting point for any affine model of the term structure is defining a data generating process to represent the macroeconomy and agents' expectations of future macroeconomic conditions. In the general case, we assume this data generating process is a vector autoregression (VAR) driven by final data:

$$X_t^{fin} = \mu^{fin} + \Phi^{fin} X_{t-1}^{fin} + \Sigma^{fin} \varepsilon_t \quad (3.1.1)$$

with p lags, where X_t^{fin} is the fully revised information for the variables in X after all major revisions have been reflected in the data. In this case, these are the final release results for X_t as of the writing of this chapter (Q1 2014). μ^{fin} is a vector of constants, Φ^{fin} is a coefficient matrix, and Σ^{fin} is a cross equation variance-covariance matrix, with ε_t assumed $\mathcal{N}(0, 1)$. It is assumed that agents solve forward for X_{t+i}^{fin} with $i \geq 1$ using the vector of constants μ^{fin} and the coefficient matrix Φ^{fin} . The VAR form is a common choice for modeling the information set governing bond markets because it is mathematically tractable and imposes few explicit restrictions. The vector X_t and its movement summarized by the VAR is assumed the complete information set governing the market decisions of bond buying agents through a pricing kernel.

This corresponds to the form that is commonly used in most affine models of the term structure and, as mentioned above, summarizes the movement of fundamentals and not necessarily market perceptions of fundamentals. The definition of Equation 3.1.1 is intended to serve as a comparison for the following real-time models.

In the same way we can define a VAR(n) information process governed by real-time data as:

$$X_t^\rho = \mu^\rho + \Phi^\rho X_{t-1}^\rho + \Sigma^\rho \varepsilon_t \quad (3.1.2)$$

For each t , X_t^ρ is the market expectation for the value of X during period t . Each lag of X^ρ , $X_{t-1}^\rho \cdots X_{t-p}^\rho$, is the release of that information for that lag of X available at time t . While X_t^ρ corresponds to a within-period expectation, $X_{t-1}^\rho \cdots X_{t-p}^\rho$ each correspond to individual releases, where the releases eventually become the final data, with the number of periods required to become final depending on the statistic. For example, if t is Q1 2000 and X^ρ contains output growth and inflation, then X_{t-1}^ρ is the first release of Q4 1999 output growth and inflation, available in Q1 2000. In the same way, X_{t-2}^ρ is the second release of output growth and inflation for Q3 1999.

If we are modeling with n factors, we write X_t^ρ as:

$$X_t^\rho = \begin{matrix} \text{Market expectation} \rightarrow \\ \text{Market expectation} \rightarrow \\ \text{Market expectation} \rightarrow \\ \text{Release 1} \rightarrow \\ \text{Release 1} \rightarrow \\ \text{Release 1} \rightarrow \\ \text{Release 2 through } (p-2) \rightarrow \\ \text{Release } p-1 \rightarrow \\ \text{Release } p-1 \rightarrow \\ \text{Release } p-1 \rightarrow \end{matrix} \begin{pmatrix} x_{r,t}^1 \\ \vdots \\ x_{r,t}^n \\ x_{r,t-1}^1 \\ \vdots \\ x_{r,t-1}^n \\ \vdots \\ x_{r,t-p+1}^1 \\ \vdots \\ x_{r,t-p+1}^n \end{pmatrix} \quad (3.1.3)$$

with the appropriate elements labeled based on their source, r referring to the period in which the values were observed and t the period of their occurrence. The elements for the current period (r and t) are based on expectations as the time period has yet to transpire and no releases of data are available. All elements for previous periods (r and $t-i$, $i \geq 1$) refer to the release of the statistic

available at r . For example, if the current observation is 2012 Q4 and x^1 is output growth, then $x_{r,t}^1$ is the market expectation for 2012 Q4 output growth in 2012 Q4 and $x_{r,t-1}^1$ is the first release of 2012 Q3 output growth. In the same way, we can write X_{t-1}^ρ as the stacked Release 1 through Release p values.

An important assumption in the construction of the real-time process is that bond pricing agents do not distinguish between adjustments to values because of information lag and adjustments to values because of changes in calculation. In each reference period, r , they take the available releases of estimates of previous period macroeconomic measures as the only information explicitly driving bond market behavior, ignoring the values that they used in previous periods. While the parameters of the data-generating process are estimated by using the entire real-time information set together, the information set of observed economic information is completely updated at each r . In other words, any fundamental changes to calculations of macroeconomic measures included in the model immediately replace the information used on both sides of Equation 3.1.2 in the quarter of the change. This is a major departure from a conventional VAR in that values are not repeated across rows in the dataset. While the impact of changes to calculations versus data revisions may have separate effects on bond markets, decomposing this effect is beyond the scope of this chapter.

The degree of departure that this real-time process has from a conventional final data driven process will be mainly driven by the frequency of updates made to the statistics. For example, Table 3.1 shows the first, second, third, and final releases of real GDP growth and civilian unemployment for Q3 1996. As can be seen, real GDP growth experiences significant revisions in every quarter while unemployment does not receive any. While there are cases where there are revisions to unemployment, they are much less common than revisions to GDP growth. This is important when comparing final and real-time processes, because some macroeconomic variables may not experience many revisions and are likely to generate very similar estimated processes as final release data. If the role of real-time data is to be tested, it is important that the variables experience enough revisions in both size and frequency to offer meaningfully different information.

Table 3.1: Quarterly Releases of Real GDP Growth and Civilian Unemployment for Q3 1996

Release	Real GDP Growth (%)	Unemployment (%)
1	2.1530	5.4
2	2.0776	5.4
3	2.0893	5.4
Final	1.0280	5.4

In addition to revisions resulting from the information gathering process, there are also en masse revisions when the calculation of the select measure changes (i.e GNP, GDP). In the final data case, there is a single calculation for each statistic and when the calculation changes, updates are made to the entire series, so for any given extract one calculation is used. In the real-time case, each statistic is observed using the calculation used in that period. This is the calculation that is used when those releases were observed. Any adjustments to the calculation of that statistic are applied to the releases available in that period, but not to any prior releases for the same statistic referencing the same period's value. This is important to keep in mind, because for every r that the real-time VAR in Equation 3.1.2 is estimated, the calculation used at r is applied for every release of the statistics observed in r . Any change in the calculation of a statistic is only applied to the values that are observed in the period of the change. As stated above, we assume that agents do not distinguish between types of revisions, but simply take whatever release is available in the period of observation.

In either the final (3.1.1) or real-time (3.1.2) driven process, μ is an $np \times 1$ vector of constants:

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{matrix} n \times 1 \\ n(p-1) \times 1 \end{matrix} \quad (3.1.4)$$

where n is the number of factors and p the number of lags, with the first n elements μ_1 through μ_n estimated and all elements below the n th element set to 0. In the same way, we can write Φ as:

$$\Phi = \left(\begin{array}{cccccccc|ccc} \Phi_{1,1,t-1} & \Phi_{1,2,t-1} & \cdots & \Phi_{1,n,t-1} & \Phi_{1,1,t-2} & \cdots & \Phi_{1,n,t-2} & \cdots & \Phi_{1,1,t-p} & \cdots & \Phi_{1,n,t-p} \\ \Phi_{2,1,t-1} & \Phi_{2,2,t-1} & \cdots & \Phi_{2,n,t-1} & \Phi_{2,1,t-2} & \cdots & \Phi_{2,n,t-2} & \cdots & \Phi_{2,1,t-p} & \cdots & \Phi_{2,n,t-p} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \Phi_{n,1,t-1} & \Phi_{n,2,t-1} & \cdots & \Phi_{n,n,t-1} & \Phi_{n,1,t-2} & \cdots & \Phi_{n,n,t-2} & \cdots & \Phi_{n,1,t-p} & \cdots & \Phi_{n,n,t-p} \\ \hline & & & \mathbf{I}_{n*(p-1) \times n*(p-1)} & & & & & \mathbf{0}_{n*(p-1) \times n} & & \end{array} \right) \quad (3.1.5)$$

where the top $n \times (np)$ array is estimated, the lower left $n(p-1) \times n(p-1)$ is an identity matrix, and the lower right $n(p-1) \times n$ is a matrix of zeros. For each element in Φ , the first subscript refers to the dependent variable predicted in X_t , the second subscript refers to the independent variable in X_{t-1} , and the third value is the relevant lag. These constructions of μ and Φ are consistent across both the final (Equation 3.1.1) and real-time data (Equation 3.1.2), but the estimated components in either are estimated using with the final or real-time data respectively. Even though the shape and position of unknown elements in μ and Φ are the same across the final and real-time VAR, it is important to note the implications of the differences in their construction.

Once the data generating process for the information driving bond buying decisions is determined, the rest of the affine model can be constructed. We continue by writing the price of any zero-coupon bond of maturity m as the expected product of the pricing kernel in period $t+1$, k_{t+1} , and the same security's price one period ahead:

$$p_t^m = E_t[k_{t+1}p_{t+1}^{m-1}] \quad (3.1.6)$$

It is assumed the pricing kernel, k_t , summarizes all information entering the pricing decisions of bonds all along the yield curve and is influenced only by the factors included in X_t in Equation 3.1.1. We assume the inter-temporal movement of the pricing kernel is conditionally log-normal and a function of the one-period risk-free rate, i_t , the prices of risk, λ_t and shocks to the VAR process in $t+1$, ε_{t+1} :

$$k_{t+1} = \exp(-i_t - \frac{1}{2}\lambda_t'\lambda_t - \lambda_t'\varepsilon_{t+1}) \quad (3.1.7)$$

We define the prices of risk as a linear function of the macroeconomic factors:

$$\lambda_t = \lambda_0 + \lambda_1 X_t \quad (3.1.8)$$

where λ_0 is $np \times 1$ and λ_1 is $np \times np$. Combining Equations 3.1.6, 3.1.7, and 3.1.8 with Equations 3.1.1 or 3.1.2 depending on the process, we can write the price of any zero-coupon bond of maturity m as:

$$p_t^m = \exp(\bar{A}_m + \bar{B}_m' X_t) \quad (3.1.9)$$

where \bar{A}_m and \bar{B}_m are recursively defined as follows:

$$\begin{aligned} \bar{A}_{m+1} &= \bar{A}_m + \bar{B}_m'(\mu - \Sigma\lambda_0) + \frac{1}{2}\bar{B}_m'\Sigma\Sigma'\bar{B}_m - \delta_0 \\ \bar{B}_{m+1}' &= \bar{B}_m'(\Phi - \Sigma\lambda_1) - \delta_1' \end{aligned} \quad (3.1.10)$$

where $\bar{A}_1 = \delta_0$ and $\bar{B}_1 = \delta_1$ and δ_0 and δ_1 relate the macro factors to the one-period risk-free rate:

$$p_t^1 = \exp(\delta_0 + \delta_1 X_t) \quad (3.1.11)$$

To derive the yield, we can rewrite Equation 3.1.9 in terms of the yield:

$$y_t^m = A_m + B_m' X_t \quad (3.1.12)$$

where $A_m = -\bar{A}_m/m$ and $B_m = -\bar{B}_m/m$.

Using a set of parameters passed in to generate A_m and B_m , Equation 3.1.12 can be used to calculate the predicted yields. The difference between the left and right-hand side of the equation is the pricing error. With distinct X_t , μ , Φ , and Σ taken from either the final process (Equation 3.1.1) or the real-time process (Equation 3.1.2) along with estimates of λ_0 , λ_1 , δ_0 and δ_1 , separate

estimates of A and B in Equation 3.1.12 are used to generate the predicted term structure. The difference between the predicted and actual term structure can be used to fit the unknown elements. The estimation process will be addressed in more detail in Section 3.3. Before moving on to some of the estimated model results of comparing a final data driven information set to a real-time data driven information set, let us describe the data that we will use.

3.2 Data

This chapter only explicitly uses macroeconomic indicators to inform the term structure. Across the models estimated, four different macroeconomic measures are used: output growth, inflation, residential investment, and unemployment. All final data are quarterly and are obtained from the Federal Reserve Bank of St. Louis (2013) website.¹ Output is measured as quarter over quarter annualized GNP/GDP growth throughout the observation period. Growth is used rather than the output level in order for there to be consistency in the measure between the final and real-time data, as consistent level information was not available across the two real-time data sources. Inflation is measured as the quarter over quarter percentage change in the GNP/GDP deflator. The change from GNP to GDP takes place in 1992. Residential investment is measured as the quarter over quarter annualized percentage change in private residential fixed investment. Unemployment is civilian unemployment. All four indicators are seasonally adjusted in both final and real-time data, as only seasonally adjusted was available consistently across both the final and real-time data.

Real-time data is taken from a combination of data from the Survey of Professional Forecasters (SPF) (2013) and the Real-Time Data Set for Macroeconomists (RTDS) (2013b) compiled by the Philadelphia Federal Reserve Bank. The American Statistical Association (ASA) started administering the SPF in 1968, asking a panelist of forecasters to submit their predictions for current quarter and up to 5 quarters in the future of key macroeconomic indicators, as well as predictions for the current and next calendar year, all seasonally adjusted. This makes the survey data particularly attractive for use in forecasting models, as it does not suffer from the fixed horizon issues that surveys such as the Blue Chip Financial Forecasts survey does.² The output and output price indices are seasonally adjusted after collection by the ASA. The main drawback of the SPF is that it is available only at a quarterly frequency, while other surveys, such as the Blue Chip survey, are available at a monthly frequency. For each data point, the median, mean, cross-sectional dispersion,

¹<http://research.stlouisfed.org/fred2/>

²For a discussion of these issues, see Chapter 2.

and individual forecasts are available. The median expectation was chosen to represent the expectation of the current quarter value, the ‘Market Expectation’ in Equation 3.1.3. The median was chosen over the mean because, unlike the mean, it is robust to outliers. It was also chosen over the cross-sectional dispersion or other quantiles that could be generated from the individual forecasts because a single estimate was needed to make it comparable to the final-data driven models. As published data for any macroeconomic measure is not available until after the completion of the quarter, this within-quarter median forecast is taken as a reasonable approximation of the market’s view of what that measure will be at the end of the period.

An alternative to SPF current quarter forecasts is the Federal Reserve internal “Greenbook” data set (2013a), also supplied by the Philadelphia Federal Reserve Bank. Greenbook data are the internal best-guess values for macroeconomic measure coincident with Federal Open Market Committee (FOMC) meetings. These data were also considered as a replacement for the SPF data, but were rejected for three reasons. First, the information is only available to those involved in FOMC decision discussions and hence are not publicly available. Second, the Greenbook current quarter forecasts for the measure is quite similar to the SPF within quarter forecasts and wouldn’t likely alter the qualitative results of this study.

Figure 3.1 shows the time series of SPF and Greenbook within quarter output growth. Visually, the two series follow a similar pattern. When the final data is included in the plot shown in Figure 3.2, it is clear to see that the difference between the real-time and final data is much larger than the difference between the two real-time series. When the real-time data after-the-fact differs from the final data, the two series both seem to differ in the same manner. Table 3.2 shows that when comparing the mean, standard deviation, and median of the output growth and inflation measures, the SPF and Greenbook statistics are very similar. When comparing either of these real-time output growth measures to the final release measure, there is a larger difference between the two. Combining these descriptive statistics with Figures 3.1 and 3.2 shows that the two real-time statistics follow a similar process, especially when the final data is used as a point of comparison. Third, Greenbook data is available only five years after the FOMC meeting in which they were used. At the current chapter’s time of writing, this would exclude much of the financial crisis of 2007-2008 and all of the Great Recession of 2008-2009. Using the SPF data allows the financial crisis and resulting downturn in growth to be included as part of the model. Because of these key differences between Greenbook and SPF data, SPF current quarter forecasts are used in favor of Greenbook current quarter forecasts in the estimation of the models.

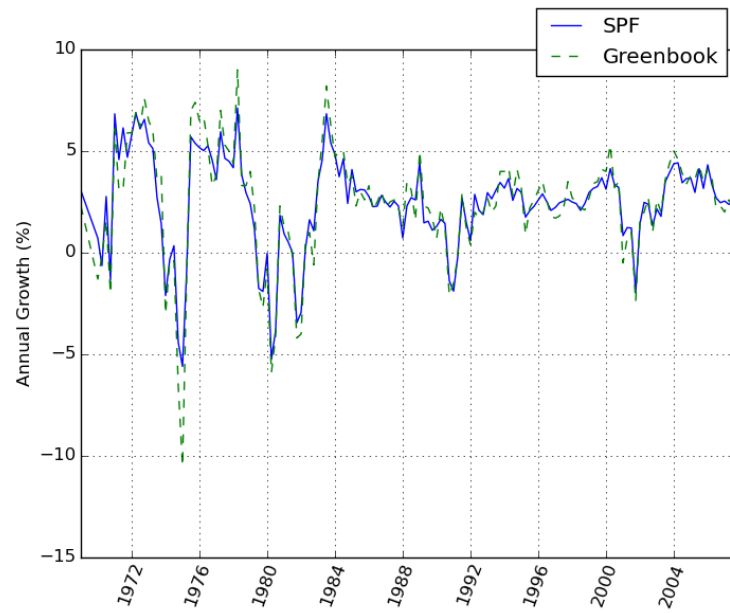


Figure 3.1: SPF and Greenbook Output Growth Statistics. SPF and Greenbook are both for within the quarter queried. Series switches from GNP to GDP in 1992.

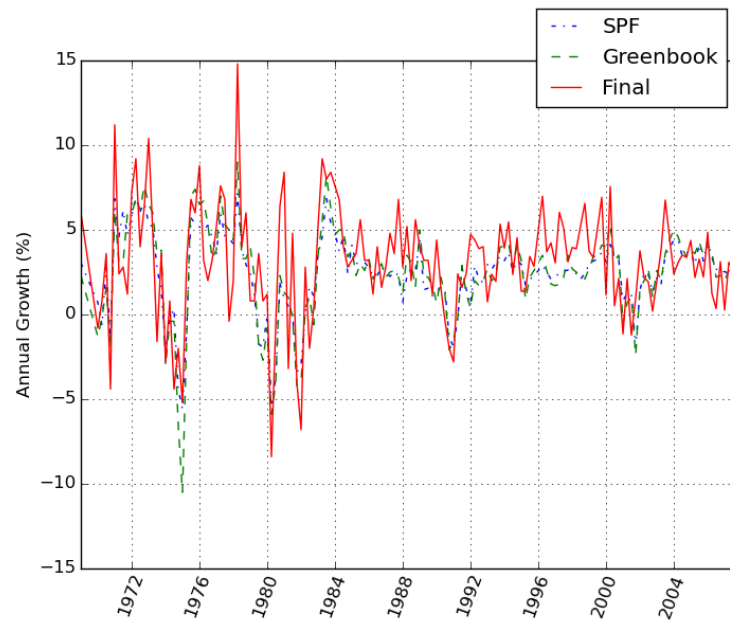


Figure 3.2: SPF, Greenbook, and Final Output Growth Statistics. SPF and Greenbook are both for within the quarter queried. Series switches from GNP to GDP in 1992.

Table 3.2: Descriptive Statistics for Output Growth and Inflation as Measured by the Median Survey of Professional Forecasters within Quarter Statistic, the Greenbook Current Quarter Statistic, and the Final Release Statistic. Data are quarterly from 1969:Q1 to 2007:Q4 as Greenbook is only available at a five year lag.

	SPF	Greenbook	Final
Output Growth			
mean	2.435	2.472	3.089
std	2.301	2.807	3.316
min	-5.598	-10.500	-8.400
25%	1.489	1.575	1.390
50%	2.585	2.600	3.200
75%	3.655	3.925	4.813
max	7.120	9.000	14.800
Inflation			
mean	3.925	4.030	3.959
std	2.223	2.572	2.479
min	1.183	0.400	0.652
25%	2.233	2.100	2.186
50%	3.247	3.300	3.098
75%	4.718	5.200	5.266
max	10.233	12.400	11.781

Previous quarter real-time information comes from the Real Time Data Set for Macroeconomists, provided by the Federal Reserve Bank of Philadelphia. This data set is compiled by manual collection of releases of macroeconomic measurements from public sources available in any given quarter. For every time period t , the releases for macroeconomic measures in $t - 1$, $t - 2$, ... available at time t is recorded. This leads to a unique time series of values for 1 to t for every t . For clarity, Table 3.3 shows an extract of real GNP. Each row signifies a statistic for a single quarter. Each column is the period in which the information is observed. Along a single row moving from left to right, another release arrives and the observation is revised. The release number is also indicated in parenthesis next to the statistic. Each diagonal represents a single release, with the first populated diagonal the first release, the diagonal above that the second release, and so on. The details of the methodology of how this data set was compiled is addressed in Croushore and Stark (2001). These data, along with the SPF median within-quarter forecast, fill out the other elements in Equation 3.1.3.

When estimating the models, quarterly data from 1969 to 2012 will be considered. 2013 data was available but the final data for 2013 had not yet passed through the major revisions and as a result were excluded. There are a few important characteristics of the data over this period. Table 3.4 offers descriptive statistics for the output and inflation measures in the final and real-time data.

Table 3.3: Sample of Real-time Data Set for Macroeconomists Real GNP

Occurrence period	Release (#)			
	1965:Q4	1966:Q1	1966:Q2	1966:Q3
1965:Q3	609.1 (1)	613.0 (2)	613.0 (3)	618.2 (4)
1965:Q4		621.7 (1)	624.4 (2)	631.2 (3)
1966:Q1			633.8 (1)	640.5 (2)
1966:Q2				644.2 (1)

Table 3.4: Descriptive Statistics of Real-time and Final Data, Quarterly Data, 1969-2012. Real-time is measured here using the within quarter SPF median forecast.

statistic	Output Growth		Inflation	
	Real-time	Final	Real-time	Final
mean	2.306	2.810	3.632	3.636
std	2.294	3.374	2.211	2.474
min	-5.598	-8.607	0.617	-0.668
25%	1.482	1.200	2.011	1.947
50%	2.494	3.103	2.821	2.776
75%	3.473	4.539	4.358	4.680
max	7.120	14.800	10.232	11.781

The real-time within-quarter median forecasts have a lower time series standard deviation than the comparison final data values for both output and inflation. Both are computed as the standard deviation of the values over the entire observation period. If the median within quarter forecast is thought of as the market's perception, it seems as though, overall, the variation in forecasters' expectation of within quarter output and inflation is lower than the variation in the ex-post final release values.

If the results of modeling with real-time information are to be compared to the results of modeling with final information, it is important that real-time data offers information that is potentially different from that in final data. As a simple indication of the potential for this, Figure 3.3 shows a time series of the residuals of regressing real-time output on final output and a constant. Specifically, the estimated relationship is:

$$\begin{aligned}
 y_t^{fin} &= E[y_t^{fin}|I_t] + \varepsilon_t \\
 &= y_{t,t}^\rho + \varepsilon_t
 \end{aligned}
 \tag{3.2.1}$$

where $E[y_t^{fin}|I_t]$ is the expectation of final output growth given the information set I , available four quarters earlier, y^{fin} is the final annualized quarter over quarter output growth and ε is the unexplained portion. We use the SPF median within quarter forecast for economic growth for

$y_{t,t}^p$. For the purpose of this exercise, ε can also be thought of as the variation in y_t^p orthogonal to variation in $E[y_t^{fin}|I_t]$. As shown in Figure 3.3, there is a considerable amount of variation in the final data that does not coincide with the real-time data measure. The thicker line shows the 8 quarter lagged rolling mean of the residuals. On a basic level, the cyclical nature of the residuals indicate there is a pattern in the real-time series not present in the final series (or vice-versa). Recessions seem to coincide with either movements down or peaks in the rolling mean of the residuals, with the exception of the July 1981 to November 1982 recession. The pattern of these residuals are a simple indication that there is the potential for information in the real-time series distinct from the final series.

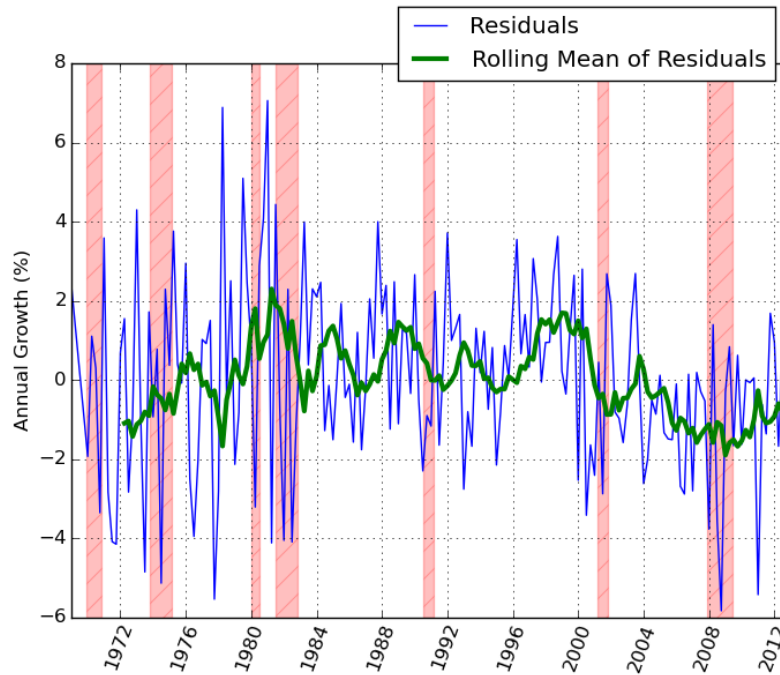


Figure 3.3: Residuals of Univariate Regression of Final Output on Real-time Output. Bold line represents 8 quarter lagged rolling average of the residuals. Highlighted areas indicate NBER (2013) recessions.

3.2.1 Yields

Yields are used to fit the relationship defined in Equation 3.1.12. In order for a single pricing kernel to recursively define yields all along a single yield curve, any differences in payouts resulting from coupons should be eliminated. Yield data are the one, two, three, four, and five year implied Fama Bliss zero-coupon yields. These yields are generated from the method described in Fama

and Bliss (1987), where yields are selected from the observed term structure and transformed into their zero-coupon form. Before presenting the modeling results, it is important to ensure that the time series of yields can be modeled using a single set of parameters. Structural break tests can help to inform this decision. If there are any structural breaks in the process governing the yields that are not modeled in the macroeconomic information used to predict the yields, separate models depending on the time period may be required. The sequential structural break approach of Banerjee et al. (1992) is used to test for the presence of a single structural break in the data. Each yield is modeled according to:

$$y_t = \mu_0 + \mu_1 \tau_{1t}(k) + \mu_2 t + \alpha y_{t-1} + \beta(L) \Delta y_{t-1} + \epsilon_t \quad (3.2.2)$$

where y_t is the yield in period t , μ_0 is a constant, μ_1 is the coefficient on the shift term, μ_2 is the coefficient on the time trend, α is the coefficient on the AR(1) term, and $\beta(L)$ is a lag polynomial. The shift term can be either a mean-shift or a trend-shift. In the case of a trend-shift, we model $\tau_{1t}(k)$ as:

$$\tau_{1t}(k) = (t - k) \mathbb{1}_{(t > k)} \quad (3.2.3)$$

where k is the breakpoint and $\mathbb{1}_{(t > k)}$ is 1 if the current time period t is past k , otherwise 0. We estimate Equation 3.2.2 for each k , with k ranging from 15% of T to 85% of T , where T is the total number of observations. We test with 4 lags in each process to allow for 4 quarters of persistence and to be consistent with the number of lags used in the VAR models. Figure 3.4 shows the time series of the F-statistics for a null hypothesis of no structural break $\mu_1 = 0$ in each of the yields. The three horizontal lines represent the 10%, 5% and 2.5% critical values for the F-statistic of testing whether a structural break exists. We use the continuous maximum function to determine the timing of the single structural break. All five yields show a structural break in the early 1980s. The one and two year yield break point is in Q3 1980 and the structural break in the three, four, and five year yield is in Q1 1981. This coincides with Paul Volker's time as Fed chairman when there was a concentrated effort to raise interest rates in order to stamp out inflation. As shown in Figures 3.5 and 3.6, none of the macroeconomic factors, final or real-time, are associated with the structural break in the yields. With a structural break appearing in the yields that does not appear in any of the macroeconomic factors, a model with time constant parameters may not be appropriate. Given

this observation, the term structure models will be estimated with an observation period beginning in 1982, after the structural break in the yields.

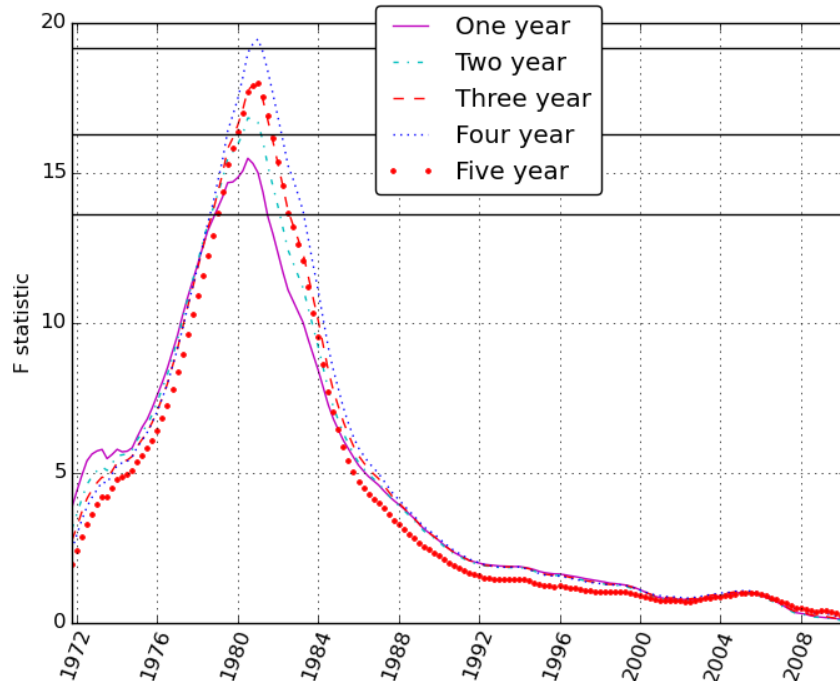


Figure 3.4: Time Series of F-statistics Used to Test for Structural Breaks in the Observed One, Two, Three, Four, and Five Year Yields. The horizontal black lines correspond to the 10%, 5% and 2.5% significance levels from bottom to top for the F-statistics taken from Banerjee et al. (1992)

3.3 Results

In the set of models below, we solve a number of models, using an information set including quarter over quarter real output growth, quarter over quarter inflation, residential investment, and unemployment. These exact measures are defined in section 3.2. We compare models using two, three, and four observed factors. The two factor model uses output growth and inflation alone, the three factor model adds residential investment, and the four factor model adds unemployment, all in that order. We use four lags in both the real-time and final data driven models in order to account for all of the real-time information and make the models comparable in their structure. We also include a three factor model with two observed factors and a single latent factor for completeness.

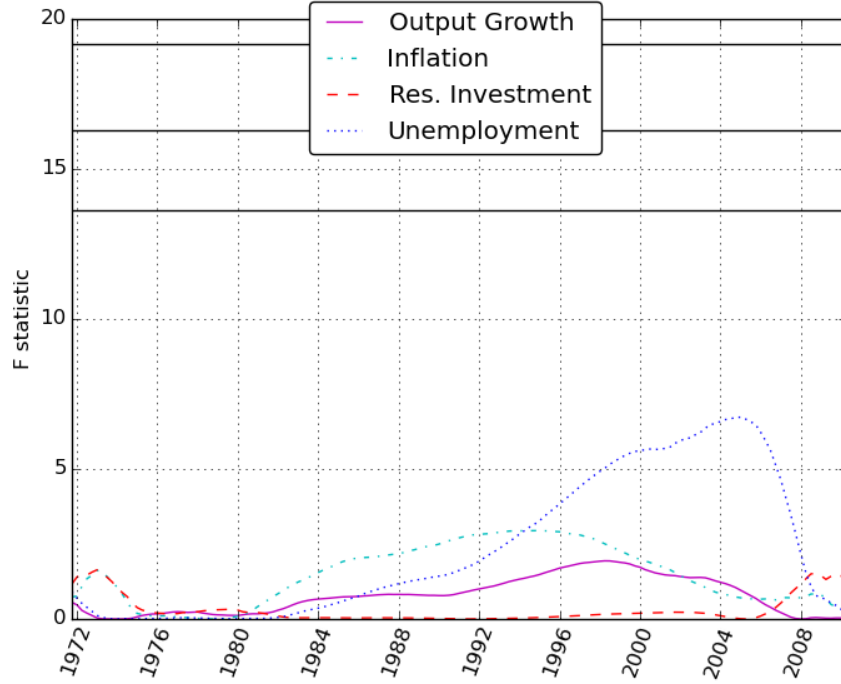


Figure 3.5: Time Series of F-statistics Used to Test for Structural Breaks in the Final Values of Output Growth, Inflation, Residential Investment, and Unemployment. The horizontal black lines correspond to the 10%, 5% and 2.5% significance levels from bottom to top for the F-statistics taken from Banerjee et al. (1992)

We make some simplifying assumptions in order to lessen the parameter space in these models. We assume that the prices of risk are non-zero only in response to the current values in X_t . With n factors, this results in block zeros below the n th element of λ_0 and outside the $n \times n$ upper left-hand block in λ_1 in Equation 3.1.8. In these models with only observed values, Equation 3.1.1 and Equation 3.1.2 can be estimated using OLS, leaving only the parameters in Equation 3.1.8 to be estimated using numerical approximation methods. Numerical approximation methods are required because there is not a closed form solution for λ_0 and λ_1 and their unknown values can only be derived based on the implied pricing error from Equation 3.1.12. Non-linear least squares is used to fit the unknown parameters in Equation 3.1.8 to minimize the sum of the square of the pricing errors, defined as:

$$\sum_m \sum_{t=1}^T (y_t^m - (A_m + B'_m X_t))^2 \quad (3.3.1)$$

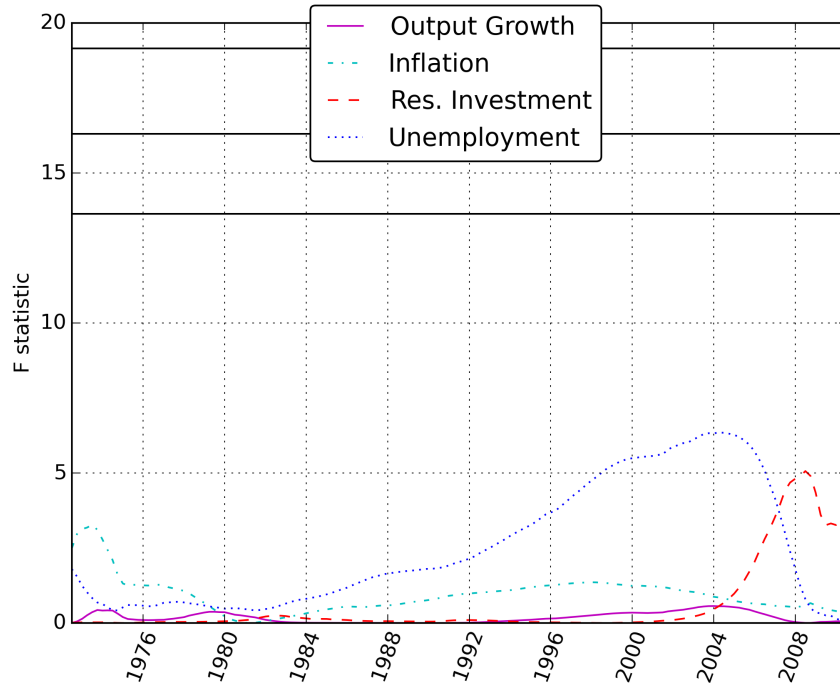


Figure 3.6: Time Series of F-statistics Used to Test for Structural Breaks in the Real-time Values of Output growth, Inflation, Residential Investment, and Unemployment. The horizontal black lines correspond to the 10%, 5% and 2.5% significance levels from bottom to top for the F-statistics taken from Banerjee et al. (1992)

where $m \in [4, 8, 12, 16, 20]$, the 5 maturities (in quarters) of zero-coupon bonds fitted in this exercise and T is the number observations. A function and parameter difference convergence threshold of 1×10^{-7} was used.

Table 3.5 presents the root-mean-square pricing error (RMSE) across multiple models, comparing models estimated with a final data process and a real-time data process. For each model, the pricing error for the yields used to fit the model is shown. The results show that there is a clear advantage, as measured by a decrease in RMSE, to modeling the information set using real-time data over final data, despite the fact there is arguably more information in the final data. Each column $F(\mathbf{n})$ and $RT(\mathbf{n})$ corresponds to the n -factor model as defined above. In the three models, the 2, 3, and 4 factors cases, the real-time model outperforms the final model. The use of multiple comparison models helps to show that it is in fact the real-time data alone that is improving the performance of these models. P-values are calculated by testing for the equivalence of means (of the

pricing errors of the two models) using a t-test that allow for different variances. These p-values are attached to the real-time columns in Table 3.5 with * for 10% and ** for 5%. As more factors are added to each model, the models improve in performance, but the advantage of using real-time data over final data increases. In the four factor models, the switch from a final process to a real-time process shows that the biggest performance difference comes in the lower maturity yields, specifically in the one and two year yields. This indicates that a real-time data driven pricing kernel is closer to the information set driving bond market decisions than a final data driven pricing kernel. This also indicates that the information may have different explanatory value at distinct ends of the yield curve. Specifically, real-time information may have more value at the shorter end of the yield curve. This particular observation will be discussed more below.

Table 3.5: RMSE for Models using Final (**F**) and Real-time (**RT**) Data. The number in parenthesis indicates the number of observed factors included and the l indicates that a single latent factor was included. Observation period is 1982-2012. *=10%, **=5%, and ***=1%, where these refer to p-values testing for the equivalence of means (of the pricing errors for the two models) using a t-test that allows for different variances.

Maturity	F (2)	RT (2)	F (3)	RT (3)	F (4)	RT (4)	F (2 + l)	RT (2 + l)
1 year	198.71	187.24	194.07	176.75	182.43	156.84**	44.25	38.21
2 years	197.35	186.51	195.67	177.25	182.84	162.33*	—	—
3 years	194.26	182.40	194.63	174.41	181.28	162.49	24.97	25.33
4 years	191.92	178.98	191.35	169.92*	178.49	162.45	43.91	40.02
5 years	189.18	175.85	186.05	166.06	173.82	159.83	57.25	55.34

For completeness an additional pair of models using final and real-time data are estimated using the first two observed factors and a single latent factor in the VAR process. Only two observed factors were included to ensure convergence of the estimated parameters. The latent factor is solved for by assuming that the two year bond is solved without error. Selecting other yields as priced without error was tried, but the two year was chosen because it struck a balance of small pricing errors for both the one year and longer maturities. An iterative solution method is used as in Ang and Piazzesi (2003), whereby initial guesses are generated for the unknown parameters holding other parameters constant and each iteration is solved via maximum likelihood. The pricing error of these models is shown in the last two columns of Table 3.5. The inclusion of the latent factor vastly decreases the pricing errors in both models, as would be expected. Another result of adding the latent factor is that the differences in the pricing error between the final and real-time models is much smaller. This may indicate that latent factor(s) in final data driven affine models of the term

structure may be compensating for the fact that real-time data is more appropriate. While the latent factor is clearly consuming much more of the pricing error than that priced by the advantage of using real-time over final data alone, this result may indicate that some of the unpriced error may be due to the inappropriate use of final data.

In order to focus on the value of final versus real-time data alone, the two four factor models driven only by observed factors will be the focus of discussion moving forward. As each column of Table 3.5 represents an individual model, there is a unique error process for each. Figure 3.7 plots the residuals of the two 4 factor models. The two processes appear very similar, confirming the close relationship between the real-time and final data.³ Upon further examination, there are some important differences in the error processes. Estimating an AR process can help to reveal differences in inter-temporal persistence in the error terms. In general, we would expect a more robust pricing kernel to better model prices and generate normally distributed errors without any inter-temporal persistence. Using BIC to determine the number of lags (AIC resulted in the same number of lags), we find that the appropriate number of lags for the final model errors was two and for the real-time models was one. An AR(2) process is also included for the real-time error in order to have a point of comparison for the final model error AR(2). The results from the estimation of these models are shown in Table 3.6. In order to get an indication of the degree of persistence in the series, we can sum the AR coefficients in each model (Andrews and Chen, 1994). These sums are provided in the last column of Table 3.6. When comparing the models using the BIC-determined number of lags, the sums of the parameters indicate a higher degree of persistence in the pricing errors generated by the final model compared to the real-time model at every one of the estimated maturities. If we compare the real-time and final data error processes using the same number of lags (2), lower persistence in the real-time model is observed in three of the five yields. The two and three year yields show close to the same persistence, while the one, four, and five year pricing errors show markedly lower persistence. While persistence is present in both the final and real-time generated errors, less persistence may indicate that there are cyclical movements in each of the yields that a real-time data informed kernel models more closely than a final-data informed kernel.

It is also interesting to note that while there is overall less persistence in the real-time data model generated pricing errors compared to those generated by the final data model, the coefficient on the first AR lag is higher with the real-time model. This may indicate that there are some

³The correlation between the four factor model final and real-time pricing error time series are 0.846, 0.891, 0.867, 0.864, and 0.850 for the one, two, three, four and five year yields, respectively.

persistent explanatory variables that are missing from the real-time data model. Because the persistence in the first lag coefficient is lower in the final data model, this may indicate that a complete model could benefit from including some final data and/or latent factors.

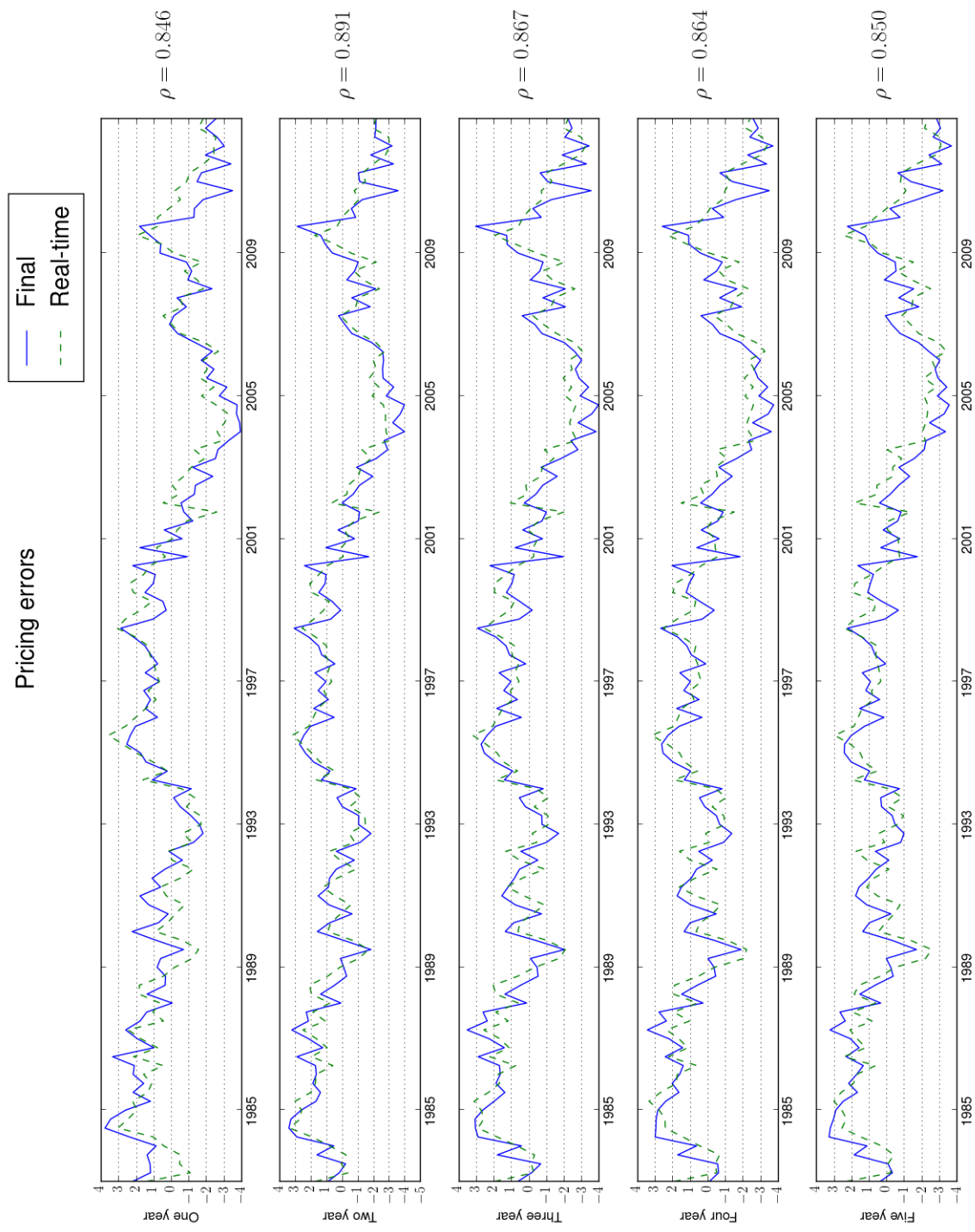


Figure 3.7: Plots of Residuals for Final (4) and Real-time (4) Models by Maturity. The correlation coefficients are shown on the right.

Table 3.6: AR Models of Pricing Errors Taken from the Four Factor Models. Number of lags selected using BIC. Parameter standard errors are shown in parenthesis. The last column shows sums of the AR coefficients.

Maturity	constant	Lag 1	Lag 2	Sum	constant	Lag 1	Sum	constant	Lag 1	Lag 2	Sum
	Final (2 lags) (BIC)				Real-time (1 lag) (BIC)			Real-time (2 lags)			
1 year	-0.0445 (0.0881)	0.5209 (0.0864)	0.3788 (0.0863)	0.900	-0.0247 (0.0827)	0.8254 (0.0528)	0.825	-0.0091 (0.0807)	0.7260 (0.0903)	0.1344 (0.0904)	0.860
2 years	-0.0286 (0.0994)	0.4895 (0.0862)	0.3820 (0.0867)	0.872	-0.0303 (0.0803)	0.8484 (0.0496)	0.848	-0.0184 (0.0792)	0.7594 (0.0912)	0.1181 (0.0914)	0.877
3 years	-0.0262 (0.1018)	0.4677 (0.0857)	0.3961 (0.0863)	0.864	-0.0328 (0.0818)	0.8410 (0.0506)	0.841	-0.0189 (0.0810)	0.7954 (0.0916)	0.0692 (0.0917)	0.865
4 years	-0.0229 (0.0970)	0.4993 (0.0865)	0.3771 (0.0875)	0.876	-0.0331 (0.0847)	0.8297 (0.0524)	0.830	-0.0178 (0.0837)	0.7872 (0.0915)	0.0676 (0.0916)	0.854
5 years	-0.0286 (0.0888)	0.5661 (0.0881)	0.3269 (0.0894)	0.893	-0.0360 (0.0827)	0.8286 (0.0520)	0.829	-0.0217 (0.0813)	0.7611 (0.0909)	0.1022 (0.0908)	0.863

In order to further investigate differences in the results of these two models, we can examine the time series of the time-varying term premium. Using the results from each fully estimated model, we can also calculate the implied term premium by taking the difference between the predicted yield and the risk-neutral yield, which is equivalent to the difference between the P-measure and Q-measure. The risk-neutral yield is the predicted yield calculated holding the prices of risk zero (λ_0 and λ_1 in Equation 3.1.8). The time-varying term premia plots are shown in Figure 3.8. While the error plots were very similar, the plots of the implied term premia show some interesting differences, specifically when comparing the shorter maturities. At the one year maturity, the time series of the term premium seems to experience much more frequent fluctuations than that modeled with the final data. There also seems to be less of a seasonal movement in the term premium. The real-time models produce more erratic movements in the term premium at the shorter maturities, with sustained seasonal movements only showing up in the longer maturities. In the final data, all of the yields seem to experience sustained seasonal positive or negative movements in the term premium.

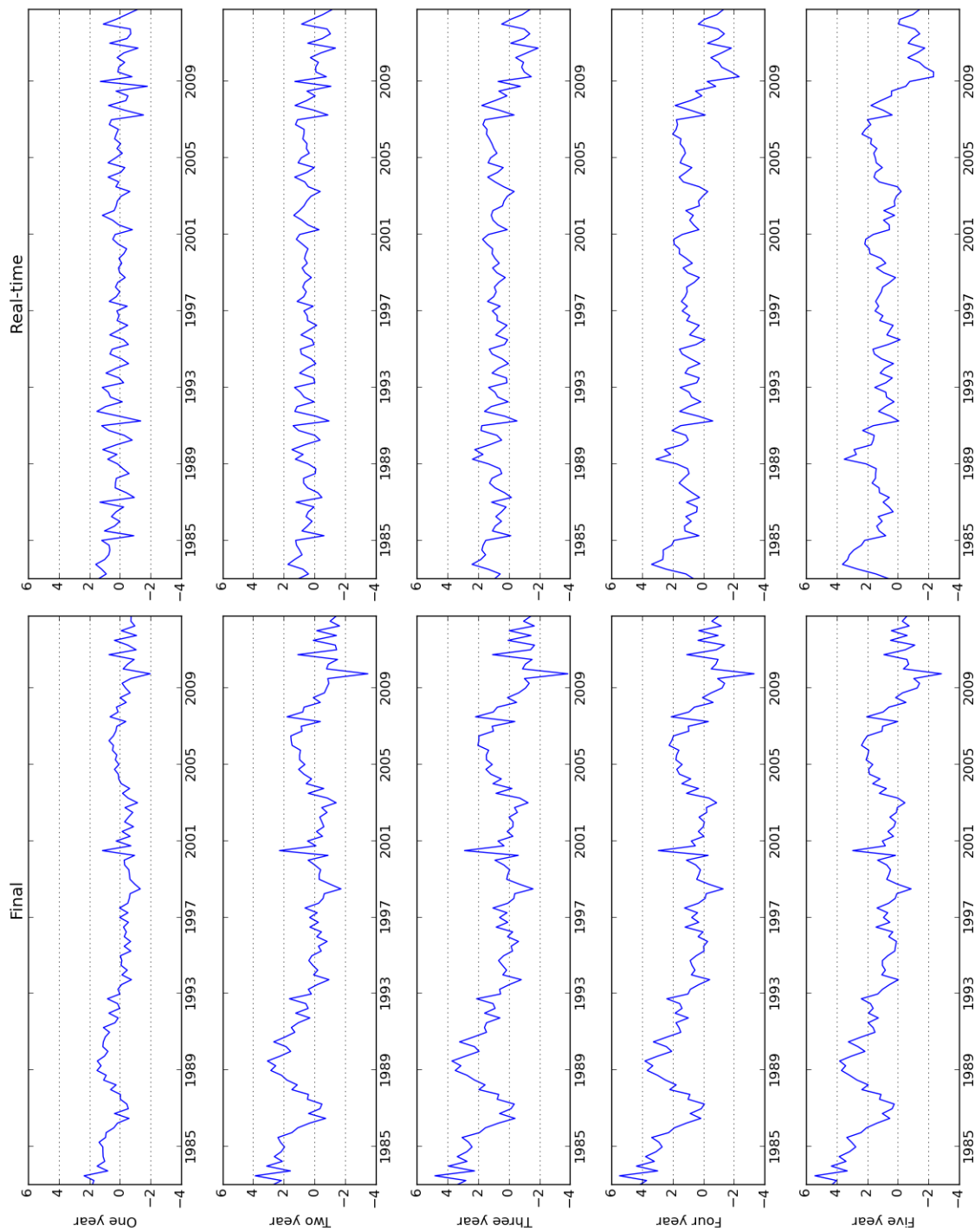


Figure 3.8: Plots of Implied Term Premium for Final (4) and Real-time (4) Models by Maturity on Left and Right Hand Side, Respectively.

These patterns in the term premium can be more formally investigated in an autocorrelation plot. Figure 3.9 presents autocorrelation plots of the time-varying implied term premium for the final and real-time models. Across the five maturities in the final model, there is persistent autocorrelation in the term premium. This pattern does not vary much according to the maturity. The autocorrelation terms are significant at the 1% level eight to ten lags back, a period of two years. The real-time model generated term premia on the other hand only develop persistence in the longer maturities. Autocorrelation estimates for the one year and two year real-time model generated term premia are not significantly different from zero at even the shortest of lags. This suggests that the term premia on the shorter maturity yields are not driven by a persistent process. For term premia associated with the three to five year maturities, the autocorrelation coefficients have a greater similarity with those generated by the final data model term premia. As we move further out the term structure to longer maturities, the term premia both become more persistent and this persistence becomes closer to that generated by the final data drive models. In fact, the five year maturity term premia for both the final and real-time models has significant autocorrelation from the 1 to the 8 quarter lag, becoming insignificant at the 8 quarter lag, even though the correlation coefficients are significantly different.

This comparison in the pattern of the term premium could indicate a difference in the pricing behavior in the markets of bonds of different maturities. The real-time portion of Figure 3.9 seems to indicate that shorter maturity bond markets respond to volatile, short-term perceived risk, while the risk attached to longer maturity yields is more consistent across periods. This follows naturally from the fact that if at least some of the agents purchasing one and two year bonds do not plan to hold them to term, the price at which they will be able to sell them will be highly dependent on the short-term economic outlook. This leads to within-period shocks to the macroeconomic factors included to inform the pricing kernel having a large impact on the perceived risk of these assets. From the results of the final and real-time models, it seems as though only real-time information can appropriately capture this risk embodied in short-term economic predictions. The model driven by final data, on the other hand, shows a very similar term premium process between all five yields, but this results in a model that does not fit as well. Given a stable VAR process governing the observed information, longer maturity premia unsurprisingly are less volatile and have a more tempered response to macroeconomic shocks. The effects of within-period shocks to the macroeconomic factors on the perceived risk of longer maturity bonds could be through the maintenance of general uncertainty about the economic horizon years ahead. In other words, while

the riskiness of short-term bonds is transitory, the riskiness of longer term bonds is more persistent. Furthermore, in these models this difference is only captured through the use of real-time data, but could often be modeled in other contexts through the use of one or more latent factors. As shown in the results in Table 3.5, a single latent factor can account for a large degree of variation in the shorter maturity yields and also lessens the advantage of using real-time over final data. Even though the use of a latent factor leads to a much tighter fitting model as measured by pricing errors across all the maturities, it also clouds the advantage of using real-time data and the subtle differences in the impact that real-time data has on shorter maturities.

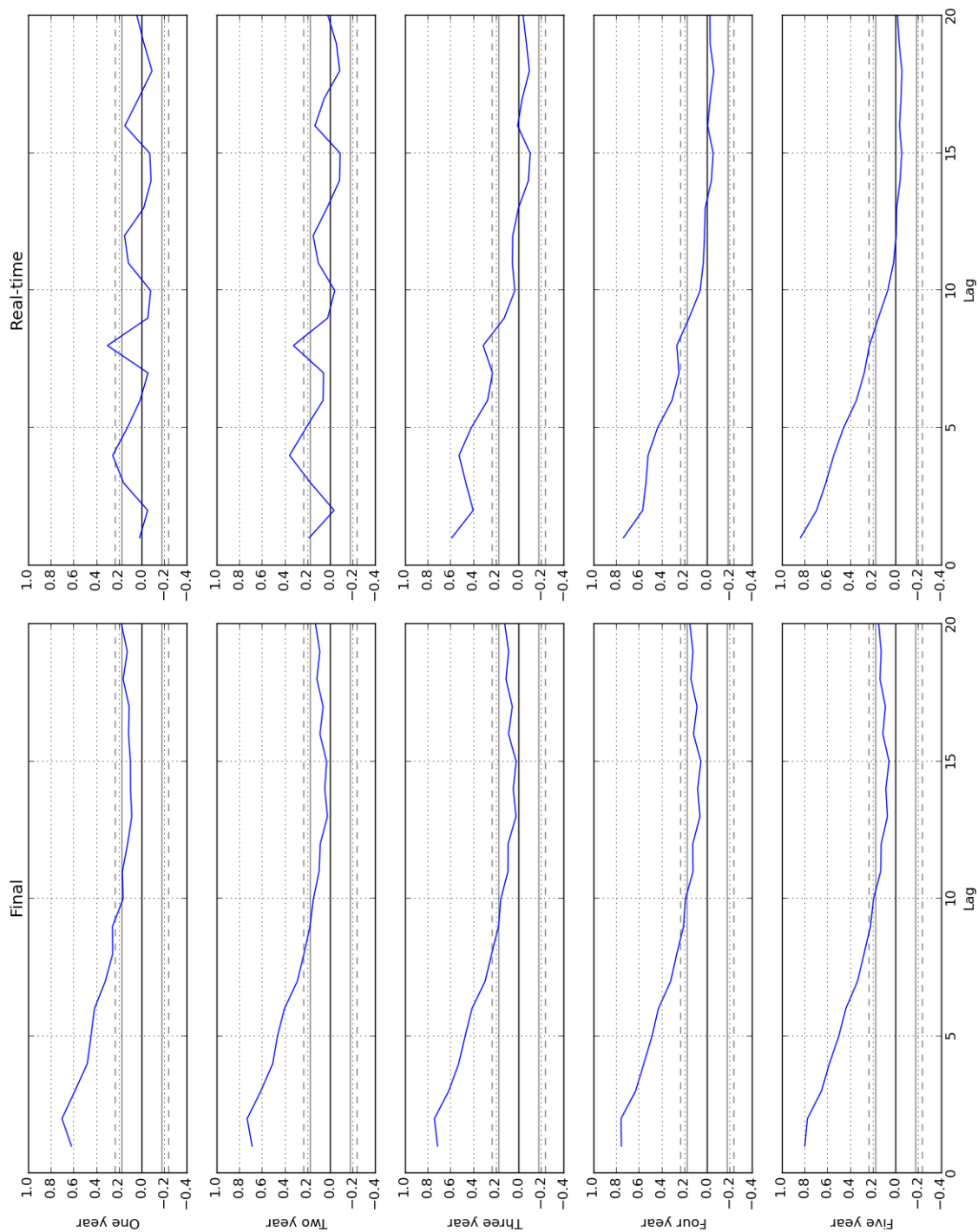


Figure 3.9: Autocorrelation Plots of Implied Time-varying Term Premium for Final (4) and Real-time (4) Models by Maturity on the Left and Right Hand Side, Respectively. The 95% confidence intervals is shown by the solid lines immediately surrounding 0 and the 99% confidence intervals are the dashed lines farther from 0.

The use of real-time data reveals a more robust structure to the differences in the risk premium of yields of different maturities. When pricing at shorter maturities, agents seem to respond more radically to changes in perceptions in macroeconomic factors and the term premium is more volatile. These perceptions are accounted for more fully through the use of real-time data. Longer maturity bond premia respond to macroeconomic factors, but through cyclical fluctuations implicit in the VAR and not through the idiosyncrasies of period-to-period movements. These should be important considerations when considering the impacts of monetary policy and attempts to manipulate the yield curve. The use of final data in term structure modeling may result in an oversimplified pricing kernel that is less robust to quick changes in perceptions of macroeconomic conditions.

3.4 Conclusion

While the use of real-time information has become very popular in most sectors of macroeconomic research, it has yet to fully penetrate affine term structure modeling of US Treasuries. This chapter shows that the use of real-time data in estimating these models reveals a fundamentally different nature to the term premia on bonds of different maturities. Estimating affine models of the term structure with real-time data resulted in lower pricing error. The real-time data driven models generated a greater variation in term premia persistence, with little to no persistence at shorter maturities and higher persistence at longer maturities. This variation was not generated by the final data driven model. Understanding what drives these differences and the different types of information contributing to the pricing decisions in bond markets of different maturities is important to predicting the impact of monetary policy intending to shape the yield curve. The use of exclusively final data can result in models that oversimplify the information driving premia and in term premia in shorter maturity bonds that are less volatile than real-time perceptions of economic measures would indicate.

Given the advantage of real-time data noted in this chapter, another avenue that this investigation could take would be to examine the ability of real-time versus final data driven affine models to forecast out-of-sample yields. While generating forecasts of the pricing kernel using final data simply involves solving forward using the VAR data generating process, this same approach is not possible with the real-time VAR specified in this chapter. While the final data VAR reuses the dependent variables as explanatory variables when forecasting future period values, the real-time data VAR does not. This makes forecasting difficult because the entire set of explanatory variables are out-of-date when the next period's value needs to be forecast. In order to obtain "real-time"

forecasts from the real-time VAR, patterns in how revisions are made to macroeconomic variables would need be modeled first. This process would then be used to generate the predictions for the real-time variables and the real-time VAR could be used to forecast out-of-sample yields.

Another area to consider for future research is whether the prominent role of latent factors in the estimation of many affine models could be an artifact of the use of final as opposed to real-time data. Results from this chapter suggest that adding even one latent factor compensates for some of the differences in pricing error generated by a final versus a real-time data driven model. Even though adding a latent factor greatly lessens the pricing error, the resulting error exhibits persistence to a similar degree found in the final data driven models. Instead, the moments of latent factors added to a final data driven model could be compared to the moments of principal components derived from a real-time data driven pricing kernel. The principal components of the real-time data driven pricing kernel may have similar properties to the latent factors estimated in combination with final data. As latent factors are becoming increasingly common, demonstrating how the information underlying latent factors relates to real-time data could lend more intuition to the descriptions of latent factors. This chapter showed how real-time data improves the performance of affine models without the use of latent factors and suggested that latent factors may compensate for a lack of real-time data in final data driven models as measured by performance.

CHAPTER 4

AN INTRODUCTION TO *AFFINE*, A PYTHON SOLVER CLASS FOR AFFINE MODELS OF THE TERM STRUCTURE

This chapter is intended to introduce and contextualize a new affine term structure modeling package, **affine**. This package consolidates a variety of approaches to estimating affine models of the term structure into a single, computational framework. Affine term structure models offer an approach for obtaining an estimate of the time-varying term premium on government bonds of various maturities, making them very attractive to those wishing to price bond risk. They also offer a method of determining what information influences government bond market agents in their pricing decisions. With the non-linear nature of these complex models, estimation is challenging, often resulting in highly customized code that is useful for estimating a specific model but not generally usable for estimating other similar models. The **affine** package is intended to provide a useful abstraction layer between the specific structure of a given affine term structure model and the components of the model that are common to all affine term structure models.

Overall, the package is designed to accomplish three main goals:

- 1) Lessen the cost of building and solving affine term structure models. Researchers in the affine term structure modeling literature build highly specialized collections of computer code that are difficult to maintain and cannot easily be adapted for use in related but separate affine models. This leads to a much smaller group of researchers to the literature than might be possible if the barrier to entry were lower.

2) Provide a meaningful computational abstraction layer for building a variety of affine term structure models. Affine term structure models come in many forms, possibly involving observed factors, unobserved latent factors, different assumptions about correlations across relationships, and different solution methodologies. This package aims to consolidate a large group of these models under a simple application programming interface (API) that will be useful to those building affine term structure models. While the theory behind affine models of the term structure has been documented across many papers in the literature, this package represents the first comprehensive computational approach for building these models in practice. This abstraction layer to affine term structure models in general is itself a new contribution to the field. The chapter will detail how models with different combinations of observed and unobserved factors in the pricing kernel, different solution methods, different numerical approximation algorithms, and different assumptions about the model structure can all be setup and built using this package.

3) Provide a single context in which many different affine models can be understood. Affine term structure models often appear self-contained, with different transformations of the same functional forms creating unnecessary differentiation in papers grounded in a single theoretical framework. This package was constructed with the intent of identifying and implementing steps to solving a model so that each can be customized by the end-user if necessary, but would continue to work seamlessly with the other parts. The unified framework allows connections between separate models to be more easily understood and compared. The framework essentially provides the building blocks for understanding and estimating an affine term structure model and the implications of certain assumptions about the functional form.

The package is written in a combination of Python and C. The application programming interface (API) is accessed entirely in Python and select components of the package are written in C for speed. The package has been tested and is currently supported on Unix-based operating systems such as Linux[®] and Macintosh[®] OS X[®] and also the Microsoft[®] Windows[®] operating system. It can be accessed entirely from a Python console or can be included in Python script. The package has hard dependencies on other Python libraries including `numpy`, `scipy`, `pandas`, and `statsmodels`. The package is currently distributed as a personal repository of the author at <https://github.com/bartbkr/affine> and is distributed under the BSD license. A proposal will be made in the future to include `affine` in `statsmodels`.

This chapter is divided into the following sections. The first section will discuss the standard affine term structure model framework and why Python was chosen as the API layer for the package.

The second section will briefly introduce the assumptions of the package about the data and the meaning of the arguments passed to the model construction and estimation objects. This can be used as a quick reference for building models. The third section describes the API in more detail, with some examples of the yield curve and factor data that are used to inform the model. This section also presents the theory behind the different estimation methods. The fourth section presents the approach behind the development, including performance issues, some profiling results, and challenges in development. The fifth section presents some scripts for executing affine term structure models in other papers in the literature, including those found in Bernanke et al. (2005) and Ang and Piazzesi (2003). Concise scripts are shown, with the full scripts included in the Appendix. The final section concludes.

4.1 A Python Framework for Affine Models of the Term Structure

Affine term structure models are a methodological tool for deriving a span of yields on securities of different maturities in terms of the information used to price those bonds. The history of affine models of the term structure begin with the assumptions laid out in Vasicek (1977). These assumptions are that: 1) the short-rate is governed by a diffusion process, specifically a Weiner process, 2) a discount bond's price is solely determined by the spot rate over its maturity, and 3) markets for the assets clear. Through these assumptions, a single factor or state variable can be derived that governs all prices of assets along the term structure. Cox et al. (1985) took the approach of Vasicek (1977) and expanded it to the case where multiple factors could be used to price the term structure. This innovation introduced by Cox et al. (1985) led to a series of papers that derived and estimated these continuous time models of the term structure. Specifically, Litterman and Scheinkman (1991), and Pearson and Sun (1994) both derive and estimated term structure models that are functions of at least two state variables, but these variables were characterized in terms of moments of the yield curve and were difficult to relate back to observed outcomes. Affine term structure models are introduced in Duffie and Kan (1996) as a subset of these models by specifying the prices of risk as an affine function of the factors.

The specification of the prices of risk as an affine transformation of the factors allows for the process governing the factors to be derived separately from the model. With the flexibility introduced by this affine specification, observed factors can be easily included. In practice, these models are estimated in discrete time and it has become common practice to explicitly include only the discrete-time specification of the models in the literature. Ang and Piazzesi (2003) introduced

a discrete-time affine term structure model, where both observed and unobserved information are included in the information governing bond markets and the process governing this information is in the form of a vector autoregression (VAR). Other important models have come in a variety of forms, including those determined solely by observed factors (Bernanke et al. 2005, Cochrane and Piazzesi 2008), solely by unobserved factors (Dai and Singleton 2002, Kim and Wright 2005), or by a combination of observed and unobserved factors (Kim and Orphanides 2005, Orphanides and Wei 2012).

In addition to the assumptions of Vasicek (1977) and the specification of an affine transformation relating the factors to the prices of risk, discrete time affine models of the term structure also assume that the pricing kernel follows a log-normal process, conditional on the prices of risk and the shocks to the process governing the factors. This assumption is made in order to make the model tractable and the pricing kernel a discrete function of the observed and unobserved components of the model.

The class of affine term structure model defined above and further specified below are that supported by the package. Modifications to the core functionality of the package to support other model types are discussed in Section 4.5.

We write the price of the security at time t maturing in n periods as the expectation at time t of the product of the pricing kernel in the next period, k_{t+1} , and the price of the same security matured one period, p_{t+1}^{n-1} :

$$p_t^n = E_t[k_{t+1}p_{t+1}^{n-1}] \quad (4.1.1)$$

This pricing kernel is defined as all information used by participants in the market to price the security beyond that defined by the maturity of the security. The pricing kernel can also be thought of as the stochastic discount factor.

In the literature, it is assumed that the pricing kernel is conditionally log-normal, a function of the one-period risk-free rate, i_t , the prices of risk, λ_t , and the unexpected innovations to the process governing the factors influencing the pricing kernel, ε_{t+1} :

$$k_{t+1} = \exp\left(-i_t - \frac{1}{2}\lambda_t'\lambda_t - \lambda_t'\varepsilon_{t+1}\right) \quad (4.1.2)$$

with λ_t $j \times 1$ where j is the number of factors used to price the term structure. Each factor is assigned an implied price of risk and these are collected in the vector λ_t . The prices of risk are estimated based on the variables included in the pricing kernel and the process specified to govern the inter-temporal movement of these variables.

In the simple case, the process governing the movement of the factors is written as a VAR(1):

$$X_t = \mu + \Phi X_{t-1} + \Sigma \varepsilon_t \quad (4.1.3)$$

where μ is a $j \times 1$ vector of constants, Φ is a $j \times j$ matrix containing the parameters on the different components of X_{t-1} , and Σ is a $j \times j$ matrix included to allow for correlations across the relationships of the individual elements of X_t . In most cases, the VAR(1) is the restructuring of a VAR(l) process with f factors, making $j = l * f$. The package allows for some flexibility in the process governing the factors, but is optimally used when the process can be simplified as a VAR(1). A VAR is commonly used in the literature to summarize the process governing the factors included in the pricing kernel for two reasons. The first reason is that a VAR is able to generate dynamics between variables without requiring the specification of a structural model. The second is that the process allows for the predicted values of the factors to be easily solved forward, where the agents are forecasting the future values of the factors and their implied contribution to the pricing kernel using the functional form specified in Equation 4.1.3. The ability to generate implied future values of the pricing kernel is essential to solving for maturities of yields all along the yield curve. X_t can be any combination of observed and unobserved (latent) factors. Observed factors are fed into the model and latent factors are recursively calculated depending on the solution method.

It is assumed that the prices of risk in time t are a linear function of the factors in time t . This assumption is what makes affine term structure models “affine”, as the prices of risk, λ_t , are an affine transformation of the factors:

$$\lambda_t = \lambda_0 + \lambda_1 X_t \quad (4.1.4)$$

where λ_0 is a $j \times 1$ vector of constants and λ_1 $j \times j$ is a parameter matrix that transforms the factors included in X_t into the risk associated with each of those factors.

In order to solve for the implied price of bonds all along the yield curve, we first define the relationship between the one period risk-free rate i_t and the factors influencing the pricing kernel:

$$i_t = \delta_0 + \delta_1' X_t \quad (4.1.5)$$

where δ_0 is 1×1 and δ_1 is a $j \times 1$ vector relating the macro factors to the one-period risk-free rate.

In order to write the price of the bond as a function of the factors and parameters of the data-generating process governing the factors, we can combine Equations 4.1.1-4.1.5¹ to write the price of any maturity zero-coupon bond as:

$$p_t^n = \exp(\bar{A}_n + \bar{B}_n' X_t) \quad (4.1.6)$$

where \bar{A}_n (1×1) and \bar{B}_n ($j \times 1$) are recursively defined as follows:

$$\begin{aligned} \bar{A}_{n+1} &= \bar{A}_n + \bar{B}_n'(\mu - \Sigma\lambda_0) + \frac{1}{2}\bar{B}_n'\Sigma\Sigma'\bar{B}_n - \delta_0 \\ \bar{B}_{n+1}' &= \bar{B}_n'(\Phi - \Sigma\lambda_1) - \delta_1' \end{aligned} \quad (4.1.7)$$

and the recursion starts with $\bar{A}_0 = \delta_0$ and $\bar{B}_1 = \delta_1$,

We can take the log of the price of a bond and divide it by the maturity of the bond to derive the continuously compounded yield, y_t^n , of a bond at any maturity:

$$\begin{aligned} y_t^n &= -\frac{\log(p_t^n)}{n} \\ &= A_n + B_n' X_t(+\epsilon_t^n) \end{aligned} \quad (4.1.8)$$

where $A_n = -\bar{A}_n/n$ and $B_n = -\bar{B}_n/n$ and ϵ_t^n is the pricing error for a bond of maturity n at time t .

This general model setup defines most discrete time affine term structure models, including the models of Chen and Scott (1993), Duffie and Kan (1996), Ang and Piazzesi (2003), Kim and Wright (2005), and Rudebusch and Wu (2008). The parameters for any given model consist of λ_0 , λ_1 , δ_0 , δ_1 , μ , Φ , and Σ . There is not a closed form solution for the unknown parameters in the model

¹For details on how these relationships are derived, see Ang and Piazzesi (2003).

given the recursive definition of A and B in Equation 4.1.8, so the unknown elements cannot be directly calculated using transformations of the set of yields, y , and factors, X . Solution methods involve a penalty function defined in terms of the pricing error, ϵ . A numerical approximation algorithm must also be chosen to determine the parameters that optimize the objective function.

In addition to the assumptions made in common with the canonical affine term structure model outlined above, the package also makes a few other theoretical assumptions:

- The data generating process for factors influencing the pricing kernel (X) can be written as a VAR(1) as in 4.1.3.
- All latent factors (if used) are ordered after observed factors in the construction of the VAR (4.1.3) governing the pricing kernel.
- In the case of Direct Maximum Likelihood, there is one yield priced without error for each latent factor.
- In the case of Kalman Maximum Likelihood, the observed factors are orthogonal to the latent factors in both the data generating process for the factors (4.1.3) and the prices of risk (4.1.4).

The first two of these assumptions are made in order to simplify the development process and could be loosened in future versions of the package. Approaches to building models where these assumptions are not appropriate are included in Section 4.5. The third and fourth assumptions refer to the specifics of two of the solution methods and are explained in more detail in Section 4.3.2.

4.1.1 Why Python?

The choice of Python as the user API layer was driven by a number of factors. First, Python is a high-level programming language that can be scripted similar to many popular linear algebra and statistical languages such as *MATLAB*® (2013), *R* (R Core Team, 2012), and *Stata*® (StataCorp, 2013). A large reason for this easy transition from other statistical languages is the existence of modules such as **numpy** (Oliphant et al., 2005–2014), **scipy** (Jones et al., 2001–2014), **pandas** (McKinney, 2005–2014), and **statsmodels** (Perktold et al., 2006–2014). These modules are all open-source and offer robust functionality for performing mathematical and statistical analysis in Python.

numpy offers a high performance linear algebra and array manipulation API very similar to *MATLAB*. Multi-dimensional arrays created using **numpy** can be manipulated by their indices and

combined with other arrays using standard linear algebra functions. **scipy**, dependent on **numpy**, ports many open-source tools for numerical integration and optimization through an easy to use API. Much of the core functionality in **scipy** comes from the ATLAS (Whaley and Petit, 2005) and LAPACK (Anderson et al., 1999) libraries, which offer high performance numerical approximation algorithms written in C and Fortran. **pandas**, a more recently developed Python module, builds in high performance DataFrame manipulation, inspired by the data-frame concept in *R*, allowing for access to elements of two-dimensional arrays by row and column labels. **statsmodels** offers basic statistical and econometric tools such as linear regression, time series methods, likelihood based approaches, sampling methods, and other tools. These modules allow for a transition to Python in the context of statistical and mathematical modeling.

Second, Python is free and open source and has a similar license in practice to the Berkeley Software Distribution (BSD), allowing it to be used in both open and closed source applications. In practice, this also makes Python free of cost, requiring only computer hardware and a modern operating system to use. It is largely platform agnostic, running on Linux, UNIX®, OS X, and Windows. Building this package in a proprietary statistical or mathematical language such as *SAS*®, *Stata*, or *MATLAB* would require a financial burden on the users of the package that would negate one of the intended purposes of this package: making affine term structure modeling accessible to a wider group of users.

Third, Python is a general purpose object-orientated programming language, a feature not shared by most statistical programming languages. This allows the package to be easily extended and modified to the need of the user. The extensibility of the package will be demonstrated in a later section of this chapter. The package can also be included in other large scope projects built in Python that may have non-statistical components such as web applications, graphical user interfaces, or distributed computing systems. These characteristics of Python make it very suitable for building package that are usable for a beginner but also extendible to one's own needs.

4.1.2 Package Logic

Before defining the API that is used to build a model, it may be useful to visualize the process that dually defines the steps through which an affine term structure model is built and estimated and the logic of the package used to initialize and solve these models. Figures 4.1 and 4.2 map the logic of this process. Figure 4.1 shows the essential arguments that must be passed to initialize a unique affine model, or in the language of the package, a unique **Affine** model

instance. These arguments include the set of yields, the factors influencing the pricing kernel, the number of latent factors, descriptors of the VAR process in Equation 4.1.3, and the structure of the parameter matrices.² These components define a single model and are fed into a single **Affine** object. Once the model object is built, the model can be solved. The solution method and numerical approximation algorithm are passed into the **solve** function. Other options related to these solution approaches are also specified here. The definition of these different solution methods and numerical approximation algorithms are specified later in the chapter. Moving on to Figure 4.2, the numerical method is selected, determining the criteria that are applied as to whether or not convergence has been achieved. Once the solution method and numerical approximation algorithm are chosen (when the **solve** method is called), the parameter set optimizing the objective function is iteratively determined. The loop in the lower half of the figure summarizes the process through which the objective function is internally calculated by the package and is included for illustrative purposes but is not modified by the end user. First, the parameter matrices, λ_0 , λ_1 , δ_0 , δ_1 , μ , Φ , and Σ are calculated using the values supplied by the numerical approximation algorithm. Then the A and B arrays are recursively calculated based on these parameter matrices. (This recursive calculation is performed either by a C or Python function depending on whether a C compiler is available. More detail on these two different approaches to calculating the same information are included in Section 4.4.) If convergence is achieved, then the parameter matrices are returned, otherwise the loop continues. There is also an implied internal dialogue that takes place to determine whether the numerical approximation algorithm generates invalid results, such as division by zero or a singular matrix. In some cases, numbers outside of the valid parameter space are passed and the process is exited.

This level of abstraction provides a comprehensive approach to building a variety of affine models of the term structure. Many of the seemingly separate approaches of affine models using different combinations of observed and unobserved factors and solution methodologies can be selected by simply passing arguments either instantiating the **Affine** object or when executing the **solve** method. Different solution methodologies can be used simply by changing these arguments rather than rewriting the entire code base, an advantage over an approach of coding every step of the estimation process.

²It is important to note that while we refer to these as matrices here, they are in fact **numpy** arrays, which support matrix operations.

Figure 4.1: Package Logic

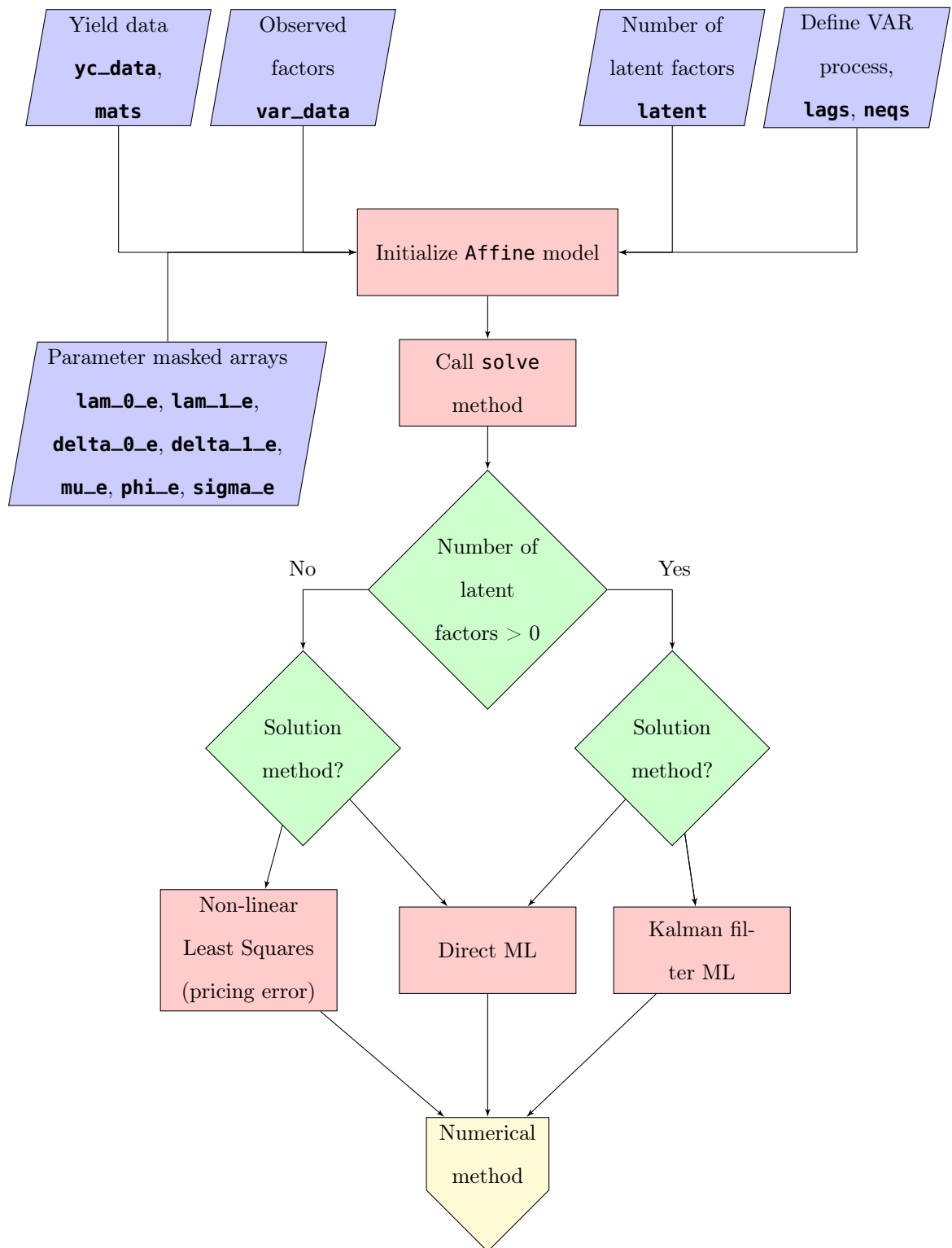
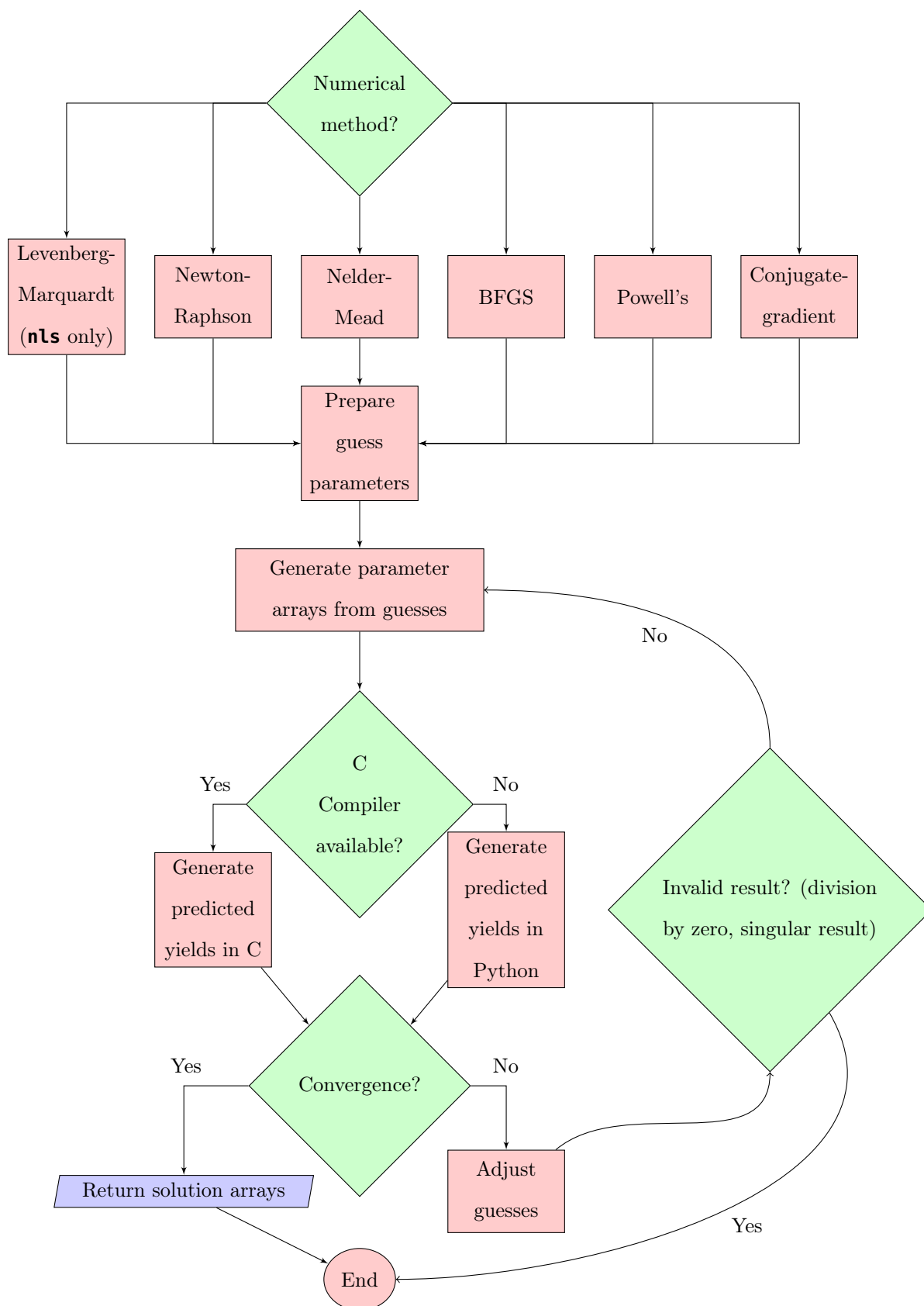


Figure 4.2: Package Logic (continued)



4.2 Assumptions of the Package

Before moving on into the details of how the package is used to build and estimate an affine term structure model, let us define the assumptions made by the package about how the data is constructed and the meaning of the arguments passed at different stages of the estimation process.

4.2.1 Data/Model Assumptions

There are two **pandas** DataFrames used to estimate the model: the yields (y in Equation 4.1.8) and the observed factors (X in Equation 4.1.3). The frequency of the observations in the observed factors and the yields must be equivalent. The DataFrame of yields is passed in through the **yc_data** argument and must have the following characteristics:

- One row per time period (month if monthly, quarter if quarterly).
- Moving forward in history from top to bottom.
- One column per yield.
- Yields are arranged in order of increasing maturity from left to right.
- All cells in the DataFrame are populated and of a numeric type.

The observed factor DataFrame is passed in through the **var_data** argument and must be organized with the following characteristics:

- One row per time period (month if monthly, quarter if quarterly).
- Moving forward in history from top to bottom.
- One column per observed factor.
- All cells in the DataFrame are populated and of a numeric type.

Both of these DataFrames are passed into the **Affine** class object. The other arguments to the **Affine** and whether they are required are:

- **lags** (required)
 - Type: integer
 - The number of lags for the VAR process specified in Equation 4.1.3.
- **neqs** (required)
 - Type: integer
 - The number of observed factors. In most cases this will be the number of columns in **var_data**.
- **mats** (required)

- Type: list of integers
- The maturities of the yields contained in `yc_data`. Each element of the list indicates the maturity of the yield in terms of the periodicity of `var_data` and `yc_data`.
- `lam_0_e` (required)
 - Type: `numpy` masked array
 - Specifies λ_0 parameter with known values filled and elements to be estimated masked.
- `lam_1_e` (required)
 - Type: `numpy` masked array
 - Specifies λ_1 parameter with known values filled and elements to be estimated masked.
- `delta_0_e` (required)
 - Type: `numpy` masked array
 - Specifies δ_0 parameter with known values filled and elements to be estimated masked.
- `delta_1_e` (required)
 - Type: `numpy` masked array
 - Specifies δ_1 parameter with known values filled and elements to be estimated masked.
- `mu_e` (required)
 - Type: `numpy` masked array
 - Specifies μ parameter with known values filled and elements to be estimated masked.
- `phi_e` (required)
 - Type: `numpy` masked array
 - Specifies Φ parameter with known values filled and elements to be estimated masked.
- `sigma_e` (required)
 - Type: `numpy` masked array
 - Specifies Σ parameter with known values filled and elements to be estimated masked.
- `latent`
 - Type: int
 - Default: 0
 - Specifies the number of latent factors to be included in the pricing kernel.
- `adjusted`
 - Type: boolean
 - Default: False
 - Specifies whether each row of `var_data` has already been transformed into an X for a VAR(1)

More detail on the implications and specification of each these arguments can be found in Section 4.3.

4.2.2 Solution Assumptions

The model is estimated by calling the `solve` method of the `Affine` model object. The first argument of the `solve` method, `guess_params`, must be a list the length of the masked elements across the parameter arguments `lam_0_e`, `lam_1_e`, `delta_0_e`, `delta_1_e`, `mu_e`, `phi_e`, and `sigma_e`. This list contains the guesses for values of the elements in these parameter arrays to begin numerical maximization of the objective function. The other parameters for the solve method define the solution method, numerical approximation algorithm, and other options applied to these methods. The solution methods where these arguments are applied are indicated by “Used In” and are ignored otherwise:

- **method** (required)
 - Type: string
 - Values: `nls` (non-linear least squares), `ml` (direct maximum likelihood), `kalman` (Kalman filter maximum likelihood)
 - Indicates solution method and objective function used to estimate model.
- **alg**
 - Type: string
 - Values: `newton` (Newton-Raphson method), `nm` (Nelder-Mead method), `bfgs` (Broyden-Fletcher-Goldfarb-Shanno algorithm), `powell` (modified Powell’s method), `cg` (non-linear conjugate gradient method), `ncg` (Newton CG method)
 - Default: `newton`
 - Used in: `ml`, `kalman`
 - Indicates numerical approximation algorithm used to optimize objective function.
- **no_err**
 - Type: list of integers
 - Used in: `ml` when `latent` \neq `False`
 - Specifies column indices of yields priced without error.
 - Zero-indexing so first column yield priced without error would be indicated with a zero.
- **maxfev**
 - Type: integer
 - Used in: `nls`, `ml`, `kalman`
 - Maximum number of function evaluations for algorithm.
- **maxiter**
 - Type: integer
 - Used in: `nls`, `ml`, `kalman`
 - Maximum number of iterations for algorithm.

- **ftol**
 - Type: float
 - Used in: `nls`, `ml`, `kalman`
 - Function value convergence criteria.
- **xtol**
 - Type: float
 - Used in: `nls`, `ml`, `kalman`
 - Parameter value convergence criteria.
- **xi10**
 - Type: list of floats
 - Used in: `kalman`
 - Starting value for Kalman latent variables.
 - Length should be number of latent factors.
- **ntrain**
 - Type: integer
 - Used in: `kalman`
 - Number of training periods for Kalman filter likelihood.
- **penalty**
 - Type: float
 - Used in: `kalman`
 - Penalty for hitting upper or lower bounds in Kalman filter likelihood
- **upperbounds, lowerbounds**
 - Type: list of floats
 - Used in: `kalman`
 - Upper and lower bounds on unknown parameters

These are the primary arguments passed into the `solve` method. Additional arguments not specified here can be passed into the method and are passed directly to the `statsmodels LikelihoodModel` object. For more information on these other arguments, please see the documentation for this method provided by `statsmodels`³. More detail on these methods and arguments can be found in Section 4.3.2.

³<http://statsmodels.sourceforge.net/devel/index.html#>

4.3 API

Let us now discuss the details of how a model would be built and estimated. For each model, a unique **Affine** class object must be instantiated. Each model is comprised of a unique set of yields, a unique set of factors used to inform these yields, a unique parameter space to be estimated, and conditions regarding whether unobserved latent factors will be estimated. The method for preparing a model is first defining the arguments required for initializing an **Affine** object. These arguments were listed in Section 4.2, but will be examined here in more detail.

yc_data is a **pandas DataFrame**⁴ of the yields, with each column signifying a single maturity. The columns of **yc_data** must be ordered by increasing maturity from left to right. Listing 4.1 shows the first rows of Fama-Bliss zero-coupon bond yields for the one, two, three, four, and five year maturities properly collected in **yc_data**. Notice how the columns are ordered from left to right in order of increasing maturity. The data must also be ordered moving forward in time from the top to the bottom of the DataFrame.

Listing 4.1: Yields DataFrame

```

1 In [2]: yc_data.head()
2 Out[2]:
3           one_yr    two_yr  three_yr  four_yr  five_yr
4 DATE
5 1952-07-01  1.923628  2.027498  2.090729  1.999381  2.139048
6 1952-10-01  1.912545  2.060032  2.148343  2.102396  2.167263
7 1953-01-01  1.973496  2.144469  2.131533  2.132738  2.267305
8 1953-04-01  2.364201  2.483449  2.194755  2.421391  2.690273
9 1953-07-01  2.291936  2.336414  2.278795  2.420576  2.715765
10
11 [5 rows x 5 columns]
```

var_data is also a DataFrame, containing the observed factors included in the VAR process specified in Equation 4.1.3 that informs the pricing kernel. In the general case, **var_data** has one column for each factor. Listing 4.2 shows how **var_data** would be structured with four observed factors: output, the price of output (**price_output**), residential investment (**resinv**), and unemployment (**unemp**).

⁴For more information on **pandas DataFrames**, see the **pandas** documentation at <http://pandas.pydata.org/pandas-docs/stable/>

Listing 4.2: Yields DataFrame

```

1 In [3]: var_data.head()
2 Out[3]:
3          output  price_output  resinv  unemp
4 DATE
5 1948-04-01  1.62508         0.88112    7.2    3.7
6 1948-07-01  0.55943         1.85048   -0.6    3.8
7 1948-10-01  0.10339         0.31250   -6.7    3.8
8 1949-01-01 -1.36232        -0.52887   -6.6    4.7
9 1949-04-01 -0.33905        -0.99782   -2.1    5.9
10
11 [5 rows x 4 columns]
```

The number of lags is specified in the **lags** argument. In this case, the number of observations in **yc_data** should be equal to the number of observations in **var_data** minus the number of lags required in the VAR. In situations where the information governing the pricing kernel does not follow a standard VAR, as in the case of a real-time estimated VAR (see Chapter 3 for the construction of a model like this), each row of **var_data** can contain current values and lagged values of each factor. In this case, the columns should be ordered in groups of lags, with each factor in the same order. Specifically, the columns should be in order:

$$x_t^1, x_t^2, \dots, x_t^f, x_{t-1}^1, \dots, x_{t-1}^f, \dots, x_{t-l}^1, \dots, x_{t-l}^f \quad (4.3.1)$$

where f is the number of observed factors and l is the number of lags. The columns are ordered from left to right going back in time. When the data are structured this way with $f * l$ columns, the **adjusted** argument should be set to **True**. In the standard VAR case, **var_data** only contains x_t^1 through x_t^f resulting in f columns (as shown in Listing 4.2) and **adjusted** is set to **False**. Of course, a standard VAR could also be passed with all current and lag columns included, so the **adjusted** flag is added for convenience. In either case, **neqs** should be set to the number of unique factors, f , used to inform the securities and **lags** should be set to the number of lags, l .

mats is a **list** of integers defining the maturities of each of the columns in **yc_data**, in a manner compatible with the frequency of the data (i.e, monthly, quarterly, annually). For example, if the model is constructed at a quarterly frequency and the columns of **yc_data** correspond to the 1, 2, 3, 4, and 5 year maturities, then **mats** would appear as in Listing 4.3.

Listing 4.3: Maturity argument

```
1 mats = [4, 8, 12, 16, 20]
```

The arguments `lam_0_e`, `lam_1_e`, `delta_0_e`, `delta_1_e`, `mu_e`, `phi_e`, and `sigma_e` are **numpy** masked arrays that are able to serve as known values of parameters defining the affine system, restrictions to the estimation process, and the location of parameters to be estimated in the arrays. These arrays can be thought of functioning like matrices in linear algebra, but they can also be one, two, or more dimensions. The name “array” primarily stems from the fact that the data within these structures are stored in C arrays, a basic C data type that allows for the storing of related data of a single type and accessed through an index.

4.3.1 Parameter Specification by Masked Arrays

As mentioned earlier, in the class of discrete-time affine term structure models addressed in this chapter, the parameters consist of λ_0 , λ_1 , δ_0 , δ_1 , μ , Φ , and Σ . These parameters map to function arguments of the initialization function, `--init--`, as:

Table 4.1: Algebraic Model Parameters Mapped to **Affine** Class Instantiation Arguments

Algebraic name	Argument Name	Dimensions	Meaning
λ_0	<code>lam_0_e</code>	$j \times 1$	Constant vector for prices of risk
λ_1	<code>lam_1_e</code>	$j \times j$	Coefficients for prices of risk
δ_0	<code>delta_0_e</code>	1×1	Constant relating factors to risk free rate
δ_1	<code>delta_1_e</code>	$j \times 1$	Coefficients relating factors risk-free rate
μ	<code>mu_e</code>	$j \times 1$	Constant vector for VAR governing factors
Φ	<code>phi_e</code>	$j \times j$	Coefficients for VAR
Σ	<code>sigma_e</code>	$j \times j$	Covariance for VAR

where the shapes of these arrays are defined with $j = f * l$, f the number of factors and l the number of lags in the VAR governing the factors informing the pricing kernel. This defines cases where X_t contains only observed factors.

The parameters are spread across these arrays. There are few cases where all of the elements of these sets of parameters are estimated, as the parameter space grows very quickly when factors are added to X_t to inform the pricing kernel. The package supports the ability, through **numpy** masked arrays, to allow only a subset of the elements in these arrays to be estimated, while others are held constant. For example, it is a common practice to restrict the prices of risk to respond to only the elements in X_t from Equation 4.1.4 that correspond to elements that actually occur in period t . For example, with X_t of shape $j \times 1$, then we might restrict the elements of λ_t in Equation

4.1.4 below the f element to zero. In this case, we would declare λ_0 and λ_1 in the script shown in Listing 4.4.

Listing 4.4: Masked array assignment

```
1 import numpy.ma as ma
2 f = 5
3 l = 4
4 j = f * l
5 lam_0_e = ma.zeros((j, 1))
6 lam_1_e = ma.zeros((j, j))
7 lam_0_e[:f, 1] = ma.masked
8 lam_1_e[:f, :f] = ma.masked
```

If we display the contents of `lam_0_e`, we see that the first f elements are “masked”, with the rest of the elements unmasked with a value of 0, as shown in Listing 4.5.⁵

Listing 4.5: Masked array access

```
1 In [1]: lambda_0
2 Out[1]:
3 masked_array(data =
4  [--]
5  [--]
6  [--]
7  [--]
8  [--]
9  [0.0]
10 [0.0]
11 [0.0]
12 ...
13 [0.0]],
14          mask =
15  [[ True]
16  [ True]
17  [ True]
18  [ True]
19  [ True]
```

⁵All variables and element examinations will be shown as they appear in IPython (Pérez and Granger, 2007). IPython is an extended Python console with many features including tab completion, in-line graphics, and many other features beyond that of the standard Python console.

```

20 [False]
21 [False]
22 [False]
23 ...
24 [False]],
25         fill_value = 1e+20)

```

For every masked **array** examined in the console, the first **array** shown is the values and the second is the masks. When the **affine** package interprets each of the seven masked arrays, it takes the masked elements as elements that need to be estimated. To be clear, elements that are masked appear as empty in the data **array** and **True** in the mask. This allows smaller parameter sets to be defined using assumptions about orthogonality between elements and other simplifying assumptions. Each of the parameter matrices must be passed in as a **numpy** masked arrays, even if all of the parameters in the **array** are set prior to the estimation process or, in the language of the package, unmasked. Guesses for starting the estimation process of the unknown values are addressed later.

The remaining undiscussed argument to the **Affine** object is **latent**, which is needed in the case of unobserved latent variables used to inform the pricing kernel. It is common practice as demonstrated in papers such as Ang and Piazzesi (2003), Kim and Wright (2005), and Kim and Orphanides (2005) to allow for the recursive definition of unobserved, latent factors in X_t . These factors are defined as statistical components of the pricing error that are drawn out through recursive definition of their implied effect on the resulting pricing error. These factors and their interpretation is discussed in more detail in Chapter 3. The **latent** argument dually identifies the inclusion of latent factors and the number of latent factors. If **latent** is **False** or **0**, then no unobserved latent factors will be estimated and X_t is solely comprised of the observed information passed in as **var_data**. In the case where **latent** > 0 , then latent factors will be estimated, according to the integer specified. It is assumed that these latent factors are ordered after any observed information in X_t . In the case of latent factors, the number of rows in λ_0 and λ_1 from Equation 4.1.4 and μ , Φ , and Σ from Equation 4.1.3 need to be increased according to the number of latent factors. If we let v be the number of unobserved latent factors included in the model and j defined as before, we can define the shape of each of the parameters as:

$$\begin{aligned}
\lambda_0 &: (j + v \times 1) \\
\lambda_1 &: (j + v) \times (j + v) \\
\delta_0 &: 1 \times 1 \\
\delta_1 &: (j + v) \times 1 \\
\mu &: (j + v) \times 1 \\
\Phi &: (j + v) \times (j + v) \\
\Sigma &: (j + v) \times (j + v)
\end{aligned} \tag{4.3.2}$$

In the case of latent factors, `var_data` is still submitted with only the observed information. The package automatically appends additional columns to `var_data` during the solution process discussed in a later section.

In both cases (with and without latent factors), after the data is imported and parameter masked arrays are defined, the **Affine** class object is created in Listing 4.6:

Listing 4.6: Affine model object instantiation

```

1 In [2]: model = Affine(yc_data=yc_data, var_data=var_data, lags=lags,
2               neqs=neqs, mats=mats, lam_0_e=lam_0_e, lam_1_e=lam_1_e,
3               delta_0_e=delta_0_e, delta_1_e=delta_1_e, mu_e=mu_e,
4               phi_e=phi_e, sigma_e=sigma_u, latent=latent)

```

Upon attempted initialization, a number of checks are applied to ensure that shapes of all of the input data and parameter masked arrays are of the appropriate size. Error messages are returned to the user if any of these consistency checks are failed and creation of the object also fails. Upon successful initialization, this method returns a class instance to which various methods and parameters are attached. In Listing 4.6, this object is defined as `model`. Any one of the input arguments can be accessed on the object after it is created. For example, if the `mats` argument defining the maturities of the yields needed to be accessed after the object has been allocated in the console, it could be done as shown in Listing 4.7.

Listing 4.7: Model attribute access

```

1 In [3]: model.mats
2 Out[3]: [4, 8, 12, 16, 20]

```

For completeness, the source for the initialization function is shown in Listing 4.8. This source code is not directly accessed by the user, but is a convenient method to reviewing what the optional and required arguments are. Required arguments appear first, do not have a default value assigned, and the order in which they are supplied matters. The requirement arguments are followed by the optional arguments that have default values which are applied when the user does not supply them.

Listing 4.8: Affine object instantiation function

```

1 class Affine(LikelihoodModel, StateSpaceModel):
2     """
3     Provides affine model of the term structure
4     """
5     def __init__(self, yc_data, var_data, lags, neqs, mats, lam_0_e, lam_1_e,
6                   delta_0_e, delta_1_e, mu_e, phi_e, sigma_e, latent=0,
7                   no_err=None, adjusted=False, use_C_extension=True):

```

4.3.2 Estimation

Once the model object is successfully instantiated, the unknown parameters can be estimated using the `solve` function of the object. The function with its arguments and documentation as they appear in the package is shown in Listing 4.9. Again, this code is included for completeness, but it is not directly accessed by the end user. For an overview of the arguments and each of their meaning and format, see Sections 4.2 and 4.3. This method takes both required and optional arguments and returns the fully estimated arrays, along with other information that defines the solution. These arguments define the method and restrictions for arriving at a unique parameter set, if possible. The starting values for the unknown elements of the parameter space are passed in as a **list** of values through the `guess_params` argument. The values specified in this list replace the unknown elements of each of the masked arrays defined in Table 4.1, in the order that they appear, and within each **array** in row-major order.⁶

Listing 4.9: Affine object estimation function

```

1         full_output=False, **kwargs):
2     """
3     Returns tuple of arrays

```

⁶Row-major refers to the order in which the elements of the arrays are internally stored, but is used here to indicate the elements are filled moving from left to right until the end of the row is reach and then the next row is jumped to.

```

4      Attempt to solve affine model based on instantiated object.
5
6      Parameters
7      -----
8      guess_params : list
9          List of starting values for parameters to be estimated
10         In row-order and ordered as masked arrays
11
12      method : string
13          solution method
14          nls = nonlinear least squares
15          ml = direct maximum likelihood
16          kalman = kalman filter derived maximum likelihood
17      alg : str {'newton', 'nm', 'bfgs', 'powell', 'cg', or 'nbg'}
18          algorithm used for numerical approximation
19          Method can be 'newton' for Newton-Raphson, 'nm' for Nelder-Mead,
20          'bfgs' for Broyden-Fletcher-Goldfarb-Shanno, 'powell' for modified
21          Powell's method, 'cg' for conjugate gradient, or 'nbg' for Newton-
22          conjugate gradient. 'method' determines which solver from
23          scipy.optimize is used. The explicit arguments in 'fit' are passed
24          to the solver. Each solver has several optional arguments that are
25          not the same across solvers. See the notes section below (or
26          scipy.optimize) for the available arguments.
27      attempts : int
28          Number of attempts to retry solving if singular matrix Exception
29          raised by Numpy
30
31      scipy.optimize params
32      maxfev : int
33          maximum number of calls to the function for solution alg
34      maxiter : int
35          maximum number of iterations to perform
36      ftol : float
37          relative error desired in sum of squares
38      xtol : float
39          relative error desired in the approximate solution
40      full_output : bool
41          non_zero to return all optional outputs
42
43      Returns

```

```

44         -----
45         Returns tuple contains each of the parameter arrays with the optimized
46         values filled in:
47         lam_0 : numpy array
48         lam_1 : numpy array
49         delta_0 : numpy array
50         delta_1 : numpy array
51         mu : numpy array
52         phi : numpy array
53         sigma : numpy array
54
55         The final A, B, and parameter set arrays used to construct the yields
56         a_solve : numpy array
57         b_solve : numpy array
58         solve_params : list
59
60         Other results are also attached, depending on the solution method
61         if 'nls':
62             solv_cov : numpy array
63                 Contains the implied covariance matrix of solve_params
64         if 'ml' and 'latent' > 0:
65             var_data_wunob : numpy
66                 The modified factor array with the unobserved factors attached
67         """
68         k_ar = self.k_ar
69         neqs = self.neqs
70         mats = self.mats

```

The **method** argument takes as a string the solution method defining the information used to determine which of a set of parameter values is closer to the true values. The options currently supported are: non-linear least squares, **nls**; direct maximum likelihood (ML), **ml**; and Kalman filter maximum likelihood, **kalman**. In cases where latent factors are added to the model, **ml** or **kalman** must be used. These two methods of calculating the likelihood also involve different assumptions about how latent factors are calculated. Specific assumptions are required to calculate the latent factors because both the parameters applied to the latent factors and the latent factors themselves are unobserved prior to estimation. The methods for calculating the unobserved factors will be discussed below in each of the method descriptions.

Non-linear Least Squares

Non-linear least squares minimizes the sum of squared pricing errors across the yields used in the estimation process and is not used in the case of latent factors. This function is formally defined as:

$$\sum_n \sum_{t=1}^T (y_t^n - (A_n + B_n' X_t))^2 \quad (4.3.3)$$

with A_n and B_n defined as in Equation 4.1.8, n choosing the maturities of yields that are used to fit the model, and T is the number of observations. This method is used in Bernanke et al. (2005) and Cochrane and Piazzesi (2008).

Unobserved factor approaches

In the case where unobserved factors are included in the pricing kernel, assumptions must be made about how these factors are calculated. Several approaches have been introduced regarding what assumptions are made to calculate the factors and two of these approaches are directly supported in the package: direct maximum likelihood and Kalman filter maximum likelihood.

The direct maximum likelihood follows directly from the term structure modeling tradition of Cox et al. (1985) and Duffie and Kan (1996), whereby only unobserved factors were used to price the term structure and by definition, these unobserved factors priced the yield curve without any error. Ang and Piazzesi (2003) extended this method to a pricing kernel composed of both observed and unobserved factors. The advantages of this method lie in the fact that a high level of precision can be achieved and that no assumptions are required concerning the starting values of the unobserved values. The disadvantages are that the parameter estimates are often highly dependent on the choice of yields priced without error. If this method is chosen, robustness tests should be performed in order to ensure that the results are not completely dependent on the yields chosen. Difficulty can also arise in direct maximum likelihood estimation because the latent factors are simultaneously estimated with the parameters applied to these latent factors. In some cases this could lead to explosive results depending on the numerical approximation algorithm used.

Maximum likelihood estimation via the Kalman filter is another method for calculating the likelihood when latent factors are included in the pricing kernel. Instead of requiring assumptions about which yields are priced without error, a Kalman state space system builds in unobserved components as part of its definition. Starting values for the latent factors combined with the

parameters values are combined to begin the recursion to solve for all values of the latent factors after the initial period. Because this method does not involve simultaneous calculation of the latent factors and the parameters applied to them, reaching a solution through numerical approximation is often times faster than in the direct maximum likelihood case. It can also be useful if the number of estimated parameters is high. Because the values of the latent factors are dependent on the starting values chosen, this can result in a loss of precision in the confidence intervals (Duffee and Stanton, 2012). If the results in the direct maximum likelihood case are very sensitive to the yields chosen to be priced without error, calculating the likelihood under the Kalman filter may result in more stable parameter estimates.

These two methods are those directly supported by the package and are discussed in greater detail below. When choosing the appropriate method, the number of free parameters, the number of latent factors, and the sensitivity of the parameter estimates to the assumptions of the likelihood should all be considered when choosing a solution method.

Direct Maximum Likelihood

In the case of direct maximum likelihood, `ml`, the log-likelihood is maximized. If any latent factors are included, they must each be matched to a yield measured without error. The number of yields used to fit the model must be greater than or equal to the number of latent factors in this case. The column indices of the yields measured without error in the `yc_data` argument must be specified in the `no_err` argument. The length of `no_err` must be equal to the number of latent variables specified during model initialization in the `latent` argument. In each of these cases, if the condition is not met, an exception is raised.

The latent factors are solved by taking both the parameters and yields estimated without error and calculating the factors that would have generated those yields given the set of parameters. Any remaining yields included in the estimation process are assumed estimated with error. This corresponds to the method prescribed in Chen and Scott (1993) and Ang and Piazzesi (2003). In the case of the yields estimated without error, we can rewrite Equation 4.1.8 as:

$$y_t^u = A_t + B_t' X_t \quad (4.3.4)$$

where u signifies a yield maturity observed without error. Once A_t and B_t are calculated for each of the y_t^u observed without error, this becomes a system of linear equations through which the unknown, latent elements of X_t can be directly solved for. Let us define E as the set of yields

priced with error. After the latent factors in X_t are implicitly defined for each t , X_t can be used to determine the pricing error for the other yields used to estimate the model:

$$y_t^e = A_t + B_t' X_t + \epsilon_t^e \quad (4.3.5)$$

where y^e signifies a yield maturity observed with error with $e \in E$ and ϵ_t is the pricing error at time t .

The likelihood is specified according to Ang and Piazzesi (2003):

$$\begin{aligned} \log(L(\theta)) = & -(T-1) \log |\det(J)| - (T-1) \frac{1}{2} \log(\det(\Sigma \Sigma')) \\ & - \frac{1}{2} \sum_{t=2}^T (X_t - \mu - \Phi X_{t-1})' (\Sigma \Sigma')^{-1} (X_t - \mu - \Phi X_{t-1}) \\ & - \frac{T-1}{2} \log \left(\sum_{e \in E} \sigma_e^2 \right) - \frac{1}{2} \sum_{t=2}^T \sum_{e \in E} \frac{(\epsilon_{t,e})^2}{\sigma_e^2} \end{aligned} \quad (4.3.6)$$

where $e \in E$ picks out the yields measured with error, $\epsilon_{t,e}$ corresponds to the resulting pricing error in Equation 4.3.5 in time t for each yield measured with error indexed by e , and σ_e is the variance of the measurement error associated with the e th yield measured with error. By definition, the number of yields measured with error is the total number of yields minus the number of yields measured without error or the length of **mats** minus the length of **no_err**.

The Jacobian of the pricing error relationships is defined as:

$$J = \begin{pmatrix} I & 0 & 0 \\ B^o & B^u & I \end{pmatrix} \quad (4.3.7)$$

where B^o is comprised of the stacked coefficient vectors for each B_n corresponding to the observed factors and B^u is comprised of the stacked coefficient vectors in B_n corresponding to the unobserved factors price without error. With γ yields priced, B^o is $\gamma \times j$ and B^u is $\gamma \times v$.

Specifically:

$$\begin{pmatrix} B^o & B^u \end{pmatrix} = \begin{pmatrix} B_{y_1}^o & B_{y_1}^u \\ B_{y_2}^o & B_{y_2}^u \\ \dots & \dots \\ B_{y_\gamma}^o & B_{y_\gamma}^u \end{pmatrix} \quad (4.3.8)$$

where each $B_{y_1}^o$ is comprised of the first j elements in B_{y_1} corresponding to the observed factors and each $B_{y_1}^u$ is comprised of the remaining v elements in B_{y_1} corresponding to the unobserved factors, with each y_1, y_2, \dots referring to the maturity of one of the γ yields priced. Each B_t^o is $1 \times j$ and each B_t^u is $1 \times v$. For a specific example, if we are using quarterly data and the one, two, three, four, and five year yields are priced, then B^o and B^u would appear as:

$$\begin{pmatrix} B^o & B^u \end{pmatrix} = \begin{pmatrix} B_4^o & B_4^u \\ B_8^o & B_8^u \\ B_{12}^o & B_{12}^u \\ B_{16}^o & B_{16}^u \\ B_{20}^o & B_{20}^u \end{pmatrix} \quad (4.3.9)$$

$$= \begin{pmatrix} B_4 \\ B_8 \\ B_{12} \\ B_{16} \\ B_{20} \end{pmatrix} \quad (4.3.10)$$

where each of B_4, B_8, B_{12}, B_{16} , and B_{20} are taken from the corresponding yield relationships in Equation 4.1.8.

Kalman Filter Maximum Likelihood

In addition to direct ML, Kalman filter derived likelihood is also available when unobserved factors are estimated. Redefining the affine system in terms of a standard state space system is relatively straightforward. After constructing the `numpy` masked arrays in Table 4.1, an observation equation is generated for each yield column in `yc_data`. We can re-use the nomenclature defined above by expanding B^o and B^u to include all yields, since the Kalman filter likelihood does not

require that any of the yields be observed without error, defining the observation equation as:

$$y_t^n = A_n + B_n^{o'} X_{t,o} + B_n^{u'} X_{t,u} + \epsilon \quad (4.3.11)$$

where B_o^n ($j \times 1$) and B_u^n ($v \times 1$) are the components of B_n corresponding to the observed and unobserved components, respectively, and ϵ is the pricing error. As noted in Section 4.2, the package assumes that the unobserved components are ordered after the observed components in the VAR system and the unobserved factors are orthogonal to the observed factors, so the state equation is written as the lower right corner ($v \times v$) portion of Φ , Φ_u :

$$X_{t+1,u} = \Phi_u X_{t,u} + \omega_{t+1} \quad (4.3.12)$$

where:

$$E(\omega_t, \omega_\tau) = \begin{cases} \Sigma_u & \text{for } t = \tau \\ \mathbf{0} & \text{otherwise} \end{cases} \quad (4.3.13)$$

and Σ_u is the lower right corner $v \times v$ portion of Σ . Values for the earliest t are initialized to begin the recursion that leads to the values for the latent factors $X_{t+r,u}$ after the initial period where r is the number of training periods. The package currently does not support estimation through the Kalman filter derived likelihood with non-zero covariance between the observed and unobserved factors.⁷ The log-likelihood for each maturity is calculated as indicated in Hamilton (1994, p. 385) and then summed over all maturities to get the total log-likelihood.

The numerical approximation algorithm for the non-linear least squares method is the Levenberg-Marquardt algorithm and is not influenced by changes to the `alg` argument, while a number of different numerical approximation algorithms can be used for both the direct and Kalman ML cases. These correspond to the different methods documented in the `scipy.optimize` module and include, but are not limited to, Newton-Raphson, Nelder-Mead, Bryoden, Fletcher, Goldfarb, and Shanno, and Powell methods. Most of these methods are accessed through the `LikelihoodModel` `statsmodels` class. The numerical approximation algorithm is chosen by the `alg` argument to the `solve` function.

⁷This is currently not possible given how the Kalman Filter likelihood is calculated. The observed factor dynamics are not included in the state space equation.

In addition to the solution method and numerical approximation algorithm, other function arguments can also be passed to the `solve` function, depending on the prior choices of solution method and numerical approximation algorithm. In the case of direct ML, the `no_err` argument is required, a **list** of indices of columns of `yc_data` assumed estimated without error. Zero-indexing is used to indicate the columns in `no_err`, with a 0 indicating that the first column be estimated without error, a 1 indicating the second column be estimated without error, etc. Zero-indexing is consistent with the rest of the Python language and any C-based language for that matter. For example, if the columns in `yc_data` are the one, two, three, four, and five year yields and the two and four year yields are estimated without error, then `no_err` can be defined as in Listing 4.10:

Listing 4.10: Assignment of columns maturities priced without error

```
1 no_err = [1, 3]
```

This would result in the two and four year yields being estimated without error and the one, three, and five year yields estimated with error. The `no_err` argument does not apply in the case of Kalman ML and even if it is passed to the `solve` function, it will be ignored.

The `xi10`, `ntrain`, and `penalty` arguments only apply to the Kalman ML method and are ignored otherwise even if they are passed. `xi10` defines the starting vector of values in the first period estimated in the state equation defined in Equation 4.3.12. This argument should be a **list** the length of the number of latent factors, equal to v and the `latent` argument in creation of the model object. `ntrain` is the number of training periods for the state space system, defining how many periods of recursion must be performed before the observations enter the calculation of the likelihood.⁸ `penalty` is a floating point number that, if supplied, determines the numerical penalty that is subtracted from the likelihood if the `upperbounds` or `lowerbounds` are hit. `upperbounds` and `lowerbounds` are both **lists** of floating point numbers whose length must be equal to the number of individual elements to be estimated across all of the parameter matrices. They define the upper bounds and lower bounds, respectively for when the `penalty` is hit for each of the parameters to be estimated.

⁸Given that the Kalman Filter recursion begins with an assumption regarding the initial value of unobserved state, a few periods simulating the system may be desired to lessen the effect of the initial state on the likelihood calculation.

Additional keyword arguments can be passed to these methods that are passed to the numerical optimization method. More detail can also be found in the `scipy optimize` documentation (Jones et al., 2001–2014).⁹

4.4 Development

Development of `affine` began with the intention of being an open-source project written in Python alone, supported by the Python modules mentioned above, namely, `numpy` (Oliphant et al., 2005–2014), `scipy` (Jones et al., 2001–2014), `pandas` (McKinney, 2005–2014), and `statsmodels` (Perktold et al., 2006–2014). Even with the many solution methods presented in the above section, performance (or lack thereof) would be a key factor to adoption in the field. As even those affine term structure models that are driven solely by observed information are still non-linear and require numerical approximation methods to solve, any steps that slow down the calculation of the objective function will inhibit performance. Details on how performance is affected and how solving that problem was approached is documented later in this section.

The general approach of optimizing every line of code will end up taking more time that it is worth. In some cases, code can be rewritten (refactored) in more efficient Python code and a desirable level of performance can be reached. In other cases, the performance issues may be a result of the high-level language nature of Python. As Python does not have static data types and performs frequent behind-the-scenes checks of implied data types, looping operations can sometimes consume more computational time than desired. In these situations, Python has a convenient feature of being able to pass objects to and from compiled C code. C is a low-level statically typed language requiring explicit memory management, but it allows for greater performance. The potential for performance increases of C over Python arise for a number of reasons.

First, static typing prevents many of the continual data type checks that Python performs behind the scenes. Static data typing forces the developer to set the data type of a single variable or an array by the time its value is assigned, defining the amount of space that a variable will take up. This prevents variables from being resized and frees up the language from needing to consistently re-evaluate the required space in memory to hold the information attached to a variable. Static typing is not available in Python unless the core language is extended with an outside package, namely, Cython (Behnel et al., 2004).

⁹The `scipy optimize` documentation can be found at <http://docs.scipy.org/doc/scipy/reference/optimize.html#>

Second, without going into too much detail, memory allocation in C allows for greater control over how information is stored. For data structures like arrays in C, there are no checks that the data entered into an **array** is within the bounds of that **array**. In contrast, a **list** can be dynamically built in Python without any bounds put on its size initially. In C, the size of an **array** must be initialized before any of its elements are assigned values, but element values can be set beyond the bounds of the **array** resulting in other addresses in random-access memory (RAM) being overwritten! C also allows explicit access to two structures in RAM, the stack and the heap. Variables allocated on the stack are quickly removed and are automatically deallocated once the scope of the variable is escaped. Variables allocated on the heap are allocated at a slightly slower pace than the stack and are not deallocated unless explicitly deleted. If variables allocated on the heap are not deallocated, the package could suffer from memory leaks. Allocation on the heap is necessary for any objects passed from C back to Python. These two RAM structures allow for any intermediate steps in our calculations to be performed using stack variables, with heap variables only used when information needs to be passed from C to Python.

Finally, pointer arithmetic in C allows for extremely high performance when iterating over arrays. If a C **array** can be assumed to be contiguous in memory, that is, occupying an uninterrupted section of memory, then this assumption can be used for a performance advantage. As an example, suppose that we wanted to create an **array** of integers that is the sum of the elements of two other arrays of the same length. We could write the operation in (at least) two ways. First, we could assign values to the summed **array** by iterating through the indices, as shown in Listing 4.11:

Listing 4.11: Explicit array iteration in C

```
1 int first_array[1000], second_array[1000], summed_array[1000];
2 /* Assign values to first_array and second_array */
3 /* ... */
4 /* Assign sum of two to summed_array */
5 for (int i = 0; i < 1000; i++) {
6     summed_array[i] = first_array[i] + second_array[i];
7 }
```

We could also perform the same operation using pointer arithmetic in Listing 4.12:

Listing 4.12: Pointer arithmetic iteration in C


```

1 int first_array[1000], second_array[1000], summed_array[1000];
2 /* Assign values to first_array and second_array */
3 /* ... */
4 /* Assign sum of two to summed_array */
5 int farray_pt = first_array;
6 int sarray_pt = second_array;
7 int sumarray_pt = summed_array;
8 for (int i = 0; i < 1000; i++) {
9     *sumarray_pt = *farray_pt + *sarray_pt;
10    sumarray_pt++;
11    farray_pt++;
12    sarray_pt++;
13 }

```

While the second option may seem more complex at first glance, it is actually more efficient. Every time that the element at a specific index is accessed, a memory address lookup operation takes place, as in `first_array[i]`. The second version of the code simply iterates over the pointers to the elements held in each of the arrays, thus allowing for quicker access. An array in C is simply a pointer to the memory address of the first element, leading to lines 5 through 7 in Listing 4.12. With each iteration, the elements in `first_array`, `second_array`, and `summed_array` are accessed through the memory address of their elements and the pointers to those addresses are incremented by the number of bytes used by the respective data type, in this case `int`. This makes full use of the fact that these arrays are allocated contiguously on the stack. This optimization becomes extremely useful when the implicit number of dimensions in an `array` is greater than one. This ability to perform pointer arithmetic in C is fully utilized in many of the C operations below in the package.

Determining what components of the package can benefit from being written in C involved some investigation. Writing in C is more difficult than Python and components should be extended into C only if there is the potential for a material performance gain. This is where code profiling tools are of great use. Code profiling tools allow developers to determine where their code is spending the most time. Given that the primary distribution of Python is written in C, the primary code profiling tool is a C extension, `cProfile`. This extension can be called when any Python script is executed and it will produce binary output that summarizes the amount of CPU time spent on

every operation performed in the code. This binary output requires other tools in order to interpret it.¹⁰

Table 4.2 is the consolidated output of profiling the `solve` function based on a model estimation process using only observed factors. Each row of the table shows a function called in the estimation process paired with the total computational time spent in that function. Only the functions with the highest total time are shown. This time reflects the amount of time spent within the function, excluding time spent in sub-functions. It is easy to see that the function that takes the most computational time is `gen_pred_coef`, which is emphasized in *italics*. A high computational time can be the result of a single execution of a function taking a long time, a single function being called many times, or a combination of both. A function that is called many times may not benefit from refactoring in C because it is taking up computational time through the fact that it is used so frequently, not because a single call is slow. In order to get a sense of which of these, the output shown in Table 4.2 can be combined with output that shows the percentage of time consumed by each call. Figure 4.3 shows the percentage of time spent on each function called by `_affine_pred`. `_affine_pred` calculates the implied yields given a set of parameters and comprises 99% of the computational time of a call to `solve`¹¹. In the figure, `gen_pred_coef` has a thicker outline and is shown to comprise 19.05% of each call to `_affine_pred`. The majority of the time in the function is spent on the `{numpy.core._dotblas.dot}` operation.

This information was used to determine what parts of the code could benefit from being written in C rather than Python. While the `{numpy.core._dotblas.dot}` function seems like a good candidate because of the amount of time spent in this function, it has already been optimized in C and thus does not qualify. The `params_to_array` function, shown in Figure 4.3, is a pure Python function, so could be a candidate. This function takes a set of parameters and generates the appropriate two dimensional arrays required to calculate the predicted yields. This function relies heavily on `numpy` masked arrays and functions providing abstract functionality in Python. Because of this dependence on abstract `numpy` functionality, it was not a good candidate for rewriting in C, as rewriting would likely involve recreating much of the core functionality already provided by `numpy`.

¹⁰RunSnakeRun (Fletcher, 2001–2013) is a popular choice for interpreting C profiling output and is easy to set up for use with Python but offers few options for displaying the output. KCacheGrind (Weidendorfer, 2002,2003) offers all of the features of RunSnakeRun and more options for output display, but involves more setup and requires the use of another tool, `pyprof2calltree` (Waller, 2013), in order to generate the appropriate output from a Python script `cProfile`.

¹¹These graphs were created using KCacheGrind (Weidendorfer, 2002,2003)

On the other hand, `gen_pred_coef` is a good candidate for passing into a C function. It requires extensive, recursive looping and only involves linear algebra operations. The code for this function is shown in Listing 4.13. This function takes the parameter arrays generated by `params_to_arrays` and generates the A_n and B_n parameters that enter into the relationship defined in Equation 4.1.8. Each of these two arrays is constructed recursively based on the set of equations specified in Equation 4.1.7. Written in pure Python, this function involves iteration and looping over multiple arrays, a series of intermediate calculations performed on multidimensional arrays, and dynamic creation of two multidimensional arrays, A_n and B_n . No matter which solution method or numerical approximation algorithm is chosen, there will be repeated instances of sets of parameters being passed into this function.

As can be seen, the `for` loop beginning in line 36 of the function iterates until the maximum maturity specified in the `mats` argument is reached. For each of these iterations, `a_pre`, `a_solve`, `b_pre`, and `b_solve` are calculated for the specific maturity, corresponding to \bar{A}_n , A_n , \bar{B}_n , and B_n , respectively, from Equations 4.1.7 and 4.1.8. A number of `array` dot products and index access operations need to be performed in each iteration. The nature of these operations and recursive form of the calculation prompted a C version of the code to be written, which is included in the Appendix in Listing D.1.

Listing 4.13: Native Python Function for generating A and B

```

1      mu : numpy array
2      phi : numpy array
3      sigma : numpy array
4
5      Returns
6      -----
7      a_solve : numpy array
8          Array of constants relating factors to yields
9      b_solve : numpy array
10         Array of coefficients relating factors to yields
11
12         """
13         max_mat = self.max_mat
14         b_width = self.k_ar * self.neqs + self.lat
15         half = float(1)/2
16         # generate predictions
17         a_pre = np.zeros((max_mat, 1))

```

```

17     a_pre[0] = -delta_0
18     b_pre = np.zeros((max_mat, b_width))
19     b_pre[0] = -delta_1[:,0]
20
21     n_inv = float(1) / np.add(range(max_mat), 1).reshape((max_mat, 1))
22     a_solve = -a_pre.copy()
23     b_solve = -b_pre.copy()
24
25     for mat in range(max_mat-1):
26         a_pre[mat + 1] = (a_pre[mat] + np.dot(b_pre[mat].T, \
27             (mu - np.dot(sigma, lam_0))) + \
28             (half)*np.dot(np.dot(np.dot(b_pre[mat].T, sigma),
29                 sigma.T), b_pre[mat]) - delta_0)[0][0]
30         a_solve[mat + 1] = -a_pre[mat + 1] * n_inv[mat + 1]
31         b_pre[mat + 1] = np.dot((phi - np.dot(sigma, lam_1)).T, \
32             b_pre[mat]) - delta_1[:, 0]
33         b_solve[mat + 1] = -b_pre[mat + 1] * n_inv[mat + 1]
34
35     return a_solve, b_solve
36
37 def opt_gen_pred_coef(self, lam_0, lam_1, delta_0, delta_1, mu, phi,
38     sigma):
39     """
40     Returns tuple of arrays
41     Generates prediction coefficient vectors A and B in fast C function
42
43     Parameters
44     -----
45     lam_0 : numpy array
46     lam_1 : numpy array

```

The change in profiling output after rewriting the `gen_pred_coef` is shown in Table 4.2 and Figure 4.4. The C extension version of `gen_pred_coef` is italicized for reference. Comparing these tables shows that the computational time spent in the function drops from 8.817 to 1.365 seconds. The function highest on the list in terms of computational time is now a core Python function `get_token` rather than a function written specifically for the package. Because it is highly unlikely that any core components of package would need to be written, the fact that the most

computationally expensive function now appears eighth on the list rather than first is a good sign that code refactoring has been effective. Figure 4.4 reinforces the conclusion that handing this function over to C was effective. Again, the function has again been given a thicker border. Instead of taking up 19.05% of each `_affine_pred` call, the function now only takes up 2.54% of each call. Because this function is called each time a new set of predicted yields need to be generated, the relative advantage of using the C based method over the original pure Python method increases as the number of iterations required for A and B goes up.

Table 4.2: Profiling Output of Pure Python Solve Function.

filename:lineno(function)	Total time
<i>affine.py:424(gen_pred_coef)</i>	8.817
{numpy.core._dotblas.dot}	6.336
{isinstance}	4.626
parser.py:59(get_token)	4.510
locale.py:363(normalize)	4.303
StringIO.py:119(read)	3.940
_strptime.py:295(_strptime)	3.196
{len}	3.165
tools.py:372(parse_time_string)	2.512
__init__.py:49(normalize_encoding)	2.433

Table 4.3: Profiling Output of Hybrid Python/C Solve Function.

filename:lineno(function)	Total time
parser.py:59(get_token)	2.841
StringIO.py:119(read)	2.270
core.py:2763(_update_from)	2.114
locale.py:363(normalize)	1.958
{getattr}	1.753
_strptime.py:295(_strptime)	1.446
parser.py:356(_parse)	1.387
<i>{affine.model._C_extensions.gen_pred_coef}</i>	1.365
{isinstance}	1.296
{len}	1.296
{numpy.core._dotblas.dot}	1.177
tools.py:372(parse_time_string)	1.132
__init__.py:49(normalize_encoding)	1.101
locale.py:347(_replace_encoding)	1.044
index.py:1273(get_loc)	1.036
parser.py:156(__init__)	0.993
{setattr}	0.958
core.py:3040(__setitem__)	0.882
{method 'get' of 'dict' objects}	0.812
parser.py:149(split)	0.788

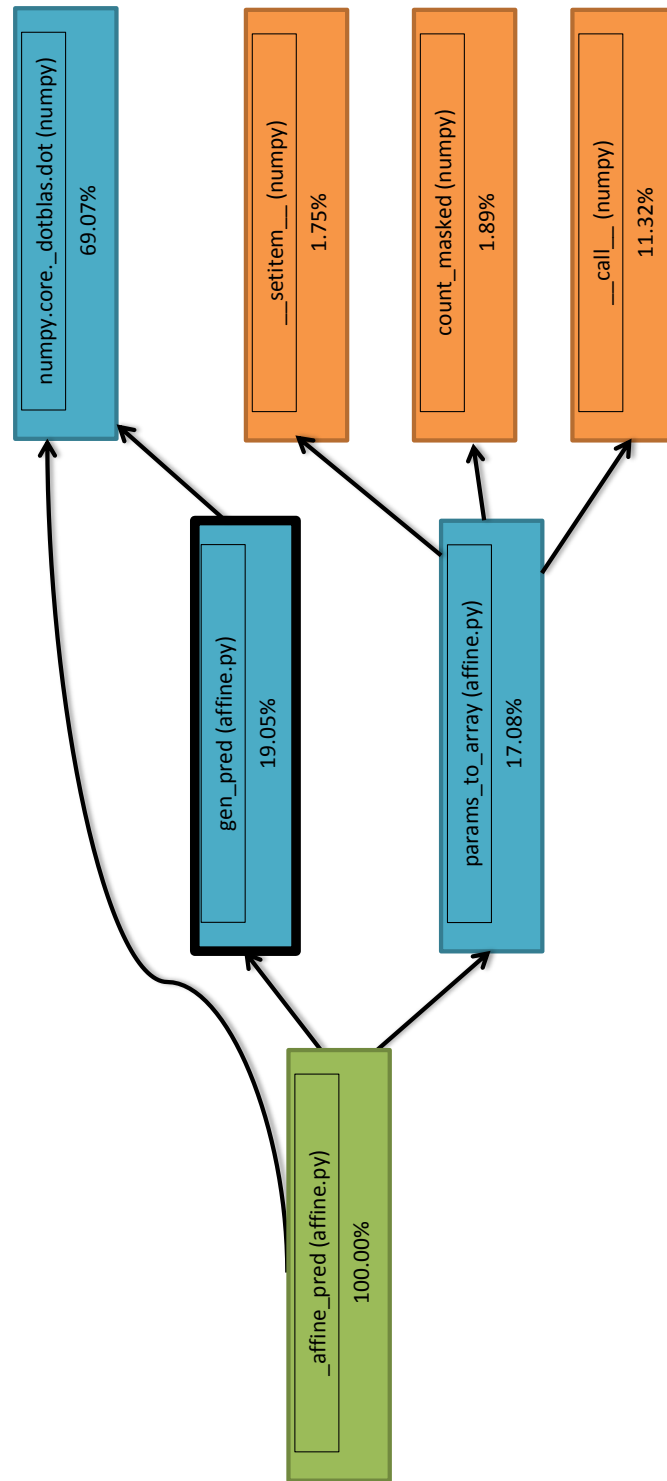


Figure 4.3: Graphical Output Profiling Pure-Python Solve Function.

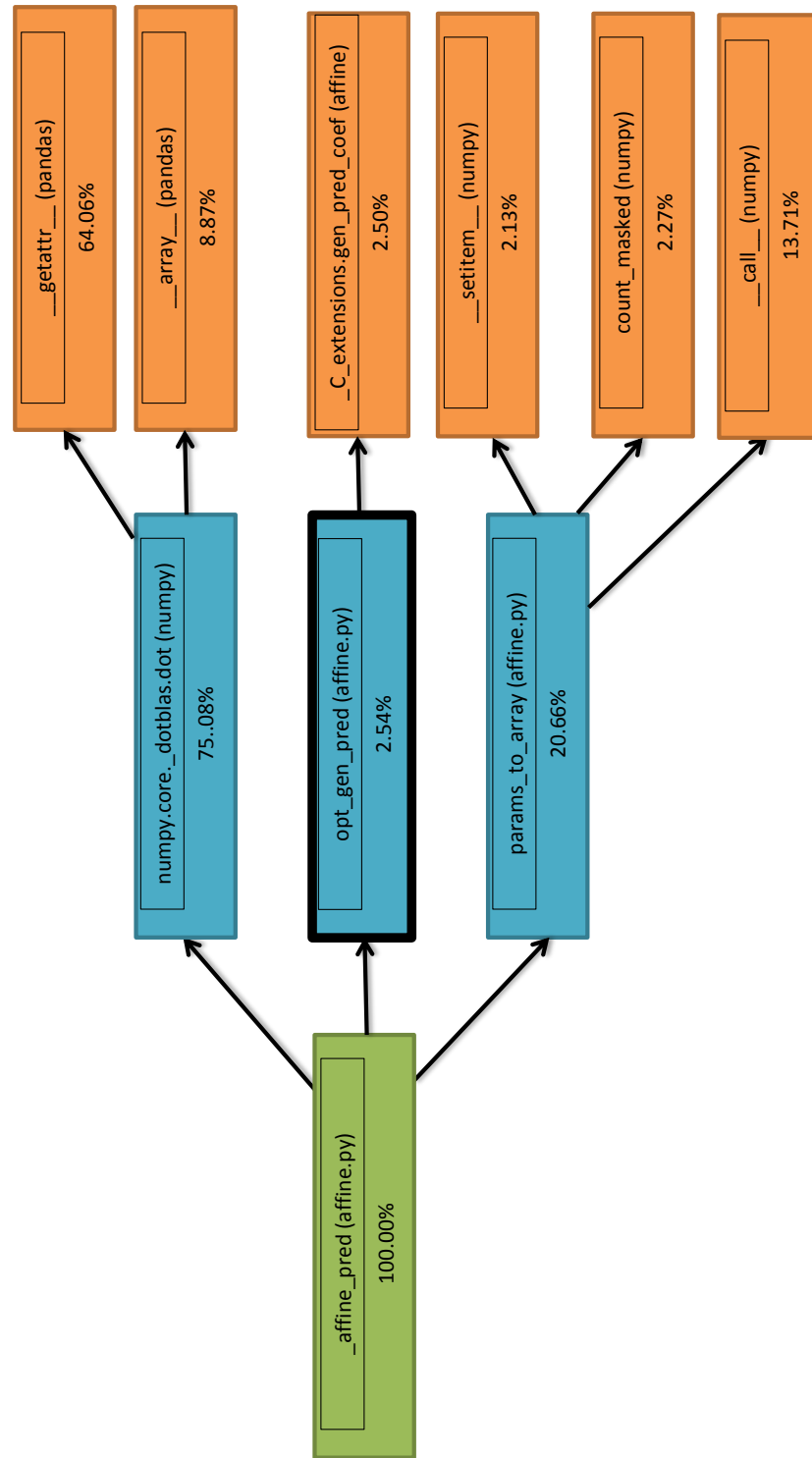


Figure 4.4: Graphical Output Profiling Hybrid Python/C Solve Function.

In order for the C code to efficiently construct the A and B arrays, various dot product functions were built that relied on pointer arithmetic. These functions are included at the top of the C code presented in Listing D.1 in the Appendix. The A and B arrays are both constructed as one dimensional arrays on the heap for two reasons. First, two-dimensional arrays are possible in C, but they are not nearly as high performing as one-dimensional arrays. A two-dimensional **array** in C is an array of pointers to pointers, with each row index referring to a separate, non-contiguous address in RAM. This means that if an element of the **array** is referred to by both indices, i.e. `array[i][j]`, this involves two lookups of the RAM location of these elements. These operations are much more efficient if the arrays are stored as one-dimensional arrays and the implicit indexing is handled by any of the operations involving these arrays. Care must be taken to ensure that an allocation of these arrays as one-dimensional is successful.¹² Second, constructing **numpy** arrays from one-dimensional arrays is much simpler than constructing them from multi-dimensional arrays, especially with the C-API provided with **numpy**. Once a single-dimensional C **array** is allocated on the heap, it only needs to be wrapped in a **numpy array** constructor, as seen in a simple example in Listing 4.14.

Listing 4.14: C array to numpy array

```

1 int rows = 3;
2 int cols = 5;
3 npy_intp dims[2] = {rows, cols};
4 /* allocate contiguous one dimensional */
5 double *myarray = (double *) malloc(rows * cols * sizeof(double));
6 /* fill in elements in myarray */
7 /* ... */
8 /* Allocate numpy array in C */
9 PyArrayObject *myNParray;
10 myNParray = (PyArrayObject *) PyArray_SimpleNewFromData(2, dims, double,
11                                                         myarray);
12 PyArray_FLAGS(myNParray) |= NPY_OWNDATA;
13 Py_DECREF(myNParray);

```

The **array** in C is assumed to be contiguous in RAM but the dimensions are passed in as a length two **array** of type `npy_intp`, a type provided by the **numpy** library. In line 5 the C array,

¹²Checking to make sure that the arrays are successfully allocated is accomplished in C by testing if the array is equal to NULL. Examples of this can be seen in the C code in the Appendix.

`myarray` is allocated on the heap. After filling in the values, the Python object to be returned from C to Python is initialized in line 9. In lines 10 and 11, the Python object is defined as a wrapper around the original C array, without needing to allocate new data for the array. It is then ensured that the Python object controls the deallocation responsibility of the object in line 12 using the `PyArray_FLAGS` function and the array flag `NPY_OWNDATA`. In order for there to be appropriate deallocation of the `numpy` array once in Python, the correct number of references must be set using the `Py_DECREF` function. The function `PyArray_SimpleNewFromData` is the preferred way of creating a `numpy` array based on data already allocated in C (Oliphant et al., 2014).

4.4.1 Testing

Testing of the package was performed with the empirical applications presented in Chapter 2 and Chapter 3. These two chapters both involved the estimation of affine term structure models, some of which were compared to other published results and others which were unique models estimated by the author. Both of these chapters depended completely on `affine` to build and estimate the models. The estimated results of the package for the models in Baker (2014a) were compared to the published results in Bernanke et al. (2005) and they generated similar pricing errors and term premium dynamics. Discrepancies between the results are minimal and are addressed in Baker (2014a). In Baker (2014b), the estimation process using `affine` generated well-fitting term structure dynamics in line with much of the literature. A test of the logic programmed to generate the prediction matrices A and B in Equation 4.1.8 was that the Python and C versions of the function were programmed based on the theory, not on each other. When it was assured that both were functioning properly independently, their results were compared and the results were identical down to the machine epsilon of the C data type `double`.

When the `Affine` object is allocated, many assertions are performed on the shape of the observed factor and yield data, the shape of each of the parameter arrays listed in Table 4.1, and the combinations of the other arguments to the objects. These other assertions include ensuring that appropriate non-error yield columns are supplied if Direct Maximum Likelihood is used as the solution method. If any of these assertions fail, creation of the `Affine` object fails and the user is notified of what caused the failure. This allows the user to modify the script and retry instantiation of the `Affine` object.

Unit tests were written in order to stabilize the core functionality of the package throughout development and across environments. As contributions are made to the package by other develop-

ers, unit tests validate that the core components of the package are functioning as expected through iterations of source code changes. These tests are included in the Appendix in Listing D.3. These tests ensure that all of the individual functions used in the package to build and solve the model are operating correctly. In some cases, the tests validate that no errors are thrown by the package when correctly formatted arguments are used. In other cases, the tests confirm that when incorrectly formatted arguments are passed to the package, an error message is raised that indicates to the user that there is an issue with the argument. Other tests run model estimation processes that are known to converge and confirm that they do in fact converge.

The unit tests are organized as functions in classes, where all of the unit test functions within a class share the same setup. In the case of the unit tests written for **Affine**, each class defines a collection of valid arguments that successfully create an **Affine** object. There are currently three classes of unit tests: **TestInitiatilize**, **TestEstimationSupportMethods**, and **TestEstimationMethods**. These classes are intended to separate unit tests with different purposes. Each test in **TestInitiatilize** begins by initializing valid arguments to an observed factor model and contains tests related to proper initial creation of an **Affine** object. The first unit test function, **test_create_correct**, passes these valid arguments to the **Affine** class and confirms that the instance of the class exists. There are then five test functions that each increment the dimensions of one of the parameter array arguments by one so that its shape is no longer valid and then verifies that an error is raised indicating that the parameter array is of incorrect shape. There are then two tests, **test_var_data_nulls** and **test_yc_data_nulls**, that replace just one of the values in the observed factor and the yield data respectively with a null value and confirm that an appropriate error is raised. The final test in this class, **test_no_estimated_values**, modifies the two parameter arrays that have masked values, unmask them, and confirms that an error is raised indicating that there are no elements in the parameter arrays to estimate.

The **TestEstimationSupportMethods** class contains tests confirming that calculations the package relies on are functioning properly. These are all in the form of positive tests, where the unit test only fails if an error is raised in the operation. The setup for the tests in this class is that of a more complex affine model with latent factors so that all of the possible calculations necessary to solve an affine model are possible. The first four tests, **test_loglike**, **test_score**, **test_hessian**, and **test_std_errs**, each confirm that with a correct model setup the likelihood, numerical score, numerical Hessian, and numerical standard errors respectively can be calculated. The next test, **test_params_to_array**, confirms that when passing values for the unknown elements in the

parameter arrays into the `params_to_array` function, the parameter arrays are returned, both when masked and standard `numpy` arrays are needed. `test_params_to_array_zeromask` performs similar testing on a function that returns arrays with unmasked elements set to zero corresponding to guess values equal to zero. The next two unit tests, `test_gen_pred_coef` and `test_opt_gen_pred_coef`, each test whether the generation of the A and B coefficients in Equation 4.1.8 for all maturities is successful, using a pure Python function or a C function respectively. The next unit test, `test_py_C_gen_pred_coef_equal`, confirms that given the model setup for this class, the Python and C functions generate the same result. The final three unit tests for this class, `test_solve_unobs`, `test_affine_pred`, and `test_gen_mat_list`, each confirm that private functions used internally by the package are operating correctly. `test_solve_unobs` confirms that the function that generates the latent factors returns valid results. `test_affine_pred` validates that the internal function used to stack the predicted yields into a one-dimensional array generates a result of the expected size. `test_gen_mat_list` tests whether or not the internal function used to determine the yields priced with and without error correctly generates these yields with the specific model setup in this class.

The final class, `TestEstimationMethods`, contains tests for running the estimation processes. For the setup, models with and without latent factors are created. The first test, `test_solve_nls`, attempts to estimate a model without latent factors. The second test, `test_solve_ml`, attempts to estimate a model with a latent factor. As in the previous unit test class, these are both positive tests, meaning that they merely test for whether convergence is possible given the current setup. If there were any issues with the outside numerical approximation libraries, these issues would cause the unit tests to fail.

After the user has installed `affine`, the entire suite of unit tests can be run using `nose`, a Python package that aids in the organization and writing of unit tests. This is accomplished by running the `nosetests` command in the top directory of the source code. Each unit test can also be run individually using a `nosetests` command specifying a path to the test in the source code¹³. For example, in order to initiate the test that verifies that an incorrectly shaped `lam_0_e` will raise an error, the following command should be run in the top level of the source code:

Listing 4.15: Running a specific unit test

```
1 nosetests affine.tests.test_model:TestInitiatilize.test_wrong_lam0_size
```

¹³For more information about `nose`, see <https://nose.readthedocs.org/en/latest/>.

In this example, the path to the Python file containing the unit tests is `affine/tests/test_model.py`, the name of the class that contains the specific test we want to run is `TestInitiatilize`, and the name of the test function is `test_wrong_lam0_size`. In cases where users want to validate the installation of the package, running all of the unit tests using the `nosetests` command is sufficient.

It is important to note that, while these unit tests do provide a reasonable amount of coverage for the basic functionality of the package, they are not an exhaustive list of all possible unit tests, nor do they cover all possible use cases. As development continues on `affine`, more unit tests will continue to be developed. It should also be noted that modifications made to the package may require changes to the unit tests in order for them to pass.

4.4.2 Issues

A few issues were encountered during development, specifically in development of the C extension. The first major issue pertained to proper allocation and reference counting of objects passed from C to Python. First, an attempt was made to create `numpy` arrays from C multi-dimensional arrays based on several online examples, but discovering a way to properly transfer ownership of these arrays to Python proved difficult. The arrays would often be returned to Python, but would be over-written in RAM before it was appropriate to do so, meaning that the reference counts to the Python array had been incorrectly set in C prior to the objects being returned to Python. After battling with this and getting inconsistent results on 32- and 64-bit architectures, single-dimensional arrays were used instead of two-dimensional arrays.

The use of one-dimensional arrays ended up leading to a significant performance improvement because pointer arithmetic could be used. This led to the writing of four bare-bones functions in C that perform the dot-product of two one-dimensional arrays (implied two-dimensional). The four functions are derived from the possibilities of transposing the first array, or the second array, neither, or both. In order for these functions to work correctly with the `numpy` arrays supplied to the C function, it must be ensured that the data referenced by the arrays is held contiguously in C. These arrays passed into the C function are initialized in Python, and there is not a guarantee that `numpy` arrays initialized in this way are held contiguously. Contiguous ordering of the data can be ensured using the `np.ascontiguousarray`, which is applied to the arrays prior to being passed into the function when the optimized C extension is successfully installed.

Another push in the way of single-dimensional arrays came with the fact that arrays of indeterminate length at compile time cannot be passed into C functions. Because the package is developed for the general case, the sizes of all of the arrays used to compute A and B are of indeterminate length at compile time. The dimensions of the array along with the pointers can be passed to the functions. When the arrays are one-dimensional and contiguous, the pointer to the first element of the `array` along with the number of rows and columns is enough information to be able to perform any kind of operation on a pair of arrays. Many of the tutorials on the use of the `numpy` C-API use multi-dimensional C arrays, but this may be based on the fact that many of the users are coming from a Python background. Single-dimensional, contiguous arrays are much better for performance and fit more naturally into C-based code development.

Another issue that was encountered in development was acceptable levels of differences between Python and C based results. One of the benefits of writing the Python and C methods for the same operations was using one to test the results of the other. Testing strict equality (`==`) in Python versus C proved problematic. After calculating the A and B arrays in both Python and C, some of the entries in arrays would be equal, while others would differ by an amount no greater than $1e-12$. The first way that I approached the issue was ensuring that the `numpy float64` data type used in `numpy` arrays was equivalent to the `NPY_DOUBLE` C data type used in the C extension. This involved going into the lower layers of `numpy` source code, eventually confirming that they were both equivalent to C `double` types. After confirming each line of code in both versions, further research led to the conclusion that these differences were driven by machine epsilon floating point comparisons. Machine epsilon refers to potential differences in the results of equivalent mathematical operations driven by floating point rounding. These specific machine epsilon differences likely resulted from differences in `libc` and built-in `numpy` functions. These differences are important to keep in mind when attempting to set convergence criteria too tightly in the numerical approximation algorithms. These are not likely to reliably hold below $1e-12$, given the recursive nature of the construction of A and B . The default convergence criteria for parameter and function differences in the package is therefore $1e-7$, as this is well above the machine epsilon but low enough to generate reliable results in most modeling exercises.

4.5 Building Models

In order to flesh out the context for this package, it may be useful to describe how the approaches of some of the important works in affine term structure modeling could be achieved

using the package. This section will focus on models that would not involve any adjustments to the core package in order to obtain the same approach, but will also present an example of a modeling approach that would involve modifications to select function in the original package. Before moving on to specific examples, it may be useful to summarize the current coverage of the package. Table 4.4 documents the papers and respective models that can be estimated using this package.

Table 4.4: Affine Term Structure Modeling Papers Matched with Degree of Support from Package

Paper	Solution method	Latent factors	Modifications required
Chen and Scott (1993)	Direct ML	Yes	No
Dai and Singleton (2000)	Simulated Method of Moments	Yes	Yes
Dai and Singleton (2002)	Direct ML	Yes	No
Ang and Piazzesi (2003)	Direct ML	Yes	No
Bernanke et al. (2005)	Non-linear least squares	No	No
Kim and Orphanides (2005)	Kalman filter ML	Yes	No
Kim and Wright (2005)	Kalman filter ML	Yes	No
Diebold et al. (2006)	Kalman filter ML	Yes	No
Cochrane and Piazzesi (2008)	Non-linear least squares	No	No
Orphanides and Wei (2012)	Direct ML	Yes	Yes

As is shown, most of the approaches of the seminal papers are directly supported by the package. The methods of Dai and Singleton (2000) and Orphanides and Wei (2012) would both require modification to the core **Affine** class. Even in these cases, the level of abstraction provided by the package allows individual components to be modified while leaving the rest of the package intact.

A few of the approaches of these papers will be discussed in subsections below, specifically in how they would be performed using **affine**. In each of these sections, the outline of the code is shown with only the key steps invoked using the package. For complete scripts for each of these methods, please see Section E of the Appendix.

4.5.1 Method of Bernanke et al. (2005)

The affine term structure model of Bernanke et al. (2005) uses a pricing kernel driven solely by observed information. The authors assume that the process governing the observed information is a VAR(4) with five macroeconomic variables using monthly data. They fit a yield curve of zero-coupon bonds using the yields on the six month, one, two, three, four, five, seven, and ten year yields. With only the use of observed factors informing the pricing kernel, the authors estimate the parameters in Equations 4.1.3 and 4.1.5 using OLS prior to estimation of the prices of risk. This vastly decreases the number of parameters to be estimated compared to models using latent factors

and leaves only the parameters in λ_0 and λ_1 to be estimated. They also assume that the prices of risk in Equation 4.1.4 are zero for all but the elements in λ_t corresponding to the contemporaneous elements in X_t .

Assuming that the data has already been imported and the other parameter arrays have been setup, the model can be initialized and estimated as shown in Listing 4.16:

Listing 4.16: Bernanke et al. (2005) model setup

```

1 import numpy.ma as ma
2 from affine.model.affine import Affine
3
4 # number of observed factors
5 n_vars = 5
6 # number of lags in VAR process
7 lags = 4
8 # maturities of yields in months
9 mats = [6, 12, 24, 36, 48, 60, 84, 120]
10
11 #import yield curve data into yc_data and macroeconomic data into var_data
12 #...
13 #fill in values of delta_0_e, delta_1_e, mu_e, phi_e, and sigma_e from OLS
14 #...
15 #initialize the lambda_0 and lambda_1 arrays
16 lam_0_e = ma.zeros([n_vars * lags, 1])
17 lam_1_e = ma.zeros([n_vars * lags, n_vars * lags])
18 #mask only contemporaneous elements (elements to be estimated)
19 lam_0_e[:n_vars, 0] = ma.masked
20 lam_1_e[:n_vars, :n_vars] = ma.masked
21
22 #instantiate model
23 model = Affine(yc_data=yc_data, var_data=var_data, mat=mats, lags=lags,
24               lam_0_e=lam_0_e, lam_1_e=lam_1_e, delta_0_e=delta_0_e,
25               delta_1_e=delta_1_e, mu_e=mu_e, phi_e=phi_e, sigma_e=sigma_e)
26 #construct guess_params
27 guess_params = [0] * model.guess_length()
28 #solve model
29 solved_model = model.solve(guess_params, method='nls')
```

Lines 15-20 ensure that the prices of risk are restricted to zero for all but the contemporaneous values of X_t . The model object is created in lines 23-25. The `solve` function is called in line 29 with the `nls` options signifying non-linear least squares, which is appropriate given that no latent factors are estimated in this model. Because the parameter space tends to be smaller in models with no latent factors, these models tend to solve in a shorter amount of time than those with latent factors. For example, at the precision levels indicated in Chapter 2, each model took around three minutes to solve. The starting values for each of the unknown parameters across λ_0 and λ_1 are set to zero and the number of unknown parameters across the parameters arrays can be generated from the object using the `guess_length()` function. The `solved_model` Python tuple contains each of the parameter arrays passed into the `Affine` class object with any masked elements solved for. In this example, the different parameter arrays are accessed in the tuple of objects returned. The estimated parameter arrays could also be accessed as attributes of the solution object along with the standard errors. The standard errors are calculated by numerically approximating the Hessian of the parameters. A future release of the package will include more user friendly presentations of the results in formatted tables. When a likelihood based approach is used, formatted tables of many of the parameter estimates and their standard errors are provided by the `statsmodels LikelihoodModel` class that `Affine` inherits from. Documentation for this formatted output is provided in `statsmodels`.

4.5.2 Method of Ang and Piazzesi (2003)

Another model setup that can easily be implemented using `affine` is that first used in Chen and Scott (1993) but more recently used in Ang and Piazzesi (2003). In this chapter, a five factor model is estimated with two observed factors summarizing movements in output and inflation, respectively, and three unobserved factors. Their method for estimating the models involves a four-step iterative process where unknown elements in individual parameter arrays are estimated in different steps. This approach is outlined in Listing 4.17. In this method, the components of μ , Φ , and Σ in Equation 4.1.3 pertaining to the observed factors are estimated with the assumption that the two observed factors are orthogonal to the unobserved factors¹⁴. The components of the short-rate relationship, Equation 4.1.5, pertaining to the observed factors are also estimated via OLS. This takes place in lines 19-21¹⁵. Beginning in Step 1 on line 30, the unknown parameters in

¹⁴This assumption is made by Ang and Piazzesi (2003) to decrease the number of estimated parameters

¹⁵For complete detail of a setup script for this method, see Listing E.2 in the Appendix.

δ_1 and Φ are estimated and a model solved object is retained in lines 38-39. In this example listing, the model solution method is indicated as direct maximum likelihood, the numerical approximation method is BFGS, and the one month, one year, and five year yields are measured without error. Using the estimated Hessian matrix from Step 1, the standard error of each parameter is estimated and, as specified in Ang and Piazzesi (2003), the insignificant parameters are set to zero in a new parameter list in lines 44-65. This parameter list is used to generate the masked arrays and parameter guesses for Step 2 and the final estimation step. In Step 2, beginning in line 69, the unknown parameters in λ_1 are estimated, holding λ_0 at 0 and δ_1 and Φ at their estimated values after Step 1. The model again is re-estimated and the insignificant parameters in λ_1 are set to zero, with the estimated value of λ_1 retained for use in Step 3 and the final estimation step. In Step 3, beginning in line 86, an analogous estimation is performed where the estimated δ_1 , Φ , and λ_1 from Step 1 and 2 are used to estimate only the unknown parameters in λ_0 . The insignificant parameters in λ_0 are set to zero, with the estimated values in λ_0 is held for the final estimation step. In the final estimation step beginning in line 93, the significant parameters across δ_1 , Φ , λ_0 , and λ_1 are all re-estimated, with the insignificant parameters in these arrays held at 0, and using the estimated values from Steps 1-3 as initial estimates. This last estimation step produces the final estimation results. This entire process took less than ten minutes to solve on a laptop with 1.8GHz CPU speed.

Listing 4.17: Ang and Piazzesi (2003) model setup

```

1 import numpy as np
2 import numpy.ma as ma
3 import scipy.linalg as la
4 from affine.model.affine import Affine
5
6 # number of observed factors
7 n_vars = 2
8 # number of lags in VAR process
9 lags = 12
10 # number of latent variables to estimate
11 latent = 3
12 #maturities of yields
13 mats = [1, 3, 12, 36, 60]
14 #indices of maturities to be estimated without error
15 no_err = [0, 2, 4]
```

```

16
17 #import yield curve data into yc_data and macroeconomic data into var_data
18 #...
19 #fill in values of delta_0_e, delta_1_e, mu_e, phi_e, and sigma_e from OLS
20 #pertaining to observed factors
21 #...
22 #initialize the lambda_0 and lambda_1 arrays
23 phi_e[-latent:, -latent:] = ma.masked
24 delta_1_e[-latent:, 0] = ma.masked
25
26 #initialize lambda arrays to all zeros, but not masked
27 lam_0_e = ma.zeros([n_vars * lags, 1])
28 lam_1_e = ma.zeros([n_vars * lags, n_vars * lags])
29
30 ##Step 1
31 model1 = Affine(yc_data=yc_data, var_data=var_data, mats=mats, lags=lags,
32                 lam_0_e=lam_0_e, lam_1_e=lam_1_e, delta_0_e=delta_0_e,
33                 delta_1_e=delta_1_e, mu_e=mu_e, phi_e=phi_e, sigma_e=sigma_e,
34                 latent=latent)
35
36 #initialize guess_params
37 #...
38 solved_model1 = model1.solve(guess_params=guess_params, no_err=no_err,
39                              method='ml', alg='bfgs')
40 parameters1 = solved_model1.solve_params
41 #calculate numerical hessian of solved_params
42 std_err = model1.std_errs(parameters1)
43
44 #create list of parameters in parameters1 that are significant based on std_err
45 #and put in sigparameters1, otherwise replace with zero
46 tval = parameters1 / std_err
47 sigparameters1 = []
48 for tix, val in enumerate(tval):
49     if abs(val) > 1.960:
50         sigparameters1.append(parameters1[tix])
51     else:
52         sigparameters1.append(0)
53
54 #retrieve new arrays with these values replaced, used for estimation in later
55 #steps

```

```

56 parameters_for_step_2 = solved_model1.params_to_array(sigparameters1,
57                                                         return_mask=True)
58 delta_1 = parameters_for_step_2[3]
59 phi = parameters_for_step_2[5]
60
61 #retrieve arrays for final step 4 estimation with only values masked that were
62 #significant
63 parameters_for_final = solved_model1.params_to_array_zeromask(sigparameters1)
64 delta_1_g = parameters_for_final[3]
65 phi_g = parameters_for_final[5]
66
67 ##End of Step 1
68
69 #Step 2
70 #Estimate only unknown parameters in lam_1_e, results in model solve object
71 #solved_model2, use arrays delta_1 and phi from above
72 lam_1_e[-latent, -latent] = ma.masked
73 lam_1_e[:n_vars, :n_vars] = ma.masked
74 #...
75
76 #set insignificant parameters equal to zero in sigparameters2
77 parameters_for_step_3 = solved_model2.params_to_array(sigparameters2,
78                                                         return_mask=True)
79 lambda_1 = parameters_for_step_3[1]
80
81 parameters_for_final = solved_model2.params_to_array_zeromask(sigparameters2,
82                                                         return_mask=True)
83 lambda_0_g = parameters_for_final[1]
84 #End of Step 2
85
86 #Step 3
87 #Estimate unknown parameters in lam_0_e, with all pre-estimated values held at
88 #estimated values using delta_1, phi, (from Step 1) and lambda_1 (from Step 2)
89
90 #collect lambda_0_g and lambda_0 similar to Step 2
91 #Step 3
92
93 #Step 4
94 #Estimate model using guesses and assumptions about insignificant arrays set
95 #equal to zero

```

```

96 model4 = Affine(yc_data=yc_data, var_data=var_data, mats=mats, lags=lags,
97                 lam_0_e=lambda_0_g, lam_1_e=lambda_1_g, delta_0_e=delta_0,
98                 delta_1_e=delta_1_g, mu_e=mu_e, phi_e=phi_g, sigma_e=sigma_u,
99                 latent=latent)
100
101 #construct guess_params from final estimated values in Steps 1-3
102 solved_model4 = model.solve(guess_params=guess_params, no_err=no_err,
103                             method='ml', alg='bfgs')

```

These two demonstrations show that much of the model building steps are abstracted by the use of the **Affine** class object. Each script easily enables one to generate plots of the respective pricing errors and time-varying term premia. The Kalman filter ML method is also supported by the package and the approach would not be much different from that presented in Listing 4.17. The only modifications required would be the **method** argument would need to be changed to **kalman** and the appropriate additional arguments specified in Section 4.2 would need to be supplied. Kalman filter ML results could be used to replicate the approaches used in Kim and Wright (2005) and Diebold et al. (2006). To make a change in the likelihood calculation approach, the **method** simply needs to be changed when calling the **solve** method.

4.5.3 Method of Orphanides and Wei (2012)

There are some approaches that have yet to be directly implemented in the package such as the Iterative ML approach used in Duffee and Stanton (2012) and Orphanides and Wei (2012). This approach could be included in future versions of the package, but could also be executed by the user by inheriting from the **Affine** class and altering the log-likelihood definition.

As an example of an approach that would require modifications to the package, let us examine the model estimated in Orphanides and Wei (2012). In this paper, the authors estimated an affine term structure model using a rolling VAR rather than a fixed parameter VAR. Because of this, the likelihood calculation needs to be changed because the package assumes that the estimated parameters in the process governing the factors (Equation 4.1.3) are constant throughout the estimation period. The suggested way of making these modifications is through inheriting from the **Affine** class¹⁶ and making modifications only to the necessary components. An outline of this approach appears in Listing 4.18. A new class, **RollingVARAffine** is created on line 34, inheriting from

¹⁶The construction of the **Affine** model object as a Python class allows the user to create a custom class that replicates the functionality of the original class, unless over-written. For more information on object-oriented programming in Python, see <https://docs.python.org/2/tutorial/classes.html>.

the `Affine` class. In line 35-39, the `loglike` function, which return the likelihood, over-writes the original method for this class of the same name. This likelihood would be replaced with the likelihood as it is calculated in Orphanides and Wei (2012), given a set of values for the unknown parameters. The actual likelihood for this method is not shown.

Once the object is modified to fit the specific affine model formulation, the setup and estimation can continue just as in the other examples. The model object is created in lines 41-44. Only μ , Φ , and Σ are estimated in the estimation step, performed in lines 49-50. The unknown parameters are passed into the newly defined likelihood just as before and the rest of the components of the estimation process are unchanged. These estimated arrays are used in the second estimation step, when λ_0 and λ_1 are estimated in Step 2 in lines 57-72.

Listing 4.18: Orphanides and Wei (2012) model setup

```

1 import numpy as np
2 import numpy.ma as ma
3 import scipy.linalg as la
4 from affine.model.affine import Affine
5
6 # number of observed factors
7 n_vars = 2
8 # number of lags in VAR process
9 lags = 2
10 # number of latent factors
11 latent = 1
12 # maturities of yields
13 mats = [4, 8, 20, 28, 40]
14 # index of yield estimated without error
15 no_err = [3]
16
17 #import yield curve data into yc_data and macroeconomic data into var_data
18 #...
19 #fill in values of delta_0_e, delta_1_e, mu_e, phi_e, and sigma_e from OLS
20 #mu_e, phi_e, and sigma_e are constructed with an extra dimension as they
21 #differ every time period
22 #initialize the lambda_0 and lambda_1 arrays
23 mu_e[-latent:, 0, :] = ma.masked
24 phi_e[-latent:, -latent:, :] = ma.masked
25 sigma_e[-latent:, -latent:, :] = ma.masked

```

```

26
27 #initialize lambda arrays to all zeros, but not masked
28 lam_0_e = ma.zeros([n_vars * lags, 1])
29 lam_1_e = ma.zeros([n_vars * lags, n_vars * lags])
30
31 #create a new class that inherits from Affine
32 #inheriting from Affine means that all methods are the same except for those
33 #redefined
34 class RollingVARAffine(Affine):
35     def loglike(self, params):
36         #here write the likelihood in terms of rolling VAR rather than fixed
37         #parameter VAR
38
39
40 #Instantiate RollingVARAffine class
41 model1 = RollingVARAffine(yc_data=yc_data, var_data=var_data, mats=mats,
42                           lags=lags, lam_0_e=lam_0_e, lam_1_e=lam_1_e,
43                           delta_0_e=delta_0_e, delta_1_e=delta_1_e, mu_e=mu_e,
44                           phi_e=phi_e, sigma_e=sigma_e, latent=latent)
45
46 #initialize guess_params
47 #...
48 #attempt to solve model
49 solved_model1 = model1.solve(guess_params=guess_params, no_err=no_err,
50                              method='ml', alg='bfgs')
51
52 #retrieve new arrays with these values replaced, used for estimation in step 2
53 mu = solve_model1[4]
54 phi = solve_model1[5]
55 sigma = solve_model1[6]
56
57 #Step 2
58 #Estimate lambda_0 and lambda_1
59 #solved_model2, use arrays mu, phi, and phi from above
60 lam_0_e[:nvars, 0] = ma.masked
61 lam_0_e[-latent, 0] = ma.masked
62 lam_1_e[-latent, -latent] = ma.masked
63 lam_1_e[:n_vars, :n_vars] = ma.masked
64
65 final_model = RollingVARAffine(yc_data=yc_data, var_data=var_data, mats=mats,

```

```

66         lags=lags, lam_0_e=lam_0_g, lam_1_e=lam,
67         delta_0_e=delta_0, delta_1_e=delta_1, mu_e=mu,
68         phi_e=phi, sigma_e=sigma, latent=latent)
69
70 #construct guess_params from final estimated values in Steps 1-3
71 fsolved_model = final_model.solve(guess_params=guess_params, no_err=no_err,
72                                   method='ml', alg='bfgs')

```

Listing 4.18 shows how the approach to modifying the core **Affine** class in order to estimate models outside of the original supported models. This approach to extending the core package could lead to more supported models and greater coverage of the affine term structure literature.

4.6 Conclusion

This chapter discussed how a variety of affine term structure models can be understood as choices among a series of permutations within a single modeling framework, including model structure, number of latent factors, solution method, and numerical approximation algorithm. This single framework was presented within the context of a new package, **affine**, that contributes to the term structure literature via its ability to simplify the process of building and solving affine models of the term structure. This technical framework within which affine term structure models can be built and understood is itself a new contribution to the literature and opens the doors for new theoretical connections to be established between previously disparate model construction and estimation approaches. This chapter demonstrated how many models could be built and estimated by only supplying data and arguments, with even more able to be built and solved with minor extensions of the package. The structure of the package lends itself naturally to extension and select parts of the solution of the process can be modified while leaving the rest of the package intact. With the theoretical background explicitly linked to the package, building models using this package should be much more simple, lowering the cost of contributing to the affine term structure model literature. The package has also been optimized for computational speed, making it easier to run a larger number of models faster.

In addition to this computational framework on its own, this chapter also detailed the development of the package and the advantages and challenges of developing computational package in Python and C. Given the current popularity of Python in mathematical modeling circles and C as a low-level computationally efficient language, the approaches to development outlined in this

chapter could serve as a useful reference for those attempting to develop computationally efficient packages in Python.

In the near future, I would like to expand the basic functionality of the package to include basic plotting of the results through the `matplotlib` library. Plotting is already supported through the core functionality used from other libraries, but specific methods could be written that would generate popular charts such as time series of the pricing error, the latent factors, and the time-varying term premium. I would also like to make the data type checks more robust and provide more feedback to the user regarding errors with the setup of their data or parameter arrays. This would include writing some robust Python exception handling classes specific to this package. Another feature I would like to include is more robust handling of errors encountered in the numerical approximation algorithms. There are times when the numerical approximation algorithms pass in invalid guesses as values, so I would like to offer the user more of a buffer from these errors, which can sometimes be cryptic. I would also like to add more well-formatted output of the estimated parameters and their standard errors.

CHAPTER 5

CONCLUSION

This dissertation has contributed to the affine term structure model literature by making suggestions for additions and modifications to the pricing kernel in the first two chapters and providing a computational modeling framework within which a wide variety of discrete-time affine models can be estimated in the third chapter. Chapter 2 demonstrated how measures of uncertainty can contribute valuable information to a pricing kernel driven by observed factors. Adding uncertainty information to the pricing kernel produced a better fitting model and generated higher term premia during recessions. This change in the term premia from the addition of uncertainty proxies to the pricing kernel suggested that, not only do different horizons of uncertainty enter the term premia, but explicitly pricing certain horizons leads to changes in the estimates of the term premia. Chapter 3 showed how real-time data used in place of final-release data produced a better performing model when measured using root-mean-square pricing error. This chapter also demonstrated that a real-time data driven affine term structure model produces an erratic term premium for shorter maturity bonds but a more inter-temporally persistent term premium for longer maturity bonds. This distinction was not generated by the equivalent model driven by final data and could be lost in a broader class of models exclusively using final-release data. This chapter also showed that some of the advantage of using real-time over final data to price the yield curve is lost with the addition of unobserved, latent factors to the pricing kernel. With the increasingly common use of latent factors in affine term structure to increase model performance, the implications of using these factors should be considered when determining how observed information enters bond markets. Together, the first two chapters showed how modeling with observed factors can reveal important information about what drives bond market decisions.

Chapter 4 provided a general framework within which affine term structure models can be built and solved and is the essential backdrop to the first two chapters. The models in Chapters

2 and 3 were both built and solved using the algorithms and approach presented in this chapter. The ease with which factors and model structure could be changed and tested within the first two chapters was a result of design choices in the package and could potentially be very useful for others building affine term structure models. Consistent term structure modeling algorithms are not in widespread use and the package presented in Chapter 4 intends to begin to fill this void. The chapter also documented the approach taken by the author to developing a package that can efficiently estimate these non-linear models and provide meaningful abstraction to those building these models. Assumptions built into the package and issues in development are both documented. The chapter provides a framework within which models based on both observed and unobserved factors can be built and understood. This framework in itself represents a unique contribution to the field that could be used by many practitioners moving forward.

This dissertation offers context to the role that observed factors may play in decomposing how the bond market behaves as a whole. With the increased use of latent factors in the affine term structure model literature, investigating how latent factors relate to and interact with these observed components could lead to a deeper understanding of the full information set that drives bond market decisions. An avenue of future research would be to continue examining both how the inclusion of specific observed factors impact estimates of latent factors and how the statistical moments of the observed factors relate to latent factors estimated within a single model. Results from Chapter 3 suggested that latent factors can somewhat compensate for information misspecification in the pricing kernel, but it is still unclear what other observed information latent factors may be pricing. Further research is required in this area to help pin down what observed information latent factors represent.

Given the observations of Chapter 2 regarding the changing role of uncertainty in recessions compared to expansions, I would also like to further research how the weights on different factors change at different points in the business cycle. The current canonical affine term structure modeling framework assumes that the prices of risk are a constant, affine transformation of the factors throughout the observation period. Loosening this restriction by allowing the prices of risk to be temporally dependent could allow for a more robust specification of factors in different parts of the business cycle. Early evidence suggesting changes in the weights on the factors could come in the form of structural break tests as suggested by Bai et al. (1998), testing changes governing the prices of risk alone. This investigation would not need to be limited to observed factor models alone and could be expanded into models integrating unobserved latent factors.

This dissertation has served as a starting point for further investigations into the roles that observed factors play in affecting the performance and attributes of affine term structure models. Specifically, this dissertation has shown that, not only does the choice of observed factors impact performance, but which observed factors are included impact the time series of the term premia. Differences in results generated by observed factor models could be obscured by the inclusion of latent factors. The flexibility to estimate many different affine term structure models introduced with the package presented in the Chapter 4 will allow for simpler testing of how observed and latent factors influence pricing decisions. The package also allows for greater flexibility in changing assumptions about the characteristics of the models and provides a single framework for understanding how a single model relates to the broader class of term structure models.

APPENDIX A

DATA FOR CHAPTER 2

All data used for Chapter 2 are at a monthly frequency.

Monthly Treasury Bill and Treasury Constant Maturities are taken from the Federal Reserve Bank of St. Louis (FRED), including 6 month, and one, two, three, five, seven, and ten year maturities.

<http://research.stlouisfed.org/fred2/categories/115>

Fama-Bliss zero-coupon yields were downloaded from Wharton Research Data Services, which is only available by subscription:

<http://wrds-web.wharton.upenn.edu/wrds/>

Total non-farm employment is taken from the BLS website:

http://data.bls.gov/pdq/SurveyOutputServlet?request_action=wh\&graph_name=CE_cesbref1

The PCE price index and federal funds rate data are taken from the FRED site:

<http://research.stlouisfed.org/fred2/categories/9>

<http://research.stlouisfed.org/fred2/series/FEDFUNDS>

Blue Chip Financial Forecast data were obtained from the individual publications available at the American University Library. The link is provided here:

<http://198.91.33.107:8080/cgi-bin/Pwebrecon.cgi?BBID=11859965>

Eurodollar futures were obtained from a Bloomberg (2012) terminal.

VIX data was obtained from the Chicago Board Options Exchange (CBOE) VIX page, as this is the authority which calculates and trades this statistic:

<http://www.cboe.com/micro/vix/historical.aspx>

APPENDIX B

DATA FOR CHAPTER 3

All data used in this chapter are at a quarterly frequency.

Final release output growth is the annualized GNP quarter over quarter growth prior to 1992 and the annualized GDP quarter over quarter from 1992 and after. Final release inflation is measured as the quarter over quarter percentage in the GNP/GDP deflator with the transition taking place in 1992 also. Residential investment is also measured as an annualized quarter over quarter percentage change. Unemployment is the civilian unemployment rate. Each of these statistics were downloaded from the FRED site:

<http://research.stlouisfed.org/fred2/series/GNP/18>

<http://research.stlouisfed.org/fred2/series/GDP>

<http://research.stlouisfed.org/fred2/series/GNPDEF>

<http://research.stlouisfed.org/fred2/series/GDPDEF/>

<http://research.stlouisfed.org/fred2/series/PRFIC96/>

<http://research.stlouisfed.org/fred2/series/UNRATE/>

The market expectations for the current quarter are taken from the Survey of Professional Forecasters which is made available by the Federal Reserve Bank of Philadelphia:

<http://www.phil.frb.org/research-and-data/real-time-center/survey-of-professional-forecasters/>

The previous quarter releases are taken from the Real-time Data Set for Macroeconomists and are available for download from the Federal Reserve Bank of Philadelphia site:

<http://www.philadelphiafed.org/research-and-data/real-time-center/real-time-data/>

As in Chapter 2, the Fama-Bliss zero-coupon yield data was downloaded from the Wharton Research Data Services:

<http://wrds-web.wharton.upenn.edu/wrds/>

APPENDIX C

ADDITIONAL FIGURES AND TABLE

FOR CHAPTER 2

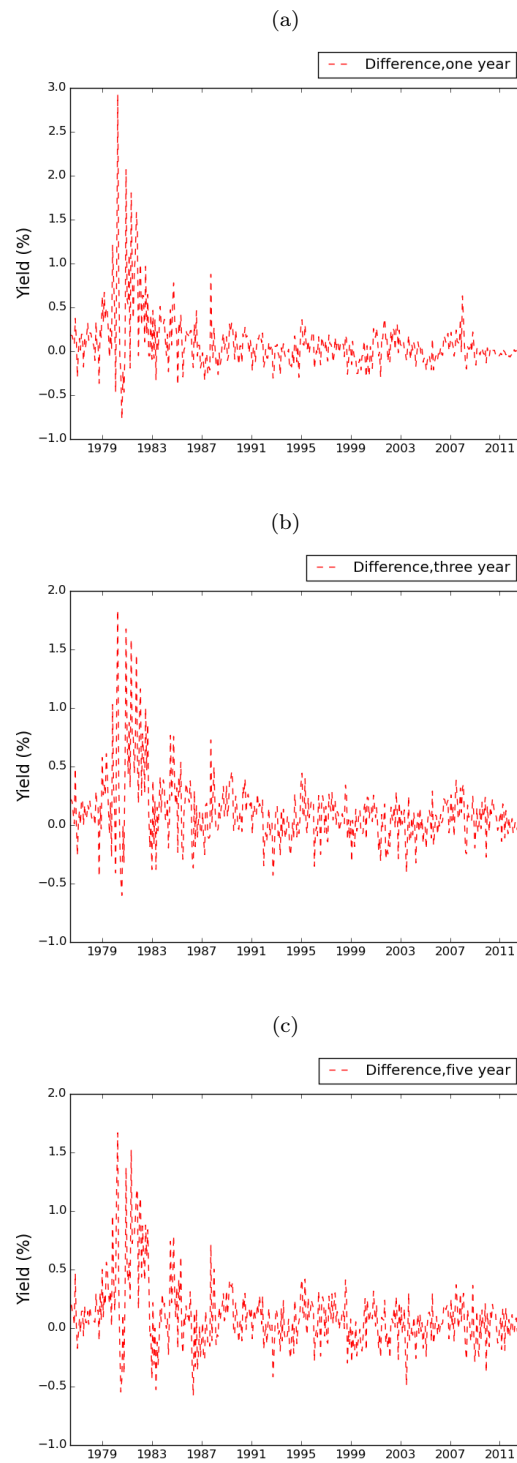
Table C.1: Maximum Five Year Term Premium by Date Range and Model. Each row represents a date range within which the maximum is calculated and each column represents an individually estimated model.

	BSR factor models		Uncertainty proxy models	
	b	b+E	b+E+D	b+E+D+V
08/90 - 05/12 (Full Sample)	3.50	2.97	3.45	3.43
03/91 - 03/01 (Expansion)	2.91	2.72	2.74	2.72
03/01 - 11/01 (Recession)	1.37	1.83	2.09	2.08
11/01 - 12/07 (Expansion)	1.96	2.02	2.05	2.02
12/07 - 06/09 (Recession)	1.26	2.20	2.64	2.62

Table C.2: Minimum Five Year Term Premium by Date Range and Model. Each row represents a date range within which the minimum is calculated and each column represents an individually estimated model.

	BSR factor models		Uncertainty proxy models	
	b	b+E	b+E+D	b+E+D+V
08/90 - 05/12 (Full Sample)	0.20	0.66	0.75	0.84
03/91 - 03/01 (Expansion)	1.43	1.25	1.35	1.33
03/01 - 11/01 (Recession)	0.86	1.40	1.69	1.69
11/01 - 12/07 (Expansion)	0.87	1.25	1.11	1.11
12/07 - 06/09 (Recession)	0.20	0.66	0.75	0.84

Figure C.1: Plots of Difference between Yields on One, Three, and Five-year Constant Maturity Government Bond Yields and Fama-Bliss Implied Zero Coupon Bond Yields.



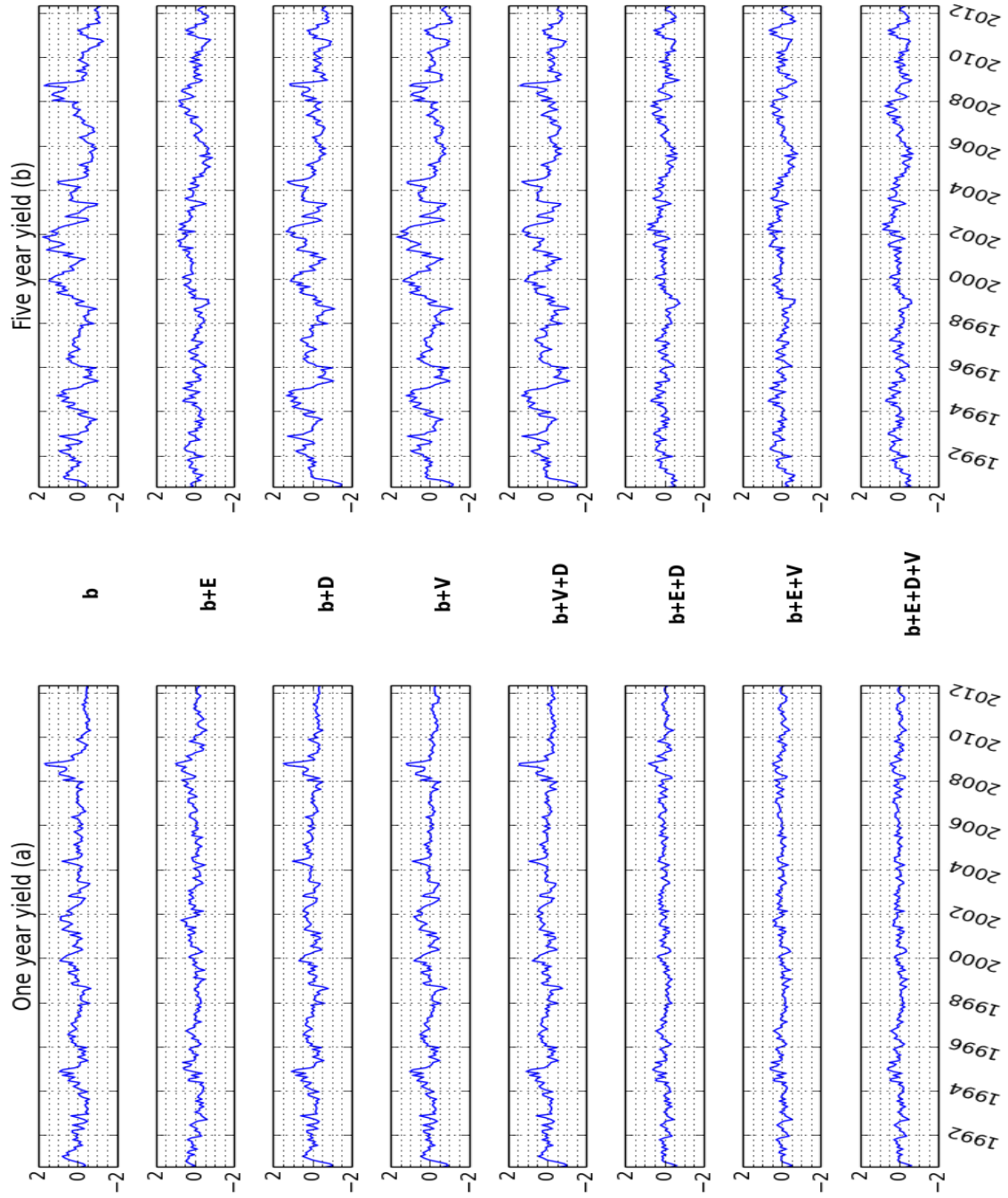


Figure C.2: Pricing Error Across Estimated Models for One and Five Year Maturity. Each row is a unique model.

APPENDIX D

SOURCE CODE FOR **AFFINE**

Listing D.1: C Code for generating A and B

```

1  #include "Python.h"
2  #include "arrayobject.h"
3  #include "C_extensions.h"
4  #include <math.h>
5  #include <stdio.h>
6
7  struct module_state {
8      PyObject *error;
9  };
10
11 /* === Constants used in rest of program === */
12 const double half = (double)1 / (double)2;
13
14
15 /* ==== Set up the methods table ===== */
16 static PyMethodDef _C_extensions_methods[] = {
17     {"gen_pred_coef", gen_pred_coef, METH_VARARGS, NULL},
18     {NULL, NULL}
19 };
20
21 #if PY_MAJOR_VERSION >= 3
22 #define GETSTATE(m) ((struct module_state*)PyModule_GetState(m))
23 #else
24 #define GETSTATE(m) (&_state)
25 static struct module_state _state;
26 #endif
27

```

```

28 // All of this is specific code for Python 3
29 #if PY_MAJOR_VERSION >= 3
30
31 static int _C_extensions_traverse(PyObject *m, visitproc visit, void *arg) {
32     Py_VISIT(GETSTATE(m)->error);
33     return 0;
34 }
35
36 static int _C_extensions_clear(PyObject *m) {
37     Py_CLEAR(GETSTATE(m)->error);
38     return 0;
39 }
40
41
42 static struct PyModuleDef moduledef = {
43     PyModuleDef_HEAD_INIT,
44     "_C_extensions",
45     NULL,
46     sizeof(struct module_state),
47     _C_extensions_methods,
48     NULL,
49     _C_extensions_traverse,
50     _C_extensions_clear,
51     NULL
52 };
53
54 /* ==== Initialize the _C_extensions functions ===== */
55 // Module name must be _C_extensions in compile and linked
56
57 #define INITERROR return NULL
58
59 PyObject *
60 PyInit__C_extensions(void)
61
62 #else
63 #define INITERROR return
64
65 void
66 init_C_extensions(void)
67 #endif

```

```

68 {
69 #if PY_MAJOR_VERSION >= 3
70     PyObject *module = PyModule_Create(&moduledef);
71 #else
72     PyObject *module = Py_InitModule("_C_extensions", _C_extensions_methods);
73 #endif
74     import_array();
75 #if PY_MAJOR_VERSION >= 3
76     return module;
77 #endif
78 }
79
80 /* Array helper functions */
81 /* ==== Matrix sum function ===== */
82 void mat_sum(int rows, int cols, double *arr1, double *arr2,
83             double *result) {
84     int mat_size, inc;
85     mat_size = rows * cols;
86     for (inc=0; inc < mat_size; inc++) {
87         *result = *arr1 + *arr2;
88         arr1++;
89         arr2++;
90         result++;
91     }
92 }
93
94 /* ==== Matrix subtraction function ===== */
95 void mat_subtract(int rows, int cols, double *arr1, double *arr2,
96                  double *result) {
97     int mat_size, inc;
98     mat_size = rows * cols;
99     for (inc=0; inc < mat_size; inc++) {
100         *result = *arr1 - *arr2;
101         arr1++;
102         arr2++;
103         result++;
104     }
105 }
106
107 /* ==== Matrix product functions ===== */

```

```

108 void mat_prodct(int row1, int col1, double *arr1,
109                 int col2, double *arr2,
110                 double *result) {
111
112     int dim1_row, dim2_col, dim1_col, col1_mod, row2_mod;
113     double sum, *arr1pt, *arr2pt;
114
115     col1_mod = 0;
116     for (dim1_row = 0; dim1_row < row1; dim1_row++) {
117         row2_mod = 0;
118         for (dim2_col = 0; dim2_col < col2; dim2_col++) {
119             arr1pt = &arr1[col1_mod];
120             arr2pt = &arr2[row2_mod];
121             sum = 0;
122             for (dim1_col = 0; dim1_col < col1; dim1_col++) {
123                 sum += (*arr1pt) * (*arr2pt);
124                 arr1pt++;
125                 arr2pt+=col2;
126             }
127             *result = sum;
128             result++;
129             row2_mod++;
130         }
131         col1_mod += col1;
132     }
133 }
134
135 /* ==== Matrix product functions tpose first argument ==== */
136 void mat_prodct_tpose1(int row1, int col1, double *arr1,
137                        int col2, double *arr2,
138                        double *result) {
139
140     int dim1_row, dim1_col, dim2_col, row1_mod, row2_mod;
141     double sum, *arr1pt, *arr2pt;
142
143     row1_mod = 0;
144     for (dim1_col = 0; dim1_col < col1; dim1_col++) {
145         row2_mod = 0;
146         for (dim2_col = 0; dim2_col < col2; dim2_col++) {
147             arr1pt = &arr1[row1_mod];

```

```

148         arr2pt = &arr2[row2_mod];
149         sum = 0;
150         for (dim1_row = 0; dim1_row < row1; dim1_row++) {
151             sum += (*arr1pt) * (*arr2pt);
152             arr1pt += col1;
153             arr2pt += col2;
154         }
155         *result = sum;
156         result++;
157         row2_mod++;
158     }
159     row1_mod++;
160 }
161 }
162
163 /* ==== Matrix product functions tpose second argument ===== */
164 void mat_prodct_tpose2(int row1, int col1, double *arr1,
165                        int row2, double *arr2,
166                        double *result) {
167
168     int dim1_row, dim2_row, dim1_col, col1_mod, col2_mod;
169     double sum, *arr1pt, *arr2pt;
170
171     col1_mod = 0;
172     for (dim1_row = 0; dim1_row < row1; dim1_row++) {
173         col2_mod = 0;
174         for (dim2_row = 0; dim2_row < row2; dim2_row++) {
175             arr1pt = &arr1[col1_mod];
176             arr2pt = &arr2[col2_mod];
177             sum = 0;
178             for (dim1_col = 0; dim1_col < col1; dim1_col++) {
179                 sum += (*arr1pt) * (*arr2pt);
180                 arr1pt++;
181                 arr2pt++;
182             }
183             *result = sum;
184             result++;
185             col2_mod += col1;
186         }
187         col1_mod += col1;

```

```

188     }
189 }
190
191 static PyObject *gen_pred_coef(PyObject *self, PyObject *args) {
192     PyArrayObject *lam_0, *lam_1, *delta_0, *delta_1, *mu, *phi, *sigma,
193         *a_fin_array, *b_fin_array;
194
195     int lam_0_cols, lam_1_cols, mu_rows, mu_cols, phi_rows, phi_cols,
196         sigma_rows, sigma_cols, mat, bp_offset, bp_noffset, next_mat, i;
197
198     const int max_mat;
199
200     double *lam_0_c, *lam_1_c, *delta_0_c, *delta_1_c, *mu_c, *phi_c,
201         *sigma_c, divisor;
202
203     // Parse input arguments to function
204     if (!PyArg_ParseTuple(args, "O!O!O!O!O!O!i",
205         &PyArray_Type, &lam_0, &PyArray_Type, &lam_1, &PyArray_Type, &delta_0,
206         &PyArray_Type, &delta_1, &PyArray_Type, &mu, &PyArray_Type, &phi,
207         &PyArray_Type, &sigma, &max_mat))
208         return NULL;
209     if (NULL == lam_0 || NULL == lam_1 || NULL == delta_0 || NULL == delta_1 ||
210         NULL == mu || NULL == phi || NULL == sigma) return NULL;
211
212     // Get dimesions of all input arrays
213     lam_0_cols=lam_0->dimensions[1];
214     lam_1_cols=lam_1->dimensions[1];
215     const int delta_1_rows=delta_1->dimensions[0];
216     mu_rows=mu->dimensions[0];
217     mu_cols=mu->dimensions[1];
218     phi_rows=phi->dimensions[0];
219     phi_cols=phi->dimensions[1];
220     sigma_rows=sigma->dimensions[0];
221     sigma_cols=sigma->dimensions[1];
222
223     // Create C arrays
224     lam_0_c = pymatrix_to_Carrayptrs(lam_0);
225     lam_1_c = pymatrix_to_Carrayptrs(lam_1);
226     delta_0_c = pymatrix_to_Carrayptrs(delta_0);
227     delta_1_c = pymatrix_to_Carrayptrs(delta_1);

```

```

228     mu_c = pymatrix_to_Carrayptrs(mu);
229     phi_c = pymatrix_to_Carrayptrs(phi);
230     sigma_c = pymatrix_to_Carrayptrs(sigma);
231
232     // Initialize collector arrays
233     npy_intp a_dims[2] = {max_mat, 1};
234     npy_intp b_dims[2] = {max_mat, delta_1_rows};
235     int b_pre_rows = delta_1_rows;
236
237     double a_pre[max_mat];
238     double b_pre[max_mat * delta_1_rows];
239     double *a_fin = (double*) malloc(max_mat*sizeof(double));
240     double *b_fin = (double*) malloc(max_mat * delta_1_rows * sizeof(double));
241
242     if (a_fin==NULL) {
243         printf("Failed to allocate memory for a_fin\n");
244     }
245     if (b_fin==NULL) {
246         printf("Failed to allocate memory for b_fin\n");
247     }
248
249     // Initialize intermediate arrays
250     // Elements for a_pre calculation
251     double dot_sig_lam_0_c[sigma_rows * lam_0_cols];
252     double diff_mu_sigl_c[mu_rows];
253     double dot_bpre_mu_sigl_c[1];
254
255     double dot_b_pre_sig_c[sigma_cols];
256     double dot_b_sigt_c[sigma_rows];
257     double dot_b_sst_bt_c[1];
258
259     // Elements for b_pre calculation
260     double dot_sig_lam_1_c[sigma_rows * lam_1_cols];
261     double diff_phi_sig_c[phi_rows * phi_cols];
262     double dot_phisig_b_c[phi_cols];
263
264     // Perform operations
265     a_pre[0] = -delta_0_c[0];
266     a_fin[0] = -a_pre[0];
267     for (i = 0; i < delta_1_rows; i++) {

```



```

268     b_pre[i] = -delta_1_c[i];
269     b_fin[i] = -b_pre[i];
270 }
271
272 double b_pre_mat_c[b_pre_rows];
273
274 // Calculate unchanging elements
275 mat_prodct(sigma_rows, sigma_cols, sigma_c,
276            lam_0_cols, lam_0_c,
277            dot_sig_lam_0_c);
278 mat_subtract(mu_rows, mu_cols, mu_c, dot_sig_lam_0_c, diff_mu_sigl_c);
279
280 for (mat = 0; mat < (max_mat - 1); mat++) {
281
282     next_mat = mat + 1;
283
284     // Setup indexes
285     bp_offset = mat * delta_1_rows;
286     bp_noffset = next_mat * delta_1_rows;
287
288     // Need this b_pre_mat for proper array reading
289     for (i = 0; i < b_pre_rows; i++) {
290         b_pre_mat_c[i] = b_pre[bp_offset + i];
291     }
292
293     mat_prodct_tpose1(b_pre_rows, 1, b_pre_mat_c,
294                      1, diff_mu_sigl_c,
295                      dot_bpre_mu_sigl_c);
296
297     mat_prodct_tpose1(b_pre_rows, 1, b_pre_mat_c,
298                      sigma_cols, sigma_c,
299                      dot_b_pre_sig_c);
300     mat_prodct_tpose2(1, sigma_cols, dot_b_pre_sig_c,
301                      sigma_rows, sigma_c,
302                      dot_b_sigt_c);
303     mat_prodct(1, sigma_rows, dot_b_sigt_c,
304               1, b_pre_mat_c,
305               dot_b_sst_bt_c);
306
307     // Divisor to prepare for b_fin calculation

```

```

308     divisor = (double)1 / ((double)next_mat + (double)1);
309
310     a_pre[next_mat] = a_pre[mat] + dot_bpre_mu_sig1_c[0] +
311         (half * dot_b_sst_bt_c[0]) - delta_0_c[0];
312     a_fin[next_mat] = -a_pre[next_mat] * divisor;
313
314     // Calculate next b elements
315     mat_prodcct(sigma_rows, sigma_cols, sigma_c,
316         lam_1_cols, lam_1_c,
317         dot_sig_lam_1_c);
318     mat_subtract(phi_rows, phi_cols, phi_c, dot_sig_lam_1_c,
319         diff_phi_sig_c);
320     mat_prodcct_tpose1(phi_rows, phi_cols, diff_phi_sig_c,
321         1, b_pre_mat_c,
322         dot_phisig_b_c);
323
324
325     for (i = 0; i < delta_1_rows; i++) {
326         b_pre[bp_noffset + i] = dot_phisig_b_c[i] - delta_1_c[i];
327         b_fin[bp_noffset + i] = -b_pre[bp_noffset + i] * divisor;
328     }
329 }
330
331 // Free core arrays
332 free(lam_0_c);
333 free(lam_1_c);
334 free(delta_0_c);
335 free(delta_1_c);
336 free(mu_c);
337 free(phi_c);
338 free(sigma_c);
339
340 a_fin_array = (PyArrayObject *) PyArray_SimpleNewFromData(2, a_dims,
341     NPY_DOUBLE,
342     a_fin);
343 PyArray_FLAGS(a_fin_array) |= NPY_OWNDATA;
344 b_fin_array = (PyArrayObject *) PyArray_SimpleNewFromData(2, b_dims,
345     NPY_DOUBLE,
346     b_fin);
347 PyArray_FLAGS(b_fin_array) |= NPY_OWNDATA;

```

```

348
349     PyObject *Result = Py_BuildValue("00", a_fin_array, b_fin_array);
350
351     // Set proper reference counts for numpy arrays
352     Py_DECREF(a_fin_array);
353     Py_DECREF(b_fin_array);
354
355     return Result;
356 }
357
358 /* ==== Create Carray from PyArray =====
359     Assumes PyArray is contiguous in memory.
360     Memory is allocated!                                     */
361 double *pymatrix_to_Carrayptrs(PyArrayObject *arrayin) {
362     double *c, *a, *inc;
363     int i, mat_size, n, m;
364
365     n = arrayin->dimensions[0];
366     m = arrayin->dimensions[1];
367     mat_size = n * m;
368     c = malloc(n * m * sizeof(*c));
369     a = (double *) arrayin->data;
370     inc = c;
371     for (i=0; i < mat_size; i++) {
372         *c = *a;
373         c++;
374         a++;
375     }
376     return inc;
377 }
378
379 /* ==== Free a double *vector (vec of pointers) ===== */
380 void free_Carrayptrs(double **v, int rows) {
381     int i;
382     for (i = 0; i < rows; i++) {
383         free(*(v + i));
384     }
385     free(v);
386 }
387

```

```
388 void free_CarrayfPy(double **v) {  
389     free((char*) v);  
390 }
```

Listing D.2: Affine package

```

1 """
2 The class provides Affine, intended to solve affine models of the
3 term structure
4 This class inherits from the statsmodels LikelihoodModel class
5 """
6
7 import numpy as np
8 import statsmodels.api as sm
9 import pandas as pa
10 import scipy.linalg as la
11 import re
12
13 from numpy import linalg as nla
14 from numpy import ma
15 from scipy.optimize import fmin_l_bfgs_b
16 from statsmodels.tsa.api import VAR
17 from statsmodels.base.model import LikelihoodModel
18 from statsmodels.regression.linear_model import OLS
19 from statsmodels.tools.numdiff import approx_hess, approx_fprime
20 from statsmodels.tsa.kalmanf.kalmanfilter import StateSpaceModel, kalmanfilter
21 from operator import itemgetter
22 from scipy import optimize
23 from util import retry
24
25 try:
26     from . import _C_extensions
27     avail_fast_gen_pred = True
28 except:
29     avail_fast_gen_pred = False
30
31 #####
32 # Create affine class system #
33 #####
34
35 class Affine(LikelihoodModel, StateSpaceModel):
36     """
37     Provides affine model of the term structure
38     """

```

```

39 def __init__(self, yc_data, var_data, lags, neqs, mats, lam_0_e, lam_1_e,
40               delta_0_e, delta_1_e, mu_e, phi_e, sigma_e, latent=0,
41               no_err=None, adjusted=False, use_C_extension=True):
42     """
43     Attempts to instantiate an affine model object
44     yc_data : DataFrame
45         yield curve data
46     var_data : DataFrame
47         data for var model
48     lags : int
49         number of lags for VAR system
50         Only respected when adjusted=False
51     neqs : int
52         Number of equations
53         Only respected when adjusted=True
54     mats : list of int
55         Maturities in periods of yields included in yc_data
56     latent: int
57         Number of latent variables to estimate
58     no_err : list of ints
59         list of the column indexes of yields to be measured without error
60         ex: [0, 3, 4]
61         (1st, 4th, and 5th columns in yc_data to be estimated without
62         error)
63
64     For all estimate parameter arrays:
65         elements marked with 'E' or 'e' are estimated
66         n = number of variables in fully-specified VAR(1) at t
67
68     lam_0_e : Numpy masked array, n x 1
69         constant vector of risk pricing equation
70     lam_1_e : Numpy masked array, n x n
71         parameter array of risk pricing equation
72     delta_0_e : Numpy masked array, 1 x 1
73         constant in short-rate equation
74     delta_1_e : Numpy masked array, n x 1
75         parameter vector in short-rate equation
76     mu_e : Numpy masked array, n x 1
77         constant vector for VAR process
78     phi_e : Numpy masked array, n x n

```

```

79         parameter array for VAR process
80     sigma_e : Numpy masked array, n x n
81         covariance array for VAR process
82     """
83     self.yc_data = yc_data
84     self.var_data = var_data
85     self.yc_names = yc_data.columns
86     self.num_yields = len(yc_data.columns)
87     self.yobs = len(yc_data)
88     self.names = names = var_data.columns
89     k_ar = self.k_ar = lags
90     if neqs:
91         self.neqs = neqs
92     else:
93         neqs = self.neqs = len(names)
94
95     self.latent = latent
96
97     self.lam_0_e = lam_0_e
98     self.lam_1_e = lam_1_e
99     self.delta_0_e = delta_0_e
100    self.delta_1_e = delta_1_e
101
102    self.mu_e = mu_e
103    self.phi_e = phi_e
104    self.sigma_e = sigma_e
105
106    # generates mats: list of mats in yield curve data
107    self.mats = mats
108    self.max_mat = max(mats)
109
110    if latent:
111        self.lat = latent
112    else:
113        self.lat = 0
114
115    self.no_err = no_err
116    if no_err:
117        # parameters for identification of yields measured without error
118        self.err = list(set(range(len(mats))).difference(no_err))

```



```

159
160     self.periods = len(self.var_data_vert)
161     self.guess_length = self._gen_guess_length()
162     assert self.guess_length > 0, "guess_length must be at least 1"
163
164     # final size checks
165     self._size_checks()
166
167     super(Affine, self).__init__(var_data_vert)
168
169     def solve(self, guess_params, method, alg="newton", attempts=5,
170               maxfev=10000, maxiter=10000, ftol=1e-8, xtol=1e-8, x1l0=[0],
171               ntrain=1, penalty=False, upperbounds=None, lowerbounds=None,
172               full_output=False, **kwargs):
173         """
174         Returns tuple of arrays
175         Attempt to solve affine model based on instantiated object.
176
177         Parameters
178         -----
179         guess_params : list
180             List of starting values for parameters to be estimated
181             In row-order and ordered as masked arrays
182
183         method : string
184             solution method
185             nls = nonlinear least squares
186             ml = direct maximum likelihood
187             kalman = kalman filter derived maximum likelihood
188         alg : str {'newton', 'nm', 'bfgs', 'powell', 'cg', or 'nlg'}
189             algorithm used for numerical approximation
190             Method can be 'newton' for Newton-Raphson, 'nm' for Nelder-Mead,
191             'bfgs' for Broyden-Fletcher-Goldfarb-Shanno, 'powell' for modified
192             Powell's method, 'cg' for conjugate gradient, or 'nlg' for Newton-
193             conjugate gradient. 'method' determines which solver from
194             scipy.optimize is used. The explicit arguments in 'fit' are passed
195             to the solver. Each solver has several optional arguments that are
196             not the same across solvers. See the notes section below (or
197             scipy.optimize) for the available arguments.
198         attempts : int

```

```

199         Number of attempts to retry solving if singular matrix Exception
200         raised by Numpy
201
202     scipy.optimize params
203     maxfev : int
204         maximum number of calls to the function for solution alg
205     maxiter : int
206         maximum number of iterations to perform
207     ftol : float
208         relative error desired in sum of squares
209     xtol : float
210         relative error desired in the approximate solution
211     full_output : bool
212         non_zero to return all optional outputs
213
214     Returns
215     -----
216     Returns tuple contains each of the parameter arrays with the optimized
217     values filled in:
218     lam_0 : numpy array
219     lam_1 : numpy array
220     delta_0 : numpy array
221     delta_1 : numpy array
222     mu : numpy array
223     phi : numpy array
224     sigma : numpy array
225
226     The final A, B, and parameter set arrays used to construct the yields
227     a_solve : numpy array
228     b_solve : numpy array
229     solve_params : list
230
231     Other results are also attached, depending on the solution method
232     if 'nls':
233         solv_cov : numpy array
234             Contains the implied covariance matrix of solve_params
235     if 'ml' and 'latent' > 0:
236         var_data_wunob : numpy
237             The modified factor array with the unobserved factors attached
238     """

```

```

239     k_ar = self.k_ar
240     neqs = self.neqs
241     mats = self.mats
242     latent = self.latent
243     yc_data = self.yc_data
244     var_data_vert = self.var_data_vert
245
246     if method == "kalman" and not self.latent:
247         raise NotImplementedError( \
248             "Kalman filter not supported with no latent factors")
249
250     elif method == "nls":
251         func = self._affine_pred
252         var_data_vert_tpose = var_data_vert.T
253         # need to stack for scipy nls
254         yield_stack = np.array(yc_data).reshape(-1, order='F').tolist()
255         # run optimization
256         solver = retry(optimize.curve_fit, attempts)
257         reslt = solver(func, var_data_vert_tpose, yield_stack, p0=guess_params,
258                       maxfev=maxfev, xtol=xtol, ftol=ftol,
259                       full_output=True, **kwargs)
260         solve_params = reslt[0]
261         solv_cov = reslt[1]
262
263     elif method == "ml":
264         assert len(self.no_err) == self.lat, \
265             "Number of columns estimated without error must match " + \
266             "number of latent variables"
267
268         if method == "bfgs-b":
269             func = self.nloglike
270             bounds = self._gen_bounds(lowerbounds, upperbounds)
271             reslt = fmin_l_bfgs_b(x0=guess_params, approx_grad=True,
272                                 bounds=bounds, m=1e7, maxfun=maxfev,
273                                 maxiter=maxiter, **kwargs)
274             solve_params = reslt[0]
275             score = self.score(solve_params)
276
277         else:
278             reslt = self.fit(start_params=guess_params, method=alg,

```

```

279             maxiter=maxiter, maxfun=maxfev, xtol=xtol,
280             ftol=ftol, **kwargs)
281         solve_params = reslt.params
282         score = self.score(solve_params)
283         self.estimate_mlresult = reslt
284
285     elif method == "kalman":
286         self.fit_kalman(start_params=guess_params, method=alg, xi10=xi10,
287                        ntrain=ntrain, penalty=penalty,
288                        upperbounds=upperbounds, lowerbounds=lowerbounds,
289                        **kwargs)
290         solve_params = self.params
291         score = self.score(solve_params)
292
293         lam_0, lam_1, delta_0, delta_1, mu, phi, sigma = \
294             self.params_to_array(solve_params)
295
296         a_solve, b_solve = self.gen_pred_coef(lam_0, lam_1, delta_0, delta_1,
297                                              mu, phi, sigma)
298
299         if latent:
300             lat_ser, jacob, yield_errs = self._solve_unobs(a_in=a_solve,
301                                                           b_in=b_solve)
302             var_data_wunob = var_data_vert.join(lat_ser)
303
304         # attach solved parameter arrays as attributes of object
305         self.lam_0_solve = lam_0
306         self.lam_1_solve = lam_1
307         self.delta_0_solve = delta_0
308         self.delta_1_solve = delta_1
309         self.mu_solve = mu
310         self.phi_solve = phi
311         self.sigma_solve = sigma
312         self.solve_params = solve_params
313
314         if latent:
315             return lam_0, lam_1, delta_0, delta_1, mu, phi, sigma, a_solve, \
316                    b_solve, solve_params, var_data_wunob
317
318     elif method == "nls":

```

```

319         return lam_0, lam_1, delta_0, delta_1, mu, phi, sigma, a_solve, \
320                b_solve, solv_cov
321
322     elif method == "ml":
323         return lam_0, lam_1, delta_0, delta_1, mu, phi, sigma, \
324                a_solve, b_solve, solve_params
325
326     def score(self, params):
327         """
328         Return the gradient of the loglike at params
329
330         Parameters
331         -----
332         params : list
333
334         Notes
335         -----
336         Return numerical gradient
337         """
338         loglike = self.loglike
339         return approx_fprime(params, loglike, epsilon=1e-8)
340
341     def hessian(self, params):
342         """
343         Returns numerical hessian.
344         """
345         loglike = self.loglike
346         return approx_hess(params, loglike)
347
348     def std_errs(self, params):
349         """
350         Return standard errors
351         """
352         hessian = self.hessian(params)
353         std_err = np.sqrt(-np.diag(la.inv(hessian)))
354         return std_err
355
356     def loglike(self, params):
357         """
358         Returns float

```

```

359     Loglikelihood used in latent factor models
360
361     Parameters
362     -----
363     params : list
364         Values of parameters to pass into masked elements of array
365
366     Returns
367     -----
368     loglikelihood : float
369     """
370
371     lat = self.lat
372     per = self.periods
373     var_data_vert = self.var_data_vert
374     var_data_vertm1 = self.var_data_vertm1
375
376     lam_0, lam_1, delta_0, delta_1, mu, phi, \
377         sigma = self.params_to_array(params)
378
379     if self.fast_gen_pred:
380         solve_a, solve_b = self.opt_gen_pred_coef(lam_0, lam_1, delta_0,
381                                                    delta_1, mu, phi, sigma)
382
383     else:
384         solve_a, solve_b = self.gen_pred_coef(lam_0, lam_1, delta_0,
385                                                delta_1, mu, phi, sigma)
386
387     # first solve for unknown part of information vector
388     lat_ser, jacob, yield_errs = self._solve_unobs(a_in=solve_a,
389                                                    b_in=solve_b)
390
391     # here is the likelihood that needs to be used
392     # use two matrices to take the difference
393     var_data_use = var_data_vert.join(lat_ser)[1:]
394     var_data_usem1 = var_data_vertm1.join(lat_ser.shift())[1:]
395
396     errors = var_data_use.values.T - mu - np.dot(phi,
397                                                    var_data_usem1.values.T)
398     sign, j_logdt = nla.slogdet(jacob)

```

```

399     j_slogdt = sign * j_logdt
400
401     sign, sigma_logdt = nla.slogdet(np.dot(sigma, sigma.T))
402     sigma_slogdt = sign * sigma_logdt
403
404     var_yields_errs = np.var(yield_errs, axis=1)
405
406     like = -(per - 1) * j_slogdt - (per - 1) * 1.0 / 2 * sigma_slogdt - \
407           1.0 / 2 * np.sum(np.dot(np.dot(errors.T, \
408           la.inv(np.dot(sigma, sigma.T))), errors)) - (per - 1) / 2.0 * \
409           np.log(np.sum(var_yields_errs)) - 1.0 / 2 * \
410           np.sum(yield_errs**2/var_yields_errs[None].T)
411
412     return like
413
414     def nloglike(self, params):
415         """
416         Return negative loglikelihood
417         Negative Loglikelihood used in latent factor models
418         """
419         like = self.loglike(params)
420         return -like
421
422     def gen_pred_coef(self, lam_0, lam_1, delta_0, delta_1, mu, phi, sigma):
423         """
424         Returns tuple of arrays
425         Generates prediction coefficient vectors A and B
426
427         Parameters
428         -----
429         lam_0 : numpy array
430         lam_1 : numpy array
431         delta_0 : numpy array
432         delta_1 : numpy array
433         mu : numpy array
434         phi : numpy array
435         sigma : numpy array
436
437         Returns
438         -----

```

```

439     a_solve : numpy array
440         Array of constants relating factors to yields
441     b_solve : numpy array
442         Array of coeffiencts relating factors to yields
443     """
444     max_mat = self.max_mat
445     b_width = self.k_ar * self.neqs + self.lat
446     half = float(1)/2
447     # generate predictions
448     a_pre = np.zeros((max_mat, 1))
449     a_pre[0] = -delta_0
450     b_pre = np.zeros((max_mat, b_width))
451     b_pre[0] = -delta_1[:,0]
452
453     n_inv = float(1) / np.add(range(max_mat), 1).reshape((max_mat, 1))
454     a_solve = -a_pre.copy()
455     b_solve = -b_pre.copy()
456
457     for mat in range(max_mat-1):
458         a_pre[mat + 1] = (a_pre[mat] + np.dot(b_pre[mat].T, \
459             (mu - np.dot(sigma, lam_0))) + \
460             (half)*np.dot(np.dot(np.dot(b_pre[mat].T, sigma),
461                 sigma.T), b_pre[mat]) - delta_0)[0][0]
462         a_solve[mat + 1] = -a_pre[mat + 1] * n_inv[mat + 1]
463         b_pre[mat + 1] = np.dot((phi - np.dot(sigma, lam_1)).T, \
464             b_pre[mat]) - delta_1[:, 0]
465         b_solve[mat + 1] = -b_pre[mat + 1] * n_inv[mat + 1]
466
467     return a_solve, b_solve
468
469 def opt_gen_pred_coef(self, lam_0, lam_1, delta_0, delta_1, mu, phi,
470     sigma):
471     """
472     Returns tuple of arrays
473     Generates prediction coefficient vectors A and B in fast C function
474
475     Parameters
476     -----
477     lam_0 : numpy array
478     lam_1 : numpy array

```



```

479         delta_0 : numpy array
480         delta_1 : numpy array
481         mu : numpy array
482         phi : numpy array
483         sigma : numpy array
484
485     Returns
486     -----
487     a_solve : numpy array
488         Array of constants relating factors to yields
489     b_solve : numpy array
490         Array of coefficients relating factors to yields
491     """
492     max_mat = self.max_mat
493
494     return _C_extensions.gen_pred_coef(lam_0, lam_1, delta_0, delta_1, mu,
495                                       phi, sigma, max_mat)
496
497 def params_to_array(self, params, return_mask=False):
498     """
499     Returns tuple of arrays
500     Process params input into appropriate arrays
501
502     Parameters
503     -----
504     params : list
505         list of values to fill in masked values
506     return_mask : boolean
507
508
509     Returns
510     -----
511     lam_0 : numpy array
512     lam_1 : numpy array
513     delta_0 : numpy array
514     delta_1 : numpy array
515     mu : numpy array
516     phi : numpy array
517     sigma : numpy array
518     """

```

```

519     lam_0_e = self.lam_0_e.copy()
520     lam_1_e = self.lam_1_e.copy()
521     delta_0_e = self.delta_0_e.copy()
522     delta_1_e = self.delta_1_e.copy()
523     mu_e = self.mu_e.copy()
524     phi_e = self.phi_e.copy()
525     sigma_e = self.sigma_e.copy()
526
527     all_arrays = [lam_0_e, lam_1_e, delta_0_e, delta_1_e, mu_e, phi_e,
528                  sigma_e]
529
530     arg_sep = self._gen_arg_sep([ma.count_masked(struct) for struct in \
531                                all_arrays])
532
533     for pos, struct in enumerate(all_arrays):
534         struct[ma.getmask(struct)] = params[arg_sep[pos]:arg_sep[pos + 1]]
535         if not return_mask:
536             all_arrays[pos] = np.ascontiguousarray(struct,
537                                                    dtype=np.float64)
538
539     return tuple(all_arrays)
540
541 def params_to_array_zeromask(self, params):
542     """
543     Returns tuple of arrays + list
544     Process params input into appropriate arrays by setting them to zero if
545     param in params in zero and removing them from params, otherwise they
546     stay in params and value remains masked
547
548     Parameters
549     -----
550     params : list
551         list of values to fill in masked values
552
553     Returns
554     -----
555     lam_0 : numpy array
556     lam_1 : numpy array
557     delta_0 : numpy array
558     delta_1 : numpy array

```

```

559     mu : numpy array
560     phi : numpy array
561     sigma : numpy array
562     guesses : list
563         List of remaining params after filtering and filling those that
564         were zero
565     """
566     paramcopy = params[:]
567     lam_0_e = self.lam_0_e.copy()
568     lam_1_e = self.lam_1_e.copy()
569     delta_0_e = self.delta_0_e.copy()
570     delta_1_e = self.delta_1_e.copy()
571     mu_e = self.mu_e.copy()
572     phi_e = self.phi_e.copy()
573     sigma_e = self.sigma_e.copy()
574
575     all_arrays = [lam_0_e, lam_1_e, delta_0_e, delta_1_e, mu_e, phi_e,
576                  sigma_e]
577
578     arg_sep = self._gen_arg_sep([ma.count_masked(struct) for struct in \
579                                all_arrays])
580
581     guesses = []
582     # check if each element is masked or not
583     for struct in all_arrays:
584         it = np.nditer(struct.mask, flags=['multi_index'])
585         while not it.finished:
586             if it[0]:
587                 val = paramcopy.pop(0)
588                 if val == 0:
589                     struct[it.multi_index] = 0
590             else:
591                 guesses.append(val)
592             it.iternext()
593
594     return tuple(all_arrays + [guesses])
595
596 def _updateloglike(self, params, xil0, ntrain, penalty, upperbounds,
597                  lowerbounds, F, A, H, Q, R, history):
598     """

```

```

599     Returns combined loglikelihood for kalman filter
600     Ignores F,A,H,Q,R,
601     """
602     paramsorig = params
603     if penalty:
604         params = np.min((np.max((lowerbounds, params), axis=0), upperbounds),
605                         axis=0)
606
607     mats = self.mats
608     per = self.periods
609     lat = self.lat
610
611     yc_data = self.yc_data
612     X = self.var_data_vertc
613
614     obsdim = self.neqs * self.k_ar
615     dim = obsdim + lat
616
617     lam_0, lam_1, delta_0, delta_1, mu, phi, sigma = \
618         self.params_to_array(params=params)
619
620     solve_a, solve_b = self.opt_gen_pred_coef(lam_0, lam_1, delta_0,
621                                              delta_1, mu, phi, sigma)
622
623     F = phi[-lat:, -lat:]
624     Q = sigma[-lat:, -lat:]
625     R = np.zeros((1, 1))
626
627     # initialize kalman to zero
628     loglike = 0
629
630     # calculate likelihood for each maturity estimated
631     for mix, mat in enumerate(self.mats):
632         obsparams = np.concatenate((solve_a[mat-1],
633                                    solve_b[mat-1][: -lat]))
634
635         A = obsparams
636         H = solve_b[mat-1][-lat:]
637         y = yc_data.values[:, mix]
638         loglike += kalmanfilter(F, A, H, Q, R, y, X, x10, ntrain, history)

```

```

639         if penalty:
640             loglike += penalty * np.sum((paramsorig-params)**2)
641
642         return loglike
643
644     def _solve_unobs(self, a_in, b_in):
645         """
646         Solves for unknown factors
647
648         Parameters
649         -----
650         a_in : list of floats (periods)
651             List of elements for A constant in factors -> yields relationship
652         b_in : array (periods, neqs * k_ar + lat)
653             Array of elements for B coefficients in factors -> yields
654             relationship
655
656         Returns
657         -----
658         var_data_c : DataFrame
659             VAR data including unobserved factors
660         jacob : array (neqs * k_ar + num_yields)**2
661             Jacobian used in likelihood
662         yield_errs : array (num_yields - lat, periods)
663             The errors for the yields estimated with error
664         """
665         yc_data = self.yc_data
666         var_data_vert = self.var_data_vert
667         yc_names = self.yc_names
668         num_yields = self.num_yields
669         names = self.names
670         k_ar = self.k_ar
671         neqs = self.neqs
672         lat = self.lat
673         no_err = self.no_err
674         err = self.err
675         no_err_mat = self.no_err_mat
676         err_mat = self.err_mat
677         noerr_cols = self.noerr_cols
678         err_cols = self.err_cols

```

```

679
680     yc_data_names = yc_names.tolist()
681     no_err_num = len(noerr_cols)
682     err_num = len(err_cols)
683
684     # need to combine the two matrices
685     # these matrices will collect the final values
686     a_all = np.zeros([num_yields, 1])
687     b_all_obs = np.zeros([num_yields, neqs * k_ar])
688     b_all_unobs = np.zeros([num_yields, lat])
689
690     a_sel = np.zeros([no_err_num, 1])
691     b_sel_obs = np.zeros([no_err_num, neqs * k_ar])
692     b_sel_unobs = np.zeros([no_err_num, lat])
693     for ix, y_pos in enumerate(no_err):
694         a_sel[ix, 0] = a_in[no_err_mat[ix] - 1]
695         b_sel_obs[ix, :] = b_in[no_err_mat[ix] - 1, :neqs * k_ar]
696         b_sel_unobs[ix, :] = b_in[no_err_mat[ix] - 1, neqs * k_ar:]
697
698         a_all[y_pos, 0] = a_in[no_err_mat[ix] - 1]
699         b_all_obs[y_pos, :] = b_in[no_err_mat[ix] - 1][:neqs * k_ar]
700         b_all_unobs[y_pos, :] = b_in[no_err_mat[ix] - 1][neqs * k_ar:]
701
702     # now solve for unknown factors using long arrays
703     unobs = np.dot(la.inv(b_sel_unobs),
704                    yc_data.filter(items=noerr_cols).values.T - a_sel - \
705                    np.dot(b_sel_obs, var_data_vert.T))
706
707     # re-initialize a_sel, b_sel_obs, and b_sel_unobs
708     a_sel = np.zeros([err_num, 1])
709     b_sel_obs = np.zeros([err_num, neqs * k_ar])
710     b_sel_unobs = np.zeros([err_num, lat])
711     for ix, y_pos in enumerate(err):
712         a_all[y_pos, 0] = a_sel[ix, 0] = a_in[err_mat[ix] - 1]
713         b_all_obs[y_pos, :] = b_sel_obs[ix, :] = \
714             b_in[err_mat[ix] - 1][:neqs * k_ar]
715         b_all_unobs[y_pos, :] = b_sel_unobs[ix, :] = \
716             b_in[err_mat[ix] - 1][neqs * k_ar:]
717
718     yield_errs = yc_data.filter(items=err_cols).values.T - a_sel - \

```

```

719         np.dot(b_sel_obs, var_data_vert.T) - \
720         np.dot(b_sel_unobs, unobs)
721
722     lat_ser = pa.DataFrame(index=var_data_vert.index)
723     for factor in range(lat):
724         lat_ser["latent_" + str(factor)] = unobs[factor, :]
725     meas_mat = np.zeros((num_yields, err_num))
726
727     for col_index, col in enumerate(err_cols):
728         row_index = yc_data_names.index(col)
729         meas_mat[row_index, col_index] = 1
730
731     jacob = self._construct_J(b_obs=b_all_obs, b_unobs=b_all_unobs,
732                             meas_mat=meas_mat)
733
734
735     return lat_ser, jacob, yield_errs
736
737     def _affine_pred(self, data, *params):
738         """
739         Function based on lambda and data that generates predicted yields
740         data : DataFrame
741         params : tuple of floats
742             parameter guess
743         """
744         mats = self.mats
745         yc_data = self.yc_data
746
747         lam_0, lam_1, delta_0, delta_1, mu, phi, sigma \
748             = self.params_to_array(params)
749
750         if self.fast_gen_pred:
751             solve_a, solve_b = self.opt_gen_pred_coef(lam_0, lam_1, delta_0,
752                                                         delta_1, mu, phi, sigma)
753
754         else:
755             solve_a, solve_b = self.gen_pred_coef(lam_0, lam_1, delta_0,
756                                                    delta_1, mu, phi, sigma)
757
758     pred = []

```

```

759         for i in mats:
760             pred.extend((solve_a[i-1] + np.dot(solve_b[i-1], data)).tolist())
761         return pred
762
763     def _gen_arg_sep(self, arg_lengths):
764         """
765         Generates list of positions
766         """
767         arg_sep = [0]
768         pos = 0
769         for length in arg_lengths:
770             arg_sep.append(length + pos)
771             pos += length
772         return arg_sep
773
774     def _gen_col_names(self):
775         """
776         Generate column names for err and noerr
777         """
778         yc_names = self.yc_names
779         no_err = self.no_err
780         err = self.err
781         noerr_cols = []
782         err_cols = []
783         for index in no_err:
784             noerr_cols.append(yc_names[index])
785         for index in err:
786             err_cols.append(yc_names[index])
787         return noerr_cols, err_cols
788
789     def _gen_mat_list(self):
790         """
791         Generate list of mats measured with and without error
792         """
793         yc_names = self.yc_names
794         no_err = self.no_err
795         mats = self.mats
796         err = self.err
797
798         no_err_mat = []

```



```

799     err_mat = []
800
801     for index in no_err:
802         no_err_mat.append(mats[index])
803     for index in err:
804         err_mat.append(mats[index])
805
806     return no_err_mat, err_mat
807
808     def _construct_J(self, b_obs, b_unobs, meas_mat):
809         """
810         Construct jacobian matrix
811         meas_mat : array
812         """
813         k_ar = self.k_ar
814         neqs = self.neqs
815         lat = self.lat
816         num_yields = self.num_yields
817         num_obsrv = neqs * k_ar
818
819         msize = neqs * k_ar + num_yields
820         jacob = np.zeros([msize, msize])
821         jacob[:num_obsrv, :num_obsrv] = np.identity(neqs*k_ar)
822
823         jacob[num_obsrv:, :num_obsrv] = b_obs
824         jacob[num_obsrv:, num_obsrv:num_obsrv + lat] = b_unobs
825         jacob[num_obsrv:, num_obsrv + lat:] = meas_mat
826
827         return jacob
828
829     def _gen_guess_length(self):
830         lam_0_e = self.lam_0_e
831         lam_1_e = self.lam_1_e
832         delta_0_e = self.delta_0_e
833         delta_1_e = self.delta_1_e
834         mu_e = self.mu_e
835         phi_e = self.phi_e
836         sigma_e = self.sigma_e
837
838         all_arrays = [lam_0_e, lam_1_e, delta_0_e, delta_1_e, mu_e, phi_e,

```

```

839             sigma_e]
840
841     count = 0
842     for struct in all_arrays:
843         count += ma.count_masked(struct)
844
845     return count
846
847     def _size_checks(self):
848         """
849         Run size checks on parameter arrays
850         """
851         dim = self.neqs * self.k_ar + self.lat
852         assert np.shape(self.lam_0_e) == (dim, 1), "Shape of lam_0_e incorrect"
853         assert np.shape(self.lam_1_e) == (dim, dim), \
854             "Shape of lam_1_e incorrect"
855
856         assert np.shape(self.delta_1_e) == (dim, 1), "Shape of delta_1_e" \
857             "incorrect"
858         assert np.shape(self.mu_e) == (dim, 1), "Shape of mu incorrect"
859         assert np.shape(self.phi_e) == (dim, dim), \
860             "Shape of phi_e incorrect"
861         assert np.shape(self.sigma_e) == (dim, dim), \
862             "Shape of sig_e incorrect"
863
864     def _gen_bounds(self, lowerbounds, upperbounds):
865         if lowerbounds or upperbounds:
866             bounds = []
867             for bix in range(max(len(lowerbounds), len(upperbounds))):
868                 tbound = []
869                 if lowerbounds:
870                     tbound.append(lowerbounds[bix])
871                 else:
872                     tbound.append(-np.inf)
873                 if upperbounds:
874                     tbound.append(upperbounds[bix])
875                 else:
876                     tbound.append(np.inf)
877                 bounds.append(tuple(tbound))
878         else:

```

879 `return None`

Listing D.3: Unit tests

```

1 """
2 Affine unit tests
3
4 For the following in the docs:
5     L = number of lags in VAR process governing pricing kernel
6     O = number of observed factors in VAR process governing pricing kernel
7     U = number of unobserved, latent factors in VAR process governing
8     pricing kernel
9 """
10 from unittest import TestCase
11
12 import unittest
13 import numpy as np
14 import numpy.ma as ma
15 import pandas as pa
16
17 from affine.constructors.helper import make_nomask
18 from affine.model.affine import Affine
19
20 # parameters for running tests
21 test_size = 100
22 lags = 4
23 neqs = 5
24 nyields = 5
25 latent = 1
26
27 class TestInitialize(TestCase):
28     """
29     Tests for methods related to instantiation of a new Affine object
30     """
31     def setUp(self):
32
33         np.random.seed(100)
34
35         # initialize yield curve and VAR observed factors
36         yc_data_test = pa.DataFrame(np.random.random((test_size - lags,
37                                                         nyields)))
38         var_data_test = pa.DataFrame(np.random.random((test_size, neqs)))

```

```

39     mats = list(range(1, nyields + 1))
40
41     # initialize masked arrays
42     self.dim = dim = lags * neqs
43     lam_0 = make_nomask([dim, 1])
44     lam_1 = make_nomask([dim, dim])
45     delta_0 = make_nomask([1, 1])
46     delta_1 = make_nomask([dim, 1])
47     mu = make_nomask([dim, 1])
48     phi = make_nomask([dim, dim])
49     sigma = make_nomask([dim, dim])
50
51     # Setup some of the elements as non-zero
52     # This sets up a fake model where only lambda_0 and lambda_1 are
53     # estimated
54     lam_0[:neqs] = ma.masked
55     lam_1[:neqs, :neqs] = ma.masked
56     delta_0[:, :] = np.random.random(1)
57     delta_1[:neqs] = np.random.random((neqs, 1))
58     mu[:neqs] = np.random.random((neqs, 1))
59     phi[:neqs, :] = np.random.random((neqs, dim))
60     sigma[:, :] = np.identity(dim)
61
62     self.mod_kwargs = {
63         'yc_data': yc_data_test,
64         'var_data': var_data_test,
65         'lags': lags,
66         'neqs': neqs,
67         'mats': mats,
68         'lam_0_e': lam_0,
69         'lam_1_e': lam_1,
70         'delta_0_e': delta_0,
71         'delta_1_e': delta_1,
72         'mu_e': mu,
73         'phi_e': phi,
74         'sigma_e': sigma
75     }
76
77     def test_create_correct(self):
78         """

```

```

79     Tests whether __init__ successfully initializes an Affine model object.
80     If the Affine object does not successfully instantiate, then this test
81     fails, otherwise it passes.
82     """
83     model = Affine(**self.mod_kwargs)
84     self.assertIsInstance(model, Affine)
85
86 def test_wrong_lam0_size(self):
87     """
88     Tests whether size check asserts for lam_0_e is implemented
89     correctly. If the lam_0_e parameter is not of the correct size,
90     which is (L * O + U) by 1, then an assertion error should be raised,
91     resulting in a passed test. If lam_0_e is of the incorrect size and
92     no assertion error is raised, this test fails.
93     """
94     mod_kwargs = self.mod_kwargs
95     # lam_0_e of incorrect size
96     mod_kwargs['lam_0_e'] = make_nomask([self.dim - 1, 1])
97     self.assertRaises(AssertionError, Affine, **mod_kwargs)
98
99 def test_wrong_lam1_size(self):
100     """
101     Tests whether size check asserts for lam_1_e is implemented correctly.
102     If the lam_1_e parameter is not of the correct size, which is (L
103     * O + U) by (L * O + U), then an assertion error should be raised,
104     resulting in a passed test. If lam_1_e is of the incorrect size and no
105     assertion error is raised, this test fails.
106     """
107     mod_kwargs = self.mod_kwargs
108     # lam_1_e of incorrect size
109     mod_kwargs['lam_1_e'] = make_nomask([self.dim - 1, self.dim + 1])
110     self.assertRaises(AssertionError, Affine, **mod_kwargs)
111
112 def test_wrong_delta_1_size(self):
113     """
114     Tests whether size check asserts for delta_1_e is implemented
115     correctly. If the delta_1_e parameter is not of the correct size, which
116     is (L * O + U) by 1, then an assertion error should be raised,
117     resulting in a passed test. If delta_1_e is of the incorrect size and
118     no assertion error is raised, this test fails.

```

```

119     """
120     mod_kwargs = self.mod_kwargs
121     # delta_1_e of incorrect size
122     mod_kwargs['delta_1_e'] = make_nomask([self.dim + 1, 1])
123     self.assertRaises(AssertionError, Affine, **mod_kwargs)
124
125     def test_wrong_mu_e_size(self):
126         """
127         Tests whether size check asserts for mu_e is implemented correctly. If
128         the mu_e parameter is not of the correct size, which is (L * 0 + U) by
129         1, then an assertion error should be raised, resulting in a passed
130         test. If mu_e is of the incorrect size and no assertion error is
131         raised, this test fails.
132         """
133         mod_kwargs = self.mod_kwargs
134         # mu_e of incorrect size
135         mod_kwargs['mu_e'] = make_nomask([self.dim + 2, 1])
136         self.assertRaises(AssertionError, Affine, **mod_kwargs)
137
138     def test_wrong_phi_e_size(self):
139         """
140         Tests whether size check asserts for phi_e is implemented correctly.
141         If the phi_e parameter is not of the correct size, which is (L * 0 + U)
142         by (L * 0 + U), then an assertion error should be raised, resulting in
143         a passed test. If phi_e is of the incorrect size and no assertion error
144         is raised, this test fails.
145         """
146         mod_kwargs = self.mod_kwargs
147         # phi_e of incorrect size
148         mod_kwargs['phi_e'] = make_nomask([self.dim + 2, self.dim - 1])
149         self.assertRaises(AssertionError, Affine, **mod_kwargs)
150
151     def test_wrong_sigma_e_size(self):
152         """
153         Tests whether size check asserts for sigma_e is implemented correctly.
154         If the sigma_e parameter is not of the correct size, which is (L
155         * 0 + U) by (L * 0 + U), then an assertion error should be raised,
156         resulting in a passed test. If sigma_e is of the incorrect size and no
157         assertion error is raised, this test fails.
158         """

```

```

159     mod_kwargs = self.mod_kwargs
160     # sigma_e of incorrect size
161     mod_kwargs['sigma_e'] = make_nomask([self.dim - 2, self.dim])
162     self.assertRaises(AssertionError, Affine, **mod_kwargs)
163
164     def test_var_data_nulls(self):
165         """
166         Tests if nulls appear in var_data whether an AssertionError is raised.
167         If any nulls appear in var_data and an AssertionError is raised, the
168         test passes. Otherwise if nulls are passed in and an AssertionError is
169         not raised, the test fails.
170         """
171         mod_kwargs = self.mod_kwargs
172         # replace a value in var_data with null
173         mod_kwargs['var_data'][1, 1] = np.nan
174         self.assertRaises(AssertionError, Affine, **mod_kwargs)
175
176     def test_yc_data_nulls(self):
177         """
178         Tests if nulls appear in yc_data whether AssertionError is raised. If
179         any nulls appear in yc_data and an AssertionError is raised, the test
180         passes. Otherwise if nulls are passed in and an AssertionError is not
181         raised, the test fails.
182         """
183         mod_kwargs = self.mod_kwargs
184         # replace a value in var_data with null
185         mod_kwargs['yc_data'][1, 1] = np.nan
186         self.assertRaises(AssertionError, Affine, **mod_kwargs)
187
188     def test_no_estimated_values(self):
189         """
190         Tests if AssertionError is raised if there are no masked values in
191         the estimation arrays, implying no parameters to be estimated. If
192         the object passed in has no estimated values and an AssertionError
193         is raised, the test passes. Otherwise if no estimated values are
194         passed in and an AssertionError is not raised, the test fails.
195         """
196         mod_kwargs = self.mod_kwargs
197         # replace a value in var_data with null
198         mod_kwargs['lam_0_e'] = make_nomask([self.dim, 1])

```



```

199         mod_kwargs['lam_1_e'] = make_nomask([self.dim, self.dim])
200         self.assertRaises(AssertionError, Affine, **mod_kwargs)
201
202     class TestEstimationSupportMethods(TestCase):
203         """
204         Tests for support methods related to estimating models
205         """
206         def setUp(self):
207
208             np.random.seed(100)
209
210             # initialize yield curve and VAR observed factors
211             yc_data_test = pa.DataFrame(np.random.random((test_size - lags,
212                                                         nyields)))
213             var_data_test = pa.DataFrame(np.random.random((test_size, neqs)))
214             mats = list(range(1, nyields + 1))
215
216             # initialize masked arrays
217             self.dim = dim = lags * neqs + latent
218             lam_0 = make_nomask([dim, 1])
219             lam_1 = make_nomask([dim, dim])
220             delta_0 = make_nomask([1, 1])
221             delta_1 = make_nomask([dim, 1])
222             mu = make_nomask([dim, 1])
223             phi = make_nomask([dim, dim])
224             sigma = make_nomask([dim, dim])
225
226             # Setup some of the elements as non-zero
227             # This sets up a fake model where only lambda_0 and lambda_1 are
228             # estimated
229             lam_0[:neqs] = ma.masked
230             lam_0[-latent:] = ma.masked
231             lam_1[:neqs, :neqs] = ma.masked
232             lam_1[-latent:, -latent:] = ma.masked
233             delta_0[:, :] = np.random.random(1)
234             delta_1[:neqs] = np.random.random((neqs, 1))
235             mu[:neqs] = np.random.random((neqs, 1))
236             phi[:neqs, :] = np.random.random((neqs, dim))
237             sigma[:, :] = np.identity(dim)
238

```

```

239     self.mod_kwargs = {
240         'yc_data': yc_data_test,
241         'var_data': var_data_test,
242         'lags': lags,
243         'neqs': neqs,
244         'mats': mats,
245         'lam_0_e': lam_0,
246         'lam_1_e': lam_1,
247         'delta_0_e': delta_0,
248         'delta_1_e': delta_1,
249         'mu_e': mu,
250         'phi_e': phi,
251         'sigma_e': sigma,
252         'latent': latent,
253         'no_err': [1]
254     }
255
256     self.guess_params = np.random.random((neqs**2 + neqs + (2 * latent),)
257                                         ).tolist()
258
259     self.affine_obj = Affine(**self.mod_kwargs)
260
261     def test_loglike(self):
262         """
263         Tests if loglikelihood is calculated. If the loglikelihood is
264         calculated given a set of parameters, then this test passes.
265         Otherwise, it fails.
266         """
267         self.affine_obj.loglike(self.guess_params)
268
269     def test_score(self):
270         """
271         Tests if score of the likelihood is calculated. If the score
272         calculation succeeds without error, then the test passes. Otherwise,
273         the test fails.
274         """
275         self.affine_obj.score(self.guess_params)
276
277     def test_hessian(self):
278         """
279         Tests if hessian of the likelihood is calculated. If the hessian

```

```

279         calculation succeeds without error, then the test passes. Otherwise,
280         the test fails.
281         """
282         self.affine_obj.hessian(self.guess_params)
283
284     def test_std_errs(self):
285         """
286         Tests if standard errors are calculated. If the standard error
287         calculation succeeds, then the test passes. Otherwise, the test
288         fails.
289         """
290         self.affine_obj.std_errs(self.guess_params)
291
292     def test_params_to_array(self):
293         """
294         Tests if the params_to_array function works correctly, with and without
295         returning masked arrays. In order to pass, the params_to_array function
296         must return masked arrays with the masked elements filled in when the
297         return_mask argument is set to True and contiguous standard numpy
298         arrays when the return_mask argument is False. Otherwise, the test
299         fails.
300         """
301         arrays_no_mask = self.affine_obj.params_to_array(self.guess_params)
302         for arr in arrays_no_mask:
303             self.assertIsInstance(arr, np.ndarray)
304             self.assertNotIsInstance(arr, np.ma.core.MaskedArray)
305         arrays_w_mask = self.affine_obj.params_to_array(self.guess_params,
306                                                         return_mask=True)
307         for arr in arrays_w_mask:
308             self.assertIsInstance(arr, np.ma.core.MaskedArray)
309
310     def test_params_to_array_zeromask(self):
311         """
312         Tests if params_to_array_zeromask function works correctly. In order to
313         pass, params_to_array_zeromask must return masked arrays with the
314         guess_params elements that are zero unmasked and set to zero in the
315         appropriate arrays. The new guess_params array is also returned with
316         those that were 0 removed. If both of these are not returned correctly,
317         the test fails.
318         """

```

```

319     guess_params_arr = np.array(self.guess_params)
320     neqs = self.affine_obj.neqs
321     guess_params_arr[:neqs] = 0
322     guess_params = guess_params_arr.tolist()
323     guess_length = self.affine_obj._gen_guess_length()
324     params_guesses = self.affine_obj.params_to_array_zeromask(guess_params)
325     updated_guesses = params_guesses[-1]
326     self.assertEqual(len(updated_guesses), len(guess_params) - neqs)
327
328     # ensure that number of masked has correctly been set
329     count_masked_new = ma.count_masked(params_guesses[0])
330     count_masked_orig = ma.count_masked(self.affine_obj.lam_0_e)
331     self.assertEqual(count_masked_new, count_masked_orig - neqs)
332
333     def test_gen_pred_coef(self):
334         """
335         Tests if Python-driven gen_pred_coef function runs. If a set of
336         parameter arrays are passed into the gen_pred_coef function and the
337         A and B arrays are returned, then the test passes. Otherwise, the test
338         fails.
339         """
340         params = self.affine_obj.params_to_array(self.guess_params)
341         self.affine_obj.gen_pred_coef(*params)
342
343     def test_opt_gen_pred_coef(self):
344         """
345         Tests if C-driven gen_pred_coef function runs. If a set of parameter
346         arrays are passed into the opt_gen_pred_coef function and the A and
347         B arrays are return, then the test passes. Otherwise, the test fails.
348         """
349         params = self.affine_obj.params_to_array(self.guess_params)
350         self.affine_obj.opt_gen_pred_coef(*params)
351
352     def test_py_C_gen_pred_coef_equal(self):
353         """
354         Tests if the Python-driven and C-driven gen_pred_coef functions produce
355         the same result, up to a precision of 1e-14. If the gen_pred_coef and
356         opt_gen_pred_coef functions produce the same result, then the test
357         passes. Otherwise, the test fails.
358         """

```

```

359     params = self.affine_obj.params_to_array(self.guess_params)
360     py_gpc = self.affine_obj.gen_pred_coef(*params)
361     c_gpc = self.affine_obj.opt_gen_pred_coef(*params)
362     for aix, array in enumerate(py_gpc):
363         np.testing.assert_allclose(array, c_gpc[aix], rtol=1e-14)
364
365     def test__solve_unobs(self):
366         """
367         Tests if the _solve_unobs function runs. If the _solve_unobs function
368         runs and the latent series, likelihood jacobian, and yield errors are
369         returned, then the test passes. Otherwise the test fails.
370         """
371         guess_params = self.guess_params
372         param_arrays = self.affine_obj.params_to_array(guess_params)
373         a_in, b_in = self.affine_obj.gen_pred_coef(*param_arrays)
374         result = self.affine_obj._solve_unobs(a_in=a_in, b_in=b_in)
375
376     def test__affine_pred(self):
377         """
378         Tests if the _affine_pred function runs. If the affine_pred function
379         produces a list of the yields stacked in order of increasing maturity
380         and is of the expected shape, the test passes. Otherwise, the test
381         fails.
382         """
383         lat = self.affine_obj.lat
384         yobs = self.affine_obj.yobs
385         mats = self.affine_obj.mats
386         var_data_vert_tpose = self.affine_obj.var_data_vert.T
387
388         guess_params = self.guess_params
389         latent_rows = np.random.random((lat, yobs))
390         data = np.append(var_data_vert_tpose, latent_rows, axis=0)
391         pred = self.affine_obj._affine_pred(data, *guess_params)
392         self.assertEqual(len(pred), len(mats) * yobs)
393
394     def test__gen_mat_list(self):
395         """
396         Tests if _gen_mat_list generates a length 2 tuple with a list of the
397         maturities estimated without error followed by those estimated with
398         error. If _gen_mat_list produces a tuple of lists of those yields

```

```

399         estimates without error and then those with error, this test passes.
400         Otherwise, the test fails.
401         """
402         no_err_mat, err_mat = self.affine_obj._gen_mat_list()
403         self.assertEqual(no_err_mat, [2])
404         self.assertEqual(err_mat, [1,3,4,5])
405
406     class TestEstimationMethods(TestCase):
407         """
408         Tests for solution methods
409         """
410         def setUp(self):
411
412             ## Non-linear least squares
413             np.random.seed(100)
414
415             # initialize yield curve and VAR observed factors
416             yc_data_test = pa.DataFrame(np.random.random((test_size - lags,
417                                                         nyields)))
418             var_data_test = pa.DataFrame(np.random.random((test_size, neqs)))
419             mats = list(range(1, nyields + 1))
420
421             # initialize masked arrays
422             self.dim_nolat = dim = lags * neqs
423             lam_0 = make_nomask([dim, 1])
424             lam_1 = make_nomask([dim, dim])
425             delta_0 = make_nomask([1, 1])
426             delta_1 = make_nomask([dim, 1])
427             mu = make_nomask([dim, 1])
428             phi = make_nomask([dim, dim])
429             sigma = make_nomask([dim, dim])
430
431             # Setup some of the elements as non-zero
432             # This sets up a fake model where only lambda_0 and lambda_1 are
433             # estimated
434             lam_0[:neqs] = ma.masked
435             lam_1[:neqs, :neqs] = ma.masked
436             delta_0[:, :] = np.random.random(1)
437             delta_1[:neqs] = np.random.random((neqs, 1))
438             mu[:neqs] = np.random.random((neqs, 1))

```

```

439     phi[:neqs, :] = np.random.random((neqs, dim))
440     sigma[:, :] = np.identity(dim)
441
442     self.mod_kwargs_nolat = {
443         'yc_data': yc_data_test,
444         'var_data': var_data_test,
445         'lags': lags,
446         'neqs': neqs,
447         'mats': mats,
448         'lam_0_e': lam_0,
449         'lam_1_e': lam_1,
450         'delta_0_e': delta_0,
451         'delta_1_e': delta_1,
452         'mu_e': mu,
453         'phi_e': phi,
454         'sigma_e': sigma
455     }
456
457     self.guess_params_nolat = np.random.random((neqs**2 + neqs)).tolist()
458     self.affine_obj_nolat = Affine(**self.mod_kwargs_nolat)
459
460     ## Maximum likelihood build
461
462     # initialize masked arrays
463     self.dim_lat = dim = lags * neqs + latent
464     lam_0 = make_nomask([dim, 1])
465     lam_1 = make_nomask([dim, dim])
466     delta_0 = make_nomask([1, 1])
467     delta_1 = make_nomask([dim, 1])
468     mu = make_nomask([dim, 1])
469     phi = make_nomask([dim, dim])
470     sigma = make_nomask([dim, dim])
471
472     # Setup some of the elements as non-zero
473     # This sets up a fake model where only lambda_0 and lambda_1 are
474     # estimated
475     lam_0[:neqs] = ma.masked
476     lam_0[-latent:] = ma.masked
477     lam_1[:neqs, :neqs] = ma.masked
478     lam_1[-latent:, -latent:] = ma.masked

```

```

479     delta_0[:, :] = np.random.random(1)
480     delta_1[:neqs] = np.random.random((neqs, 1))
481     mu[:neqs] = np.random.random((neqs, 1))
482     phi[:neqs, :] = np.random.random((neqs, dim))
483     sigma[:, :] = np.identity(dim)
484
485     self.mod_kwargs = {
486         'yc_data': yc_data_test,
487         'var_data': var_data_test,
488         'lags': lags,
489         'neqs': neqs,
490         'mats': mats,
491         'lam_0_e': lam_0,
492         'lam_1_e': lam_1,
493         'delta_0_e': delta_0,
494         'delta_1_e': delta_1,
495         'mu_e': mu,
496         'phi_e': phi,
497         'sigma_e': sigma,
498         'latent': latent,
499         'no_err': [1]
500     }
501
502     self.guess_params_lat = np.random.random((neqs**2 + neqs +
503                                                (2 * latent),)).tolist()
504     self.affine_obj_lat = Affine(**self.mod_kwargs)
505
506
507     def test_solve_nls(self):
508         """
509         Tests whether or not basic estimation is performed for non-linear least
510         squares case without any latent factors. If the numerical approximation
511         method converges, this test passes. Otherwise, the test fails.
512         """
513         guess_params = self.guess_params_nolat
514         method = 'nls'
515         solved = self.affine_obj_nolat.solve(guess_params, method=method,
516                                              alg='newton')
517
518     def test_solve_ml(self):

```



```
519     """
520     Tests whether or not model estimation converges is performed for direct
521     maximum likelihood with a single latent factor. If the numerical
522     approximation method converges, this test passes. Otherwise, the test
523     fails.
524     """
525     guess_params = self.guess_params_lat
526     method = 'ml'
527     self.affine_obj_lat.solve(guess_params, method=method, alg='bfgs',
528                               xtol=0.1, ftol=0.1)
529
530     ##Need test related to Kalman filter method
531
532 if __name__ == '__main__':
533     unittest.main()
```

APPENDIX E

SAMPLE SCRIPTS FOR EXECUTING MODELS AND VIEWING RESULTS

Listing E.1: Example script executing Bernanke et al. (2005) approach

```

1 """
2 This script estimates a model with final data
3 """
4 import numpy as np
5 import numpy.ma as ma
6 import scipy.linalg as la
7 import pandas as pa
8 import datetime as dt
9 import matplotlib.pyplot as plt
10
11 from statsmodels.api import OLS
12 from statsmodels.tsa.api import VAR
13 from pandas.tseries.offsets import *
14 from affine.model.affine import Affine
15
16 import ipdb
17
18 #####
19 # Create function for unmask all elements np array #
20 #####
21 def unmask_zarray(dims):
22     array = ma.zeros(dims)
23     array[:, :] = ma.masked
24     array[:, :] = ma.nomask

```

```

25     return array
26
27 latent = [False,
28           1,
29           2,
30           3]
31
32 #####
33 # Prepare macro data for VAR #
34 #####
35 macro_data = pa.read_csv("./data/macro_data_final/macro_data_quarterly.csv",
36                          index_col = 0, parse_dates=True, sep=";")
37
38 macro_data.rename(columns={'GDP': 'output', 'Prices': 'price_output',
39                          'Resinv': 'resinv'}, inplace=True)
40
41 macro_data = macro_data[['output', 'price_output', 'resinv', 'unemp']]
42 #macro_data = macro_data[['output', 'price_output']] * 4
43 macro_vars = macro_data.columns.tolist()
44
45 neqs = len(macro_vars)
46 lags = 4
47 lat = 0
48 dim = neqs * lags + lat
49
50 #####
51 # Grab yield curve data #
52 #####
53 quarters = [4, 8, 12, 16, 20]
54 #set 36 mth as estimated with no error
55 no_err = [3]
56
57 ycdata = pa.read_csv("./data/fama-bliss/fama-bliss_formatted.csv",
58                     index_col=0, parse_dates=True)
59
60 yc_cols = ['TMYTM_1', 'TMYTM_2', 'TMYTM_3', 'TMYTM_4', 'TMYTM_5']
61 mod_yc_data = ycdata[yc_cols]
62 mod_yc_data['year'] = mod_yc_data.index.year
63 mod_yc_data['month'] = mod_yc_data.index.month
64 mod_yc_data['day'] = 1

```

```

65 mod_yc_data = mod_yc_data[mod_yc_data['month'].isin([1, 4, 7, 10])]
66 mod_yc_data['index'] = mod_yc_data.apply(
67     lambda row: dt.datetime(int(row['year']),
68                             int(row['month']),
69                             int(row['day'])), axis=1)
70 mod_yc_data = mod_yc_data.set_index('index')[yc_cols]
71
72 var_dates = pa.date_range("1/1/1982", "10/1/2012",
73                           freq="QS").to_pydatetime()
74 yc_dates = var_dates[lags:]
75
76 use_macro_data = macro_data.ix[var_dates]
77 #demean and standadize the data
78 use_macro_data = (use_macro_data - use_macro_data[lags:].mean()) / \
79                 use_macro_data[lags:].std()
80 use_yc_data = mod_yc_data.ix[yc_dates]
81
82 mu_e = unmask_zarray((dim, 1))
83 phi_e = unmask_zarray((dim, dim))
84 sigma_u = unmask_zarray((dim, dim))
85
86 #####
87 # Create mu_e, phi_e and sigma masked arrays #
88 #####
89 var_fit = VAR(use_macro_data, freq="Q").fit(maxlags=lags)
90 coefs = var_fit.params.values
91 mu_e[:neqs] = coefs[0, None].T
92 if lat:
93     phi_e[:neqs, :-lat] = coefs[1:].T
94 else:
95     phi_e[:neqs] = coefs[1:].T
96
97 phi_e[neqs:dim - lat, :(lags - 1) * neqs] = np.identity((lags - 1) * neqs)
98 sigma_u[:neqs, :neqs] = np.linalg.cholesky(var_fit.sigma_u)
99 if lat:
100     sigma_u[neqs:-lat, neqs:-lat] = np.identity((lags - 1) * neqs)
101     sigma_u[:-lat, :-lat:] = np.identity(lat)
102 else:
103     sigma_u[neqs:, neqs:] = np.identity((lags - 1) * neqs)
104

```

```

105 #####
106 # Create lambda masked arrays #
107 #####
108 lambda_0_e = unmask_zarray((dim, 1))
109 lambda_1_e = unmask_zarray((dim, dim))
110
111 #####
112 # Create delta masked arrays #
113 #####
114 final_data = pa.read_csv("./data/macro_data_final/macro_data.csv", sep=";",
115                          index_col=0, parse_dates=True, na_values="M")
116 rf_rate = final_data["fed_funds"].ix[yc_dates]
117
118 delta_ind = use_macro_data.ix[yc_dates]
119 delta_ind["constant"] = 1
120 delta_model = OLS(rf_rate, delta_ind).fit()
121 delta_0 = unmask_zarray((1, 1))
122 delta_1_e = unmask_zarray((dim, 1))
123 delta_0[0, 0] = delta_model.params[-1]
124 delta_1_e[:neqs, 0] = delta_model.params[:-1]
125
126 df_tp = ['one_yr_tp_final',
127          'two_yr_tp_final',
128          'three_yr_tp_final',
129          'four_yr_tp_final',
130          'five_yr_tp_final']
131 df_errs = ['one_yr_errs_final',
132            'two_yr_errs_final',
133            'three_yr_errs_final',
134            'four_yr_errs_final',
135            'five_yr_errs_final']
136
137 one_yr_tp_final = pa.DataFrame(index=yc_dates)
138 two_yr_tp_final = pa.DataFrame(index=yc_dates)
139 three_yr_tp_final = pa.DataFrame(index=yc_dates)
140 four_yr_tp_final = pa.DataFrame(index=yc_dates)
141 five_yr_tp_final = pa.DataFrame(index=yc_dates)
142
143 one_yr_errs_final = pa.DataFrame(index=yc_dates)
144 two_yr_errs_final = pa.DataFrame(index=yc_dates)

```

```

145 three_yr_errs_final = pa.DataFrame(index=yc_dates)
146 four_yr_errs_final = pa.DataFrame(index=yc_dates)
147 five_yr_errs_final = pa.DataFrame(index=yc_dates)
148
149 xtol = 0.00001
150 ftol = 0.00001
151
152 # Setup model
153 np.random.seed(101)
154
155 print "xtol " + str(xtol)
156 print "ftol " + str(ftol)
157 print "Begin " + str(yc_dates[0])
158 print "End " + str(yc_dates[-1])
159 print "latent = " + str(lat)
160
161 rerun = False
162
163 #####
164 # Final iteration #
165 #####
166
167 lambda_0_e[:neqs, 0] = ma.masked
168 lambda_1_e[:neqs, :neqs] = ma.masked
169 model = Affine(yc_data=use_yc_data, var_data=use_macro_data,
170               lam_0_e=lambda_0_e, lam_1_e=lambda_1_e, delta_0_e=delta_0,
171               delta_1_e=delta_1_e, mu_e=mu_e, phi_e=phi_e,
172               sigma_e=sigma_u, mats=quarters)
173
174
175 if rerun:
176     guess_length = model.guess_length
177     guess_params = [0.0000001] * guess_length
178     out_bsr = model.solve(guess_params=guess_params, method='nls', ftol=ftol,
179                          xtol=xtol, maxfev=10000000, maxiter=1000000,
180                          full_output=False, alg='nm')
181     lam_0, lam_1, delta_0, delta_1, mu, phi, sigma, a_s, b_s, \
182     solve_params = out_bsr
183     #maybe make function later to do this
184     pa.DataFrame(lam_0).to_csv("./results/final/lam_0.csv", index=False)

```

```

185     pa.DataFrame(lam_1).to_csv("./results/final/lam_1.csv", index=False)
186     pa.DataFrame(delta_0).to_csv("./results/final/delta_0.csv", index=False)
187     pa.DataFrame(delta_1).to_csv("./results/final/delta_1.csv", index=False)
188     pa.DataFrame(mu).to_csv("./results/final/mu.csv", index=False)
189     pa.DataFrame(phi).to_csv("./results/final/phi.csv", index=False)
190     pa.DataFrame(sigma).to_csv("./results/final/sigma.csv", index=False)
191 else:
192     #read from csv
193     lam_0 = pa.read_csv("./results/final/lam_0.csv").values
194     lam_1 = pa.read_csv("./results/final/lam_1.csv").values
195     delta_0 = pa.read_csv("./results/final/delta_0.csv").values
196     delta_1 = pa.read_csv("./results/final/delta_1.csv").values
197     mu = pa.read_csv("./results/final/mu.csv").values
198     phi = pa.read_csv("./results/final/phi.csv").values
199     sigma = pa.read_csv("./results/final/sigma.csv").values
200
201
202 #####
203 # Collect results #
204 #####
205
206 a_rsk, b_rsk = model.gen_pred_coef(lam_0=lam_0, lam_1=lam_1, delta_0=delta_0,
207                                   delta_1=delta_1, mu=mu, phi=phi,
208                                   sigma=sigma)
209
210 #generate no risk results
211 lam_0_nr = np.zeros([dim, 1])
212 lam_1_nr = np.zeros([dim, dim])
213 sigma_zeros = np.zeros_like(sigma)
214 a_nrsk, b_nrsk = model.gen_pred_coef(lam_0=lam_0_nr, lam_1=lam_1_nr,
215                                       delta_0=delta_0, delta_1=delta_1, mu=mu,
216                                       phi=phi, sigma=sigma_zeros)
217 X_t = model.var_data_vert
218 per = model.yc_data.index
219 act_pred = pa.DataFrame(index=per)
220 for i in quarters:
221     act_pred[str(i) + '_act'] = model.yc_data['TMYTM_' + str(i / 4)]
222     act_pred[str(i) + '_pred'] = a_rsk[i-1] + np.dot(b_rsk[i-1], X_t.T)
223     act_pred[str(i) + '_nrsk'] = a_nrsk[i-1] + np.dot(b_nrsk[i-1].T, X_t.T)
224     act_pred[str(i) + '_err'] = (act_pred[str(i) + '_act'] - \

```

```

225         act_pred[str(i) + '_pred'])
226     act_pred[str(i) + '_sqer'] = (act_pred[str(i) + '_act'] - \
227         act_pred[str(i) + '_pred'])*2
228     act_pred[str(i) + '_tp'] = act_pred[str(i) + '_pred'] - \
229         act_pred[str(i) + '_nrsk']
230 one_yr = act_pred.reindex(columns = filter(lambda x: '4' in x, act_pred))
231 two_yr = act_pred.reindex(columns = filter(lambda x: '8' in x, act_pred))
232 three_yr = act_pred.reindex(columns = filter(lambda x: '12' in x, act_pred))
233 four_yr = act_pred.reindex(columns = filter(lambda x: '16' in x, act_pred))
234 five_yr = act_pred.reindex(columns = filter(lambda x: '20' in x, act_pred))
235
236 #generate st dev of residuals
237 yields = ['one_yr', 'two_yr', 'three_yr', 'four_yr', 'five_yr']
238 for yld in yields:
239     print yld + " & " + str(np.sqrt(np.mean(eval(yld).filter(
240         regex= '.*sqer$').values)) * 100)
241     tp = yld + '_tp_final'
242     err = yld + '_errs_final'
243     eval(tp)[yld] = eval(yld).filter(regex='.*tp$')
244     eval(err)[yld] = eval(yld).filter(regex='.*err$')
245
246     # if xix == len(xtols) - 1 and fix == len(ftols) - 1:
247     #     for df in df_tp:
248     #         eval(df).to_csv('./results/final/' + df + '.csv',
249     #             float_format='%0.8f')
250     #     for df in df_errs:
251     #         eval(df).to_csv('./results/final/' + df + '.csv',
252     #             float_format='%0.8f')
253
254
255 #out of sample forecasting
256 solve_forward = 10
257 yc_data = model.yc_data.copy()
258 yc_data_cols = yc_data.columns.tolist()
259 for per in range(10):
260     row = mu + np.dot(phi, X_t[-1:].T)
261     date = X_t[-1:].index.to_pydatetime()[0] + MonthBegin() + \
262         MonthBegin() + MonthBegin()
263     X_t.loc[date, :] = row.T
264     for qix, quart in enumerate(quarters):

```



```
265         col = yc_data_cols[qix]
266         yc_data.loc[date, col] = a_rsk[quart-1] + np.dot(b_rsk[quart-1], row)
```

[illegible]

```

39
40 macro_data = macro_data[['output', 'price_output']]
41 macro_vars = macro_data.columns.tolist()
42
43 neqs = len(macro_vars)
44 lags = 4
45 lat = 1
46 dim = neqs * lags + lat
47
48 #####
49 # Grab yield curve data #
50 #####
51 quarters = [4, 8, 12, 16, 20]
52 #set 36 mth as estimated with no error
53
54 ycdata = pa.read_csv("./data/fama-bliss/fama-bliss_formatted.csv",
55                      index_col=0, parse_dates=True)
56
57 yc_cols = ['TMYTM_1', 'TMYTM_2', 'TMYTM_3', 'TMYTM_4', 'TMYTM_5']
58 mod_yc_data = ycdata[yc_cols]
59 mod_yc_data['year'] = mod_yc_data.index.year
60 mod_yc_data['month'] = mod_yc_data.index.month
61 mod_yc_data['day'] = 1
62 mod_yc_data = mod_yc_data[mod_yc_data['month'].isin([1, 4, 7, 10])]
63 mod_yc_data['index'] = mod_yc_data.apply(
64     lambda row: dt.datetime(int(row['year']),
65                             int(row['month']),
66                             int(row['day'])), axis=1)
67 mod_yc_data = mod_yc_data.set_index('index')[yc_cols]
68
69 var_dates = pa.date_range("1/1/1982", "10/1/2012",
70                           freq="QS").to_pydatetime()
71 yc_dates = var_dates[lags:]
72
73 use_macro_data = macro_data.ix[var_dates]
74 #demean and standadize the data
75 use_macro_data = (use_macro_data - use_macro_data[lags:].mean()) / \
76                 use_macro_data[lags:].std()
77 use_yc_data = mod_yc_data.ix[yc_dates]
78

```

```

79 mu_e = unmask_zarray((dim, 1))
80 phi_e = unmask_zarray((dim, dim))
81 sigma_u = unmask_zarray((dim, dim))
82
83 #####
84 # Create mu_e, phi_e and sigma masked arrays #
85 #####
86 var_fit = VAR(use_macro_data, freq="Q").fit(maxlags=lags)
87 coefs = var_fit.params.values
88 mu_e[:neqs] = coefs[0, None].T
89 if lat:
90     phi_e[:neqs, :-lat] = coefs[1:].T
91 else:
92     phi_e[:neqs] = coefs[1:].T
93
94 phi_e[neqs:dim - lat, :(lags - 1) * neqs] = np.identity((lags - 1) * neqs)
95 sigma_u[:neqs, :neqs] = np.linalg.cholesky(var_fit.sigma_u)
96 if lat:
97     sigma_u[neqs:-lat, neqs:-lat] = np.identity((lags - 1) * neqs)
98     sigma_u[-lat:-lat:] = np.identity(lat)
99 else:
100     sigma_u[neqs:, neqs:] = np.identity((lags - 1) * neqs)
101
102 #####
103 # Create lambda masked arrays #
104 #####
105 lambda_0_e = unmask_zarray((dim, 1))
106 lambda_1_e = unmask_zarray((dim, dim))
107
108 #####
109 # Create delta masked arrays #
110 #####
111 final_data = pa.read_csv("./data/macro_data_final/macro_data.csv", sep=";",
112                          index_col=0, parse_dates=True, na_values="M")
113 rf_rate = final_data["fed_funds"].ix[yc_dates]
114
115 delta_ind = use_macro_data.ix[yc_dates]
116 delta_ind["constant"] = 1
117 delta_model = OLS(rf_rate, delta_ind).fit()
118 delta_0 = unmask_zarray((1, 1))

```

```

119 delta_1_e = unmask_zarray((dim, 1))
120 delta_0[0, 0] = delta_model.params[-1]
121 delta_1_e[:neqs, 0] = delta_model.params[:-1]
122
123 df_tp = ['one_yr_tp_final',
124          'two_yr_tp_final',
125          'three_yr_tp_final',
126          'four_yr_tp_final',
127          'five_yr_tp_final']
128 df_errs = ['one_yr_errs_final',
129            'two_yr_errs_final',
130            'three_yr_errs_final',
131            'four_yr_errs_final',
132            'five_yr_errs_final']
133
134 one_yr_tp_final = pa.DataFrame(index=yc_dates)
135 two_yr_tp_final = pa.DataFrame(index=yc_dates)
136 three_yr_tp_final = pa.DataFrame(index=yc_dates)
137 four_yr_tp_final = pa.DataFrame(index=yc_dates)
138 five_yr_tp_final = pa.DataFrame(index=yc_dates)
139
140 one_yr_errs_final = pa.DataFrame(index=yc_dates)
141 two_yr_errs_final = pa.DataFrame(index=yc_dates)
142 three_yr_errs_final = pa.DataFrame(index=yc_dates)
143 four_yr_errs_final = pa.DataFrame(index=yc_dates)
144 five_yr_errs_final = pa.DataFrame(index=yc_dates)
145
146 xtol = 1e-5
147
148 ftol = 1e-5
149
150 max = 50
151 min = 0.0000001
152
153 no_err = [1]
154 # Setup model
155 np.random.seed(101)
156
157 print "xtol " + str(xtol)
158 print "ftol " + str(ftol)

```

```

159 print "Begin " + str(yc_dates[0])
160 print "End " + str(yc_dates[-1])
161 print "latent = " + str(lat)
162 print "noerr = " + str(no_err)
163
164 #only need iterative process if latent variables are estimated
165 #####
166 # First iteration, estimate phi and others, hold lambdas const #
167 #####
168 phi_e[-lat,-lat] = ma.masked
169 delta_1_e[-lat, 0] = ma.masked
170
171 print "First estimation"
172 model = Affine(yc_data=use_yc_data, var_data=use_macro_data,
173               lam_0_e=lambda_0_e, lam_1_e=lambda_1_e,
174               delta_0_e=delta_0, delta_1_e=delta_1_e, mu_e=mu_e,
175               phi_e=phi_e, sigma_e=sigma_u, mats=quarters,
176               latent=1)
177
178 guess_length = model.guess_length
179 guess_params = np.linspace(0.5, 1.5, guess_length)
180
181 #try random stuff for all params
182 #guess_params = (np.random.random((guess_length,)) * (1.0 / 10000)).tolist()
183
184 out_bsr = model.solve(guess_params=guess_params, method='ml',
185                       ftol=ftol, xtol=xtol, maxfev=10000000,
186                       maxiter=1000000, full_output=False, alg='nm',
187                       no_err=no_err)
188 a, b, c, d, e, f, g, a_s, b_s, solve_params, var_data_wunob = out_bsr
189
190 hessian = model.hessian(solve_params)
191 std_err = np.sqrt(-np.diag(la.inv(hessian)))
192
193 print std_err
194
195 tval = solve_params / std_err
196 rep_vals = []
197 for tix, val in enumerate(tval):
198     if abs(val) > 1.960:

```

```

199         rep_vals.append(solve_params[tix])
200     else:
201         rep_vals.append(0)
202
203     #fill in relevant values of phi_e and delta_1
204     a, b, c, delta_1, d, phi, e = model.params_to_array(rep_vals,
205                                                         return_mask=True)
206     a, b, c, delta_1_g, d, phi_g, e, dphi_guesses = \
207         model.params_to_array_zeromask(rep_vals)
208
209     #####
210     # Second iteration, estimate lambda_1, hold lambda_0 at zero #
211     #####
212     print "Second estimation"
213     lambda_1_e[-lat, -lat] = ma.masked
214     lambda_1_e[:, neqs, :neqs] = ma.masked
215
216     model = Affine(yc_data=use_yc_data, var_data=use_macro_data,
217                   lam_0_e=lambda_0_e, lam_1_e=lambda_1_e,
218                   delta_0_e=delta_0, delta_1_e=delta_1, mu_e=mu_e,
219                   phi_e=phi, sigma_e=sigma_u, mats=quarters,
220                   latent=True)
221
222     guess_length = model.guess_length
223     guess_params = np.linspace(0.5, 1.5, guess_length)
224
225     out_bsr = model.solve(guess_params=guess_params, method='ml',
226                          ftol=ftol, xtol=xtol, maxfev=10000000,
227                          maxiter=1000000, full_output=False, alg='nm',
228                          no_err=no_err)
229     a, b, c, d, e, f, g, a_s, b_s, solve_params, var_data_wunob = out_bsr
230
231     hessian = model.hessian(solve_params)
232     std_err = np.sqrt(-np.diag(la.inv(hessian)))
233
234     print std_err
235
236     tval = solve_params / std_err
237     rep_vals = []
238     for tix, val in enumerate(tval):

```

```

239     if abs(val) > 1.960:
240         rep_vals.append(solve_params[tix])
241     else:
242         rep_vals.append(0)
243
244     #fill in relevant values of lambda_1
245     a, lambda_1, b, c, d, e, f = model.params_to_array(rep_vals,
246                                                         return_mask=True)
247     a, lambda_1_g, b, c, d, e, f, ll_guesses = \
248         model.params_to_array_zeromask(rep_vals)
249
250     #####
251     # Third iteration, estimate lambda_0, holding lambda_1 at prior values #
252     #####
253     print "Third estimation"
254     lambda_0_e[:neqs, 0] = ma.masked
255     lambda_0_e[-lat:, 0] = ma.masked
256
257     model = Affine(yc_data=use_yc_data, var_data=use_macro_data,
258                   lam_0_e=lambda_0_e, lam_1_e=lambda_1, delta_0_e=delta_0,
259                   delta_1_e=delta_1, mu_e=mu_e, phi_e=phi, sigma_e=sigma_u,
260                   mats=quarters, latent=True)
261
262     guess_length = model.guess_length
263     guess_params = np.linspace(0.5, 1.5, guess_length)
264
265     out_bsr = model.solve(guess_params=guess_params, method='ml',
266                           ftol=ftol, xtol=xtol, maxfev=10000000,
267                           maxiter=1000000, full_output=False, alg='nm',
268                           no_err=no_err)
269     a, b, c, d, e, f, g, a_s, b_s, solve_params, var_data_wunob = out_bsr
270
271     hessian = model.hessian(solve_params)
272     std_err = np.sqrt(-np.diag(la.inv(hessian)))
273
274     print std_err
275
276     tval = solve_params / std_err
277     rep_vals = []
278     for tix, val in enumerate(tval):

```



```

279     if abs(val) > 1.960:
280         rep_vals.append(solve_params[tix])
281     else:
282         rep_vals.append(0)
283
284 #fill in relevant values of lambda_1
285 lambda_0, a, b, c, d, e, f = model.params_to_array(rep_vals,
286                                                     return_mask=True)
287 lambda_0_g, a, b, c, d, e, f, l0_guesses = \
288     model.params_to_array_zeromask(rep_vals)
289
290 #####
291 # Final iteration #
292 #####
293 print "Fourth estimation"
294
295 model = Affine(yc_data=use_yc_data, var_data=use_macro_data,
296               lam_0_e=lambda_0_g, lam_1_e=lambda_1_g,
297               delta_0_e=delta_0, delta_1_e=delta_1_g, mu_e=mu_e,
298               phi_e=phi_g, sigma_e=sigma_u, mats=quarters,
299               latent=True)
300
301 guess_params = l0_guesses + l1_guesses + dlphi_guesses
302 out_bsr = model.solve(guess_params=guess_params, method='ml',
303                       ftol=ftol, xtol=xtol, maxfev=100000,
304                       maxiter=100000, full_output=False, alg='nm',
305                       no_err=no_err)
306 lam_0, lam_1, delta_0, delta_1, mu, phi, sigma, a_s, b_s, solve_params, \
307     var_data_wunob = out_bsr
308
309 #####
310 # Collect results #
311 #####
312
313 a_rsk, b_rsk = model.gen_pred_coef(lam_0=lam_0, lam_1=lam_1, delta_0=delta_0,
314                                   delta_1=delta_1, mu=mu, phi=phi,
315                                   sigma=sigma)
316
317 #generate no risk results
318 lam_0_nr = np.zeros([dim, 1])

```

```

319 lam_1_nr = np.zeros([dim, dim])
320 sigma_zeros = np.zeros_like(sigma)
321 a_nrsk, b_nrsk = model.gen_pred_coef(lam_0=lam_0_nr, lam_1=lam_1_nr,
322                                     delta_0=delta_0, delta_1=delta_1, mu=mu,
323                                     phi=phi, sigma=sigma_zeros)
324 if lat:
325     X_t = var_data_wunob
326 else:
327     X_t = model.var_data_vert
328 per = model.yc_data.index
329 act_pred = pa.DataFrame(index=per)
330 for i in quarters:
331     act_pred[str(i) + '_act'] = model.yc_data['TMYTM_' + str(i / 4)]
332     act_pred[str(i) + '_pred'] = a_rsk[i-1] + np.dot(b_rsk[i-1], X_t.T)
333     act_pred[str(i) + '_nrsk'] = a_nrsk[i-1] + np.dot(b_nrsk[i-1].T, X_t.T)
334     act_pred[str(i) + '_err'] = (act_pred[str(i) + '_act'] - \
335                                 act_pred[str(i) + '_pred'])
336     act_pred[str(i) + '_sqer'] = (act_pred[str(i) + '_act'] - \
337                                 act_pred[str(i) + '_pred'])*2
338     act_pred[str(i) + '_tp'] = act_pred[str(i) + '_pred'] - \
339                                act_pred[str(i) + '_nrsk']
340 one_yr = act_pred.reindex(columns = filter(lambda x: '4' in x, act_pred))
341 two_yr = act_pred.reindex(columns = filter(lambda x: '8' in x, act_pred))
342 three_yr = act_pred.reindex(columns = filter(lambda x: '12' in x, act_pred))
343 four_yr = act_pred.reindex(columns = filter(lambda x: '16' in x, act_pred))
344 five_yr = act_pred.reindex(columns = filter(lambda x: '20' in x, act_pred))
345
346 #generate st dev of residuals
347 yields = ['one_yr', 'two_yr', 'three_yr', 'four_yr', 'five_yr']
348 for yld in yields:
349     print yld + " & " + str(np.sqrt(np.mean(eval(yld).filter(
350                                     regex= '.*sqer$').values)) * 100)
351     tp = yld + '_tp_final'
352     err = yld + '_errs_final'
353     eval(tp)[yld] = eval(yld).filter(regex='.*tp$')
354     eval(err)[yld] = eval(yld).filter(regex='.*err$')
355
356 #plot five year pricing errors
357 five_yr_errs_final['five_yr'].plot(subplots=True)
358 #plot one year pricing errors

```

```
359 one_yr_errs_final['five_yr'].plot(subplots=True)
360 #plot one year time-varying term premium
361 one_yr_tp_final['five_yr'].plot(subplots=True)
362 #plot five year time-varying term premium
363 five_yr_tp_final['five_yr'].plot(subplots=True)
```

REFERENCES

- Adrian, Tobias, Arturo Estrella, and Hyun Song Shin (2010). “Monetary cycles, financial cycles, and the business cycle.” Technical report, Staff Report, Federal Reserve Bank of New York.
- Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1999). *LAPACK Users’ Guide* (Third ed.). Philadelphia, PA: Society for Industrial and Applied Mathematics.
- Andrews, Donald WK and Hong-Yuan Chen (1994). “Approximately median-unbiased estimation of autoregressive models.” *Journal of Business & Economic Statistics* 12(2), 187–204.
- Ang, Andrew and Monika Piazzesi (2003). “A no-arbitrage vector autoregression of term structure dynamics with macroeconomic and latent variables.” *Journal of Monetary Economics* 50(4), 745–787.
- Bai, Jushan, Robin L Lumsdaine, and James H Stock (1998). “Testing for and dating common breaks in multivariate time series.” *The Review of Economic Studies* 65(3), 395–432.
- Baker, Barton (2014a). *An Extension and Replication of Bernanke, Reinhart, and Sack(2005)*. Ph.D. thesis, American University.
- Baker, Barton (2014b). *Real-time data and informing affine models of the term structure*. Ph.D. thesis, American University.
- Baker, Scott, Nicholas Bloom, and Steven Davis (2013). “Measuring economic policy uncertainty.” *Chicago Booth Research Paper* (13-02).
- Banerjee, Anindya, Robin L Lumsdaine, and James H Stock (1992). “Recursive and sequential tests of the unit-root and trend-break hypotheses: theory and international evidence.” *Journal of Business & Economic Statistics* 10(3), 271–287.
- Batchelor, Roy and Pami Dua (1992). “Conservatism and consensus-seeking among economic forecasters.” *Journal of Forecasting* 11(2), 169–181.
- Behnel, S., R. Bradshaw, L. Dalcin, M. Florisson, V. Makarov, and D. Seljebotn (2004). “Cython: C-extensions for python.” <http://cython.org/>.
- Bernanke, Ben S, Vincent R Reinhart, and Brian P Sack (2005). “Monetary policy alternatives at the zero bound: An empirical assessment.” *Brookings Papers on Economic Activity* 2004(2), 1–100.
- Bloom, Nicholas (2009). “The impact of uncertainty shocks.” *Econometrica* 77(3), 623–685.
- Bloomberg (2012). “Commodity quote: Eurodollar rate.” January 1980 - December 2012, via Bloomberg, LP, accessed December 2012.
- BLS (2012, 12). “Bureau of labor statistics.” <http://www.bls.gov/>.
- Bomberger, William A (1996). “Disagreement as a measure of uncertainty.” *Journal of Money, Credit and Banking* 28(3), 381–392.

- CBOE (2009). "The CBOE volatility index - VIX." Technical report, Chicago Board Options Exchange, Incorporated.
- Chen, Ren-RAW and Louis Scott (1993). "Maximum likelihood estimation for a multifactor equilibrium model of the term structure of interest rates." *Journal of Fixed Income* 3, 14–31.
- Chun, Albert Lee (2011). "Expectations, bond yields, and monetary policy." *Review of Financial Studies* 24(1), 208–247.
- Cochrane, J.H. and M. Piazzesi (2008). "Decomposing the yield curve." *Graduate School of Business, University of Chicago, Working Paper*.
- Cox, John C, Jr Ingersoll, Jonathan E, and Stephen A Ross (1985, March). "A theory of the term structure of interest rates." *Econometrica* 53(2), 385–407.
- Croushore, Dean and Tom Stark (2001). "A real-time data set for macroeconomists." *Journal of Econometrics* 105(1), 111–130.
- CRSP, Center for Research in Security Prices. Graduate School of Business, The University of Chicago (2013). "Fama-Bliss discount bonds - monthly only." Used with permission. All rights reserved. <http://www.vrplumber.com/programming/runsnakerun/>.
- Dai, Qiang and Kenneth J. Singleton (2000, October). "Specification analysis of affine term structure models." *Journal of Finance* 55(5), 1943–1978.
- Dai, Qiang and Kenneth J Singleton (2002). "Expectation puzzles, time-varying risk premia, and affine models of the term structure." *Journal of Financial Economics* 63(3), 415–441.
- Diebold, Francis X., Glenn D. Rudebusch, and S. Borağan Aruoba (2006). "The macroeconomy and the yield curve: a dynamic latent factor approach." *Journal of Econometrics* 131(1-2), 309–338.
- Diebold, Francis X and Kamil Yilmaz (2008). "Macroeconomic volatility and stock market volatility, worldwide." Technical report, National Bureau of Economic Research.
- Doh, Taeyoung (2011). "Yield curve in an estimated nonlinear macro model." *Journal of Economic Dynamics and Control* 35(8), 1229–1244.
- Dovern, Jonas, Ulrich Fritsche, and Jiri Slacalek (2012). "Disagreement among forecasters in G7 countries." *Review of Economics and Statistics* 94(4), 1081–1096.
- Duffee, Gregory R and Richard H Stanton (2012). "Estimation of dynamic term structure models." *The Quarterly Journal of Finance* 2(02), 1–51.
- Duffie, Darrell and Rui Kan (1996). "A yield-factor model of interest rates." *Mathematical Finance* 6(4), 379–406.
- Fama, Eugene F and Robert R Bliss (1987). "The information in long-maturity forward rates." *The American Economic Review*, 680–692.
- Federal Reserve Bank of Philadelphia (2013a, 12). "Greenbook data sets." <http://www.phil.frb.org/research-and-data/real-time-center/greenbook-data/>.
- Federal Reserve Bank of Philadelphia (2013b, 12). "Real-time data set for macroeconomists." <http://www.phil.frb.org/research-and-data/real-time-center/real-time-data/>.
- Federal Reserve Bank of Philadelphia (2013, 12). "Survey of professional forecasters." <http://www.phil.frb.org/research-and-data/real-time-center/survey-of-professional-forecasters/>.

- Federal Reserve Bank of St. Louis (2013, 12). "Federal reserve economic data." <http://research.stlouisfed.org/fred2/>.
- Fleming, Jeff, Chris Kirby, and Barbara Ostdiek (1998). "Information and volatility linkages in the stock, bond, and money markets." *Journal of Financial Economics* 49(1), 111–137.
- Fletcher, Mike (2001–2013). "RunSnakeRun: Gui viewer for python profiling runs." <http://www.vrplumber.com/programming/runsnakerun/>.
- Giordani, Paolo and Paul Söderlind (2003). "Inflation forecast uncertainty." *European Economic Review* 47(6), 1037–1059.
- Grishchenko, Olesya and Jing-zhi Jay Huang (2012). "The inflation risk premium: Evidence from the TIPS market." *SSRN 1108401*, http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1108401.
- Hamilton, James Douglas (1994). *Time Series Analysis*. Princeton, New Jersey: Princeton University Press.
- Jones, Eric, Travis Oliphant, Pearu Peterson, et al. (2001–2014). "SciPy: Open source scientific tools for Python." <http://www.scipy.org/>.
- Joslin, Scott, Kenneth J Singleton, and Haoxiang Zhu (2011). "A new perspective on gaussian dynamic term structure models." *Review of Financial Studies* 24(3), 926–970.
- Keynes, John M (1936). *The General Theory of Employment, Interest and Money*. London: The Macmillan Press.
- Kim, Don H. and Athanasios Orphanides (2005, November). "Term structure estimation with survey data on interest rate forecasts." CEPR Discussion Papers 5341, C.E.P.R. Discussion Papers.
- Kim, Don H. and Jonathan H. Wright (2005). "An arbitrage-free three-factor term structure model and the recent behavior of long-term yields and distant-horizon forward rates." Finance and Economics Discussion Series 2005-33, Board of Governors of the Federal Reserve System (U.S.).
- Krishnamurthy, Arvind and Annette Vissing-Jorgensen (2011). "The effects of quantitative easing on interest rates: channels and implications for policy." Technical report, National Bureau of Economic Research.
- Litterman, Robert and José Scheinkman (1991). "Common factors affecting bond returns." *The Journal of Fixed Income* 1(1), 54–61.
- Mankiw, N Gregory, Ricardo Reis, and Justin Wolfers (2004). "Disagreement about inflation expectations." In *NBER Macroeconomics Annual 2003, Volume 18*, pp. 209–270. The MIT Press.
- McCallum, Bennett T (1976). "Rational expectations and the estimation of econometric models: An alternative procedure." *International Economic Review* 17(2), 484–490.
- McKinney, Wes (2005–2014). "Python data analysis library." <http://pandas.pydata.org/>.
- Minsky, Hyman P (1986). *Stabilizing an unstable economy*. New Haven: Yale University Press.
- NBER (2013, 12). "National bureau of economic research." <http://www.nber.org/>.
- Oliphant, Travis et al. (2005–2014). "Numpy: Open source scientific tools for Python." <http://www.numpy.org/>.
- Oliphant, Travis et al. (2014). "How to extend numpy." <http://docs.scipy.org/doc/numpy/user/c-info.how-to-extend.html>.

- Orphanides, Athanasios (2001). "Monetary policy rules based on real-time data." *American Economic Review* 91(4), 964–985.
- Orphanides, Athanasios and Min Wei (2012). "Evolving macroeconomic perceptions and the term structure of interest rates." *Journal of Economic Dynamics and Control* 36(2), 239–254.
- Pearson, Neil D and Tong-Sheng Sun (1994). "Exploiting the conditional density in estimating the term structure: An application to the Cox, Ingersoll, and Ross model." *The Journal of Finance* 49(4), 1279–1304.
- Pérez, Fernando and Brian E. Granger (2007, May). "IPython: a system for interactive scientific computing." *Computing in Science and Engineering* 9(3), 21–29.
- Perktold, J., S. Seabold, and W. McKinney (2006–2014). "statsmodels: Statistics in Python." <http://statsmodels.sourceforge.net/>.
- Piazzesi, Monika and Martin Schneider (2007, June). "Equilibrium yield curves." In *NBER Macroeconomics Annual 2006, Volume 21*, NBER Chapters, pp. 389–472. National Bureau of Economic Research, Inc.
- R Core Team (2012). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0, <http://www.R-project.org/>.
- Rich, Robert, Joseph Song, and Joseph Tracy (2012). "The measurement and behavior of uncertainty: Evidence from the ECB survey of professional forecasters." *SSRN 2192510*, http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2192510.
- Rich, Robert and Joseph Tracy (2010). "The relationships among expected inflation, disagreement, and uncertainty: evidence from matched point and density forecasts." *The Review of Economics and Statistics* 92(1), 200–207.
- Rudebusch, Glenn D, Brian P Sack, and Eric T Swanson (2007). "Macroeconomic implications of changes in the term premium." *Review-Federal Reserve Bank of Saint Louis* 89(4), 241.
- Rudebusch, Glenn D and Tao Wu (2008). "A macro-finance model of the term structure, monetary policy and the economy." *The Economic Journal* 118(530), 906–926.
- StataCorp (2013). *Stata Statistical Software: Release 13*. College Station, Texas. <http://www.stata.com/>.
- Stock, James H and Mark W Watson (2003). "Has the business cycle changed and why?" In *NBER Macroeconomics Annual 2002, Volume 17*, pp. 159–230. MIT press.
- Taylor, John B (1993). "Discretion versus policy rules in practice." In *Carnegie-Rochester conference series on public policy*, Volume 39, pp. 195–214. Elsevier.
- The MathWorks Inc. (2013). *MATLAB, Version 8.1 (R2013a)*. Natick, Massachusetts.
- Vasicek, Oldrich (1977, November). "An equilibrium characterization of the term structure." *Journal of Financial Economics* 5(2), 177–188.
- Waller, Peter (2013). "pyprof2calltree." <https://pypi.python.org/pypi/pyprof2calltree>.
- Weidendorfer, Josef (2002,2003). "Kcachegrind; call graph viewer." <http://kcachegrind.sourceforge.net/html/Home.html>.
- Whaley, R. Clint and Antoine Petitet (2005, February). "Minimizing development and maintenance costs in supporting persistently optimized BLAS." *Software: Practice and Experience* 35(2), 101–121. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>.

- Wickens, Michael R (1982). "The efficient estimation of econometric models with rational expectations." *The Review of Economic Studies* 49(1), 55–67.
- Wright, Jonathan H (2011). "Term premia and inflation uncertainty: Empirical evidence from an international panel dataset." *The American Economic Review* 101(4), 1514–1534.
- Zarnowitz, Victor and Louis A Lambros (1987). "Consensus and uncertainty in economic prediction." *National Bureau of Economic Research*, <http://www.nber.org/papers/w1171>.