Predictive Tracking for Augmented Reality

TR95-007
February 1995

Ronald T. Azuma

Department of Computer Science
CB #3175, Sitterson Hall
UNC-Chapel Hill
Chapel Hill, NC 27599-3175

Predictive Tracking for Augmented Reality

by

Ronald Tadao Azuma

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science

Chapel Hill

1995

Approved by:

_____ Adviser
T. Gary Bishop

_____ Reader
Vernon Chi

_____ Reader
Frederick P. Brooks, Jr.

ABSTRACT

Ronald Tadao Azuma. Predictive Tracking for Augmented Reality

(Under the direction of T. Gary Bishop.)

In Augmented Reality systems, see-through Head-Mounted Displays (HMDs) superimpose virtual three-dimensional objects on the real world. This technology has the potential to enhance a user's perception of and interaction with the real world. However, many Augmented Reality applications will not be accepted unless virtual objects are accurately registered with their real counterparts. Good registration is difficult, because of the high resolution of the human visual system and its sensitivity to small differences. Registration errors fall into two categories: *static* errors, which occur even when the user remains still, and *dynamic* errors caused by system delays when the user moves. Dynamic errors are usually the largest errors. This dissertation demonstrates that predicting future head locations is an effective approach for significantly reducing dynamic errors.

This demonstration is performed in real time with an operational Augmented Reality system. First, evaluating the effect of prediction requires robust static registration. Therefore, this system uses a custom optoelectronic head-tracking system and three calibration procedures developed to measure the viewing parameters. Second, the system predicts future head positions and orientations with the aid of inertial sensors. Effective use of these sensors requires accurate estimation of the varying

prediction intervals, optimization techniques for determining parameters, and a system built to support real-time processes.

On average, prediction with inertial sensors is 2 to 3 times more accurate than prediction without inertial sensors and 5 to 10 times more accurate than not doing any prediction at all. Prediction is most effective at short prediction intervals, empirically determined to be about 80 milliseconds or less. An analysis of the predictor in the frequency domain shows the predictor magnifies the signal by roughly the square of the angular frequency and the prediction interval. For specified head-motion sequences and prediction intervals, this analytical framework can also estimate the maximum possible time-domain error and the maximum tolerable system delay given a specified maximum time-domain error.

Future steps that may further improve registration are discussed.

ACKNOWLEDGEMENTS

I thank my advisor, Gary Bishop, and my committee members, Frank Biocca, Frederick Brooks, Vern Chi, Henry Fuchs, and Jonathan Marshall, for their advice and guidance in this work.

I would also like to thank the following people:

- Mark Ward, for doing most of the mechanical and electronic design of the optoelectronic tracking system that was critical to this project.
- Brad Bennett and Stefan Gottschalk for writing much of the software for the optoelectronic tracking system.
- Brad Bennett for his help with the low-level software, installation and debugging of the single-board computers and Pixel-Planes 5.
- John Thomas, John Hughes, Kurtis Keller, and Jack Kite for making mechanical parts for this project and expediting the ordering of equipment.
- Vern Chi and Steven Brumback for designing analog circuitry used in this project.
- David Harrison, Brennan Stephens, Elliot Poger, and Peggy Wetzel for their help with video recording and editing.
- Jack Goldfeather and John F. "Spike" Hughes for explaining some of the mathematics to me.
- Russell Taylor and Mark Finch for letting me use equipment from the Scanning Tunneling Microscope project to test our inertial sensors.
- Marc Olano and Jonathan Cohen for creating low-latency rendering code on Pixel-Planes 5.
- Carl Mueller, Marc Olano, and David Ellsworth for their advice on programming Pixel-Planes 5.
- Mike Bajura, Andrei State, Rich Holloway, Jannick Rolland, and Ulrich Neumann for discussions about see-through registration strategies.
- Devesh Bhatnagar, Suresh Balu, and the other Tracker group team members for their general support on the tracking equipment and software.

• Fay Ward and Kathy Tesh for always being there when I needed to mail something overnight, fill out a form that was due that day, and for squeezing me onto my professors' schedules when necessary.

Financial help came from the following sources:

• ONR Contract N00014-86-K-0680
• the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract Number 8920219
• ARPA Contract DABT63-93-C-C048 "Enabling Technologies and Application Demonstrations for Synthetic Environments"
• a Pogue Fellowship

Last, but certainly not least, I thank my parents for their support and understanding.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A/D	Analog to Digital conversion
ACM	Association for Computing Machinery
AR	Augmented Reality
CPU	Central Processing Unit
CRT	Cathode Ray Tube
DEC	Digital Equipment Corporation
EKF	Extended Kalman Filter
FFT	Fast Fourier Transform
FOHMD	Fiber Optic Helmet-Mounted Display
GP	Graphics Processor (in Pixel-Planes 5)
GPS	Global Positioning System
HIF	Host InterFace board (for Pixel-Planes 5)
HMD	Head-Mounted Display
HUD	Head-Up Display
Hz	Hertz (cycles per second)
I/O	Input and Output
ID	Identification
IPD	Interpupillary Distance
ISA	Industry Standard Architecture (bus for PC's)
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MByte	Megabyte (10^6 bytes)
MHz	Megahertz (10^6 Hz)

MIMD	Multiple Instruction, Multiple Data
ms	milliseconds (10^{-3} seconds)
MS-DOS	Microsoft Disk Operating System
ns	nanoseconds (10^{-9} seconds)
NTSC	National Television System Committee
ODE	Ordinary Differential Equation
OS	Operating System
PC	Personal Computer
Pxpl5	Pixel-Planes 5
RMS	Root-Mean-Square
ROS	Ring Operating System (for Pixel-Planes 5)
RP	Remote Processor
SIGGRAPH	Special Interest Group on Computer Graphics (a division of ACM). Also refers to an annual conference sponsored by that group.
SIMD	Single Instruction, Multiple Data
TAXI	Transparent Asynchronous Xmitter-receiver Interface
TV	Television
UNC	University of North Carolina
μsec	microseconds (10^{-6} seconds)
VE	Virtual Environment
VME	VERSAbus Module European
VR	Virtual Reality

LIST OF SYMBOLS

Q, *x, h*()	Italicized mathematical expressions are scalar variables, labels, or functions if accompanied with parentheses
M, **X**	Boldfaced mathematical expressions are vectors, matrices, or quaternions
x	Cross product: **M** x **V**
·	Dot product: **M · X**
•	Quaternion multiplication: $\mathbf{Q_1} \bullet \mathbf{Q_2}$
space	Implied multiplication, scalar *p w*, or matrix **M X**
$\dot{r}$	Derivative of *r* with respect to time
$\ddot{r}$	Double derivative of *r* with respect to time
$\mathbf{A}^T$	Transpose of **A**
$\mathbf{A}^{-1}$	Inverse of **A**
G[0]	The first scalar element inside vector **G**. Indices are zero-origin.
A[2, 0]	Scalar element at row #2 and column #0 inside matrix **A**. Indices are zero-origin.
ω	Omega, a 3 by 1 vector representing angular velocity (Used in Chapters 4 and 5.)
ω	Angular frequency. $\omega = 2\pi f$, where *f* is the frequency in Hertz. (Used in Chapter 6.)

1. Introduction

1.1 Background

Although tremendous advances in rendering three-dimensional graphic objects have been made in the past two decades, very little has changed in the way that typical users manipulate and view these 3-D objects. In 1975, graphic displays typically showed unshaded line drawings with no hidden surfaces. Today, a consumer can buy a rendering package for her personal computer which generates images that, in certain cases, approach photorealism. But both today's user and her counterpart 20 years ago look at these objects through a monitor and manipulate them with dials, keyboards, joysticks, light pens, and other such devices. The computer graphic objects exist in their own separate world, visible only "through the looking glass," remote and difficult to interact with. Consider how hard it would be to accomplish everyday tasks if you could only look at objects in your surrounding environment by viewing a monitor attached to a TV camera, and you could only grasp and move those objects by manipulating dials, keyboards, joysticks, etc.

In 1965, Ivan Sutherland proposed a different metaphor for human-computer interaction. Sutherland's Ultimate Display is one that immerses humans inside a three-dimensional computer-generated world indistinguishable from reality [Sutherland65]. In this virtual world, one could grab or sit down on a virtual chair. Being shot by a virtual bullet might prove fatal. Inside this virtual world, humans directly view and interact with 3-D objects using intuitive, natural skills developed since birth: walking, grasping, etc.

Sutherland's Ultimate Display is far beyond the capabilities of current technologies. However, it is possible to approximate parts of it. In 1968,

Sutherland and his team built the first *Head-Mounted Display* (HMD) [Sutherland68]. The HMD system consists of three main components: the HMD itself, a scene generator, and a tracking system. The HMD is a helmet-like device, worn on the head, which holds two tiny display devices in front of the user's eyes. These two displays project stereo image pairs, generated in real time by the *scene generator*, which is a computer specialized for the rapid generation of graphic images. The tracker mounted on the HMD tells the scene generator the position and orientation of the user's head in 3-D space. With this tracker information, the scene generator can generate images of what the user should see at that particular position and orientation. If the system responds quickly to user motion and generates images of sufficient fidelity, the user experiences the illusion of being immersed inside an intangible, virtual world that surrounds her. Now, instead of gazing into the looking glass, she has gone through the looking glass, entering the same space that the 3-D graphic objects inhabit. Figure 1.1 shows a picture of Sutherland's original HMD, and Figure 1.2 shows a user inside a more modern HMD viewing representations of molecules.

Figure 1.1: Sutherland's HMD

Figure 1.2: User viewing molecular models with an HMD

Recent media publicity has stirred public interest in HMDs, which are often lumped together with other technologies into an area called *Virtual Reality* (VR). Many researchers prefer the term *Virtual Environment* (VE) as a more accurate description of the effect achieved. The hype surrounding this field may have deluded the public into believing that the current state-of-the-art is closer to Sutherland's Ultimate Display than it really is. Modern systems fall far short of the goal of generating completely convincing Virtual Environments. There is much room for improvement in each of the three main technologies that drive HMDs: the display devices, the scene generator, and the tracking system. Equally important for gaining user acceptance is the need to demonstrate serious applications that significantly benefit from the use of HMDs.

Some of these serious applications may come from a variant of Virtual Environments, called *Augmented Reality* (AR). The difference between Virtual Environments and Augmented Reality is in their treatment of the real world. Virtual Environments use HMDs that completely replace the real world with a computer-generated synthetic environment. Removing the electrical power from such a *closed-view HMD* would effectively blind the user wearing

it. In contrast, Augmented Reality *supplements*, rather than supplants, the real world. Augmented Reality systems use *see-through HMDs* that superimpose virtual objects upon the wearer's view of the real world. In theory, these virtual objects can be combined with real world objects so that the two worlds merge together seamlessly. For example, consider a virtual vase sitting on a real chair. When the user walks behind the chair so that the back of the chair covers most of the vase, the see-through HMD should only display the part of the vase that sticks up above the back of the chair, preserving the illusion that both objects exist in the same space.

1.2 Motivation

What serious applications would benefit from Augmented Reality technology, if it were available? Augmented Reality enhances a user's perception of and interaction with the real world. The virtual objects display information that the user cannot directly detect with her senses. Two areas where this technology could help are medical applications and the assembly and repair of complicated mechanical devices.

Doctors could use Augmented Reality as a visualization aid during surgery. For example, a research group at UNC has conducted trial runs of scanning the womb of a pregnant woman with an ultrasound sensor, generating a real-time 3-D representation of the fetus inside the womb and displaying that in a see-through HMD (Figure 1.3). The goal is to endow the doctor with "X-ray vision," granting her the ability to see the moving, kicking fetus lying inside the womb [Bajura92] [State94]. Another potential application is needle biopsy. A doctor wearing a see-through HMD could see a virtual object specifying the exact location of a tiny tumor within the patient, helping her perform the biopsy. See-through HMDs might also help make minimally-invasive surgery easier. While minimally-invasive techniques are less stressful to the patient, they are difficult because of the reduced visibility inside the patient. Combining see-through HMDs with real-time 3-D sensors that scan the inside of the patient would enable the surgeon to see what she was doing without cutting the patient open, reducing the trauma of the operation.

Figure 1.3: Virtual fetus superimposed on pregnant patient

Another potential category of Augmented Reality applications is the assembly, maintenance, and repair of complex machinery. Instructions might be easier to understand if they were available, not in the form of manuals with text and pictures, but as 3-D drawings superimposed upon the actual equipment, showing step-by-step the tasks that need to be done and how to do them. Steve Feiner's group at Columbia demonstrated this in a laser printer maintenance application [Feiner93]. Storing these instructions in electronic form might also save space and reduce costs. A group at Boeing uses a see-through HMD to guide a technician in building a wiring harness that forms part of an airplane's electrical system. Currently, technicians use large physical layout boards to construct such harnesses, and Boeing uses several warehouses to store all these boards. Such space might be emptied for better use if this application proves successful [Janin93]. See-through HMDs might aid other mechanical tasks as well. An architect with a see-through HMD might be able to look out a window and see how a proposed new skyscraper would change her view. Figure 1.4 shows a conceptual diagram of what a jet engine maintenance application might look like.

Figure 1.4: Conceptual drawing of engine maintenance application. Diagram drawn by Andrei State.

Robinett speculates that Augmented Reality may be useful in any application that requires displaying information not directly available or detectable by human senses by making that information visible (or hearable, feelable, etc.) [Robinett92a]. Appropriate head-based sensors might extend the user's visual range into the infrared or ultraviolet frequencies, and remote sensors would let users view objects hidden by walls or hills. Conceptually, anything not detectable by human senses but detectable by machines might be transduced into something that a user can sense inside a see-through HMD.

1.3 The problem

Although Augmented Reality offers tantalizing potential for increasing worker productivity, existing technology is not able to support these tasks. Today, Augmented Reality is barely at the demonstration phase. Existing demos offer only a glimpse into future applications; they are not robust and powerful tools at present. Many problems must be overcome before the full potential of Augmented Reality is realized. One of the most basic is the

registration problem. The objects in the real and virtual worlds must be properly aligned with respect to each other, or the illusion that the two worlds coexist will be compromised. More seriously, many applications *demand* accurate registration. For example, recall the needle biopsy application. If the virtual object is not where the real tumor is, the surgeon will miss the tumor and the biopsy will fail. Without accurate registration, Augmented Reality will not be accepted in many applications.

Registration problems also exist in Virtual Environments, but they are not nearly as serious because they are harder to detect than in Augmented Reality. Since the user only sees virtual objects in VE applications, registration errors result in visual-kinesthetic and visual-proprioceptive conflicts. Such conflicts between different human senses may be a source of motion sickness [Pausch92]. Because the kinesthetic and proprioceptive systems are much less sensitive than the visual system, visual-kinesthetic and visual-proprioceptive conflicts are less noticeable than visual-visual conflicts. For example, a user wearing a closed-view HMD might hold up her real hand and see a virtual hand. This virtual hand should be displayed exactly where she would see her real hand, if she were not wearing an HMD. But if the virtual hand is wrong by five mm, she may not detect that unless actively looking for such errors. The same error is much more obvious in a see-through HMD, where the conflict is visual-visual.

Furthermore, a phenomenon known as *visual capture* [Welch78] makes it even more difficult to detect such registration errors. Visual capture is the tendency of the brain to believe what it sees rather than what it feels, hears, etc. That is, visual information tends to override all other senses. When watching a television program, a viewer believes the sounds come from the mouths of the actors on the screen, even though they actually come from a speaker in the TV. Ventriloquism works because of visual capture. Similarly, a user might believe that her hand is where the virtual hand is drawn, rather than where her real hand actually is, because of visual capture. This effect increases the amount of registration error users can tolerate in Virtual Environment systems. If the errors are systematic, users might even be able to adapt to the new environment, given a long exposure time of several hours or days [Welch78].

Augmented Reality demands much more accurate registration than Virtual Environments [Azuma93]. Imagine the same scenario of a user holding up her hand, but this time wearing a see-through HMD. Registration errors now result in visual-visual conflicts between the images of the virtual and real hands. Such conflicts are easy to detect because of the resolution of the human eye and the sensitivity of the human visual system to differences. Even tiny offsets in the images of the real and virtual hands are easy to detect.

What angular accuracy is needed for good registration in Augmented Reality? A simple demonstration will show the order of magnitude required. Take out a dime and hold it at arm's length, so that it looks like a circle. The diameter of the dime covers about 1.2 to 2.0 degrees of arc, depending on your arm length. In comparison, the width of a full moon is about 0.5 degrees of arc! Now imagine a virtual object superimposed on a real object, but offset by the diameter of the dime. Such a difference would be easy to detect. Thus, the angular accuracy required is a small fraction of a degree. The lower limit is bounded by the resolving power of the human eye itself. The central part of the retina is called the *fovea*, which has the highest density of color-detecting cones, about 120 per degree of arc, corresponding to a spacing of half a minute of arc [Jain89]. Observers can differentiate between a dark and light bar grating when each bar subtends about one minute of arc, and under specialized circumstances they can detect even smaller differences [Doenges85]. However, existing HMD trackers and displays are not capable of providing one minute of arc in accuracy or resolution, so the present achievable accuracy is much worse than that ultimate lower bound. In practice, errors of a few pixels are detectable in modern HMDs.

These requirements are difficult to meet. The current state-of-the-art for registration with see-through HMDs, as reported in the text and pictures of [Bajura92] [Feiner93] [Janin93], achieves errors on the order of 13 mm for objects at arm's length, viewed by a stationary head from a restricted range of viewpoints. This corresponds to about 1.1 degrees of arc for a 68 cm arm length.

1.4 Sources of error

Registration errors are difficult to adequately control because of the high accuracy requirements and the numerous sources of error. The main sources are:

- Distortion in the HMD optics
- Mechanical misalignments in the HMD
- Errors in the head-tracking system
- Incorrect viewing parameters (field of view, tracker-to-eye position and orientation, interpupillary distance)
- End-to-end system delays

The first four are *static* errors, because they cause registration errors even when the user's head remains still. The fifth category, end-to-end delays, is a *dynamic* error, because it has no effect until the user's head moves.

Although static errors are important, dynamic errors currently contribute the most to registration errors. While previous work has demonstrated static errors of about 13 mm, dynamic errors can easily exceed 100 mm in magnitude, or about 8.3 degrees of arc for a 68 cm arm length.

Dynamic errors occur because of system delays, or lags. I define the *end-to-end system delay* as the time difference between the moment that the tracking system measures the position and orientation of the user's head to the moment when the generated images corresponding to that position and orientation appear in the HMD. These delays exist because each component in an Augmented Reality system requires some time to do its job. The delays in the tracking subsystem, the communication delays, the time it takes the scene generator to draw the appropriate images in the frame buffers, and the scanout time from the frame buffer to the displays all contribute to end-to-end lag. On our systems, end-to-end delays typically exceed 100 ms. Simpler systems can have less delay, but other systems have more. Delays of 250 ms or more can exist on slow, heavily loaded, or networked systems.

End-to-end system delays cause registration errors only when the user moves her head. Assume that the user stands still. Then the lag does not

cause registration errors. No matter how long the delay is, the images generated are appropriate for the user's position and orientation, since the head has not moved since the time the tracker measurement was taken. Compare this to the case when the user moves her head. Say that the tracker measures the head at an initial time *t*. The images corresponding to time *t* will not appear until some future time t_2, because of the end-to-end system delays. During this delay, the user's head remains in motion, so when the images computed at time *t* finally appear, the user sees them at a different location than the one they were computed for. Thus, the images are incorrect for the time they are actually viewed. To the user, the virtual objects appear to "swim around" and "lag behind" the real objects. This was graphically demonstrated in the videotape of UNC's ultrasound experiment, shown at SIGGRAPH '92 [Bajura92]. Figure 1.5 shows what the registration looks like when everything stands still. The virtual gray trapezoidal region represents what the ultrasound wand is scanning. This virtual trapezoid should be attached to the tip of the real ultrasound wand. This is the case in Figure 1.5, where the tip of the wand is visible at the bottom of the picture, to the left of the "UNC" letters. But when the head or the wand moves, large dynamic registration errors occur, as shown in Figure 1.6. The tip of the wand is now far away from the virtual trapezoid. Also note the motion blur in the background, which is caused by the user's head motion.

Figure 1.5: Ultrasound static registration: Virtual gray trapezoid registered with the wand tip.

Figure 1.6: Dynamic misregistration due to system delays. Virtual trapezoid is now separated from the wand tip.

How rapidly do people move their heads while wearing an HMD? This will vary with the user and the application. I collected head motion from naive users who tried a demonstration of one of our HMD systems. The application consisted of a virtual room of interesting objects. The users walked around inside this environment, grabbing and manipulating various objects.
Figure 1.7 shows a histogram of the measured angular velocities for a slow-moving head. Figure 1.8 shows a histogram for a fast motion sequence. Both distributions are similar in form, with each user moving slowly on average. The main difference is in the peaks: 70 degrees per second for the slow sequence, and 120 degrees per second for the fast sequence.

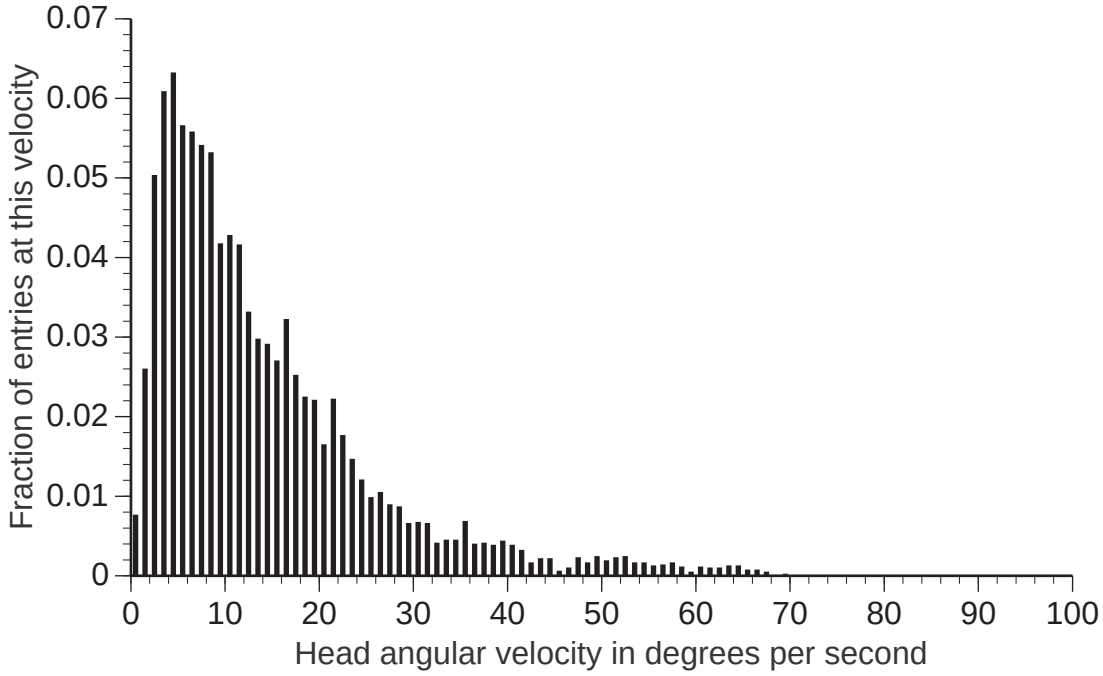


Figure 1.7: Histogram of head angular velocities, slow motion sequence

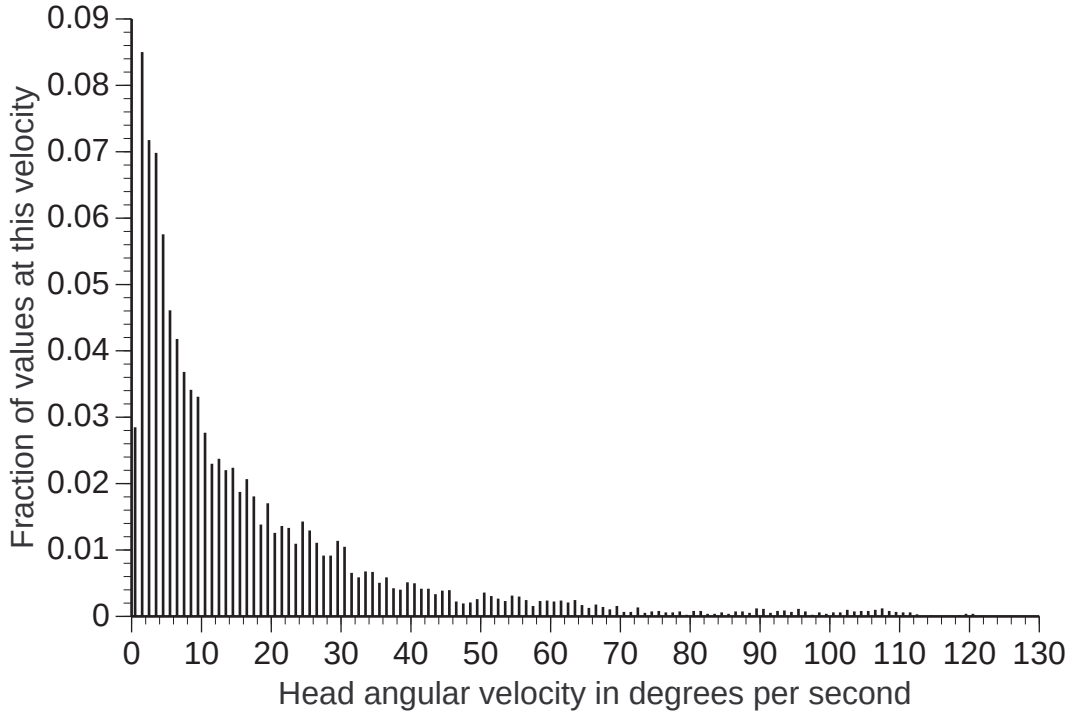


Figure 1.8: Histogram of head angular velocities, fast motion sequence

Some previous works have also measured rates of head motion. Uwe List reports peak angular velocities ranging from 76 to 240 degrees per

second in a flight-simulator application where pilots were asked to fixate on a target [List83]. Smith reports that peak velocities of 300 degrees per second are seen with the CAE flight simulator [Smith84].

I can also plot the data in Figures 1.7 and 1.8 in the form of cumulative density functions. In this case, a *cumulative density function* specifies the fraction of all values in the dataset that are at or below the specified angular velocity. Figure 1.9 shows the cumulative density functions for the slow and fast motion sequences. In these demonstration applications, the users spend most of their time moving slowly. About half of the values are under 10 degrees per second. However, in the fast motion sequence, a small but significant amount of time is spent at fast velocities. Ten percent of the values range from 40 to 120 degrees per second.

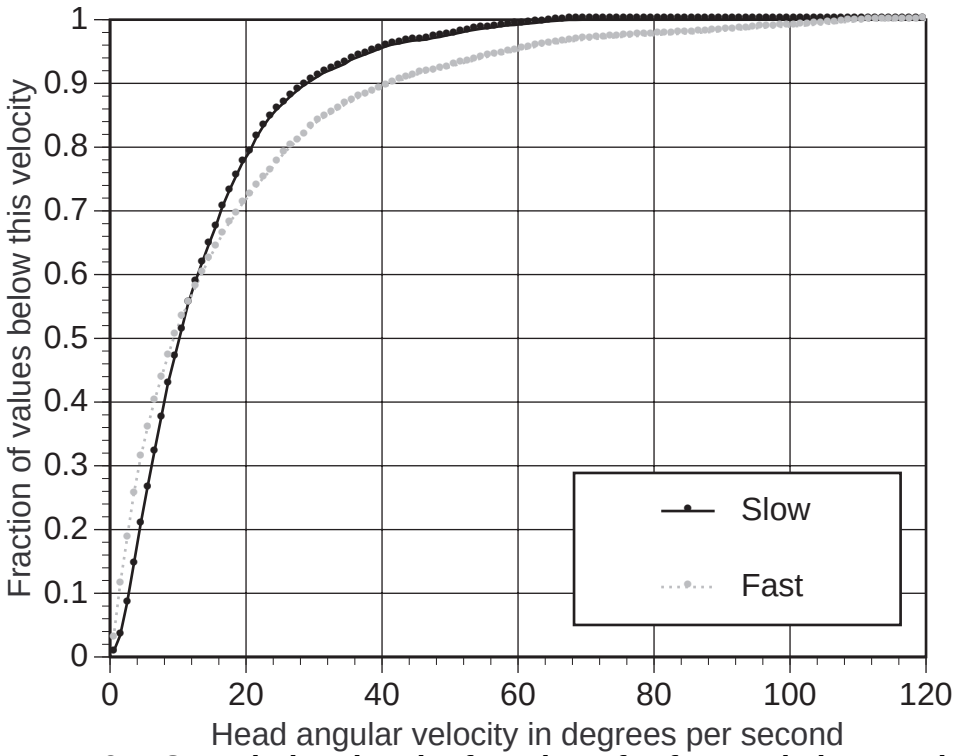


Figure 1.9: Cumulative density functions for fast and slow motion sequences

System delays seriously hurt the illusion that the real and virtual worlds coexist because they cause large registration errors. With a typical end-to-end lag of 100 ms and a moderate head rotation rate of 50 degrees per second, the angular dynamic error is 5 degrees. At a 68 cm arm length, this

results in registration errors of almost 60 mm. This is 4.5 times larger than the 13 mm static registration errors achieved in previous work.

System delays are not likely to disappear anytime soon. Some believe that the current course of technological development will automatically solve this problem. Unfortunately, it is difficult to reduce system delays to the point where they are no longer an issue. Recall that registration errors must be kept to a small fraction of a degree. At the moderate head rotation rate of 50 degrees per second, system lag must be 10 ms or less to keep angular errors below 0.5 degrees. Just scanning out a frame buffer to a display at 60 Hz requires 16.67 ms. Modern scene generators tend to use pipelined architectures to increase throughput, at the cost of increasing delay [Foley90]. It might be possible to build an HMD system with less than 10 ms of lag, but the drastic cut in throughput and the expense required to construct the system would make such a solution unattractive. Minimizing system delay is important, but reducing delay to the point where it is no longer a source of registration error is not practical with existing technology.

1.5 Contribution

Since sufficiently reducing end-to-end delay is too difficult, another approach is to compensate for delay. When the tracker reads the head location, what the scene generator should use is not the measured head location but rather the head location at the time when the user views the images. If this future head location can be perfectly determined, then the system delay does not cause any registration errors. That is, compensation requires predicting future motions of the user's head.

Predicting the future, whatever the subject, is a difficult task. Forecasts cannot be perfect, since unanticipated events will make predictions wrong. However, it is my belief that for this particular problem, reasonably accurate predictions are feasible. Therefore, the goal of this dissertation is to establish:

Thesis Statement

Predictive tracking is an effective means of significantly reducing registration errors caused by delays in Augmented Reality systems.

To defend this thesis, I make the following contributions:

- Improved static registration
- Improved dynamic registration, using predictive tracking
- Evaluation of inertial-based prediction
- Demonstration of the static and dynamic improvements in a real system
- Report of lessons learned based on experience with this system
- Autocalibration and optimization techniques for determining system parameters
- Characterization of the theoretical behavior of this predictor

I now provide a summary of these contribution areas and where they are covered in the rest of this dissertation. Chapter 2 begins by describing the system at a high level and defining the registration task.

Improved static registration (Chapter 3): Pictures and videos of existing Augmented Reality systems show registration from only a small range of viewpoints. The user is not allowed to translate or rotate the HMD very far from the initial viewpoint. There are two reasons for this limitation. First, most commercially-available head-tracking systems do not provide sufficient accuracy and range to permit such movement without greatly increasing the static registration errors. Second, determining robust viewing parameters is non-trivial. *Viewing parameters* are the values required to generate the graphic images, given the measurements from the head tracker. They include field-of-view, the position and orientation offsets from the head tracker to the user's eyes, and the locations of the real objects with respect to the HMD. Determining viewing parameters that work from just one viewpoint is much easier than determining parameters that work from many different viewpoints. The parameters must be correct to work at many different viewpoints, but at a single viewpoint it is possible to find several other

combinations of parameters where the errors happen to cancel at that particular viewpoint. For example, assume the position offset from the tracker to an eye is wrong by 10 cm, but the location of the real object is incorrect by 10 cm in exactly the opposite direction. Therefore, parameters that yield good registration from one viewpoint may result in static errors of a few degrees at another viewpoint.

While the thesis of this dissertation is not explicitly about reducing static registration errors, good static registration is important for evaluating dynamic registration errors. This evaluation is based on how closely the real and virtual objects are aligned. If the alignment is accurate when the user remains still but inaccurate when the head is in motion, then the user can easily see the effect of dynamic errors. However, if the alignment is inaccurate even in the static case, then accurate registration never occurs, making it more difficult to see the effect of prediction. Thus, good static registration is helpful for a convincing demonstration of the effectiveness of prediction.

My system is capable of keeping a virtual and real object closely aligned across many widely-spaced viewpoints and view directions. The tracker is a custom optoelectronic system built at UNC, which has less distortion and longer range than most commercially-available trackers. I developed calibration techniques for determining the viewing parameters. The robust static registration is demonstrated by several still photographs taken from a single video sequence in which the user walked 270 degrees around the registration object. The static errors are usually within ±5 mm from most viewpoints (0.42 degrees for a 68 cm arm length), which is less than the ±13 mm shown in previous work.

Improved dynamic registration (Chapter 4): I developed prediction algorithms that reduce dynamic errors caused by the system delays. Inertial sensors (three angular rate gyroscopes and three linear accelerometers) were added to the see-through HMD. These sensors and the head tracker measure the head's position, orientation, angular velocity, and linear acceleration. To generate estimates of the velocity and acceleration for both orientation and position, I use a Kalman filter, which is described further in Chapter 4. Given the sensor readings, the noise in the original signals and a

model of the user's motion, the Kalman filter produces the desired estimates. Then the predictor uses these estimates to extrapolate head position and orientation the required interval into the future.

This is not the first research effort to use predictive tracking. How does my predictor differ from previous attempts at using prediction to compensate for delays in HMD systems? It is the first to use *both* gyroscopes and accelerometers to aid prediction. Most previous works based their predictions only on the tracker readings, without inertial sensors. Of the two that included inertial sensors, both predicted orientation only, using angular accelerometers. Also, no previous work has attempted to use and evaluate prediction in the context of reducing registration errors in see-through HMDs.

Evaluation of inertial-based prediction (Chapter 4): No previous work evaluates how much inertial sensors aid head-motion prediction. I offer three objective error metrics and compare the results of prediction on representative motion traces that were captured in real time with the actual system. My inertial-based predictor reduces dynamic errors by a factor of 5 to 10 over not doing any prediction at all and by a factor of 2 to 3 over a representative predictor that does not use inertial sensors. This is also demonstrated in a videotape that shows what the user sees inside the HMD.

Demonstration in a real system (Chapter 5): This work was not done purely as a theoretical study or implemented solely in a simulation. I built a real, working Augmented Reality system that runs in real time to test and evaluate my registration techniques. My prediction routines are tested on actual collected motion data taken from the head tracker and the head-mounted inertial sensors, not on artificially-created simulated data. This shows that prediction can work in real time in a real system.

Report of lessons learned from the real system (Chapter 5): In a simulator or in a theoretical study, certain difficulties can be avoided by redefining the problem, assuming perfect knowledge of the system, changing definitions, or making different assumptions. But when building a real system, these difficulties cannot be avoided. These lessons about what is hard in building an effective Augmented Reality system have not been covered by the

previous work in this field. They are an important contribution to pass on to anyone else who works in this area.

Two main lessons have come out of this system. First, accurate prediction requires accurate knowledge of how *far* to predict. While this sounds like an innocent truism, it has serious ramifications for AR system design. Timestamps must be available at several stages in the pipeline. Since the various components in our system run asynchronously, the prediction interval varies from iteration to iteration, requiring continual adjustments. Accurate estimation of this varying prediction interval requires removing all uncontrollable sources of latency, like operating system resources and communication pathways shared with other users. All these requirements are not usually found in existing VE and AR systems.

The second lesson is that prediction is not a total panacea. One cannot predict arbitrary intervals into the future and expect good results. Prediction errors grow rapidly with increasing prediction intervals, and jitter in the predicted outputs is disturbing at long prediction intervals. Attempts to reduce these problems by filtering or other techniques failed because they introduce additional lag, the very problem the predictor tries to eliminate. In practice, the only solution is to keep prediction intervals short, below ~80 ms.

Autocalibration and optimization (Chapter 5): The prediction module requires several parameters: the locations of the inertial sensors, the bias and scale factors to use when processing the inertial signals, and estimates of how much to trust the sensors and the motion model. If the parameters are not set properly, prediction accuracy will suffer. Therefore, determining these parameters is an important part of effective prediction. I developed optimization and autocalibration techniques to set most of these parameters. These routines take recorded motion runs and use nonlinear search techniques to find the parameters that best match the recorded data. Some of these work better than others. An evaluation of the effectiveness of these routines is provided.

Theoretical behavior of the predictor (Chapter 6): No previous works have provided a theoretical description of how their prediction methods behave. I characterize the behavior of the Kalman filter and the predictor I

use by analyzing them in the frequency domain. This involves deriving transfer functions for the predictor, the Kalman filter, and the combination of the two. These transfer functions describe how the filter and predictor modify the head-motion signals in the frequency domain, allowing the exploration of the prediction parameter space by analysis rather than purely empirical observations. This theoretical framework demonstrates where the observed jitter comes from and shows what happens to the predicted signals as the prediction interval increases. High-frequency components in the original signal contribute the most to jitter because the predictor preferentially magnifies signals roughly by the square of the angular frequency. Measuring acceleration appears to increase prediction accuracy more than having measured velocity available does. These functions also estimate the largest possible time-domain error, given the spectral characteristics of the original signal. That allows designers to specify the maximum tolerable prediction error and then determine the longest system delay that will meet the specification, given the specific predictor and the type of motion that users perform. Unfortunately, specifying such a bound given the choice of any *arbitrary* predictor is an intractable problem, but this framework might be modified for future predictors, assuming they are linear or linearizable.

The combination of these static and dynamic techniques results in registration that is better than any previously demonstrated. Registration errors have been called "the swimming problem," because the virtual objects "swim around" the real ones, making it difficult to believe that any rigid relationship between the real and virtual objects exists. This dissertation achieves a milestone in that viewers now see real and virtual objects as "staying closely attached," rather than "swimming around." Noticeable misregistrations still occur, both static and dynamic, but this dissertation brings the field within striking distance of the ideal goal of "No Swimming," which would achieve adequately precise registration for all conditions observed in actual applications. Further work must be done to achieve this goal, and I suggest several areas to explore in Chapter 7.

2. Problem statement

Before describing the methods used to attack the registration problem, I will describe the system and define the registration task. This description will be at an abstract level, specifying system components by what they do and not how they do it, whenever possible. For example, the tracking system description will not specify the technology: optical, magnetic, mechanical, etc. These abstract descriptions will be sufficient for understanding the methods presented in Chapters 3 and 4 and allow these methods to be applied in different contexts.

2.1 See-through HMD systems

Augmented Reality uses see-through HMD systems to blend real and virtual worlds together. Such systems consist of a see-through HMD, a tracking system, and a scene generator. The tracker measures the position and orientation of the head. With this information and models of the virtual objects, the scene generator can generate appropriate images of these virtual objects, as they would appear if viewed from the measured head location. The see-through HMD blends these virtual images with a view of the real world, completing the illusion.

The two main ways of accomplishing this blending are by optical or video technologies. Thus, see-through HMDs are categorized as either *optical see-through HMDs* or *video see-through HMDs*. The next three sections describe each technology and compare their strengths and weaknesses, justifying the choice of an optical see-through HMD for exploring the registration problem.

2.1.1 Optical see-through

Optical see-through HMDs work by placing optical combiners in front of the user's eyes. These combiners are partially transmissive, so that the user can look directly through them to see the real world. If the power is turned off, the user can still see the real world. The combiners are partially reflective, so that the user can also see virtual images bounced off the combiners from head-mounted monitors. This approach is similar in nature to *Head-Up Displays* (HUDs) commonly used in military aircraft, except that the combiners are attached to the head. Thus, optical see-through HMDs have sometimes been described as a "HUD on a head" [Wanstall89]. Figure 2.1 shows a conceptual diagram of an optical see-through HMD.

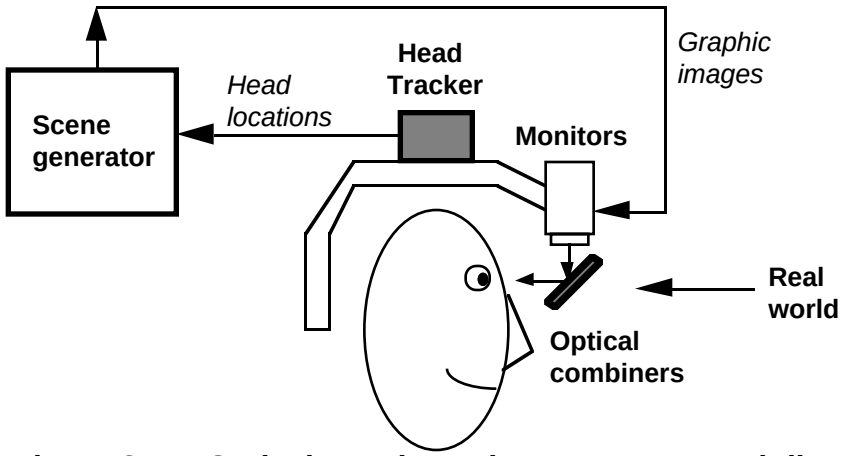


Figure 2.1: Optical see-through HMD conceptual diagram

2.1.2 Video see-through

In contrast, video see-through HMDs work by combining a closed-view HMD with one or two head-mounted video cameras. The video cameras provide the user's view of the real world. Video from these cameras is combined with the graphic images created by the scene generator, blending the real and virtual. The result is sent to the monitors in front of the user's eyes in the closed-view HMD. Unlike optical see-through HMDs, the user has no direct view of the real world, so if the power is turned off, the user is effectively blinded. Figure 2.2 shows a conceptual diagram of a video see-through HMD.

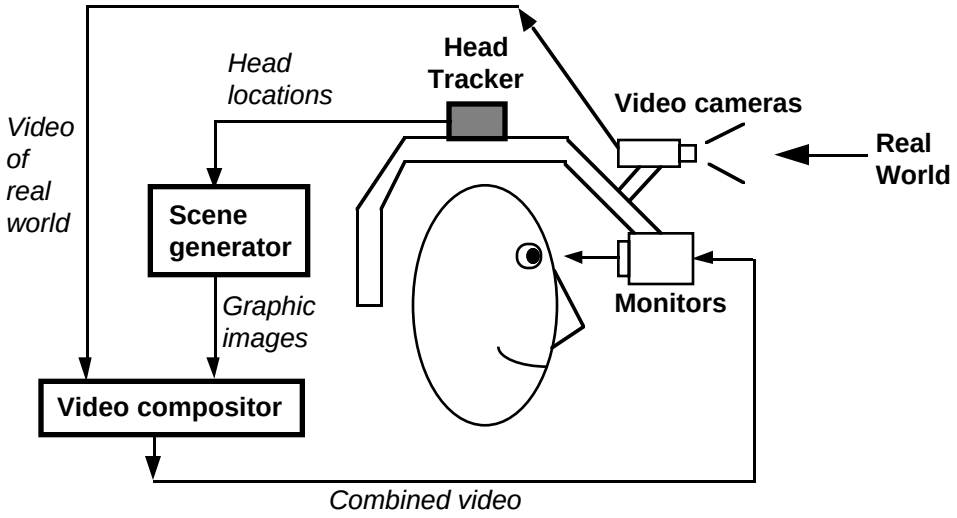


Figure 2.2: Video see-through HMD conceptual diagram

Video composition can be done in more than one way. A simple way is to use chroma-keying, the same technique used in many video special effects. The background of the computer graphic images is set to a specific color, say green, which none of the virtual objects use. Then the combining step replaces all green areas with the corresponding parts from the video of the real world. This has the effect of superimposing the virtual objects over the real world. A more sophisticated composition would use depth information. If the system had depth information at each pixel for the real world images, it could combine the real and virtual images by a pixel-by-pixel depth comparison. This would allow real objects to cover virtual objects and vice-versa. For example, a virtual lamp located behind a real table would be partially obscured by the real table, just as it should be if it was really there.

2.1.3 Comparison of optical and video see-through approaches

Video see-through offers the following advantages over optical see-through:

1) Flexibility in composition strategies: A basic problem with optical see-through is that the virtual objects do not completely obscure the real world objects, because the optical combiners allow light from both the virtual and real sources. Building an optical see-through HMD that can selectively shut out the light from the real world is difficult. Thus, the virtual objects appear ghost-like and semi-transparent. This damages the illusion of reality

because occlusion is one of the strongest depth cues. In contrast, video see-through is far more flexible about how it merges the real and virtual images. Since both the real and virtual are available in digitized form, video see-through compositors can, on a pixel-by-pixel basis, take the real, or the virtual, or some blend between the two to simulate transparency. Because of this flexibility, video see-through may ultimately produce more compelling environments than optical see-through approaches.

2) Wide field-of-view: Distortions in optical systems are a function of the radial distance away from the optical axis. The further one looks away from the center of the view, the larger the distortions get. A digitized image taken through a distorted optical system can be undistorted by applying image processing techniques to unwarp the image, provided that the optical distortion is well characterized. This requires significant amounts of computation, but this constraint will be less important in the future as computers become faster. It is harder to build wide field-of-view displays with optical see-through techniques. Any distortions of the user's view of the real world must be corrected *optically*, rather than digitally, because the system has no digitized image of the real world to manipulate. Complex optics are expensive and add weight to the HMD.

3) Real and virtual view delays can be matched: Video see-through offers another approach for reducing the registration errors caused by delays. Instead of predicting the virtual images, delay the video of the real world to match the delay in the virtual image stream. This approach does not apply to optical see-through, because that gives the user a direct view of the real world. However, eliminating dynamic error through this approach comes at the cost of delaying *both* the real and virtual views, so it appears to the user as if *everything* lags behind. Such problems are common in telepresence systems and are not easily solved, because "predicting" future images of the real environment is a non-trivial task.

Optical see-through has the following advantages over video see-through:

1) Simplicity: Optical see-through is a simpler approach than video see-through. Optical approaches have only one "stream" of video to worry

about: the graphics images. The real world is seen directly through the combiners, and that time delay is generally a few nanoseconds. Video see-through, on the other hand, must deal with separate video streams for the real and virtual images. Both streams have inherent delays in the tens of milliseconds. The two streams must be properly synchronized, or temporal distortion results. Also, optical see-through HMDs with narrow field-of-view combiners offer views of the real world that are basically undistorted. Video see-through cameras almost always have some amount of distortion that must be compensated for, along with any distortion from the optics in front of the display devices.

2) Resolution: Video see-through limits the resolution of what the user sees, both real and virtual, to the resolution of the display devices. With current displays, this resolution is far less than the resolving power of the fovea. Optical see-through also shows the graphic images at the resolution of the display device, but the user's view of the real world is not degraded. Thus, video see-through reduces the resolution of the real world, while optical see-through does not.

3) No eye offset: With video see-through, the user's view of the real world is provided by the video cameras. In essence, this puts his "eyes" where the video cameras are. In most configurations, the cameras are not located exactly where the user's eyes are, creating an offset between the cameras and the real eyes. This offset introduces displacements from what the user sees compared to what he expects to see. For example, if the cameras are above the user's eyes, he will see a view from a vantage point slightly taller than he is used to. Video see-through can avoid the eye offset problem with the use of mirrors to create another set of optical paths that mimic the paths directly into the user's eyes. Using those paths, the cameras will see what the user's eyes would normally see without the HMD. This adds complexity to the HMD design, however. Offset is generally not a difficult design problem for optical see-through displays.

To tackle the registration problem, I chose to use optical see-through instead of video see-through, primarily because the former is simpler. Digitizing video images usually adds at least one *frame time* of delay to the video stream, where a frame time is how long it takes to completely update an

image. A monitor that completely refreshes the screen at 60 Hz has a frame time of 16.67 ms. Dealing with two delayed video streams instead of just one makes it harder to synchronize the two streams and to accurately determine the required prediction intervals. For the purpose of tackling the registration problem, the disadvantages of optical see-through HMDs are minor, while the advantages of a single video stream and having fewer sources of optical distortions are significant.

Note, however, that video see-through can support strategies for achieving registration that are unavailable in optical see-through, because the former has digitized images of the real world. This dissertation focuses only on registration techniques implementable with optical see-through HMDs. Additional approaches for video see-through that merit future investigation are briefly described in Chapter 7.

2.2 Tracker

The HMD has a head tracker attached to it, which returns the position and orientation of the HMD. I assume that the tracker is rigidly attached to the HMD, so that once the user puts on the HMD and adjusts it, the displays remain at fixed and constant offsets from the head tracker. The tracker is not assumed to operate at a constant rate. Reports of head locations must include timestamps, but they may arrive at variable intervals. Each report consists of an HMD position and orientation measurement (which I also refer to as "head" position and orientation for brevity). Head position, which is returned in meters, is the offset required to move the origin of the World coordinate system to the origin of the Tracker (i.e., HMD-centered) coordinate system, as measured in World space. Head orientation is a quaternion that rotates the Tracker coordinate axes so that they share the same orientation as the World coordinate axes. The coordinate systems are right-handed. Figure 2.3 shows how yaw, pitch, and roll are defined with respect to the Tracker coordinate system.

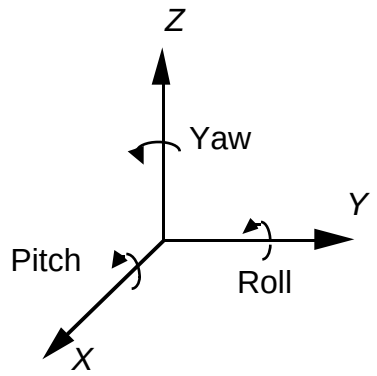


Figure 2.3: Definition of yaw, pitch, and roll with respect to Tracker space

Besides the head tracker, the HMD has inertial sensors to detect velocity and acceleration information. Two clusters of sensors exist, one for orientation sensors and the other for acceleration. Each cluster consists of three one-dimensional sensors, mounted in a mutually orthogonal configuration. Since most three-dimensional velocity and acceleration sensors are really a cluster of three 1-D sensors internally, this is a reasonable model to use. For orientation, angular accelerometers or angular rate gyroscopes are available. I chose to use angular rate gyroscopes, which detect angular velocity, because that requires only one integration step to recover orientation, compared with two integration steps for acceleration information. While also including angular accelerometers would provide direct measurement capability of angular acceleration, I found I was able to reasonably estimate that from the angular rate information, as discussed further in Section 5.3. For position, the only choice available is linear accelerometers. Linear rate sensors do not appear to exist, since no force is based on linear rates. Each cluster of sensors is mounted rigidly on the HMD, in separate locations. The position and orientation of each cluster with respect to the Tracker coordinate system does not change with time. Each cluster is assigned its own coordinate system: Gyroscope space and Accelerometer space. Figure 2.4 gives examples of where these coordinate systems may be located.

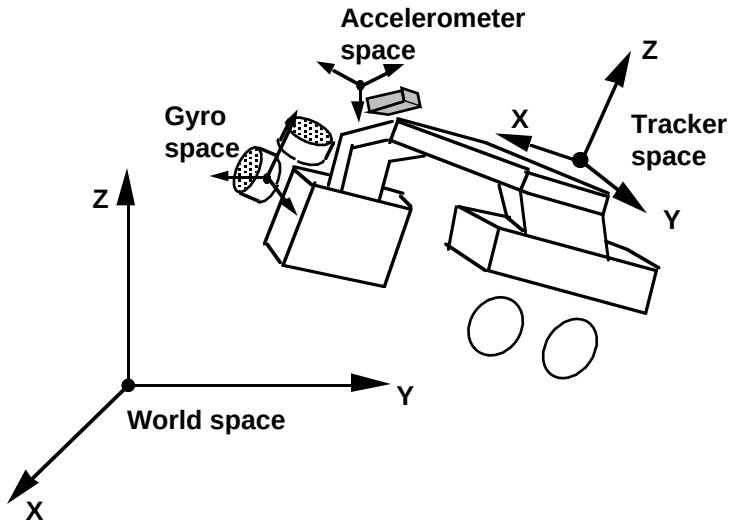


Figure 2.4: Example of coordinate systems

Each 1-D inertial sensor produces a signal that represents the angular rate or accelerations that it detects. These signals must be digitized by an A/D board and assigned a timestamp. To recover the angular rate in degrees per second or the acceleration in meters per second squared, I take a digitized value, subtract a bias, then multiply by a scale factor. Each sensor has its own bias and scale values, which may drift with time. This drift is due to thermal variations and *one-over-f* or *flicker* noise, which is ubiquitous to electronic circuits. The distribution of one-over-f noise is more concentrated at low frequencies than high frequencies, so noise at very low frequencies causes drift over the periods of minutes, hours and even days.

2.3 System operation

Figure 2.5 provides a high-level diagram of the entire system. This section steps through the basic operation of this abstract system. Details about the specific system I implemented will be left for Chapter 5.

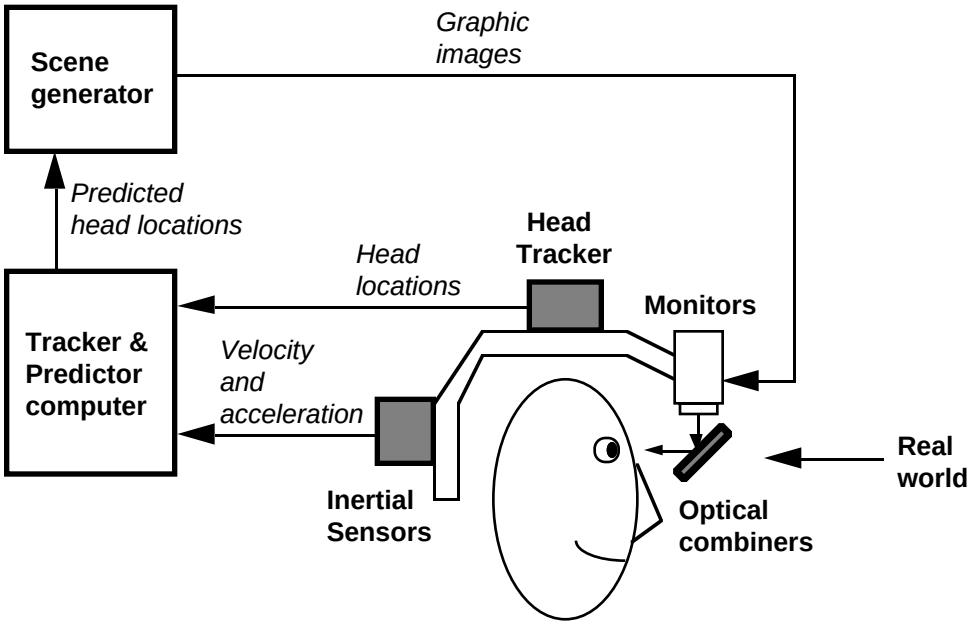


Figure 2.5: High-level system diagram

The user wears the HMD and moves his head. The head tracker and inertial sensors detect this motion and report it to the tracker and predictor computer. These reports do not have to arrive at regular intervals, but they must have associated timestamps. The tracker and predictor computer's job is to process the readings from the various sensors and provide predicted future head locations as requested. Each time the scene generator is ready to begin drawing a new set of graphic images, the tracker and predictor computer provides it with an estimated future head location to use in generating the viewpoint. How this is done is the subject of Chapter 4.

Once the future head location is provided, the scene generator must use that to generate appropriate images to display in the HMDs. This requires knowing certain parameters, primarily the field-of-view of the displays and the position and orientation offsets between the head tracker origin and the user's eyes. Determining these parameters is the subject of Chapter 3.

Note that this system is asynchronous. Both the tracker and the scene generator run as quickly as they can, without synchronization. This maximizes throughput, but it also means that the system delays will vary with time, which makes the prediction routines more complicated. This is discussed further in Chapters 4 and 5. Most real VE and AR systems are asynchronous. The exceptions are flight simulators, which are often

synchronized and place guaranteed limits on system performance [Reisman90].

2.4 Registration task

All that remains is to specify a registration task that will be used for evaluation purposes. This task should be something easy to replicate, sensitive to small errors, and not tied to any particular application. Since I am attacking the registration problem for optical see-through HMDs in general, I want the registration task to be abstract, rather than being taken from a specific AR application.

The task I use is the following: place a hollow wooden crate in the environment, at a height that is slightly shorter than a standing user, so that the user can look down upon or along the top edges of the crate to check registration. The crate measures 15" by 20.5" by 18.75". The goal is to put three virtual axes so that they are aligned with three edges of the crate, and they intersect at one corner. The three virtual axes are mutually perpendicular, forming a virtual coordinate system. Figure 2.6 is a conceptual diagram of what the crate and virtual axes should look like with proper registration. Figures 2.7 show what the actual crate looks like, and Figure 2.8 shows what good registration looks like as recorded inside the actual see-through HMD. The black cylinder attached to one corner of the wooden crate is there for calibration purposes and is explained in Chapter 3.

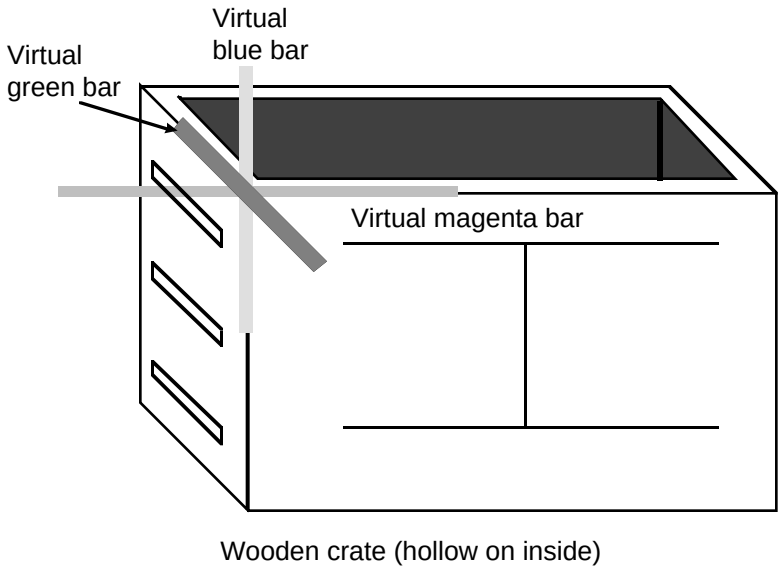


Figure 2.6: Conceptual diagram of desired registration

Figure 2.7: A picture of the actual wooden crate

Figure 2.8: Desired registration as seen inside the see-through HMD

This task satisfies the criteria previously outlined. It turns out that this task is sensitive to both position and orientation errors, because it is easy to see such misalignments when looking down the long edges of the wooden crate. The virtual axes are extruded rectangles of specified thickness, which basically act as spatial error bars. If the thick virtual bars always cover the real edges as the user moves around, then I can claim that the apparent registration is accurate within the width of those bars.

3. Static registration

This chapter describes the methods developed to achieve good static registration. Chapter 1 introduced the concept of static and dynamic registration errors. The main contribution of this dissertation lies in the reduction of dynamic errors. But without good static registration, it is difficult for a user to evaluate the reduction in dynamic error, as Chapter 1 explained. Therefore, I also had to develop methods for attacking static error.

This chapter first lists the main sources of static error, then discusses previous work. It describes the calibration methods and ends with an evaluation of what was achieved, how that compares to previous work, and what problems remain.

3.1 Sources of error

Static error was defined in Section 1.4, which listed the main sources of static error as:

- Distortion in the HMD optics
- Mechanical misalignments in the HMD
- Errors in the tracker
- Incorrect viewing parameters

Optical distortions exist in most HMD systems, especially ones with wide field-of-view displays. They are a function of the radial distance away from the optical axis. Near the center of the field-of-view, images are relatively undistorted, but far away from the center, image distortion can be large. For example, straight lines may appear curved. In a see-through HMD with narrow field-of-view displays, the optical combiners add virtually no distortion, so the user's view of the real world is not warped. However, the

optics used to focus and magnify the graphic images from the display monitors can introduce distortion. This mapping of distorted virtual images on top of an undistorted view of the real world causes static registration errors.

Mechanical misalignments are discrepancies between the model or specification of the HMD and the actual physical properties of the real HMD. For example, the combiners, optics, and monitors may not be at the expected distances or orientations with respect to each other. If the frame is not sufficiently rigid, the various component parts may change their relative positions as the user moves around, causing errors. Mechanical misalignments can cause subtle changes in the position and orientation of the projected virtual images that are difficult to compensate.

Errors in the reported outputs from the head tracking system are the most serious type of static registration errors. As noted in Chapter 1, Augmented Reality systems require highly accurate trackers. Almost all commercially-available trackers lack the required accuracy, and no tracker currently exists that provides high accuracy at long ranges in real time. For example, commonly-used magnetic trackers give distorted readings at long ranges because any metal in the environment warps the magnetic fields. Today, scene generators and see-through HMDs are available with sufficient performance to support Augmented Reality applications, but the necessary tracking and sensing technologies are not available. More research needs to be done to develop such technologies.

Incorrect viewing parameters are the last major source of static registration errors. Viewing parameters specify how to convert the reported head locations into viewing matrices used by the scene generator to draw the graphic images. These parameters include:

- Center of projection and viewport dimensions
- Offset, both in translation and orientation, between the location of the head tracker and the user's eyes
- Field of view

Incorrect viewing parameters cause systematic static errors. Take the example of the vertical translation offsets between the head tracker and the

user's eyes. If these offsets are too small, all the virtual objects will appear lower than they should.

3.2 Previous work

Registration of real and virtual objects is not limited to HMDs. Special-effects artists seamlessly integrate computer-generated 3-D objects with live actors in film and video. The difference lies in the amount of control available. With film, a director can carefully plan each shot, and artists can spend hours per frame, adjusting each by hand if necessary, to achieve perfect registration. HMDs are a far more difficult medium to work with. The Augmented Reality system cannot control the motions of the HMD wearer. The user looks where she wants, and the system must respond within tens of milliseconds.

Deering demonstrated an impressive registration of a virtual and real ruler with a head-tracked stereo system [Deering92]. In this system, a user wears stereo glasses and views images displayed on a high-resolution monitor. Monitor-based registration has also been shown for mechanical [Drascic93] [Oyama93] and medical applications [Taubes94]. The registration problem is significantly easier in head-tracked stereo systems than in HMD-based systems, because the graphic images do not change nearly as much in the former for the same amount of head rotation [Cruz-Neira93]. The reason is that the images displayed in a head-tracked stereo system are primarily determined by the eye positions, rather than the head orientation.

An extensive body of literature exists in the robotics and photogrammetry communities on camera calibration techniques; see the references in [Lenz88] for a start. Such techniques compute a camera's viewing parameters by taking several pictures of an object of fixed and sometimes unknown geometry. These pictures must be taken from different locations. Matching points in the 2-D images with corresponding 3-D points on the object sets up mathematical constraints. With enough pictures, these constraints determine the viewing parameters and the 3-D location of the calibration object. Alternately, they can serve to drive an optimization routine that will search for the best set of viewing parameters that fits the collected

data. These techniques should be directly applicable to video see-through HMDs. However, it is not clear how to directly apply these techniques to an *optical* see-through HMD, where no camera exists. In a typical camera calibration routine, each image may contain a dozen or so points that must be identified and their 2-D coordinates extracted. I judged that asking a user to perform such a task to be unreasonably difficult. She would have to keep her head still while simultaneously identifying the 2-D coordinates of several points in the real world, repeating this from many different viewpoints.

At least four different people here at UNC Chapel Hill have attempted registration with see-through HMDs in a non-systematic fashion. Such approaches proceed as follows: place a real object in the environment and attempt to register a virtual object with that real object. While wearing the HMD, stand at one viewpoint or a few selected viewpoints and manually adjust the location of the virtual object and the other viewing parameters until the registration "looks right." These approaches require a skilled user and generally do not achieve robust results. Achieving good registration from one single viewpoint is much easier than registration from a wide variety of viewpoints using a single set of parameters. Usually what happens is satisfactory registration at one viewpoint, but when the user walks to a significantly different viewpoint, the registration is inaccurate because of incorrect viewing parameters or tracker distortions. This means many different sets of parameters must be used, which is a less than satisfactory solution.

The only two published works describing static calibration methods come from Boeing. The printer maintenance application from Columbia [Feiner93] and UNC's ultrasound application [Bajura92] do not describe the calibration methods used. The first Boeing reference [Caudell92] describes a calibration rig and a user viewing task that conceptually should measure most of the viewing parameters. This paper does not an attempt an evaluation of how well the method works in practice. The other Boeing reference [Janin93] describes a method used with an optical see-through HMD and a magnetic tracker: the Ascension Big Bird. In a personal conversation in August 1994, Janin reported errors of up to 1.5 inches in the tracker outputs at long ranges. Because he does not entirely trust his tracker, he spreads out the registration

errors evenly across the entire tracker range, thus minimizing the maximum errors at the cost of raising typical errors. His calibration approach is based on camera calibration techniques, asking a user to identify the 2-D location of a single real point from about 200 different viewpoints. The results are sent into an optimizer that searches for the parameters that minimize the maximum error.

The best static registration in a see-through HMD claimed by previous work, in the text and pictures of [Bajura92] [Feiner93] [Janin93], is ±0.5 inches, or about ±13 mm.

The initial inspiration for my boresight operation came from methods used to align helmet-mounted sights used on helicopter gunships.

3.3 Basic approach

My goals and my choice of technology guided the design of calibration techniques for static registration. Although the main contribution of this dissertation comes from the reduction of dynamic errors, good static registration is helpful for the evaluation of dynamic errors, as described in Section 1.5. Therefore, the goal is to achieve static registration that keeps the virtual and real aligned closely enough that the user believes the two to be linked. I judged this should ideally be within ±2-3 mm, based on viewing simulations. Also, my choice of an optical see-through HMD meant that each user must potentially run several calibration routines. Video see-through HMDs may require fewer calibration steps. The offset between the camera and tracker does not change from user to user. The only parameter that requires adjustment for each user is the interpupillary distance (IPD), which is easily measured by an optometrist's tool. But in optical see-through, several viewing parameters change with different users. Therefore, I set the following guidelines:

1) The calibration tasks must be simple. Each user must be able to perform the calibration tasks. Since most users will not be familiar with the details of the system and how the parameters work, the

calibration tasks must be easily explained to and performed by any user, not just an expert.

2) The static registration must be good, but it does not have to be perfect. To effectively demonstrate the reduction of dynamic errors, I want the following situation: With no prediction, the virtual axes "swim around" the corner of the crate. With prediction turned on, the virtual axes stick closely to the real corner, maintaining the illusion that the two are linked. Based on simulations, this definitely occurs when errors are kept under 2-3 mm, and it often occurs with somewhat larger errors, such as half a centimeter. Since a small amount of static error is tolerable, I did not feel it necessary to tackle all sources of static registration errors, just the biggest ones.

3) The calibration routines should trust the tracker and ask the user to perform view-based tasks that depend on geometric constraints to determine the parameters. While every tracking system has errors, I believed that a tracker available at UNC was accurate enough to allow such an approach. Furthermore, compensating for tracker errors is most appropriately done by improving the tracker itself, not by having the applications attempt to handle distorted data. The emphasis on view-based tasks comes from the belief that those can be simple to perform, satisfying criterion #1, and accurate enough to satisfy criterion #2. In view-based tasks, users attempt to position certain virtual lines to match corresponding real lines. If the users can perform such tasks accurately, the resulting registration might be good enough for my purposes. A simple test of moving my head so that my right eye looked straight down an edge of a bookcase convinced me that such tasks might be feasible.

I chose not to attempt a stereo system because of the color mismatch between the two displays. The monitors are Sony Watchmans: tiny TV sets aimed at the consumer market. When sending color stereo images into the two displays, I found the color mismatches to be so bad that I could not fuse the stereo images. No matter how I adjusted the color and tint settings in the two monitors, the mismatches prevented me from fusing the images.

Therefore, I only use the right eye of the HMD, treating it as a monocular device.

I chose not to compensate for the optical distortions in the HMD. If the distortions are known, it is possible to predistort the graphic images to remove the effects of distortion [Robinett92c]. However, this requires considerable computational resources to run in real time. The Pixel-Planes 5 implementation uses eight Intel i860 processors running in parallel and adds latency to the overall system. The extra complexity and lag were major factors in my decision not to do compensation. Also, the optical distortions within the HMD are relatively small, becoming obvious only when the gaze direction is significantly off the central optical axis. Looking 20 degrees off-axis can bend a virtual object at arm's length about half a centimeter from where it should be. Since I did not need perfect registration, I decided that it would be simplest to attempt good registration only in the center of the field of view. If the optical errors proved large enough to prevent me from evaluating the improvement in dynamic errors, I could implement optical distortion compensation later.

I chose to focus on the remaining two sources of static errors: the tracking system and the viewing parameters. For tracking, I made use of a custom optoelectronic tracker built at UNC Chapel Hill. To determine the viewing parameters, I developed several calibration techniques that ask the user to perform certain tasks. Section 3.4 briefly describes the tracker, and Section 3.5 explains the calibration techniques.

3.4 Optoelectronic tracker

UNC's custom optoelectronic head-tracker is scalable to any room size, without the gross distortions seen in commonly-used magnetic-based trackers. This accuracy at long ranges makes it attractive for use with Augmented Reality systems. Figure 3.1 shows a conceptual drawing of the system. Head-mounted optical sensors view beacons mounted in ceiling panels above the user's head. The beacon locations are known, as is the geometry of the sensors on the HMD. Theoretically, the sensors must see at least three beacons to provide enough information to compute the unknown

position and orientation of the HMD. In practice, viewing about a dozen beacons with three or more sensors yields the best results. This configuration of sources and sensors provides good orientation sensitivity while also covering any room-sized volume. To increase the range, simply add more panels to the ceiling. Figure 3.2 shows the actual system, covering a 10' by 12' area. Figures 3.3 and 3.4 show closeups of the HMD, outfitted with four optical sensors. Figure 3.5 was taken with a camera sensitive to infrared light, so it shows a pattern of LEDs lit in the ceiling as the user walks underneath it. For details, see [Azuma91] [Gottschalk93] [Ward92]. Some additional details will be discussed in Chapter 5.

Figure 3.1: Conceptual diagram of optical tracking system. Diagram drawn by Mark Ward.

Figure 3.2: The actual system in operation

Figure 3.3: Side view of HMD equipped with four optical sensors

Figure 3.4: A pair of views of HMD equipped with four optical sensors

Figure 3.5: Lit LEDs in the ceiling

The optoelectronic tracker's performance makes it suitable for Augmented Reality applications. Update rates are typically 60-80 Hz, with typical lags of 15-30 ms. These numbers depend upon the number of beacons viewed and how long it takes to light those beacons and read the sensors. Thus, the update rate and lag vary with time. Absolute accuracy has never been measured, but the resolution is well under 0.2 degrees and 2 mm under good tracking conditions. Relative accuracy has been measured in certain test cases. For example, the HMD was mounted rigidly to a mechanical turntable. The turntable was accurate to one minute of arc. By rotating the turntable a certain amount, then comparing that against what the optoelectronic tracker reported, I found that the relative accuracy for that task was within 0.2 degrees. For a similar translation task that slid the HMD along a horizontal bar with ruled markers, I found the relative accuracy to be under 2 mm. Because of these results, the optoelectronic tracker seemed accurate enough for Augmented Reality applications. However, Section 3.6 will describe some distortions that were discovered during attempts to use this tracker with Augmented Reality applications.

For the AR system, I used the optical sensors from the HMD shown in Figures 3.3 and 3.4, but I remounted them on a specially-designed platform that holds the four sensors in a symmetrical pattern, looking upward. Because this platform looks vaguely like a hat, it is called the "4-hat." Figure 3.6 shows the 4-hat by itself, and Figure 3.7 shows the 4-hat with optical sensors attached. The 4-hat is rigidly mounted to the rear of the optical see-through HMD, providing double duty as tracking mechanism and counterweight. Figures 3.8 and 3.9 show the 4-hat attached to the optical see-through HMD.

Figure 3.6: 4-hat platform for mounting optical sensors

Figure 3.7: 4-hat equipped with four optical sensors

Figure 3.8: Front view of optical see-through HMD with 4-hat

Figure 3.9: Rear view of optical see-through HMD with 4-hat

3.5 Calibration techniques

Static calibration requires knowledge of the position and orientation of the objects in the real world, the field-of-view, the center of the field-of-view, and the position and orientation offsets from the tracker to the user's right eye. For this system, the only real world object of interest is the wooden crate. These values are measured in the following order:

1) Position and orientation of the crate
2) Center of the field-of-view
3) Orientation offset between the tracker and the right eye
4) Position offset between the tracker and the right eye
5) Field-of-view

The next four subsections describe these steps in detail.

3.5.1 Crate location

To measure the position and orientation of the crate, I attach a probe to the 4-hat, as shown in Figure 3.10. The optoelectronic tracker returns the position and orientation of the 4-hat at the origin shown in Figure 3.11. The probe, built by Kurtis Keller, was built so that the probe tip lies exactly along the *Z*-axis of the 4-hat coordinate system. Since the length of the probe is known, I can compute the location of the probe tip through a simple transformation, given the position and orientation of the 4-hat. Therefore, I can use the 4-hat and the probe to measure the locations of arbitrary points in space.

Figure 3.10: 4-hat with probe attached

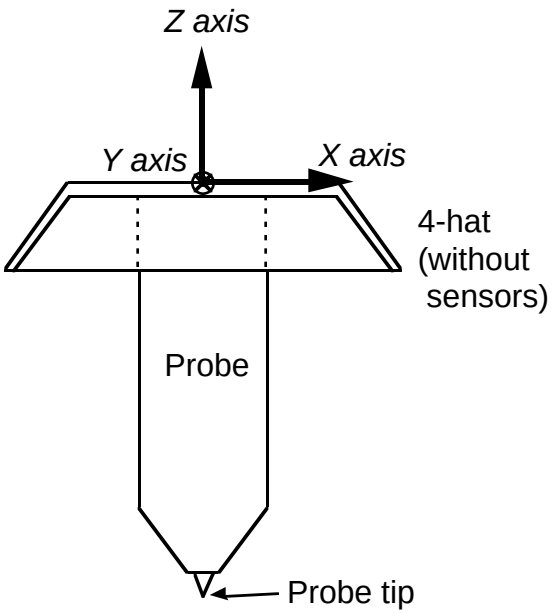


Figure 3.11: Diagram of 4-hat coordinate system and probe

With this tool, I measure eight points on the top of the crate. These points, labeled **A** through **H**, are shown in Figure 3.12. The goal is to determine the locations of the two edges that points **A**-**H** lie on. Therefore, these points must lie along the specified edges and be spaced well apart. It is not important that the user measure exactly the same set of points across different digitization runs. To measure each point, the user holds the probe still and presses one button, waits 1-2 seconds, then presses another button. The digitization program averages the measurements taken during that time interval to reduce the noise in the measured probe tip location.

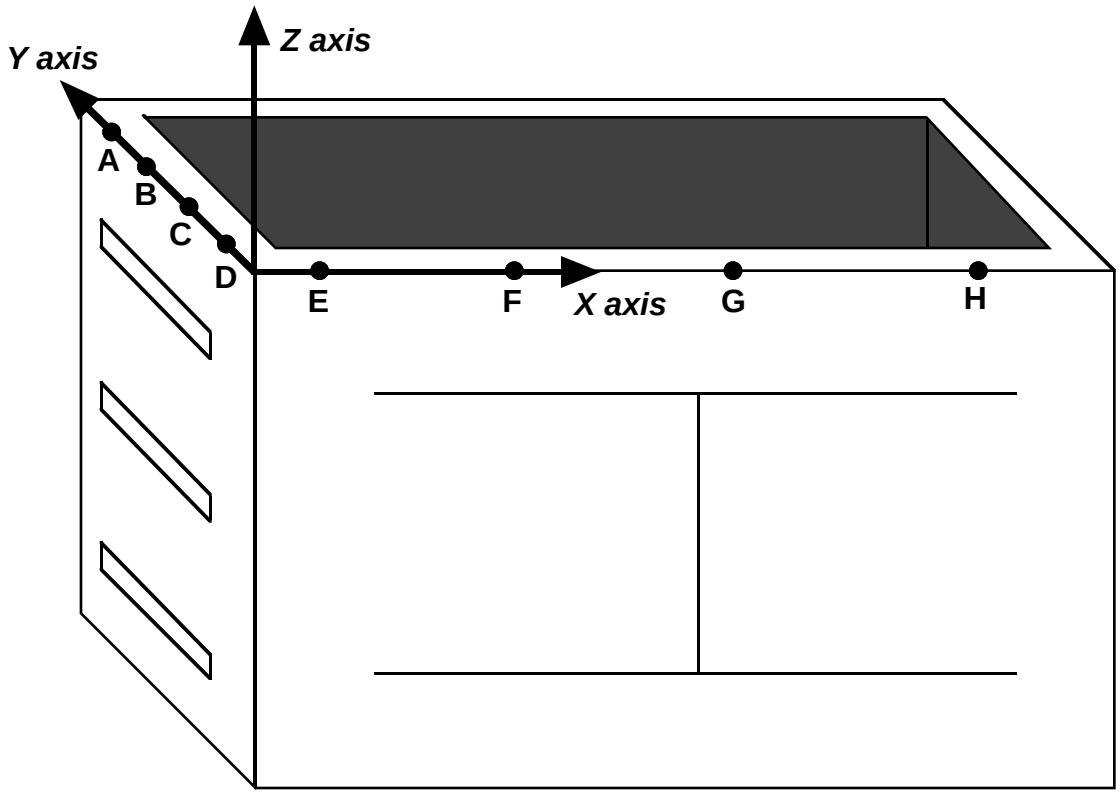


Figure 3.12: Digitized points on two edges of the crate

The locations of the eight points are then sent into an optimization program. It finds the two 3-D lines that minimize the least-squares distance to the points **A**-**D** and **E**-**H**, respectively. Then I force both lines to intersect and be orthogonal with each other. For each line, I find the point that is closest to the other line. The average of these two points is declared to be the origin of both lines. Then I adjust the vectors of both lines to force them to be orthogonal. Normalizing the two vectors and taking their dot product yields their angular difference. By taking their cross product, I find the normal to the plane formed by those two lines. This normal, along with the origin of both lines, defines the line about which I will rotate both lines to force them to be orthogonal. Each line is rotated by half of the difference between 90 degrees and their angular difference. Once this is done, I have two intersecting and orthogonal lines. The cross product of the vectors yields the third axis, *Z*.

The three mutually orthogonal vectors and the origin define a local coordinate system for the wooden crate. Thus, the World→Crate transformation is now known.

This method assumes that the edges of the crate are mutually orthogonal. To check this, I measured the pairs of diagonals on each of the three faces adjoining the corner of the crate where the three vectors meet. This measurement was done with a ruler. The largest difference between the pairs was about 1.5 mm, on the 18.75" by 20.5" face. This corresponds to around 0.13 degrees of error.

3.5.2 Crosshair

The center of the frame buffer does not necessarily coincide with the center of the virtual image seen by the right eye. Therefore, off-center projections are required to render the graphics images [Foley90], and for that I need to know which pixel in the frame buffer corresponds to the center of the field-of-view.

The optics limit the extent of the virtual images. Beyond some angular deviation away from the center of the field-of-view, it is not possible to see any of the virtual image due to the limited range of coverage provided by the optics. Assume that the frame buffer covers the entire range visible through the optics. Then it is possible to measure the center of the field-of-view by the following procedure.

In the frame buffer, draw a non-head-tracked 2-D crosshair, as shown in Figures 3.13, 3.14 and 3.15. The crosshair is a purely 2-D object, unaffected by head position or orientation. The crosshair is specified by four numbers: the (*X, Y*) coordinates of the center and the X_{radius} and Y_{radius}, in pixels. A program allows the user to modify each of the four numbers. First, the user adjusts the *X* coordinate of the center and the X_{radius} until the leftmost and rightmost lines are equally spaced from the visible edges of the field of view. This is tested by increasing the X_{radius}; both lines should disappear simultaneously or the *X* coordinate of the center is incorrect. Similarly, the user finds the *Y* coordinate of the center by adjusting it along with the Y_{radius} and watching the top and bottom lines of the crosshair.

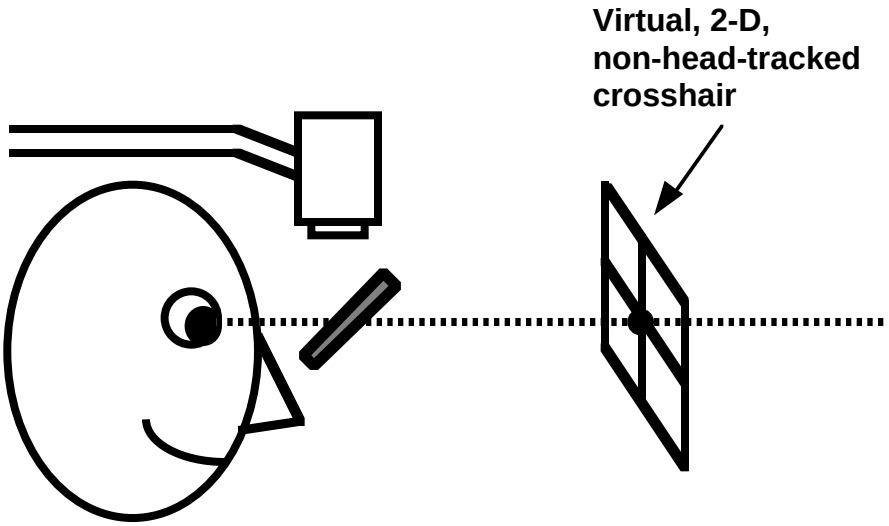


Figure 3.13: Conceptual view of the virtual crosshair

Figure 3.14: Actual view of crosshair, as seen inside the HMD

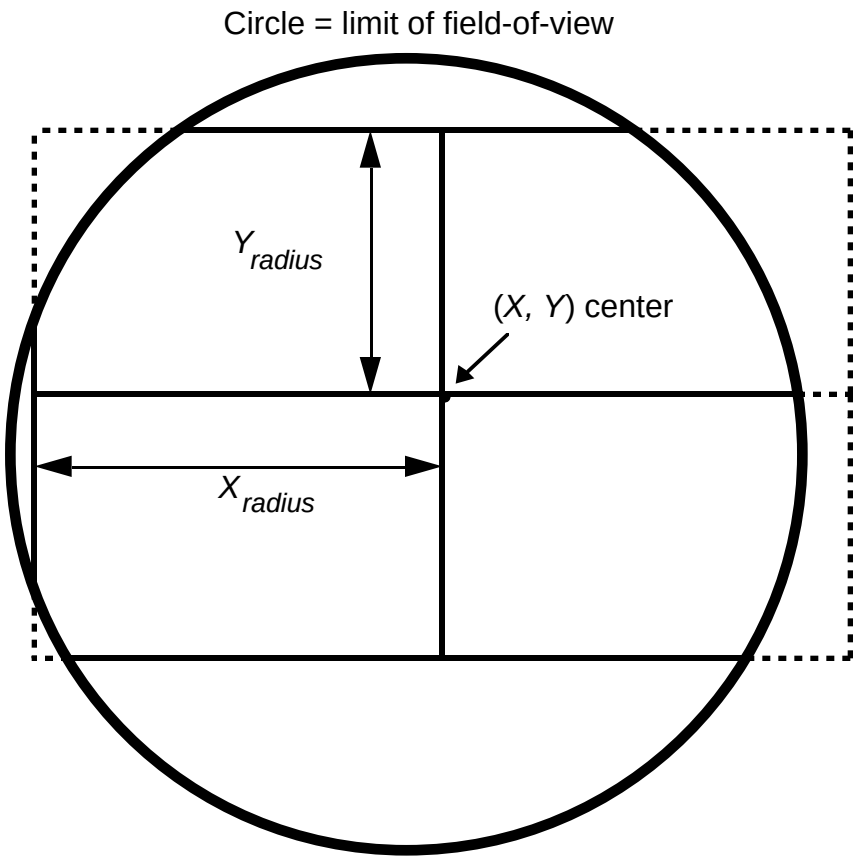


Figure 3.15: Calibration of center of the field-of-view

In practice, the frame buffer is 640 by 512 pixels, generating an NTSC signal to the monitor. I measured the center of the field-of-view at (330, 255), which is about 10 pixels away from the center of the frame buffer.

3.5.3 Boresight

The boresight operation determines both the orientation and position offsets from the head tracker to the right eye. These offsets determine the transformation to apply to the tracker readings to get the viewpoint at the user's right eye. Figure 3.16 shows a conceptual drawing of the boresight operation. The user looks straight down the left edge of the crate with her right eye. A 0.25" diameter pipe sticking out from the edge helps the user tell when she is properly aligned with the left edge. Simultaneously, she also centers the virtual crosshair with the corner of the crate and lines up the middle horizontal and vertical lines of the crosshair with corresponding edges of the wooden crate. Figure 3.17 is an external view of this operation,

showing the 0.25" diameter pipe sticking out from the corner. Figure 3.18 shows what the user sees as she performs the boresight.

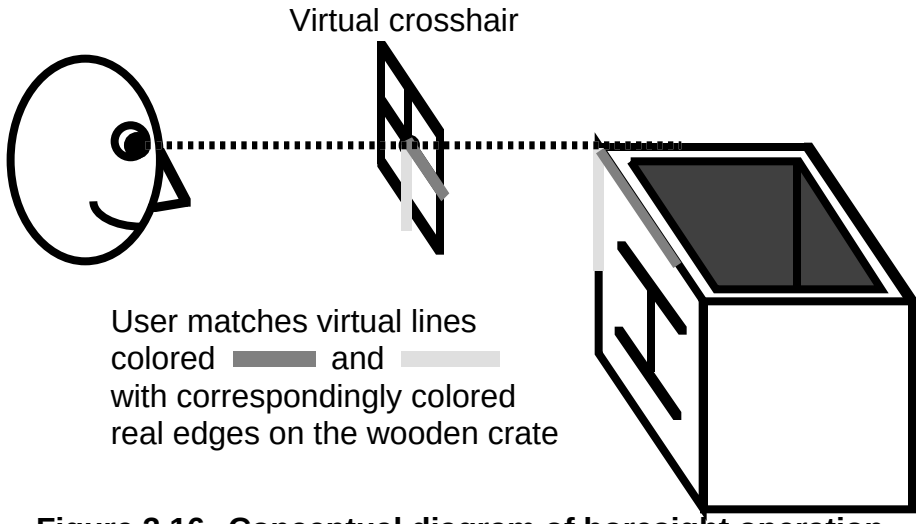


Figure 3.16: Conceptual diagram of boresight operation

Figure 3.17: External view of boresight

Figure 3.18: Internal view of boresight

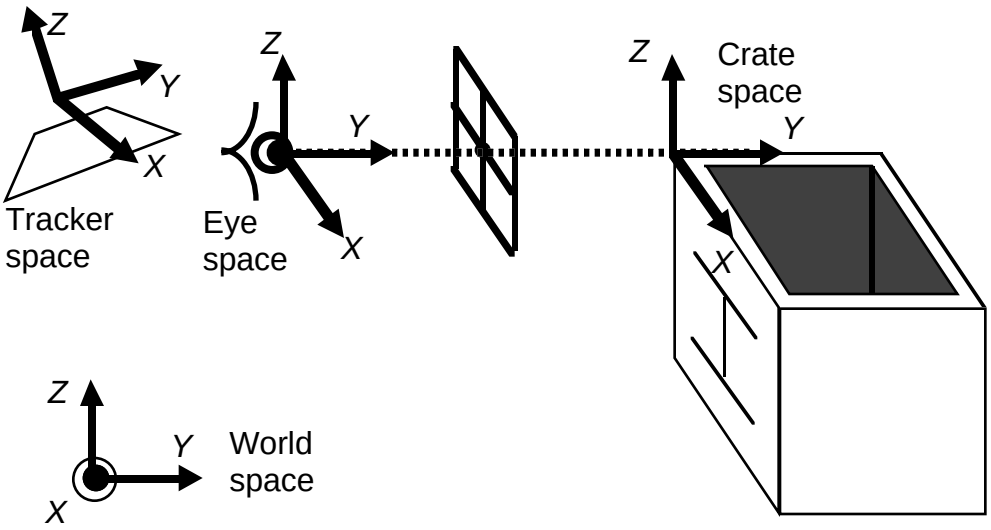


Figure 3.19: Coordinate systems in boresight

The boresight determines the orientation offset because it locks the user's Eye-space orientation to the Crate-space orientation. Figure 3.19 shows the various coordinate systems involved. If the boresight is performed correctly, the Eye and Crate coordinate systems share the same orientation. That is, it establishes:

$$\mathbf{Q_{wc}} = \mathbf{Q_{we}}$$

where $\mathbf{Q_{wc}}$ is defined as the quaternion that rotates points and vectors from Crate space to World space, and $\mathbf{Q_{we}}$ rotates points and vectors from Eye space to World space [Robinett92b]. Quaternions are a way of representing orientation and rotation operations; for details, please see [Chou92] [Shoemake89]. Also note from the List of Symbols at the beginning of this dissertation that the symbol '•' represents quaternion multiplication. Then the desired orientation offset $\mathbf{Q_{et}}$ is computed as follows:

$$\mathbf{Q_{te}} = \mathbf{Q_{tw}} \bullet \mathbf{Q_{we}} \qquad \text{[by identity]}$$

$$\mathbf{Q_{te}} = (\mathbf{Q_{wt}})^{-1} \bullet \mathbf{Q_{wc}} \qquad \text{[substitution for } \mathbf{Q_{we}} \text{]}$$

$$\mathbf{Q_{et}} = (\mathbf{Q_{te}})^{-1}$$

$\mathbf{Q_{wt}}$ is what the head tracker returns, and $\mathbf{Q_{wc}}$ was determined in the Crate location measurement step in Section 3.5.1, so I have enough information to compute $\mathbf{Q_{et}}$.

All that remains is to determine the Eye→Tracker translation offset. As defined so far, the boresight operation only establishes that the user's right eye lies somewhere along the ray extending from the left edge of the wooden crate, as shown in Figure 3.16. The boresight does not measure the distance between the eye and the corner of the frame. For that, I have to add another constraint. Along with the previously mentioned tasks in the boresight operation, the user must now also line up two nails mounted top of the wooden crate, as shown in Figure 3.20. A red LED is placed on the rear nail to help the user determine when both nails are lined up. When the front nail covers the LED on the rear nail, the user knows that they are aligned. These two nails are visible in the upper right part of Figure 3.18, showing what they look like when they are not aligned. Lining up the nails forces the eye to be at a specific distance along the ray extending from the left edge, fully determining the position of the eye at point **P**.

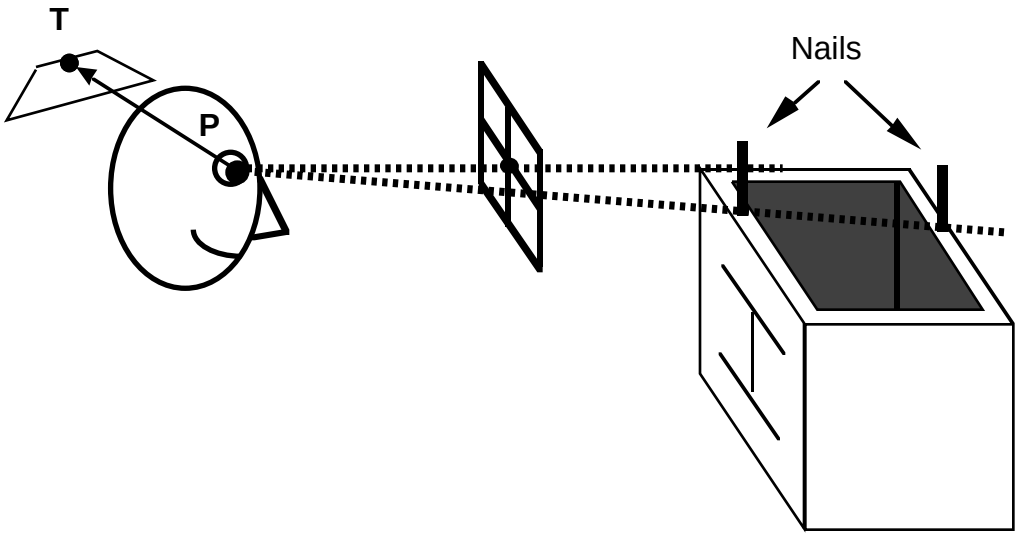


Figure 3.20: Nails specify distance along ray

With the known location of the nails on the wooden crate, the location of point **P** is known in Crate space. This location can be expressed in World space by using the measurements determined in Section 3.5.1. Point **T**, the origin of the tracker, is also known in World space because the head tracker provides that information. The desired translation offset expresses how to move points and vectors from Tracker space to Eye space, or equivalently how to translate the Eye coordinate system origin to coincide with the origin of the Tracker coordinate system, where the translation takes place in Eye space. Therefore, all that remains is to take the vector $\vec{\mathbf{PT}}$, currently

expressed in World space, and rotate it to find its value in Eye space. That is, I need the quaternion $\mathbf{Q_{ew}}$, which is computed by:

$$\mathbf{Q_{ew}} = \mathbf{Q_{et}} \bullet \left(\mathbf{Q_{wt}}\right)^{-1}$$

where $\mathbf{Q_{et}}$ was previously computed, and $\mathbf{Q_{wt}}$ is what the head tracker provides.

In practice, the user performs two separate boresights. The first is performed at a moderate distance from the wooden crate (about 2-3 feet), for greater orientation sensitivity. Since this step only determines the orientation offset, the user does not line up the nails. The second boresight is for the position offset, requiring the user to line up both nails. This is performed at close range (under a foot) for greater translation sensitivity.

3.5.4 Field-of-view

The final viewing parameter to measure is the field-of-view (FOV). It suffices to measure the FOV along the vertical direction in screen space, since scaling that by the frame buffer's aspect ratio yields the horizontal FOV, assuming the display optics are not anamorphic. Figure 3.21 shows a conceptual diagram of the operation. Three lines are drawn on the front surface of the wooden crate. The locations of these lines in Crate space are known. The crosshair's Y_{radius} is set to 125 pixels so that the top and bottom lines of the crosshair are easily visible. The user stands in front of the crate and lines up the top and bottom lines of the virtual crosshair with corresponding real lines drawn on the wooden crate. Note that the distance between the right eye and the virtual crosshair is not known, but that will not be required.

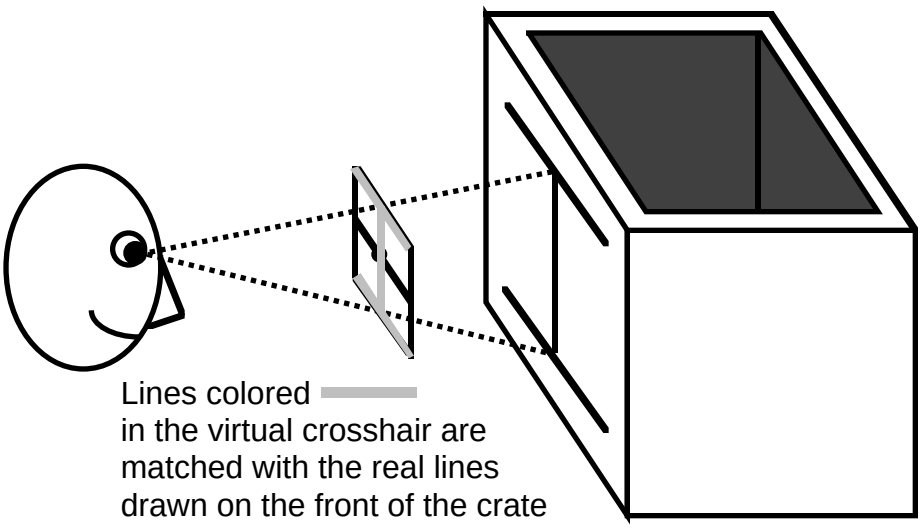


Figure 3.21: Conceptual diagram of FOV calibration

This operation forces the Eye-space *X* axis to be parallel to the Crate-space *X* axis, because otherwise the pair of virtual lines will not appear to be parallel with the pair of real lines. Therefore, I can reduce this to a 2-D situation by intersecting each line with the *X*=0 plane in Eye space. This 2-D situation is shown in Figure 3.22.

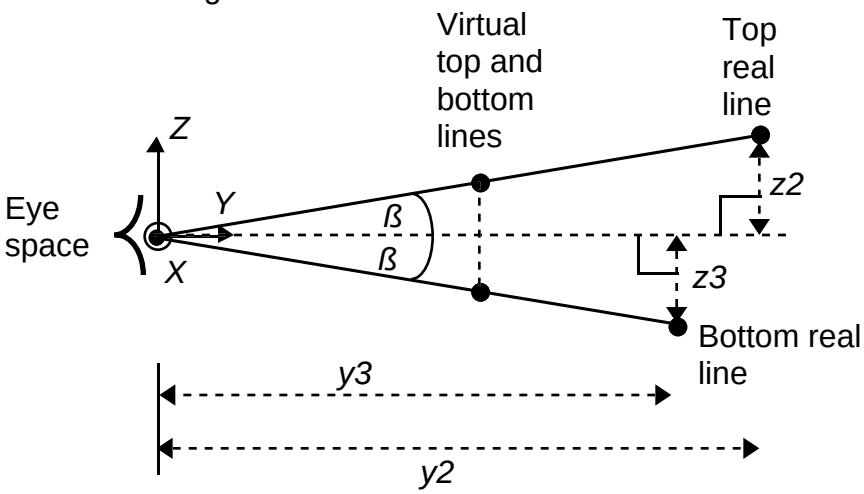


Figure 3.22: 2-D side view of FOV calibration, in the *X=0* plane

The locations of the top and bottom real lines drawn on the wooden crate are known, in Crate space. From previously determined transformations, I can convert these lines from Crate space to World space, and then to Eye space. Intersecting each line with the plane *X*=0 in Eye space yields the two points representing the top and bottom real lines, as shown in Figure 3.22. These are specified by the numbers *y2, y3, z2* and *z3*. Note that as drawn, *y2, y3* and *z2* are positive, but *z3* is negative. Also note

that generally speaking, *y2* does not equal *y3*, nor does the magnitude of *z2* equal the magnitude of *z3,* because the Eye-space *Y* axis is generally not parallel to the Crate-space *Y* axis. However, the crosshair is parallel to the Eye-space *Z* axis and bisects the viewing angle, due to the definition of the crosshair in Section 3.5.2.

The angle *ß* is computed by either of the following:

$$ß = \tan^{-1}\left(\frac{z2}{y2}\right)$$

$$ß = -\tan^{-1}\left(\frac{z3}{y3}\right)$$

However, two times *ß* is not the entire vertical FOV. Recall that the crosshair's Y_{radius} was set to 125 pixels, so the vertical extent covered by the crosshair is only 250 pixels. The frame buffer has 512 pixels in the vertical direction, and the normal Y_{radius} that covers the entire visible FOV is much larger than 125 pixels. Figure 3.23 shows the situation for computing the entire vertical FOV.

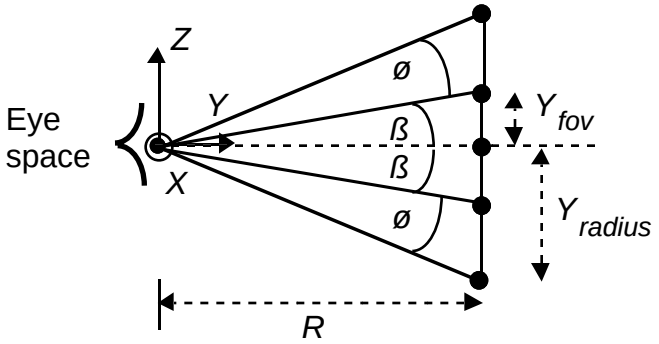


Figure 3.23: Computing the total FOV

Y_{fov} is set to 125 pixels. Y_{radius} is set to whatever was determined to cover the visible FOV in Section 3.5.2, assuming the viewport was set to that vertical extent. Otherwise, the viewport was left at the full frame buffer vertical extent of 512 pixels, so Y_{radius} is set to 512 pixels. Then the goal is to compute *ß* + *ø*, which is half of the total FOV.

$$\tan(ß) = \frac{Y_{fov}}{R}$$

$$\tan(ß + ø) = \frac{Y_{radius}}{R}$$

$$R\tan(ß) = Y_{fov}$$

$$R\tan(ß + ø) = Y_{radius}$$

$$\frac{\tan(ß + ø)}{\tan(ß)} = \frac{Y_{radius}}{Y_{fov}}$$

$$ß + ø = \tan^{-1}\left(\frac{Y_{radius}}{Y_{fov}}\tan(ß)\right)$$

$$\text{FOV} = 2(ß + ø)$$

3.6 Evaluation

The calibration steps provide enough information to render images of the virtual objects. The location of the wooden crate specifies where the three virtual axes should be in World space. The position and orientation offsets determine the transformation that yields the viewpoint at the right eye, given tracker data. The FOV has been measured. The only unusual aspect in rendering the images is the need for an off-center projection, as specified by the measurement of the center of the FOV.

To demonstrate static registration from a variety of viewpoints, I walked 270 degrees around the corner of the wooden frame where the three virtual axes intersect and recorded a videotape of what the user saw. This recording required putting a small video camera inside a bust of a human head, where the right eye is supposed to be (Figure 3.24). Then the optical see-through HMD is strapped to this bust and carried around (Figure 3.25). The incandescent lights used during filming added noise to the outputs of the optoelectronic tracker, because of the increased infrared background light. This makes the virtual axes appear to "jump around" more than they normally would, but it does not change the overall static registration.

Figure 3.24: Bust with hole in right eye and video camera

Figure 3.25: Carrying the bust with see-through HMD attached

Figures 3.27 and 3.28 show nine pictures taken from the videotape of the walkaround. The approximate viewpoints where each picture was taken are shown in Figure 3.26.

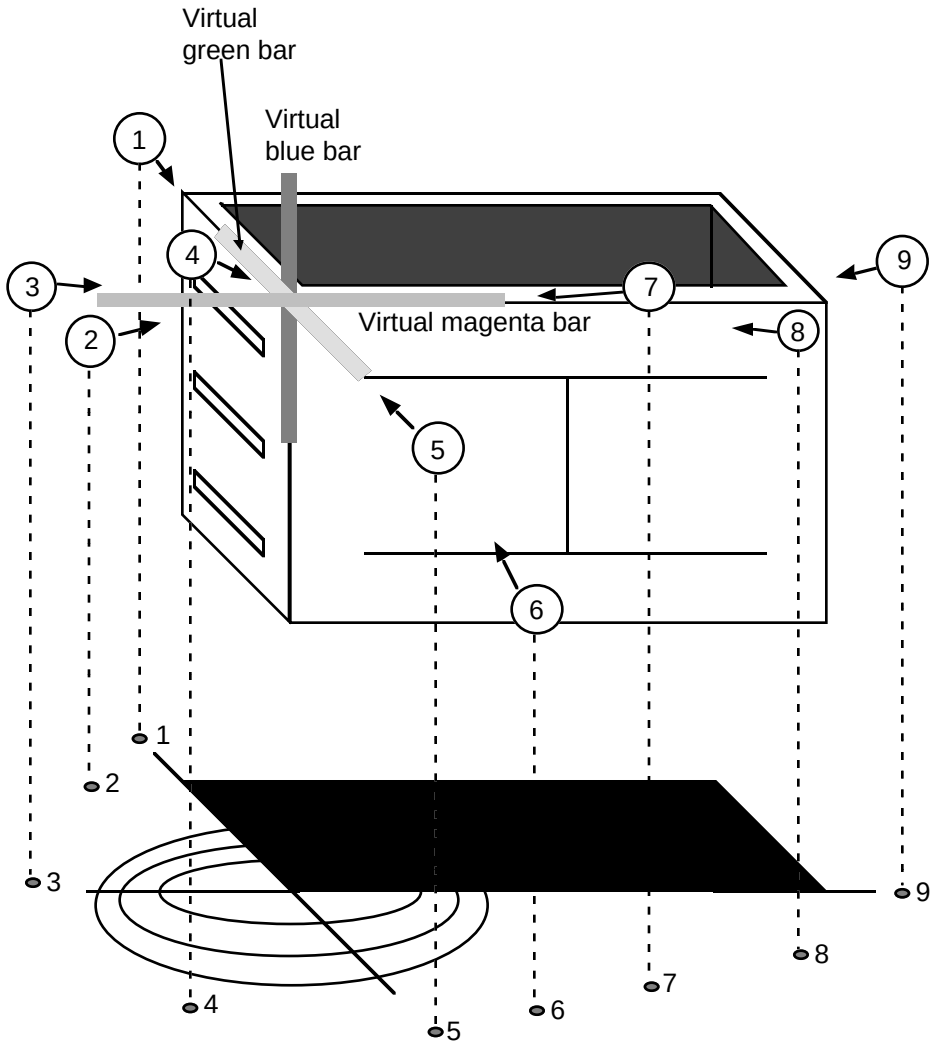


Figure 3.26: Static registration viewpoints during walkaround

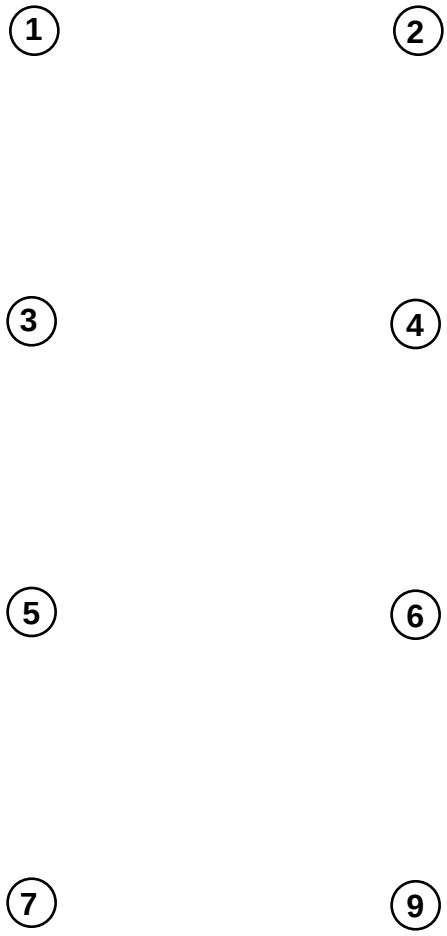


Figure 3.27: Views from static registration viewpoints #1-7 and #9

Figure 3.28: View from static registration viewpoint #8

Note that the corners and edges of the frame usually stay within the width of the thick virtual axes, which act as spatial error bars. The magenta and green bars are extruded rectangles with 5 mm by 5 mm cross-section. The blue bar has a 7 mm by 7 mm cross-section because that color was harder to see than the other two. Those dimensions put the registration within ±4 mm for the red and green bars and ±5 mm for the blue bar. These are not strict boundaries, however. Figure 3.28 shows one viewpoint where the blue bar does not cover its corresponding vertical edge, so the static error is higher than 5 mm there.

The main improvements demonstrated in this work over the previous work discussed in Section 3.2 is the reduction of typical static registration errors to ±5 mm, and the demonstration of this registration from a wide variety of viewpoints, not just one or two. Cross-system comparisons are tricky, however. With a different see-through HMD, tracking system, and calibration techniques, one cannot conclude that the calibration techniques alone were responsible for the improvements.

For example, the calibration techniques previously described are applicable to any tracking system and any see-through HMD where the frame buffer covers the entire FOV. However, the registration will only be as good as the tracker. The results achieved with the custom-built optoelectronic tracker will probably not be duplicated with a different tracker that may have

greater distortions. The calibration techniques assume that the tracker is accurate and do not take tracker errors into account. Therefore, I do not recommend their use with trackers that have large distortions. In that situation, Janin's approach may be more suitable.

The calibration steps assume that the virtual crosshair is projected correctly. That is, when the user looks at the center of the crosshair, the crosshair should be orthogonal to the *XZ* plane in Eye space, with the virtual vertical lines parallel to the *Z* axis and the virtual horizontal lines parallel to the *X* axis. In practice, this may not be the case, due to mechanical misalignments in the HMD. While my calibration steps can compensate somewhat for roll (rotation about the *Y* axis in Eye space), they will not detect other rotation errors caused by mechanical misalignments.

I did not compensate for the optical distortions in the HMD, for the reasons described in Section 3.3, and that may affect the FOV calibration. Optical distortion is not a problem for most of the calibration steps, because most of them involve lining things up at the center of the FOV. The distortion is a function of the radial distance away from the center of the FOV, so at the center itself the distortion should be practically nonexistent. If the distortion is purely a radial function, then that should not hurt the determination of the center of the FOV, since the distortion would affect the edges equally. There is virtually no distortion of the real world as seen through the combiners, so optical distortion does not affect aligning the two nails in the boresight operation. The only calibration step that could be significantly hurt by optical distortion is the measurement of the FOV, because that involves using virtual lines at some distance away from the center of the FOV. To reduce this effect, the user should try to match the top and bottom lines at the points closest to the middle of his FOV, at the points intersected by the middle vertical line in the crosshair.

The optoelectronic tracker is not as accurate as it needs to be for AR registration, requiring compensation or correction. Small distortions exist that were not discovered until several people at UNC tried using this tracker for AR applications. Andrei State discovered one by putting the 4-hat on a mechanical rig that rotates it in place about its origin. Since the origin does not change as the 4-hat rotates, the reported position should remain constant.

Unfortunately, that is not what happens. As the 4-hat rotates, the *Z* position changes by less than one or two millimeters, but the *X* and *Y* coordinates slowly trace out an ellipse where the largest diameter is about two centimeters. One such ellipse is shown in Figure 3.29. Such errors would easily be visible in the registration task.

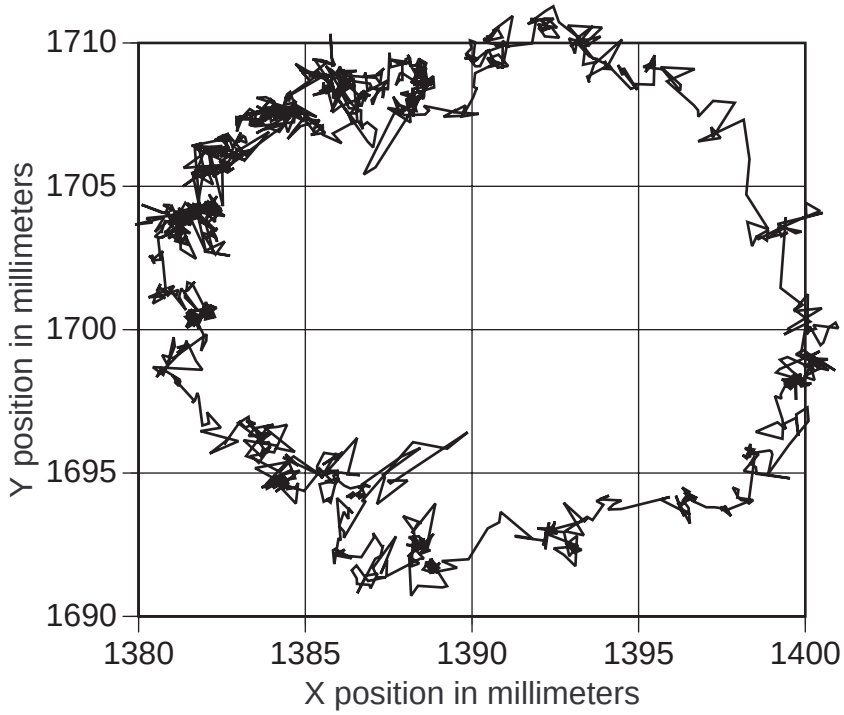


Figure 3.29: Elliptical path traced out in *XY* plane as 4-hat rotates 360 degrees about its origin

Since this error is very systematic and tightly coupled to the head orientation, I was able to compensate for it. I measured the ellipse at the approximate height of the top of the wooden crate. The calibration tasks all occur at basically the same orientation, where the user looks toward the front face of the crate. Therefore, I defined the position reported at this base orientation to be "truth" and used the numbers from the ellipse to compute offsets to this position. The result is a function, based on the yaw orientation, that generates offsets added to the reported tracker *X* and *Y* positions to compensate for the distortion.

Unfortunately, this elliptical error is not the only distortion that exists in the optoelectronic tracker. Many other subtle ones occur as the user moves around, and they are not well understood. The source of these distortions is probably inadequate calibration of the optical sensors and errors in the

location of the sensors on the 4-hat. This problem is currently under investigation. A significant fraction of the remaining static registration errors, such as the one seen in Figure 3.28, probably comes from these distortions.

Registration accuracy depends on how well the user can perform the calibration tasks. Users reported some difficulty in keeping their heads still during the boresight and the FOV measurement operations. Averaging measurements reduces the noise. Once the user presses a button to announce that the boresight or the FOV operation has been performed, I average the results computed from the 60 most recent tracker reports. To learn what variation remained, I asked three users to repeat the boresight and FOV operations five times. They moved their heads away in between each operation. The results are shown in Table 3.1.

	Quaternion offset standard deviation (in degrees)	Position offset standard deviation (in millimeters)	FOV standard deviation (in degrees)
User #1	0.27	2.6	0.12
User #2	0.18	1.1	0.17
User #3	0.52	10.9	0.03

Table 3.1: Variance in repeated boresight and FOV operations

The average standard deviations in computed orientation offsets, position offsets, and FOV were 0.32 degrees, 4.8 mm, and 0.1 degrees, respectively. Most of the position variation was along the *Y* axis in Eye space, the distance that the operation is least sensitive to. The variation in the orientation offset is too large. Errors in the orientation offset cause equivalent registration errors, and an error of 0.32 degrees is easily detectable. In practice, some users had to repeat the calibration steps more than once before achieving satisfactory registration, probably because of this variation. It may be the case that the boresight task is simply too difficult to perform consistently, because it involves aligning several things simultaneously. Or users may not be able to keep their heads sufficiently still, especially when burdened with a heavy HMD. This suggests that alternate approaches for measuring the viewing parameters deserve exploration.

The optoelectronic tracker is a line-of-sight system that loses tracking if the head-mounted optical sensors are not aimed at the ceiling panels. It also loses accuracy when few sensors are aimed at the ceiling, or few LEDs can be seen, or if the LEDs are seen at grazing angles or long distances. This means that I cannot pitch or roll the HMD steeply and retain accurate tracking. That is why the images in Figures 3.27 and 3.28 do not show the wooden crate from such viewing angles.

The HMD is not as rigidly attached to the user's head as it should be, but that has not turned out to be a major problem. The HMD is attached to a flexible support "web" that straps onto the user's head. This support web is similar to the one inside construction or baseball helmets. Since the web is flexible and attached to the HMD basically at only one point, the HMD can rotate and slide around with respect to the user's head as the user moves. Initially I feared that this non-constant relationship would ruin the registration, but in practice it seems to have little effect after calibration. Even when the HMD slides on the user's head, the HMD itself stays rigid, so the relationship between the right eye display and the head tracker remains constant. The user compensates for the sliding HMD by rotating her right eyeball. The net change in the Tracker→Eye transformation is small, causing little difference in apparent registration.

4. Dynamic registration

The largest source of registration error is the dynamic error caused by end-to-end system delays as the head rotates and translates. This chapter describes and evaluates a method of compensating for these delays: predicting future head locations with the aid of head-mounted inertial sensors. Inertial-based prediction reduces average dynamic errors by a factor of 5 to 10 over doing no prediction and a factor of 2 to 3 over non-inertial-based prediction. Without prediction, virtual objects "swim around" their real counterparts; with prediction, they stay close enough that users perceive them to stick together. A possible future direction for further improving accuracy is described.

4.1 Basic approach

The goal of this dissertation is to demonstrate that predicting future head locations is an effective way to reduce dynamic registration errors, as explained in Sections 1.4 and 1.5. In this section, I survey the prediction problem in general, describe some basic approaches, and explain which approach I took and why.

Prediction is like driving a car when the only available view is the rear view mirror. The driver can see what has happened in the past, but he has no direct view of the future. This is called a *causal* situation, where all past values, but none of the future, are known. To keep the car on the road, the driver must anticipate, or predict, where the road will be, based solely on the past observations and his knowledge of roads in general. The difficulty of this task depends on the shape of the road and how fast the car is going. If the road is straight and remains so, then the task is easy. If the road twists and turns rapidly, the task may be nearly impossible. Or the difficulty may lie

somewhere in between. Thus, the first question to ask about head-motion prediction is what type of "road" it is — trivial, intractable, or tractable?

Predicting head motions for HMD systems is an interesting and potentially tractable problem because of the motion characteristics and the required prediction intervals. Some motions are very well characterized and easy to accurately predict. For example, the dates and times of solar eclipses can be predicted centuries in advance. Other motions are so random that they are basically unpredictable except in a broad statistical sense, like predicting the location of one particle undergoing Brownian motion. Head motion lies somewhere in between these two extremes. It exhibits strong temporal and spatial coherence, but the motion is not so simple that one can easily find a model that completely characterizes it. Figures 4.1 through 4.8 show parts of two head motion segments. They were recorded from naive users running demonstration applications of our HMD systems, so they are called Demo1 and Demo2. The graphs show one position and one orientation trace from each sequence, with one closeup of each section. Each segment is about 150 seconds long, and each closeup is about 8 seconds long. In the first run, the maximum angular velocity was 70 degrees per second, while the second run had a maximum of 121 degrees per second. In typical VE and AR systems, the required prediction intervals range from 50 to 250 ms. By studying the closeup views, one can see that prediction might be possible on such curves for small prediction intervals. Note that some of the curves are noisy, which will pose problems. Chapter 6 will discuss how prediction performance is affected by noise and long prediction intervals.

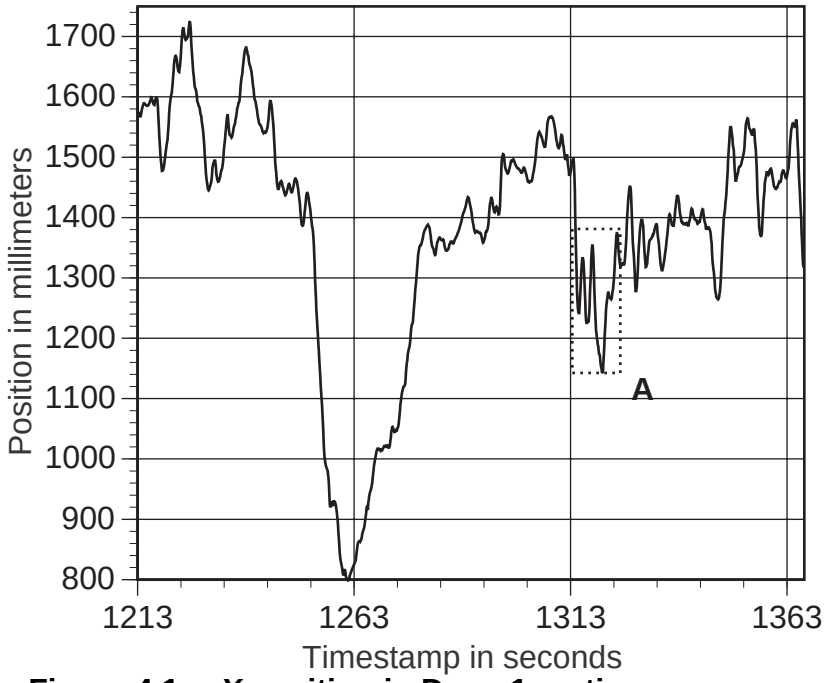


Figure 4.1: *X* position in Demo1 motion sequence

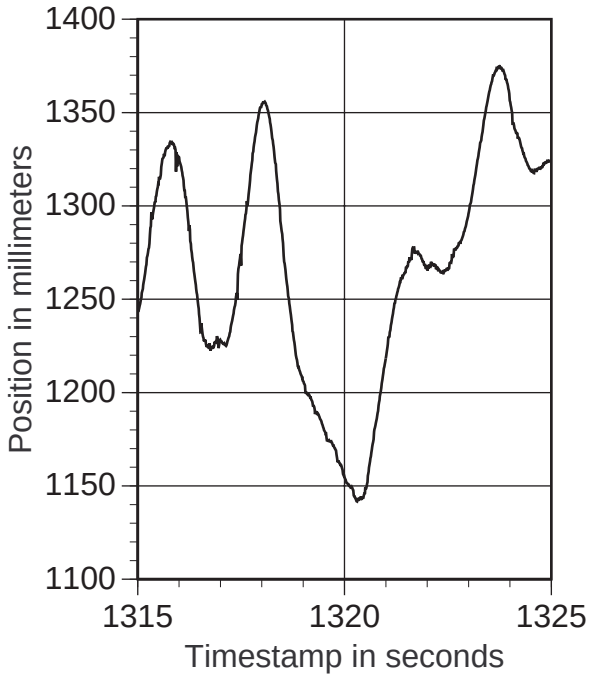


Figure 4.2: Closeup of region A in Figure 4.1

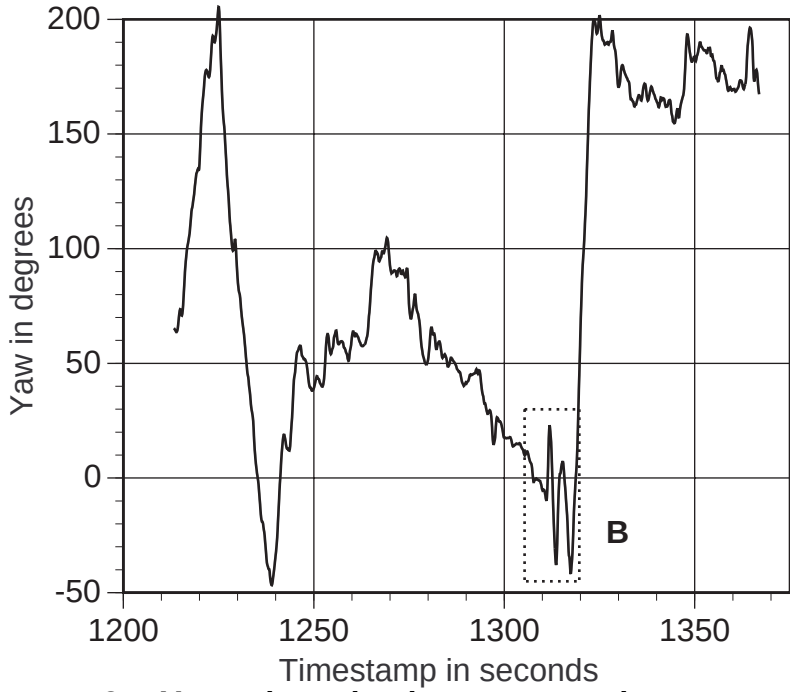


Figure 4.3: Yaw orientation in Demo1 motion sequence

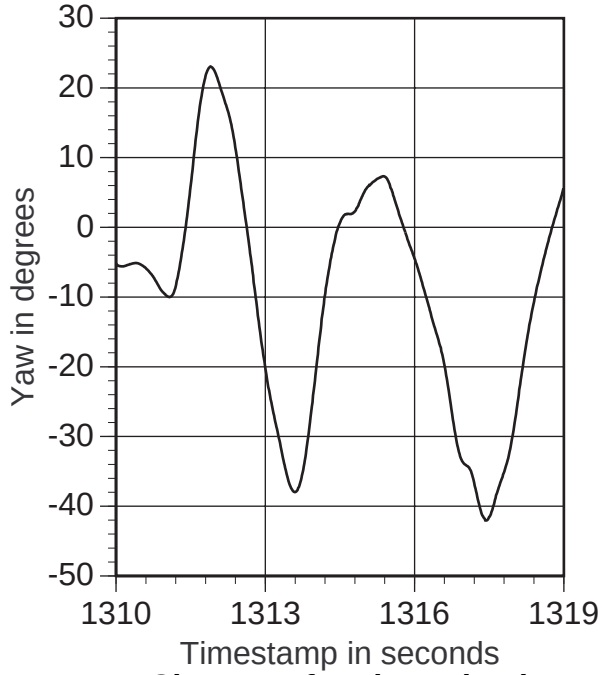


Figure 4.4: Closeup of region B in Figure 4.3

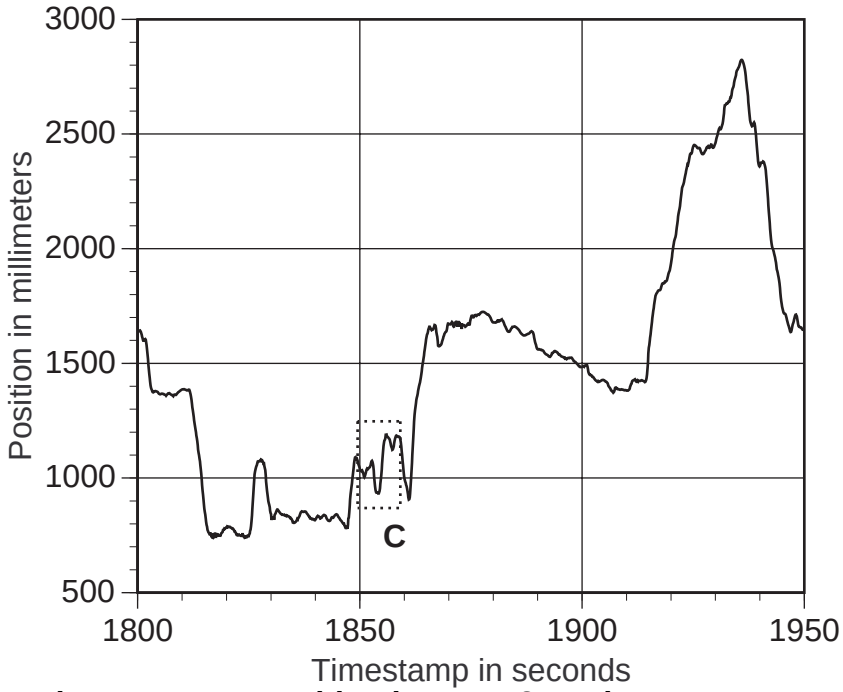


Figure 4.5: *Y* position in Demo2 motion sequence

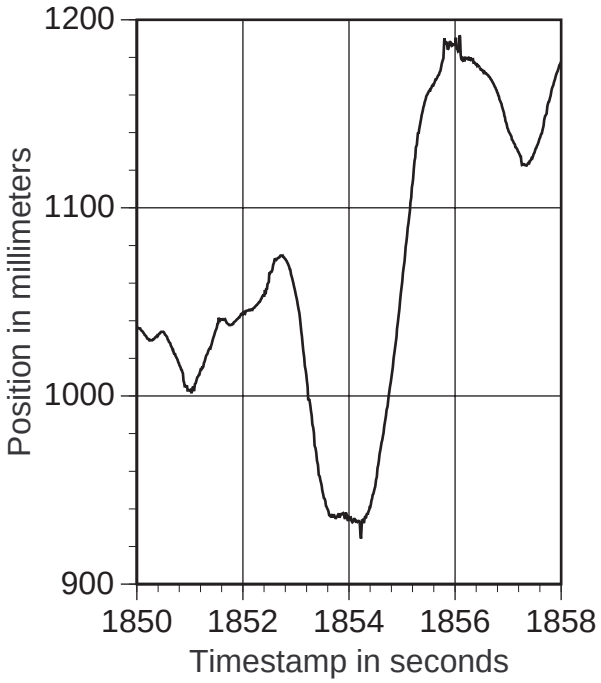


Figure 4.6: Closeup of region C in Figure 4.5

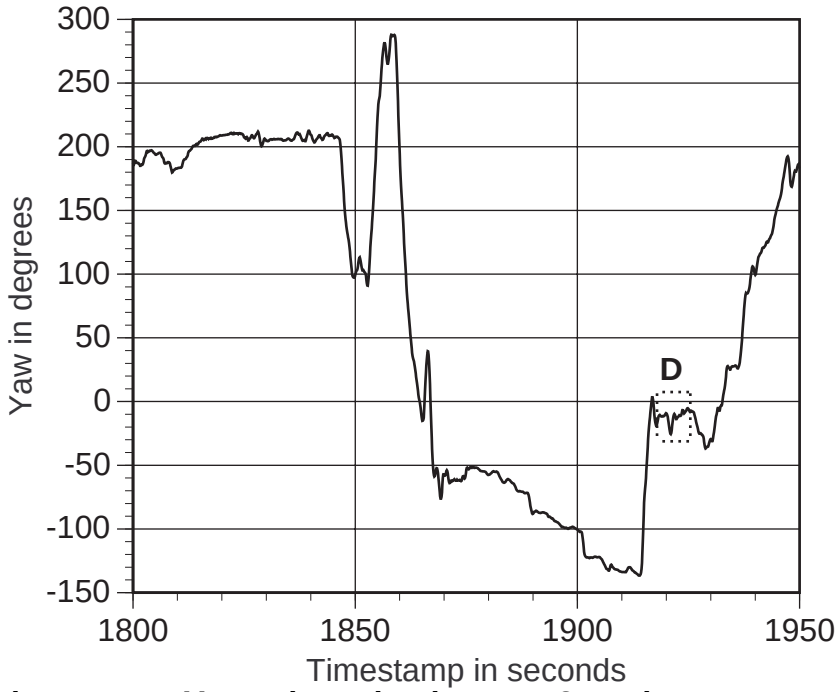


Figure 4.7: Yaw orientation in Demo2 motion sequence

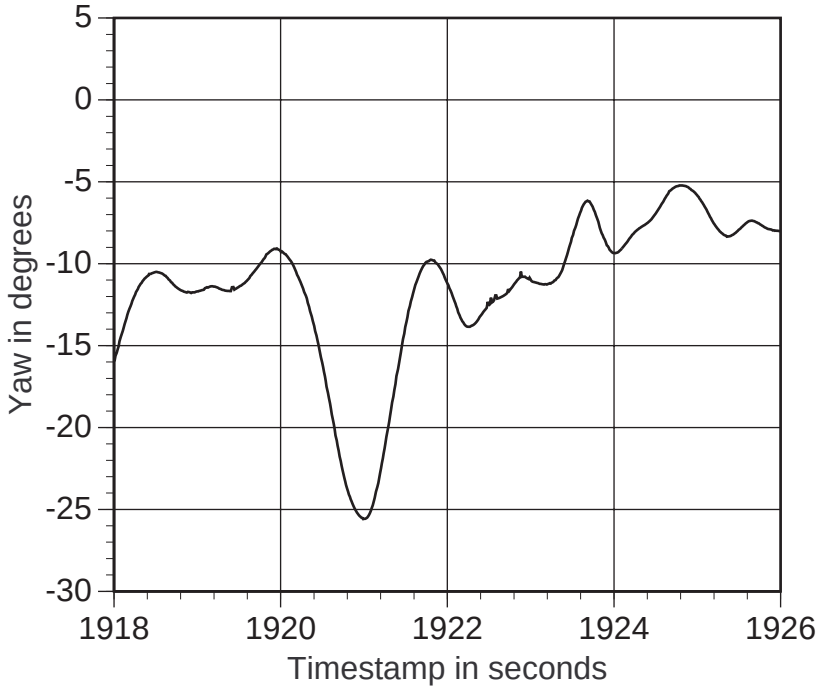


Figure 4.8: Closeup of region D in Figure 4.7

Another way of looking at the same data is to transform it into the frequency domain through Fourier analysis. (See Sections 6.2 and 6.6.3 for a discussion of frequency analysis.) Figures 4.9 through 4.12 show typical results for translation and orientation curves from a slow and fast head motion

sequence. The *power spectrum* of a curve displays the average square magnitudes of the coefficients, plotted against frequency. These graphs show the square root of the power spectrum values, or average magnitudes versus frequency. Note that virtually all the signal energy in each of the four graphs exists at frequencies under 2 Hz, which corroborates similar data collected by [So92]. The 2 Hz value is one way of quantitatively expressing that heads burdened with HMDs are limited in how quickly they can accelerate, decelerate, and change direction. It also suggests that predicting long intervals into the future is essentially intractable. Trying to accurately predict 10 seconds into the future is difficult with a signal that has significant energy up to 2 Hz. The signal could change direction several times within that 10 second period.

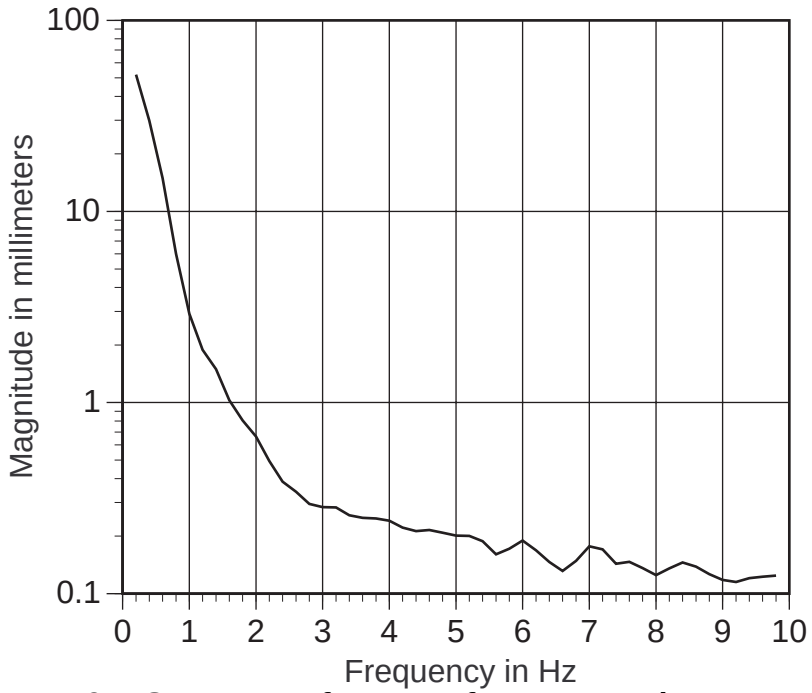


Figure 4.9: Spectrum of *X* curve from 1st motion sequence

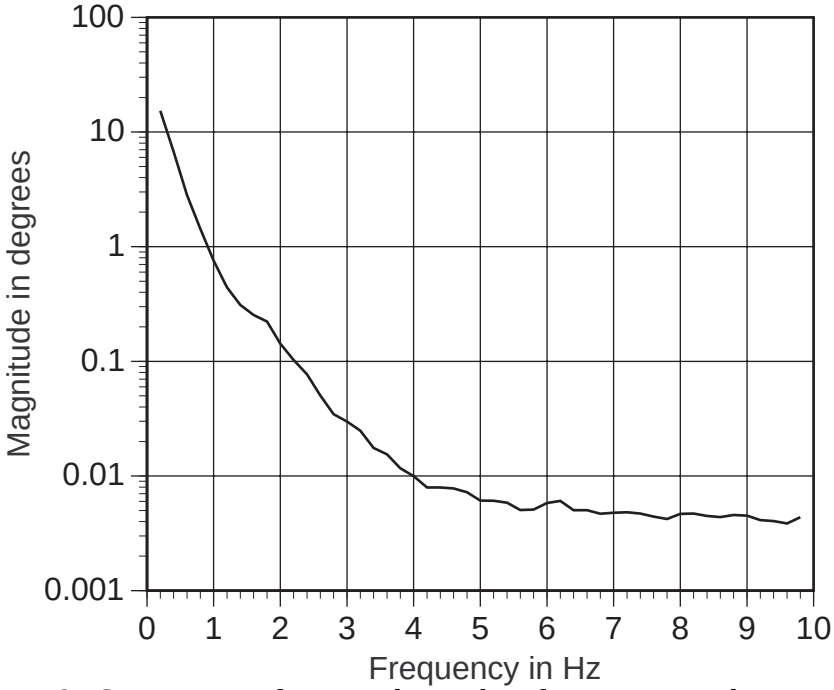


Figure 4.10: Spectrum of yaw orientation from 1st motion sequence

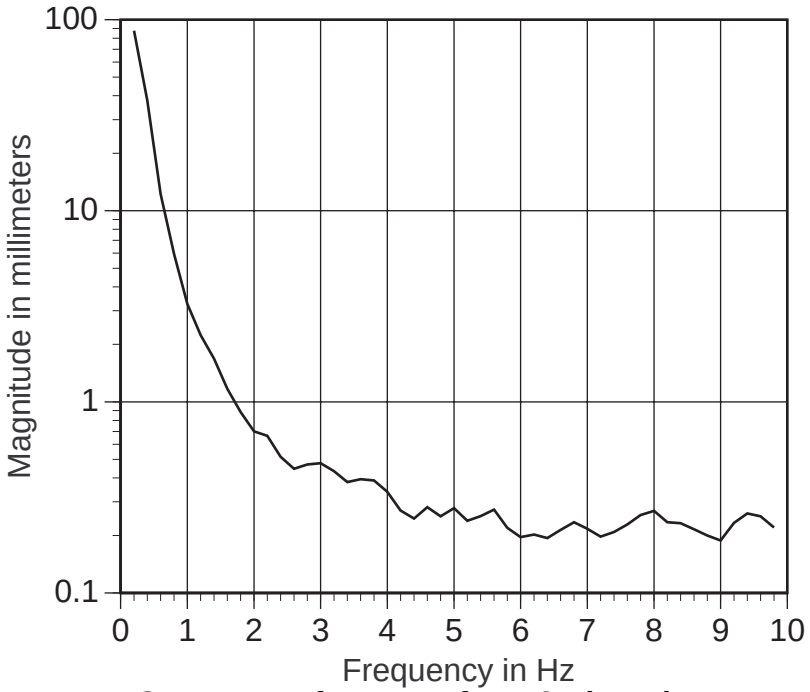


Figure 4.11: Spectrum of *Y* curve from 2nd motion sequence

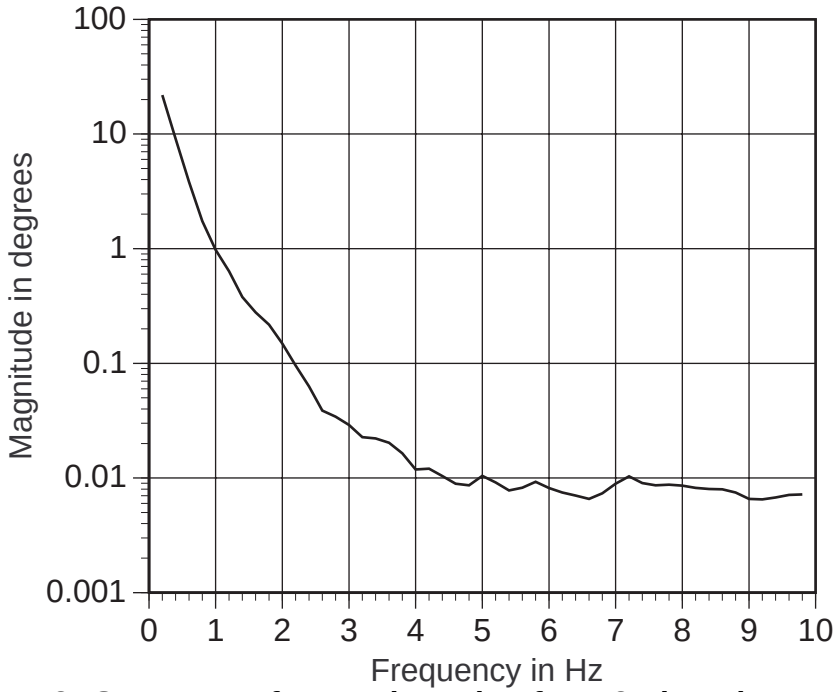


Figure 4.12: Spectrum of yaw orientation from 2nd motion sequence

Predicting head motion is a specific example of a much larger class of prediction and estimation problems that have received a great deal of attention. I draw upon this previous work, adapting general techniques to this problem as appropriate. The characteristics of this problem determine the most appropriate approach to take. What are the characteristics of head motion?

First, head motion is a collection of multidimensional signals that are difficult to perfectly model. Translation may be represented by a linear model. If orientation is represented by quaternions, then the orientation model is nonlinear because the four quaternion terms are nonlinearly dependent on each other. However, determining a model that closely fits head motion is nontrivial, linear or not.

Orientation is not intrinsically nonlinear, at least in a local neighborhood around the current orientation. One way to convert a nonlinear representation of orientation, such as quaternions, into a linear one is to use the following steps. First, change the initial quaternion into an equivalent Euler angle, where the three terms represent yaw, pitch, and roll. Then by using the small angle approximation, a small rotation away from the initial orientation can be specified by linear yaw, pitch, and roll operations, where the order of the

rotations is unimportant. Integrating these small rotations over time yields accurate yaw, pitch, and roll orientation curves that avoid gimbal-lock problems commonly associated with Euler angles. The integration timestep can be set as small as needed to push the integration error as low as desired. This produces three Euler angle curves representing orientation that could be handled by three 1-D linear predictors. However, this integration is expensive and is not practical in real time.

Second, head motion is *nonstationary*, which means that the statistical properties of the curves may change with time. For example, a user may keep his head still for a long time, then suddenly start moving around at rapid velocities.

Finally, the measurements of head motion will be corrupted by noise, generally of a complicated nature. A *bandlimited white noise process* is one that has equal magnitudes for all frequencies below a certain limit and zero energy for all frequencies above the limit. White noise is well behaved, but it is not generally the case that the tracker and inertial sensor measurement errors are accurately modeled solely by adding white noise; the inaccuracies are usually more complicated than that. The existence of noise requires the use of estimators to extract the best guess of the true value of the signal. The combination of difficult modeling, nonstationary signals and noise makes it difficult to apply standard prediction and estimation techniques without first simplifying the problem or violating assumptions. I now discuss how well the following general approaches match the head-motion prediction problem:

- Curve-fitting
- Control theory
- Information theory
- Time-series analysis
- Wiener filters
- Kalman filters

Fitting curves or splines to data is a reasonable method for smoothing but generally gives unsatisfactory results when used for prediction. Because the data points are corrupted with noise, techniques that approximate the points should be used, instead of ones that exactly match the data points.

Since high-order polynomials can introduce "wiggles" that are not representative of the original motion, locally-applied low-order curves like quadratic or cubic polynomials are usually the techniques of choice. I tried extrapolating future data points with such curves and found that they were inaccurate predictors.

Several previous works use control theory to aid predictor design; unfortunately it is difficult to apply that to this particular problem. *Control theory* applies to problems where a system attempts to follow a known input signal as accurately as possible, within the physical limitations of the system. For example, imagine a robot arm. The input signal tells the arm to move to a new position. Since the arm has inertia and the motor cannot generate an infinite amount of force, the arm cannot instantaneously move to the new position. It requires some time to execute the move. Control theory is the science of designing controllers that make the arm move in a way that is optimal with respect to some criterion, such as minimizing the time required or the energy consumed. Control theory is useful in flight simulators because airplanes have inertial characteristics and input signals that are easily measured. A plane cannot instantaneously change direction or speed, and these limitations are modeled by a *transfer function*. The input signals come from the pilot's controls in the cockpit. Figure 4.13 compares the flight simulator application against the problem of head-motion prediction in an Augmented Reality system. Within the AR system itself, there is no "inertia" between the input and output head locations. The AR system only adds delay. The inertia exists in the human head. To use control theory with this problem, I would need to somehow measure the neural signals that control the muscles that move the head. Since I do not have that information available, it is difficult to apply standard control-theory designs to this particular problem.

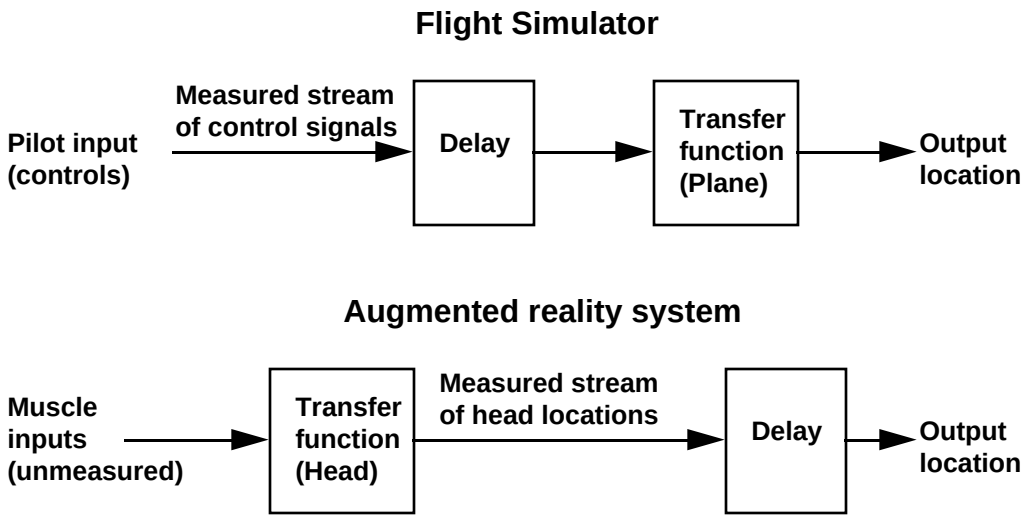


Figure 4.13 Flight simulator vs. Augmented Reality system

Information theory provides some interesting results for the prediction problem under specific conditions. Assume a 1-D signal is bandlimited, stationary, and free of noise. *Bandlimited* means that the signal has no energy in the frequency domain above a specified frequency. *Stationary* means that the statistical characteristics of a signal, such as the mean and covariances, remain constant with time. With these three assumptions, it is theoretically possible to predict *arbitrarily far* into the future with complete accuracy. This requires knowledge of all past values of the signal and that the signal be sampled faster than the Nyquist rate. This result makes use of the bandwidth restriction, not of any statistical properties of the signal. Conceptually, this result is true because a stationary bandlimited signal will eventually start repeating itself if observed long enough. By looking into the past far enough, a predictor can extract enough information to essentially reconstruct the entire function, allowing prediction for arbitrary intervals into the future. If the number of past values is limited, formulas exist that minimize the prediction error based on the number of available values. For details, see [Splettstösser82] [Mugler90] and other related works.

Unfortunately, these theoretical results are not useful in practice because measured head-motion signals are noisy and nonstationary. With nonstationary signals, looking deeply into the past may not be useful for predicting future values. The existence of noise makes the prediction formulas impractical. The formulas are extremely sensitive to any noise in the data. Even the limited precision of floating-point numbers causes problems in

practice, restricting effective prediction distances to fractions of the sampling period, which is not a useful range for the head-motion prediction problem.

Time-series approaches work on noisy data by approximately fitting specific models to the data. Almost all the literature in this area assumes stationary, linear signals. Given that values in the incoming signals are somehow correlated from current time t to a future time $t + dt$, the goal is to find a model that fits the data so well that the difference between the model and the data is white noise. In general, this is impossible, so the task becomes one of finding or picking a model that matches reasonably well. Standard models include linear ramps, impulse functions, constant values, and other curves. Time-series analysis is often used in economic applications, such as forecasting the demand for heating oil four months from now, so models often include cyclical components like sinusoidal functions. Regression methods can be used to find parameters that best fit a specific model to provided data. By looking at recorded motion sequences in Figures 4.1 through 4.8, it is clear that such simple models do not accurately fit head motion data for more than short time intervals. More complicated models include autoregressive (AR), moving average (MA), autoregressive moving-average (ARMA), and Box-Jenkins approaches [Montgomery90].

Basic time-series approaches do not use any information about the derivatives of the signals, since in most time-series applications the only measurements available are the signals themselves. Clearly, having sensors that directly measure the velocity or acceleration of the user's head should make the prediction problem easier. While it is possible to estimate derivative information from position signals, numerical differentiation is a noisy operation, and the derivative estimates will be delayed in time from their true values. That is, a derivative estimator requires some history of position values to make an accurate velocity estimate, which introduces significant lag. In contrast, sensors that directly measure velocity provide velocity measurements with very little delay. Therefore, I chose to augment the HMD with inertial sensors, and the chosen prediction method should make use of the additional information those sensors provide.

Two general classes of optimal linear estimators exist: Wiener and Kalman filters. They generally supersede the ARMA and Box-Jenkins models

used in time-series forecasting, at the cost of additional complexity. Both Wiener and Kalman filters are optimal in the sense that they minimize the expected mean-square error, given certain assumptions. Wiener filtering assumes that the signal to be detected is described by a white-noise process, and this signal is corrupted by a different white-noise source. Both the noise and the signal processes must be characterized by their known autocovariance and cross-covariance functions. The Kalman filter has a different set of assumptions. It assumes that the signals can be modeled by a set of variables that capture the state of the system at any time, along with a process that determines how these state variables change with time in the absence of any inputs. The output of this model, when subtracted from the signal, results in white noise, and the covariances of that noise are known. The signals are assumed to be corrupted with white noise of known covariances. The initial values of the state variables and their covariances are known. The signals can be nonstationary. Then the Kalman filter will take the measurements and return the optimal estimate for the state variables at any desired time.

Kalman filtering is a better choice than Wiener filtering for the head-motion prediction problem. Wiener filtering assumes a noiselike signal, and the graphs in Figures 4.1 - 4.8 show that head motion signals have far too much structure to be accurately modeled as a noiselike signal. Wiener filtering is also more computationally intensive than Kalman filtering, especially when simultaneously extracting multiple outputs from a set of signals. Kalman filtering has an efficient recursive formulation that is suitable for computer implementation, an important factor when the predictor has to run in real time. Finally, the Kalman filter can use the derivative information provided by the head-mounted inertial sensors.

A linear Kalman filter may be adequate for noisy, nonstationary translation signals, but quaternion-based orientation terms are nonlinear. Optimal nonlinear estimation in general is either intractable or too computationally expensive to be practical. Any nonlinear estimators that run in real time must therefore be suboptimal. Chang [Chang84] surveys such approaches, including the finite memory filter, the fading memory filter, and the constant gain filter, but he recommends the Extended Kalman Filter (EKF)

for most purposes. A variation of the standard Kalman filter, the EKF linearizes the model about the operating point specified by the current set of state variables, using that approximation to make the problem tractable.

A premium should be placed on simple and fast prediction methods. The time it takes to run the prediction routine is added directly to the end-to-end system delay. The difficulty of the prediction problem grows rapidly with increasing end-to-end delay (see Chapter 6). Increasing the prediction interval from 50 ms to 500 ms does not make the problem ten times harder; it makes the problem virtually intractable. Therefore, execution time is a significant factor in predictor design. A complicated predictor that takes 50 ms longer to execute than a simple predictor must be much more accurate than the simple predictor, just to break even in terms of performance.

Because of these factors, I chose to use linear Kalman filters to estimate and predict translation terms and an EKF to estimate and predict orientation terms. The Kalman filter appears to be the most natural formulation for my particular problem, which is perhaps why it is also used in several previous works. Although the uncertainty in both the measurements and the model are not adequately represented by white noise, the Kalman filter tends to perform well even when those underlying assumptions are violated. The next section describes how the Kalman filter works.

4.2 Kalman filters

This section gives an introduction to the Kalman filter. This description is intended to provide the reader with a basic idea of what the filter does; the actual equations are listed in Section 4.4. Full details about the Kalman filter are beyond the scope of this dissertation; for that please read [Lewis86] or the original papers [Kalman60] [Kalman61]. A good description of the history and background is in [Sorenson70]. The example I use is based on [Lewis86] and [Maybeck79].

I will illustrate the basic operation of the Kalman filter with a simple example. Say that a train is travelling on a long, straight section of track. Therefore, the position of the train can be described by one number *d*: the

distance of the train from some milestone. The train is equipped with a speedometer and a sensor that measures the train's position, such as a Global Positioning System (GPS) receiver. Say that the GPS readings are only occasionally available. Each sensor also has limited accuracy. The problem is to determine the position of the train at any specified time.

The Kalman filter is an estimator that combines data from sensors and a motion model in a computationally-efficient manner. If certain assumptions hold, the Kalman filter provides estimates that are optimal in the sense of minimizing the expected mean-square error. The estimates are of variables that describe the basic characteristics of the system. In this example, these are position *d* and velocity *v*. These two variables are held in a 2 by 1 vector **X**, called the *state vector*:

$$\mathbf{X} = \begin{bmatrix} d \\ v \end{bmatrix}$$

The values in the state vector are only approximations of the unknown true values. This is due to limited sensor accuracies and the fact that the measurements are not always available. The inaccuracies in the measurements are assumed to be accurately described by additive white noise. The Kalman filter models this uncertainty by a Gaussian probability distribution, as shown in Figure 4.14. This probability distribution comes from the assumption that all noise processes are white.

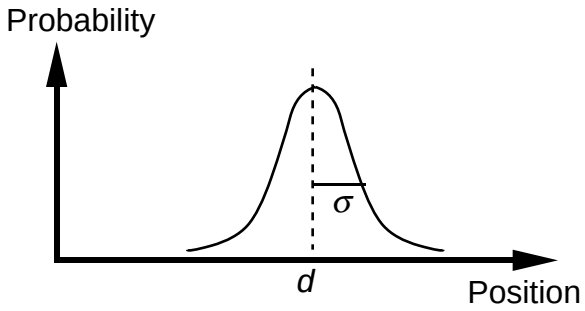


Figure 4.14 Gaussian probability distribution for position *d*

The Gaussian represents the distribution of values where the true position might be. The most likely position is *d*, the position in the state vector. The width of the Gaussian is specified by the standard deviation σ. Large standard deviations yield "fat" Gaussians, indicating that the estimate of *d* is

not very accurate. Smaller standard deviations yield "narrow" Gaussians, representing a more accurate estimate.

The Kalman filter stores these uncertainties in a *covariance matrix* **P**. **P** is an *N* by *N* matrix, where *N* is the number of entries in the state vector **X**. For this example, **P** is a 2 by 2 matrix. The diagonal terms are the variances of the state variables, while the non-diagonal terms are the covariances. Let σ_d be the standard deviation of variable *d*, and let σ_v be the standard deviation of variable *v*. Then **P** is:

$$\mathbf{P} = \begin{bmatrix} \sigma_d{}^2 & \sigma_d\,\sigma_v \\ \sigma_d\,\sigma_v & \sigma_v{}^2 \end{bmatrix}$$

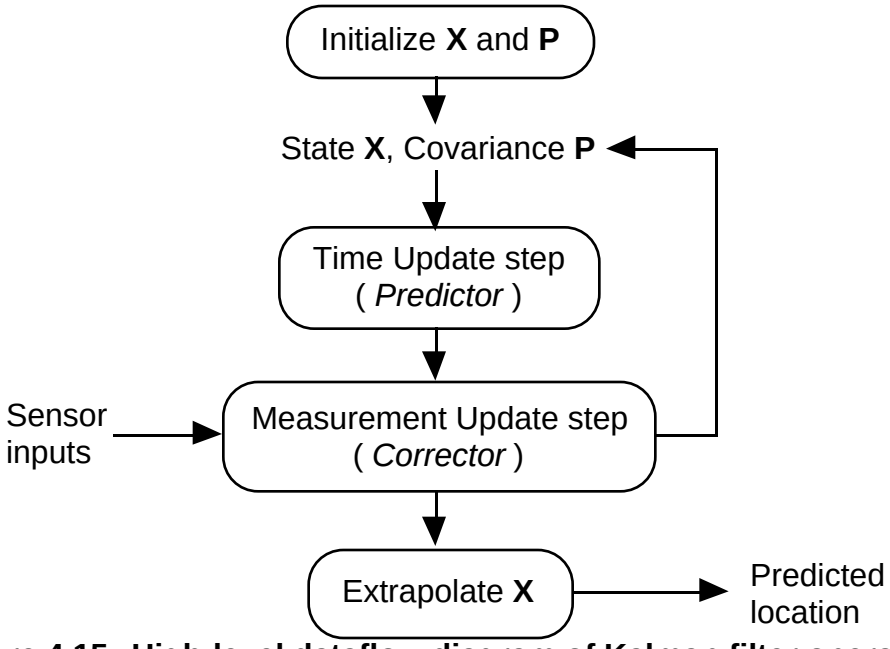


Figure 4.15 High-level dataflow diagram of Kalman filter operation

Figure 4.15 shows how the Kalman filter updates **X** and **P** as measurements are taken of the train's position. The Kalman filter starts by setting **X** and **P** to their initial values. The filter also sets the current time *t* to the initial timestamp. Now every time a GPS measurement is taken, the Kalman filter modifies **X** and **P** to include this new information. It does this in two steps that are similar in form to predictor-corrector methods used in numerical integrators. Say the GPS measurement is taken at time t_g, which must be more recent than the current time *t* associated with **X** and **P**. The Kalman filter runs a *time update* step (or predictor) that uses a motion model

to generate new estimates of **X** and **P** at time t_g. Then the filter runs a *measurement update* step (or corrector) that blends the GPS measurement with the new estimates of **X** and **P**. The result is the final estimate of **X** and **P**. The current time *t* associated with those matrices is set to t_g, and the filter is now ready for a new measurement.

The Kalman filter extrapolates the values in **X** to predict positions at any specified time. Generally speaking, a request for an estimate of **X** will not be for precisely the current time *t*, but for some future time t_f. Therefore, the filter must take the current value of **X** at time *t* and extrapolate that to a predicted value at time t_f. This is normally done by running a time update step from *t* to t_f, but other predictors can be used.

The motion model describes how the values of **X** and **P** change in the absence of any input. The model is assumed to accurately specify the future behavior of the system, except for the addition of white noise. For this example, the model is quite simple:

$$p_g = p + v\left(t_g - t\right)$$
$$\dot{v} = ß\, w(t)$$

where p_g is the new position at future time t_g, *ß* is some constant, and *w*(*t*) is a white noise process. The values in the covariance matrix **P** always increase as a result of running the motion model in the time update step. That is due to the noise in the motion model. As the time update step predicts further into the future without new measurements, the uncertainty of the estimates in **X** must grow. This is similar to a particle undergoing Brownian motion. Say that the particle's initial position is known, but none of the future positions are measured. Then the distribution of possible particle locations grows larger with time, as do the associated variances.

Figure 4.16 shows one example of how the estimated position and standard deviation change as a result of the time and measurement update steps. The initial position *d* at current time *t*, with associated standard deviation σ_d, is on the left side of the diagram. A new measurement d_m is taken at time t_g. That measurement has an associated standard deviation σ_{dm}. The train is assumed to be moving from left to right on this diagram. The time update step takes the initial position *d* and predicts a new position d_t

at time t_g. The new standard deviation σ_{dt} is larger than the initial standard deviation, for reasons previously described. Then the measurement update step combines the time update estimate with the measurement, to form the final estimate d_f with standard deviation σ_{df}. Note that the final estimate is a blend of the time-update estimate and the measurement, and that its associated standard deviation σ_{df} is smaller than either σ_{dt} or σ_{dm}. By combining the time-update estimate and the measurement, the measurement update step provides a more accurate estimate.

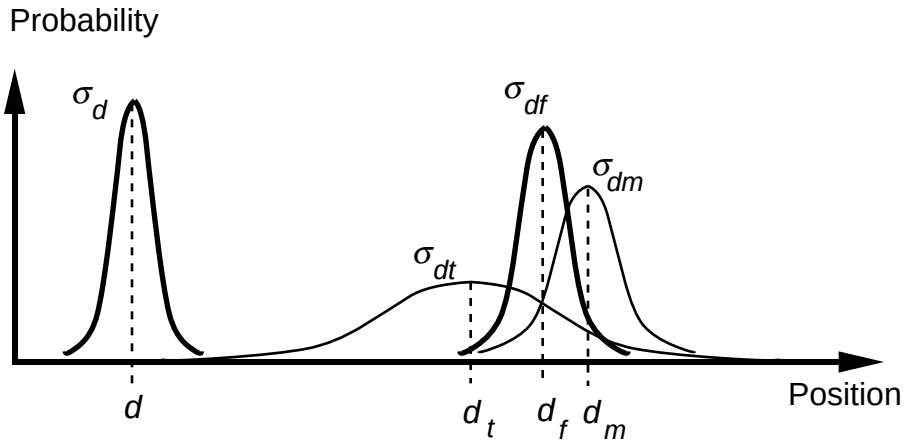


Figure 4.16 An example of how position and standard deviation change during the time and measurement update steps

The Kalman filter is efficient to implement because it mostly requires matrix operations and has a recursive formulation. Generally speaking, producing optimal position estimates might require knowledge of all past positions. As the set of past positions grows larger, such an algorithm would take longer to run. The Kalman filter avoids this by a recursive formulation that captures the information it needs to know about all past values in just two matrices, **X** and **P**, maintained at one timestamp *t*.

This example provides a high-level description of what the Kalman filter does and how it operates. The actual equations used to implement the filter are described in Section 4.4.

4.3 Previous work

This dissertation is not the first attempt at predicting head motion for HMDs. This section surveys previous work on predicting head motion or related motions. The more general prediction and estimation approaches were discussed in Section 4.1.

4.3.1 Prediction of head motion

Two papers use head-motion prediction with head-tracked stereo displays [Deering92] [Paley92]. They extrapolate future head positions based on past positions reported by the head tracker. For example, Paley averages the last two measured positions and adds that to the current position to form his predicted location.

Several papers predict head motion for HMDs. Albrecht developed an adaptive predictor to predict the yaw and pitch components of orientation [Albrecht89]. The predictor was not operated in real time. Instead, it ran in simulation on head-motion data recorded from an F-15 flight simulator that used a Kaiser Agile Eye HMD and a Polhemus head tracker. His method keeps the *N* most recent measurements and assigns a weight to each, summing the weighted measurements to form the predicted output. The weights can change with time, making this an adaptive filter. The filter uses a fixed prediction interval of 100 ms and assumes the tracker inputs are evenly spaced in time.

Rebo used a Kalman filter to predict head orientation for an HMD [Rebo88]. His filter model assumes that head acceleration behaves like white noise. He states that prediction is accurate when head motion is very slow, but he also notes that when the user comes to an abrupt halt, the predicted locations oscillate.

Liang also uses a Kalman filter to predict future head orientations and positions for an HMD [Liang91]. The motion model assumes that head rotations are infrequent and that head translations mostly occur along the gaze direction, leading to the choice of a Gauss-Markov model for acceleration:

$$A = -ß\,V + K\,w(t)$$

where *ß* and *K* are constant parameters, *A* is acceleration, *V* is velocity, and *w*(*t*) is a unit white-noise sequence. A separate lowpass filter reduces jitter in the position values because of the noise in their Polhemus tracker. They note a tradeoff between the smoothness of the predicted outputs versus the accuracy of the prediction. The predictor assumes that tracker readings are evenly spaced in time and uses a constant prediction interval.

Smith developed an orientation-only predictor for HMDs [Smith84]. He takes the rotation matrices that represent the last two orientations reported from the head tracker, subtracts one matrix from the other, and uses the resulting difference matrix to linearly extrapolate, component-by-component, a future rotation matrix.

Rediffusion predicted future head orientations on an HMD-based flight simulator [Murray85]. Angular rates are estimated from the orientations reported by a Polhemus tracker. The combination of measured head orientations and the estimated head angular velocities are used to predict future head orientations. The method appears to be proprietary, as no other details or references are provided.

Image deflection is a clever technique for reducing the amount of apparent system delay for systems that only use head orientation [So92] [Regan94] [Riner92]. It is not a prediction method per se. Instead, it is a way to incorporate more recent orientation measurements into the late stages of the rendering pipeline. Therefore, it is a feed-forward technique. The scene generator renders an image much larger than needed to fill the display. Then just before scanout, the system reads the most recent orientation report. Alternately, one can do prediction based on the recent orientation values, drastically reducing the required prediction intervals. The orientation value is used to select the fraction of the frame buffer to send to the display, since orientation changes are equivalent to shifting the frame buffer output. More work needs to be done to extend this approach to handle head translations.

So far, all the methods I have described base their predictions on the reported tracker positions and orientations. Only two methods also make use

of inertial sensors to aid head-motion prediction. Both used prototypes of CAE's Fiber Optic Helmet-Mounted Display (FOHMD) and predict head orientation. Uwe List mounted angular accelerometers on an HMD and integrated the acceleration values to provide estimates of angular velocities [List84]. These velocities are used to extrapolate future head orientations along each local Euler axis. CAE's 1986 technical report [Welch86] uses three angular accelerometers to cover all three orientation axes. The accelerometer readings are filtered, then integrated to provide estimates of future angular velocities and orientations. The predictions are blended with estimated orientations and velocities derived from the head tracker to reduce drift and other systematic errors. Several constant coefficients control the blending between these values.

The prediction technique actually used in the production version of the FOHMD is considered proprietary, so no published references are available. On August 11, 1993, I had the opportunity to personally try the FOHMD at NASA Ames. That unit had three angular accelerometers, mounted in a mutually orthogonal configuration, on the back of the helmet. I wore the helmet both with prediction turned on and prediction turned off. The predicted motion definitely had much less apparent delay than the non-predicted motion. However, since the FOHMD is not set up to support Augmented Reality applications, it was extremely difficult to quantify how much error remained after the delay compensation. Without registered pairs of real and virtual objects, I had nothing to use as a basis of comparison. A technician mentioned that in the past, they also used angular rate gyroscopes in place of the angular accelerometers.

4.3.2 Prediction of related motion

Friedmann uses Kalman filters to predict the position of a drumstick [Friedmann92]. With a Polhemus sensor taped to the drumstick, he tries to detect when the drumstick strikes a surface so he can play a computer-generated sound. System and tracker delays force him to predict when the contact occurs to maintain synchronization between the graphics and the generated sound. He uses a Multiple Model or Generalized Likelihood

approach that runs several Kalman filters in parallel and chooses the output of the one that best seems to match observed measurements in the recent past.

Chu Wang mentions but does not describe a prediction system for tracked hand motion [WangC90]. He notes problems with overshoots caused by rapid acceleration or deceleration.

Teleoperation is a related field that also has problems with system delays. Numerous papers in this area describe delay compensation techniques; one example is [Bejczy92]. Some teleoperation systems, but not all, involve head motion. Since the recorded head motion is known, all are control problems rather than estimation problems. The goal is to make the slave device follow the recorded motion as quickly and accurately as possible. Most teleoperation systems have an order of magnitude more lag than typical Virtual Environment or Augmented Reality systems. Delays of a second or two are not unusual in teleoperation applications. When delays become that long, the goal changes from improving real-time matching to avoiding a complete breakdown of system usefulness.

The flight simulation community has also tackled system delay problems; examples include [Crane84] [McFarland86] [McFarland88] [Sobiski87]. Cardullo surveys these and other delay compensation methods [Cardullo90]. These methods model an airplane and the pilot controlling it in a simulator, modifying the pilot's control signals to increase stability and reduce the apparent lag in the system. Sobiski's and McFarland’s papers, for example, focus on aircraft roll rate.

A wealth of literature exists on prediction and tracking problems for ships, planes, missiles, and other vehicles, primarily for military purposes. Typical applications include tracking the path of an enemy airplane or predicting the path of an incoming ballistic missile. Representative examples include [Berg83] [Blom84] [Bolger87] [Chang84] [Tugnait87]. The range and pattern of motions of the human head are significantly different from those of aircraft, ships, and missiles.

4.3.3 Characteristics of head motion

It may be possible to achieve more accurate head motion prediction by developing more sophisticated models of head motion. The references I have listed so far basically treat the head as a rigid body and extrapolate based on the estimated position, velocity, and sometimes acceleration. If head motion contains other higher-order characteristics, an accurate model may be able to exploit those to improve prediction accuracy. Not much work has been done in this area.

The same group at the University of Alberta that wrote [Liang92] described plans for running experiments to identify higher-order head motion characteristics in [Shaw92]. The chosen task starts with a user keeping his head still. A target is drawn at a random location in his field-of-view. The user rotates his head to center the target in his display. For this task, they hope to show that rotation paths follow a great arc of a circle and that the velocity curves are symmetric about the temporal midpoint of the motion.

A few researchers have used Fitts' Law to predict head motion. A *Fitts' task*, when applied to HMDs, is the following: Place two targets in virtual space around the user. The user moves his head so that he aims it at one target, then the other, then back to the original target, oscillating between the two targets. Fitts empirically derived a function that, given the size of the targets and the distance separating the two targets, will predict the time it takes to complete the task [Fitts54]. This formula, called *Fitts' Law*, has been found to apply to a wide variety of motor-control tasks [Meyer88], and head motion is no exception [Andres89] [Jagacinski85] [Radwin90]. Note that Fitts' Law does not describe the path the user's head travels, just the time that it takes to switch between targets.

Several studies of pilots wearing Head-Mounted Sights have been done; see [Wells87] for an overview. A pilot wearing a Helmet-Mounted Sight can aim a gun simply by turning his head to look at the target. A typical study presents a target moving in a somewhat random pattern to the pilot and measures how well the pilot is able to track the target. Such data may indicate some characteristics of head motion, at least for the target-tracking task.

A book edited by Peterson describes the physiology of head movement and may be of use to anyone interested in developing physically-based models of head motion [Peterson88].

4.4 Prediction method

The predictor is one component of the "Tracker and predictor computer" module shown in Figure 2.5. It receives head locations, angular velocities and linear accelerations from the head tracker and the inertial sensors. When the scene generator is ready to create a new set of graphic images, it asks the predictor to produce a predicted head location for use in generating those images. These measurement inputs and requests for predictions can occur at any time and will not occur at evenly-spaced intervals. This section first defines the inputs and outputs for this module, then explains how the translation terms are handled, followed by the orientation terms. This explanation includes the equations used to implement the method.

To make this explanation simpler, certain implementation details are not covered until Chapter 5. Please see Section 5.3 for those details.

4.4.1 Overview

At startup, the prediction module initializes itself. After that, it responds to two events: 1) an input of tracker and inertial data and 2) a request to provide a predicted output.

Each input packet supplies the following information:

Timestamp	[When these values were recorded, in seconds]
qw qx qy qz	[Tracker orientation, as a quaternion]
tx ty tz	[Tracker position, in meters]
ω0 ω1 ω2	[Tracker angular velocity in radians per second]
ax ay az	[Tracker linear acceleration in meters per second^2]

The quaternion representing Tracker orientation is the quaternion required to rotate the World coordinate axes so that it shares the same

orientation as the Tracker coordinate axes. The Tracker position measurement tells how far to translate the World coordinate axes with respect to the World coordinate system so that the origin of the World coordinate system coincides with the origin of the Tracker coordinate system. Thus taken together, the Tracker orientation and position describe the transformation that moves points and vectors from Tracker space to World space.

Angular velocity is reported as omega, a 3 by 1 vector that represents the instantaneous angular rate of head rotation. Throughout the rest of this chapter, ω represents omega:

$$\omega = \begin{bmatrix} \omega 0 \\ \omega 1 \\ \omega 2 \end{bmatrix}$$

Omega is reported in Tracker space. The 3 by 1 vector that omega specifies is the axis about which the angular rotation takes place, and the rate of rotation about that axis, in radians per second, is the magnitude of omega. The rotation direction is specified by the right-hand rule applied to the axis of rotation.

Linear acceleration is the translational acceleration, reported in the same space as the Tracker positions are. The accelerometers do not actually return this value (see Section 5.3.1), but to make the filter simpler, I assume there is a "virtual sensor" that returns that value.

There is no direct output from a measurement packet. Instead, the predictor updates its internal state based on the reported measurements.

Each request for a predicted location comes with a single parameter: a timestamp specifying the time for the desired predicted location. In return, the predictor sends the scene generator the following:

qw qx qy qz	[Tracker orientation, as a quaternion]
tx ty tz	[Tracker position, in meters]

The definitions of the Tracker orientation and position are the same as those in the input measurements.

4.4.2 Translation

The predictor runs separate filters for the translation and orientation components. The translation terms are broken up further into the three axes *X*, *Y*, and *Z*. Separating the three translation terms greatly simplifies the filters at the cost of ignoring potential correlations across the three axes. Section 4.5 discusses the ramifications of this decision.

Since the three filters that handle the three axes are identical in form, I shall describe only the one that handles the *Y* axis.

Translation is handled by a linear *Continuous-Discrete* Kalman Filter. That means the time update step is handled in continuous time, but the measurements are available only at discrete intervals. The filter requires a 3 by 1 state vector **X** that holds the state variables representing the current position, velocity, and acceleration along the *Y* axis.

$$\mathbf{X} = \begin{bmatrix} y \\ \dot{y} \\ \ddot{y} \end{bmatrix}$$

Associated with this state vector is a 3 by 3 covariance matrix **P** that represents how accurate the filter believes the state variables are. The matrices **X** and **P** are maintained at time *t*, which is considered the current time. As new measurements arrive, *t* is advanced and **X** and **P** are updated to reflect the new information from the measurements. But before that happens, the filter must be initialized.

1) Initialization: The first measurement packet that arrives initializes the filter. If *y* is the reported *Y* axis position in the measurement packet, then **X** is initialized to:

$$\mathbf{X} = \begin{bmatrix} y \\ 0 \\ 0 \end{bmatrix}$$

Matrix **P** is initialized to $\mathbf{P_0}$, which makes the initial position, velocity and acceleration covariances large so that the initial values in **X** will be replaced by incoming measurements as those arrive.

$$\mathbf{P_0} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 500 & 0 \\ 0 & 0 & 500 \end{bmatrix}$$

The current time *t* is set to the timestamp from the measurement packet.

After the first measurement, any subsequent measurements are incorporated into the filter by a *time update* step, followed by a *measurement update* step. Say the new measurement occurs at timestamp t_1, where $t_1 > t$. The time update step advances the state variables from time *t* to time t_1, based upon the motion model. Then the measurement update step blends in the new measurements taken at time t_1. Finally, the current time *t* is set equal to new time t_1.

2) Time update: The time update step uses a 4th-order Runge-Kutta ODE solver [Press88] to integrate the derivatives of **X** and **P** from time *t* to time t_1. The derivatives are:

$$\dot{\mathbf{X}} = \mathbf{A}\,\mathbf{X}$$
$$\dot{\mathbf{P}} = \mathbf{A}\,\mathbf{P} + \mathbf{P}\,\mathbf{A}^{\mathsf{T}} + \mathbf{E_{ty}}$$

A is a 3 by 3 matrix that specifies how to get the derivatives of the state variables from the state variables themselves. Since **X** contains position, velocity and acceleration information, the matrix **A** simply sets the derivative of position to velocity and the derivative of velocity to acceleration. The derivative of acceleration is set to zero. This is the simplest reasonable motion model given state variables of position, velocity, and acceleration.

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$\mathbf{E_{ty}}$ is a 3 by 3 matrix that specifies the uncertainty in the motion model.

3) Measurement update: The measurement update step takes the measured position and acceleration and incorporates that into the **X** and **P** matrices. The 2 by 1 matrix **Z** holds the measured values.

y_m = the measured position
$\ddot{y}_m$ = the measured acceleration

$$\mathbf{Z} = \begin{bmatrix} y_m \\ \ddot{y}_m \end{bmatrix}$$

X is updated as follows:

$$\mathbf{X} = \mathbf{X} + \mathbf{K}(\mathbf{Z} - \mathbf{H}\,\mathbf{X})$$

H is a 2 by 3 matrix that describes how the measurements relate to the state variables, by the expression **Z** = **H X**. In this case, **H** simply assigns the measured position to the state variable position and the measured acceleration to the state variable acceleration.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

K is a 3 by 2 matrix called the Kalman gain, which controls the blending between the model and the new measurements.

$$\mathbf{K} = \mathbf{P}\,\mathbf{H}^{\mathsf{T}}\left(\mathbf{H}\,\mathbf{P}\,\mathbf{H}^{\mathsf{T}} + \mathbf{R_{ty}}\right)^{-1}$$

$\mathbf{R_{ty}}$ is a 2 by 2 covariance matrix specifying the uncertainty in the measurements. The last step is to update covariance matrix **P** as follows:

$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\,\mathbf{H})\mathbf{P}, \text{ where } \mathbf{I} = \text{ the 3 by 3 identity matrix}$$

Section 5.3.2 discusses how to set covariance matrices $\mathbf{E_{ty}}$ and $\mathbf{R_{ty}}$.

4) Prediction: Prediction is possible anytime after the filter has been initialized. The scene generator requests a predicted location at time *p*, where *p* > current time *t*. In the Kalman filter formulation, the optimal way to extrapolate future values is to run a time update step from time *t* to time *p*, except that the **P** matrix is not modified. Thus, extrapolation is based on the same motion model used by the filter. If the model matches the motion

except for added white noise, then this prediction is the best possible in the sense of minimizing mean-square error. However, other predictors can be used. In that case, the Kalman filter simply serves to provide the best estimates of the state variables for the predictor.

Because the model is simple, it is possible to solve the differential equations and write the predictor as a closed-form expression. The estimated position, velocity, and acceleration are drawn from the current values in **X**.

Let y_t = the estimated position at time t
Let $\dot{y}_t$ = the estimated velocity at time t
Let $\ddot{y}_t$ = the estimated acceleration at time t
Let y_p = the computed position at time p
Then $y_p = y_t + \dot{y}_t (p - t) + \frac{1}{2} \ddot{y}_t (p - t)^2$

This result is the same as the expression for computing the position of a falling body on Earth. This is not surprising, because both situations assume constant acceleration from time *t* to time *p*. One can also think of it as a 2nd-order Taylor expansion of a function *y*() about the point *t*.

In reality, this simple motion model is not a perfect fit to actual head motion. Another possibility is discussed in Section 4.6.

4.4.3 Orientation

Orientation is handled by a single EKF. The 10 by 1 state vector **X** holds the orientation, angular velocity, and angular acceleration terms. **Q** is a 4 by 1 vector representing the orientation quaternion. As previously defined, ω is the 3 by 1 vector representing angular velocity.

$$\mathbf{Q} = \begin{bmatrix} qw \\ qx \\ qy \\ qz \end{bmatrix}, \qquad \omega = \begin{bmatrix} \omega 0 \\ \omega 1 \\ \omega 2 \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} qw & qx & qy & qz & \omega 0 & \omega 1 & \omega 2 & \dot{\omega}0 & \dot{\omega}1 & \dot{\omega}2 \end{bmatrix}^T$$

Associated with this state vector is a 10 by 10 covariance matrix **P** that represents how accurate the filter believes the state variables are. As in the translation case, the matrices **X** and **P** are maintained at time *t*, which is considered the current time. As new measurements arrive, *t* is advanced and **X** and **P** are updated to reflect the new information from the measurements. Each prediction request is based on extrapolating readings from the current values in **X**. So the three operations that must be supported are initialization, incorporating a new measurement, and prediction.

1) Initialization: The first measurement packet that arrives initializes the filter. The four quaternion terms in **X** are set to the measured quaternion, while all other terms in **X** are set to zero. All non-diagonal terms in the covariance matrix **P** are set to zero. The first four diagonal terms of **P**, which are for the quaternion, are set to 1, while the other diagonal terms of **P**, which are for the angular velocity and acceleration, are set to 50. As in the translation case, these initial values for **P** are deliberately large so that the initial values in **X** will be updated as new measurements arrive. The current time *t* is set to the timestamp from the measurement packet.

After the first measurement, any subsequent measurements are incorporated into the filter by a *time update* step, followed by a *measurement update* step, just as in the translation case. Let the new measurement occur at timestamp t_1, where $t_1 > t$. The time update step advances the state variables from time *t* to time t_1, based upon the motion model. Then the measurement update step blends in the new measurements taken at time t_1. Finally, the current time *t* is set equal to new time t_1.

2) Time update: The time update step integrates the derivatives of **X** and **P** from time *t* to time t_1 with the same ODE solver used in the translation case. The derivatives are:

$$\dot{\mathbf{X}} = a(\mathbf{X}, t)$$
$$\dot{\mathbf{P}} = \mathbf{A}\,\mathbf{P} + \mathbf{P}\,\mathbf{A}^{\mathsf{T}} + \mathbf{E}_{\mathbf{orient}}$$

Note that the expression for the derivative of the state vector **X** is different from the translation case. The derivative is provided by the nonlinear

function $a(\mathbf{X}, t)$ which computes the quaternion's derivative by the following formula:

$$\dot{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} \bullet \omega)$$

The multiplication between **Q** and ω is a quaternion multiplication, and the 3 by 1 vector ω is written as a quaternion with the *qw* term set to zero and the *qx, qy,* and *qz* terms set to the $\omega 0$, $\omega 1$, and $\omega 2$ angular velocity values extracted from **X**, respectively [Chou92]. The function $a(\mathbf{X}, t)$ provides the derivatives of the angular velocity terms by returning the angular acceleration terms in **X**. The derivatives of the angular acceleration terms are set to zero.

$\mathbf{E}_{\mathbf{orient}}$ is a 10 by 10 matrix that specifies the uncertainty in the motion model.

The EKF is a suboptimal nonlinear estimator that linearizes the system about the operating point. Say that the current estimate of the state is $\mathbf{X_c}$. Then the Taylor expansion of the model around that point is:

$$a(\mathbf{X},t) = a(\mathbf{X_c},t) + \left.\frac{\partial a}{\partial \mathbf{X}}\right|_{\mathbf{X}=\mathbf{X_c}} (\mathbf{X} - \mathbf{X_c}) + \ldots$$

To linearize this, ignore all terms after the first-order term. The result is stored in a 10 by 10 Jacobian matrix **A**. If the state vector **X** has *N* variables, x_1 through x_N, then the nonlinear $a(\mathbf{X}, t)$ function is an *N*-valued function. Decompose that into *N* single-valued functions $a_1(\mathbf{X}, t)$, $a_2(\mathbf{X}, t)$, ... $a_N(\mathbf{X}, t)$. Then the Jacobian matrix **A** is defined as:

$$\mathbf{A} = \begin{bmatrix} \frac{\partial a_1}{\partial x_1} & \frac{\partial a_1}{\partial x_2} & \cdots & \frac{\partial a_1}{\partial x_N} \\ \frac{\partial a_2}{\partial x_1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial a_N}{\partial x_1} & \cdots & \cdots & \frac{\partial a_N}{\partial x_N} \end{bmatrix}$$

In this case, the Jacobian matrix is set using the *qw, qx, qy, qz, ω0, ω1* and *ω2* terms from the **X** vector as follows:

$$\mathbf{A} = \begin{bmatrix} 0 & \frac{-\omega 0}{2} & \frac{-\omega 1}{2} & \frac{-\omega 2}{2} & \frac{-qx}{2} & \frac{-qy}{2} & \frac{-qz}{2} & 0 & 0 & 0 \\ \frac{\omega 0}{2} & 0 & \frac{\omega 2}{2} & \frac{-\omega 1}{2} & \frac{qw}{2} & \frac{-qz}{2} & \frac{qy}{2} & 0 & 0 & 0 \\ \frac{\omega 1}{2} & \frac{-\omega 2}{2} & 0 & \frac{\omega 0}{2} & \frac{qz}{2} & \frac{qw}{2} & \frac{-qx}{2} & 0 & 0 & 0 \\ \frac{\omega 2}{2} & \frac{\omega 1}{2} & \frac{-\omega 0}{2} & 0 & \frac{-qy}{2} & \frac{qx}{2} & \frac{qw}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3) Measurement update: The measurement update takes the measured orientation and angular velocity and incorporates those into the **X** and **P** matrices. The 7 by 1 matrix **Z** holds the measured values. Let the subscript *m* denote the measured values:

$$\mathbf{Z} = \begin{bmatrix} qw_m & qx_m & qy_m & qz_m & \omega 0_m & \omega 1_m & \omega 2_m \end{bmatrix}^{\mathsf{T}}$$

The measurement update generates new **X** and **P** matrices as follows:

$$\mathbf{K} = \mathbf{P}\,\mathbf{H}^{\mathsf{T}}\left(\mathbf{H}\,\mathbf{P}\,\mathbf{H}^{\mathsf{T}} + \mathbf{R}_{\mathbf{orient}}\right)^{-1}$$

$$\mathbf{X} = \mathbf{X} + \mathbf{K}\left(\mathbf{Z} - h(\mathbf{X})\right)$$

$$\mathbf{P} = \left(\mathbf{I} - \mathbf{K}\,\mathbf{H}\right)\mathbf{P}, \text{ where } \mathbf{I} \text{ is the 10 by 10 identity matrix}$$

$\mathbf{R_{orient}}$ is a 7 by 7 covariance matrix specifying the uncertainty in the measurements. **K** is the 10 by 7 Kalman gain matrix. The nonlinear function *h*(**X**) describes how the measurements relate to the state variables by:

$$\mathbf{Z} = h(\mathbf{X})$$

Z is composed of the measured quaternion $\mathbf{Q_m}$ and the measured omega $\omega_\mathbf{m}$. The function *h*(**X**) computes those from the **Q** and ω terms in the state vector **X** as follows:

$$\mathbf{Q_m} = \text{Normalize}(\mathbf{Q})$$
$$\omega_\mathbf{m} = \omega$$

H is the 10 by 7 Jacobian matrix for that function, and it is set using the *qw, qx, qy,* and *qz* terms from **X** as follows:

$$\text{Let } D = qw^2 + qx^2 + qy^2 + qz^2$$
$$\text{Let L } = \sqrt{D}$$

$$\text{Then } \mathbf{H} = \begin{bmatrix} \frac{L - \frac{qw^2}{L}}{D} & \frac{-qw\ qx}{L\ D} & \frac{-qw\ qy}{L\ D} & \frac{-qw\ qz}{L\ D} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{-qx\ qw}{L\ D} & \frac{L - \frac{qx^2}{L}}{D} & \frac{-qx\ qy}{L\ D} & \frac{-qx\ qz}{L\ D} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{-qy\ qw}{L\ D} & \frac{-qy\ qx}{L\ D} & \frac{L - \frac{qy^2}{L}}{D} & \frac{-qy\ qz}{L\ D} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{-qz\ qw}{L\ D} & \frac{-qz\ qx}{L\ D} & \frac{-qz\ qy}{L\ D} & \frac{L - \frac{qz^2}{L}}{D} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

At the end of the measurement update step, the quaternion part of **X** is explicitly renormalized. This is not a standard part of the Kalman filter, but without this extra step the quaternion term quickly becomes unnormalized, due to accumulating numerical errors caused by the linearization.

Section 5.3.2 discusses how to set covariance matrices $\mathbf{E_{orient}}$ and $\mathbf{R_{orient}}$. Section 5.3.1 describes how to recover $\omega_{\mathbf{m}}$ from the raw gyroscope measurements.

4) Predictor: The scene generator requests a predicted location at time *p*, where *p* > *t*. As in the translation case, the standard method of extrapolation is to do a time update from time *t* to time *p* for vector **X** only. This integrates the differential equations under the assumption that angular acceleration remains constant from time *p* to time *t*. It is possible to derive a closed-form solution for this problem.

Let the current time be t_c and the ending time be *p*. This notation will let *t* become a dummy variable. It is possible to rewrite the expression for the derivative of a quaternion as a matrix multiplication:

$$\dot{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} \bullet \omega) = \mathbf{M}(t)\,\mathbf{Q}$$

While the multiplication of **Q** and ω is a quaternion multiplication, the multiplication of **M**(*t*) and **Q** is a matrix multiplication. The 4 by 4 matrix **M**(*t*) that satisfies this equation is defined as follows:

$$\dot{\mathbf{Q}} = \mathbf{M}(t)\,\mathbf{Q}$$

$$\begin{bmatrix} \dot{qw} \\ \dot{qx} \\ \dot{qy} \\ \dot{qz} \end{bmatrix} = \begin{bmatrix} 0 & \frac{-\omega 0(t)}{2} & \frac{-\omega 1(t)}{2} & \frac{-\omega 2(t)}{2} \\ \frac{\omega 0(t)}{2} & 0 & \frac{\omega 2(t)}{2} & \frac{-\omega 1(t)}{2} \\ \frac{\omega 1(t)}{2} & \frac{-\omega 2(t)}{2} & 0 & \frac{\omega 0(t)}{2} \\ \frac{\omega 2(t)}{2} & \frac{\omega 1(t)}{2} & \frac{-\omega 0(t)}{2} & 0 \end{bmatrix} \begin{bmatrix} qw \\ qx \\ qy \\ qz \end{bmatrix}$$

Note that the $\omega 0$, $\omega 1$, and $\omega 2$ terms change with time because the assumption of constant angular acceleration implies that angular velocity is a linear function with respect to time. That is:

$$\omega 0(t) = \omega 0(t_c) + (p - t_c)\,\dot{\omega 0}(t_c)$$

$$\omega 1(t) = \omega 1(t_c) + (p - t_c)\,\dot{\omega 1}(t_c)$$

$$\omega 2(t) = \omega 2(t_c) + (p - t_c)\,\dot{\omega 2}(t_c)$$

The general form of the solution to the differential equation $\dot{\mathbf{Q}} = \mathbf{M}(t)\,\mathbf{Q}$ is:

$$\mathbf{Q_p} = e^{\int \mathbf{M}(t)\,dt}\ \mathbf{Q_{tc}}$$

where $\mathbf{Q_{tc}}$ is the original quaternion at current time t_c, and $\mathbf{Q_p}$ is the predicted quaternion at time p. Integrating $\mathbf{M}(t)$ is done on a component-by-component basis. This requires integrating $\omega 0(t)$, $\omega 1(t)$ and $\omega 2(t)$ from time t_c to time p.

$$\int_{t_c}^{p} \omega 0(t)\,dt = \int_{t_c}^{p} \left[\omega 0(t_c) + (p - t_c)\,\dot{\omega 0}(t_c) \right] dt$$

$$= (p - t_c)\,\omega 0(t_c) + \tfrac{1}{2}(p - t_c)^2\,\dot{\omega 0}(t_c)$$

Similarly,

$$\int_{t_c}^{p} \omega 1(t)\,dt = (p - t_c)\,\omega 1(t_c) + \tfrac{1}{2}(p - t_c)^2\,\dot{\omega 1}(t_c)$$

$$\int_{t_c}^{p} \omega 2(t)\,dt = (p - t_c)\,\omega 2(t_c) + \tfrac{1}{2}(p - t_c)^2\,\dot{\omega 2}(t_c)$$

Now define a, b, c, d and $\mathbf{B}$ to be the following:

$$a = \tfrac{1}{2}\int_{t_c}^{p} \omega 0(t)\,dt = \tfrac{1}{2}(p - t_c)\,\omega 0(t_c) + \tfrac{1}{4}(p - t_c)^2\,\dot{\omega 0}(t_c)$$

$$b = \tfrac{1}{2}\int_{t_c}^{p} \omega 1(t)\,dt = \tfrac{1}{2}(p - t_c)\,\omega 1(t_c) + \tfrac{1}{4}(p - t_c)^2\,\dot{\omega 1}(t_c)$$

$$c = \tfrac{1}{2}\int_{t_c}^{p} \omega 2(t)\,dt = \tfrac{1}{2}(p - t_c)\,\omega 2(t_c) + \tfrac{1}{4}(p - t_c)^2\,\dot{\omega 2}(t_c)$$

$$d = \sqrt{a^2 + b^2 + c^2}$$

$$\mathbf{B} = \int_{t_c}^{p} \mathbf{M}(t)\,dt = \begin{bmatrix} 0 & -a & -b & -c \\ a & 0 & c & -b \\ b & -c & 0 & a \\ c & b & -a & 0 \end{bmatrix}$$

Now the problem is reduced to:

$$\mathbf{Q_p} = e^{\mathbf{B}}\,\mathbf{Q_{tc}}$$

Since **B** is now known, what remains is to find a closed-form expression for $e^{\mathbf{B}}$. By applying a Taylor expansion to $e^{\mathbf{B}}$, I get the following:

$$e^{\mathbf{B}} = \mathbf{I} + \mathbf{M} + \frac{\mathbf{M}^2}{2!} + \frac{\mathbf{M}^3}{3!} + \frac{\mathbf{M}^4}{4!} + \ldots$$

where **I** is the 4 by 4 identity matrix. Note the following relationships hold:

$$\mathbf{M}^2 = -d^2\,\mathbf{I}$$
$$\mathbf{M}^3 = -d^2\,\mathbf{M}$$
$$\mathbf{M}^4 = d^4\,\mathbf{I}$$
$$\mathbf{M}^5 = d^4\,\mathbf{M}$$
$$\mathbf{M}^6 = -d^6\,\mathbf{I}$$
$$\mathbf{M}^7 = -d^6\,\mathbf{M}$$

etc.

Therefore:

$$e^{\mathbf{B}} = \mathbf{I}\left(1 - \frac{d^2}{2!} + \frac{d^4}{4!} - \frac{d^6}{6!}\ldots\right) + \frac{\mathbf{M}}{d}\left(d - \frac{d^3}{3!} + \frac{d^5}{5!} - \frac{d^7}{7!}\ldots\right)$$

$$e^{\mathbf{B}} = \mathbf{I}\cos(d) + \frac{\mathbf{M}}{d}\sin(d)$$

So the closed-form solution for $\mathbf{Q_p}$ is:

$$\boxed{\mathbf{Q}_{\mathbf{p}} = \left[\mathbf{I}\cos(d) + \frac{\mathbf{M}(t_c)}{d}\sin(d)\right]\mathbf{Q}_{\mathbf{tc}}}$$

Once $\mathbf{Q_p}$ is computed, I explicitly renormalize it before sending it to the scene generator.

4.5 Evaluation

From the user's perspective, prediction changes dynamic registration from "swimming around the real object" to "staying close." Without prediction, dynamic registration errors are large enough to strain the illusion that the real and virtual coexist. The virtual objects "swim around" the real objects as the user moves his head, making it difficult to believe that any registration exists. With prediction turned on, the real and virtual objects stay close enough to each other that the user perceives them to be together. Although prediction does not remove all dynamic errors, it demonstrably improves the dynamic registration.

My prediction method reduces average dynamic errors by a factor of 5 to 10 over not doing any prediction at all, and it reduces errors by a factor of 2 to 3 over doing prediction without the aid of inertial sensors. The larger factors occur when the head is moving rapidly. At slower speeds, the ratio of improvement is not as large because of the "noise floor" caused by the noise in the original signals. These factors are based on three objective error metrics: angular error, position error, and screen error, where the tracker-reported locations are assumed to be the true locations.

Angular error is the difference, in degrees, between the predicted orientation and the actual orientation. Let $\mathbf{Q_{actual}}$ be the quaternion that represents the actual orientation and $\mathbf{Q_{predicted}}$ be the quaternion representing the predicted orientation. Then the angular error E_{ang} (in degrees) is computed as follows:

$$\mathbf{Q_{diff}} = \mathbf{Q_{actual}} \bullet \left(\mathbf{Q_{predicted}}\right)^{-1}$$

$$E_{ang} = \frac{2(180)}{\pi} \cos^{-1}\left(\mathbf{Q_{diff}}[0]\right)$$

The $\cos^{-1}()$ function returns values in radians, and $\mathbf{Q_{diff}}[0]$ is the *qw* term from the $\mathbf{Q_{diff}}$ quaternion.

Position error is the difference, in millimeters, between the predicted position and the actual position. Screen error measures errors based upon what the user sees, so it is perhaps the best overall metric. Screen error is

the difference, in pixels, between the 2-D coordinates of the intersection of the three virtual axes and the 2-D projection of the real corner of the crate. This is computed in simulation. An X-window program draws both the "real" crate, based on the reported head locations, and the virtual axes, based on the predicted head locations, on a hypothetical 512 by 512 screen covering a 30 degree FOV.

The angular, position and screen error metrics trust that the reported head locations from the head tracker are the true locations, so these metrics only measure dynamic registration error, not static error. The program that calculates screen error computes the projection of the crate based on the reported head locations and the measured crate location. The simulator assumes these measurements are exact and that the see-through HMD is perfect, resulting in perfect static registration. While this does not match reality, it makes it possible to separate the effects of dynamic registration error from static registration error.

I recorded three motion sequences that I consider representative for testing the registration of the virtual axes with the corner of the real wooden crate. These sequences are called Walkaround, Rotation, and Swing. During each motion sequence, the user kept the corner of the crate visible in the field-of-view. In the Walkaround sequence, the user slowly walks around the corner of the crate, using motions similar to the 270 degree walkaround used to test static registration in Chapter 3. The Rotation sequence has the user yawing, pitching, and circling his head while standing in one place. The Swing motion sequence combines fast orientation and translation motions as the user aggressively moves and turns his head to look at the corner of the crate from many different vantage points during a short time span. The user does not stand in one place during the Swing motion sequence. The position and orientation traces are shown in Figures 4.17 - 4.22.

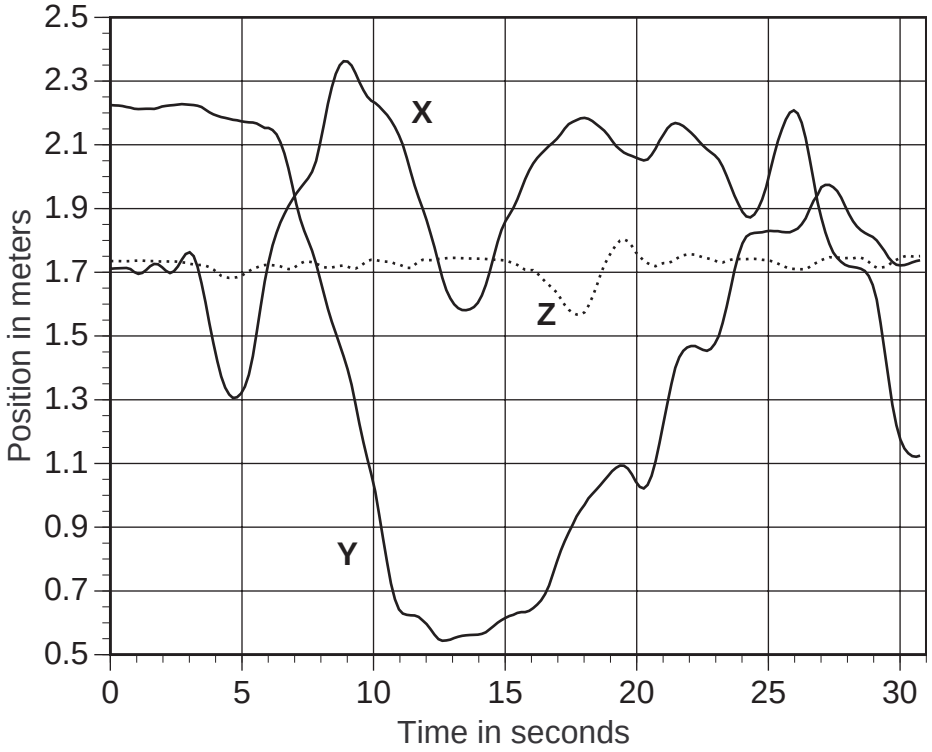


Figure 4.17: Walkaround motion sequence: Translation curves

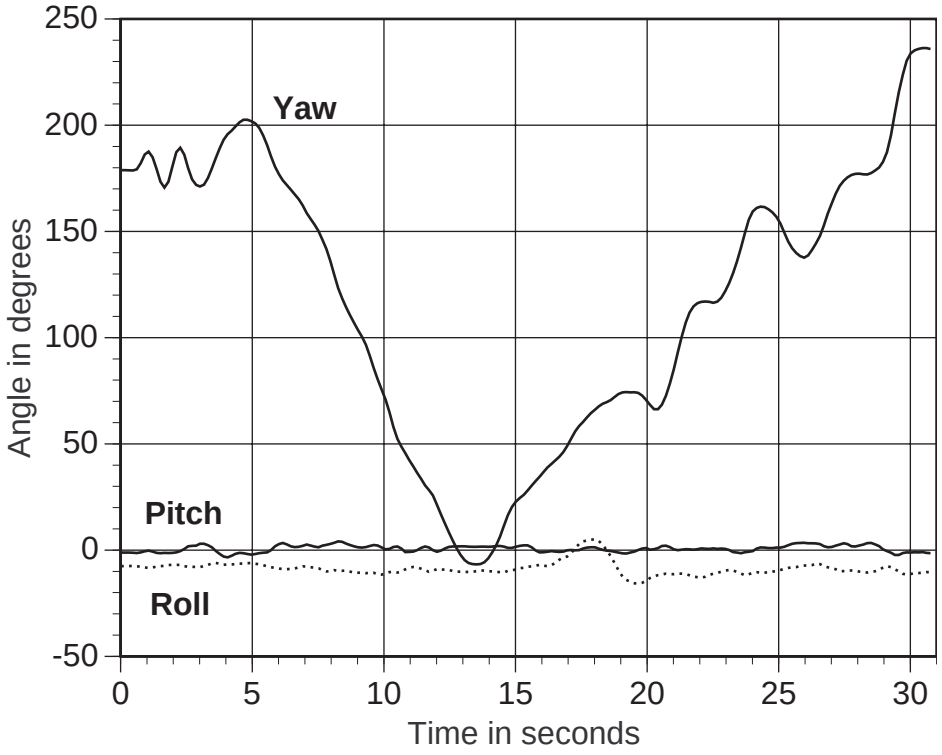


Figure 4.18: Walkaround motion sequence: Orientation curves

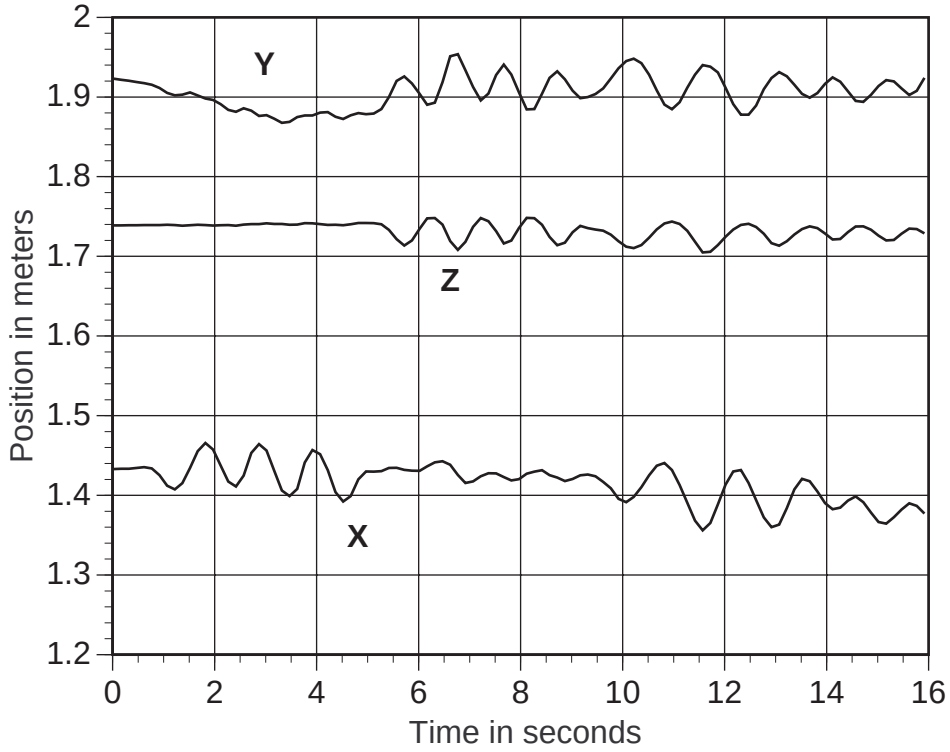


Figure 4.19: Rotation motion sequence: Translation curves

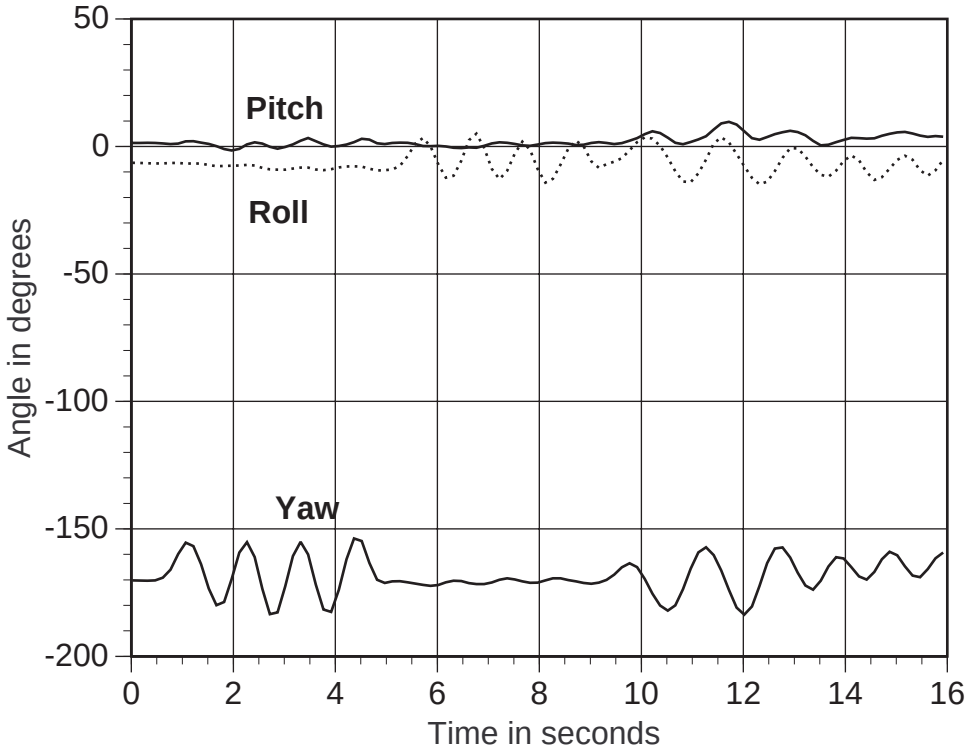


Figure 4.20: Rotation motion sequence: Orientation curves

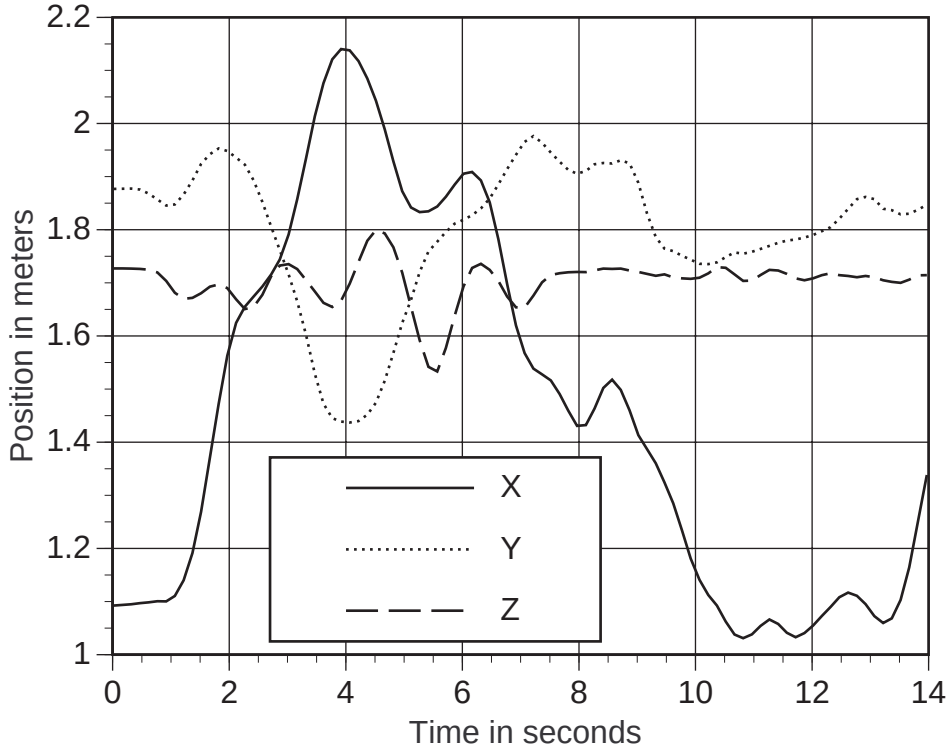


Figure 4.21: Swing motion sequence: Translation curves

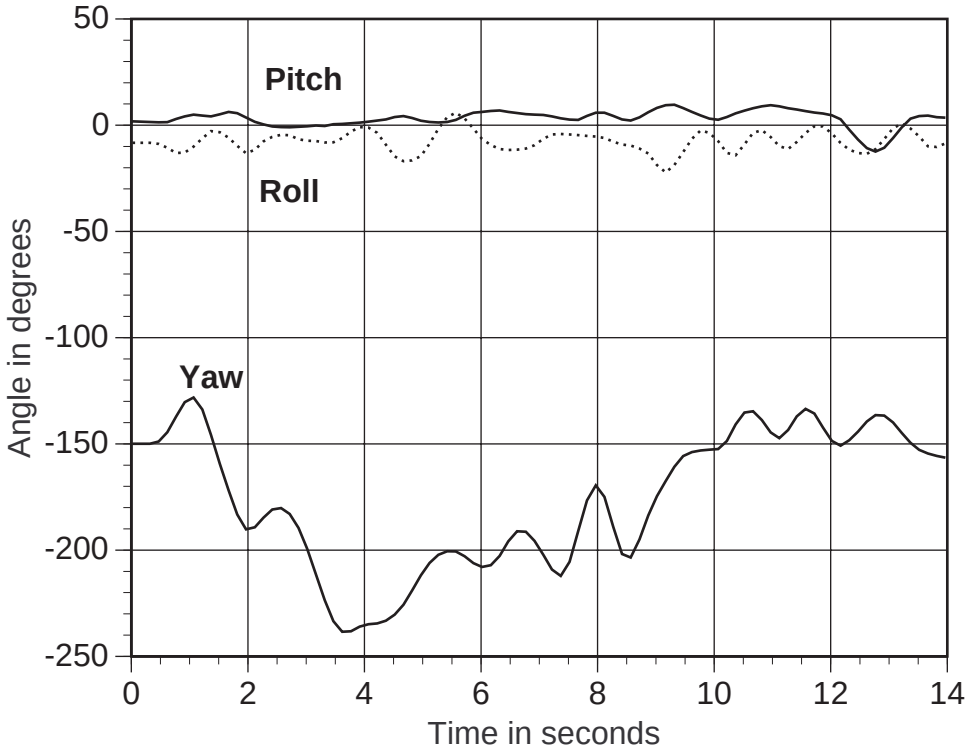


Figure 4.22: Swing motion sequence: Orientation curves

I compared my inertial-based predictor against doing no prediction and against doing prediction without the use of inertial sensors on these three motion sequences. After examining all previous works that performed head-motion prediction without inertial sensors and implementing some of them, I found that predictors of the form that Alberta [Liang91] used gave the most accurate prediction. Therefore, that is the basis of comparison. Note, however, that this is not a comparison between my system and Alberta's, because such cross-system comparisons are difficult to do. They have a completely different scene generator, HMD, tracker, and application. Instead, this comparison is within my system framework, where the only difference is the prediction module used. All the system details covered in Chapter 5 that improve prediction accuracy also apply to my implementation of non-inertial prediction, which is based on the predictor described in the Alberta paper. I found parameters that worked well on several different motion sequences, using the same search techniques as those used to find the **E** and **R** matrices for my predictor, as described in Section 5.3.2. Thus, the non-inertial predictor that I compare against is not a representative implementation of previous work; it is an improved version of previous work. What is really being determined by this comparison is how much inertial sensors improve prediction accuracy when the predictor uses a simple motion model.

This comparison is performed by offline simulations on the three recorded motion sequences, with the results displayable in an X-window program that simulates what the user would see. The predictors were also implemented and run in real time in the actual system, with the results captured on video. The simulation results are comparable to the video taken in the actual system. The advantage of the simulator is that it allows head-to-head comparisons on *exactly* the same motion sequences, something that is difficult to arrange in the actual system.

	Walkaround			Rotation			Swing		
	Ang	Pos	Screen	Ang	Pos	Screen	Ang	Pos	Screen
No prediction	**1.3**	**14.3**	**9.3**	**2.2**	**6.6**	**33.6**	**2.5**	**17.8**	**37.1**
	4.3	**38.0**	**62.0**	**5.3**	**17.6**	**92.1**	**6.5**	**46.0**	**118.6**
Prediction without Inertial	**0.2**	**2.5**	**4.5**	**0.6**	**3.3**	**13.6**	**0.6**	**5.2**	**16.2**
	0.8	**9.0**	**26.7**	**1.6**	**11.7**	**51.0**	**1.8**	**17.1**	**62.8**
Prediction with Inertial	**0.1**	**1.1**	**2.7**	**0.18**	**1.6**	**5.2**	**0.2**	**2.7**	**7.2**
	0.4	**6.1**	**15.1**	**0.57**	**9.8**	**36.1**	**0.7**	**17.8**	**30.1**

Average error Peak error

Angular error in degrees
Position error in mm
Screen error in pixels
Prediction interval set at constant 60 ms for all runs

Table 4.1: Summary of prediction errors on three motion sequences

Table 4.1 summarizes the average and peak errors for all three objective error metrics on the three recorded motion sequences. The average inertial-based prediction errors are lower by a factor of 5-10 than the average "no prediction" errors. The average inertial-based prediction errors are also lower by a factor of 2-3 than the average non-inertial prediction errors, using any of the error metrics. Figures 4.23 - 4.25 show the actual error curves for the Swing motion sequence. Figures 4.26 - 4.28 show a small segment of the yaw orientation curve from the Swing motion sequence with overlaid predicted yaw curves that result from no prediction, non-inertial prediction, and inertial-based prediction. Figures 4.29 - 4.31 do the same for a small part of the *Z* translation curve from the Swing motion sequence.

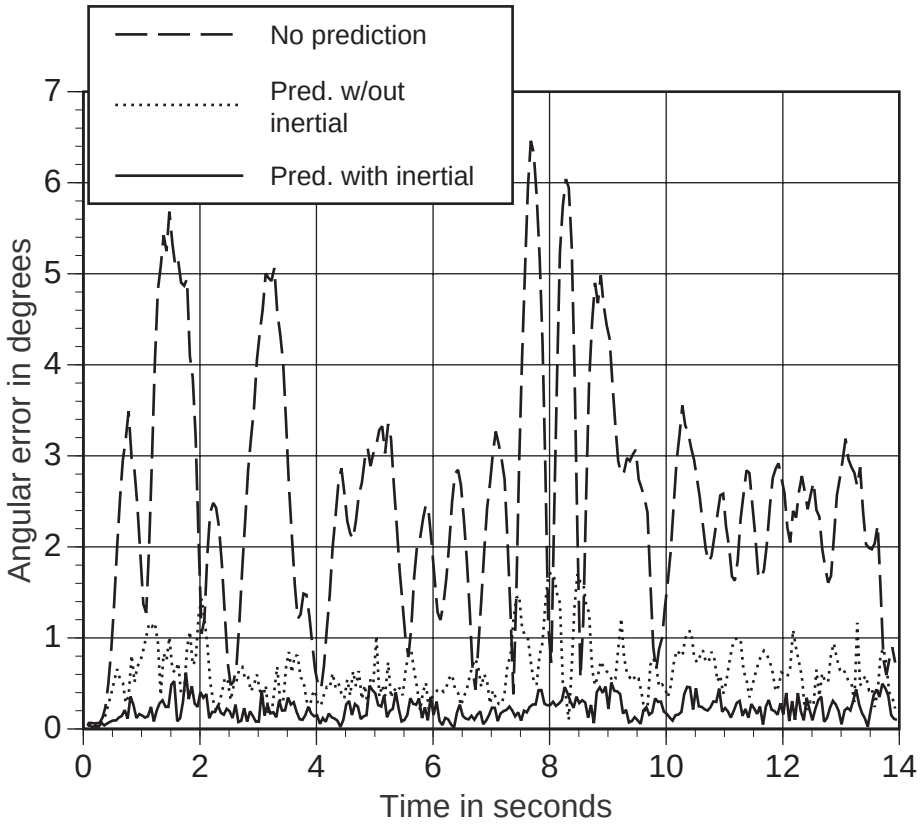


Figure 4.23: Angular errors for Swing motion sequence

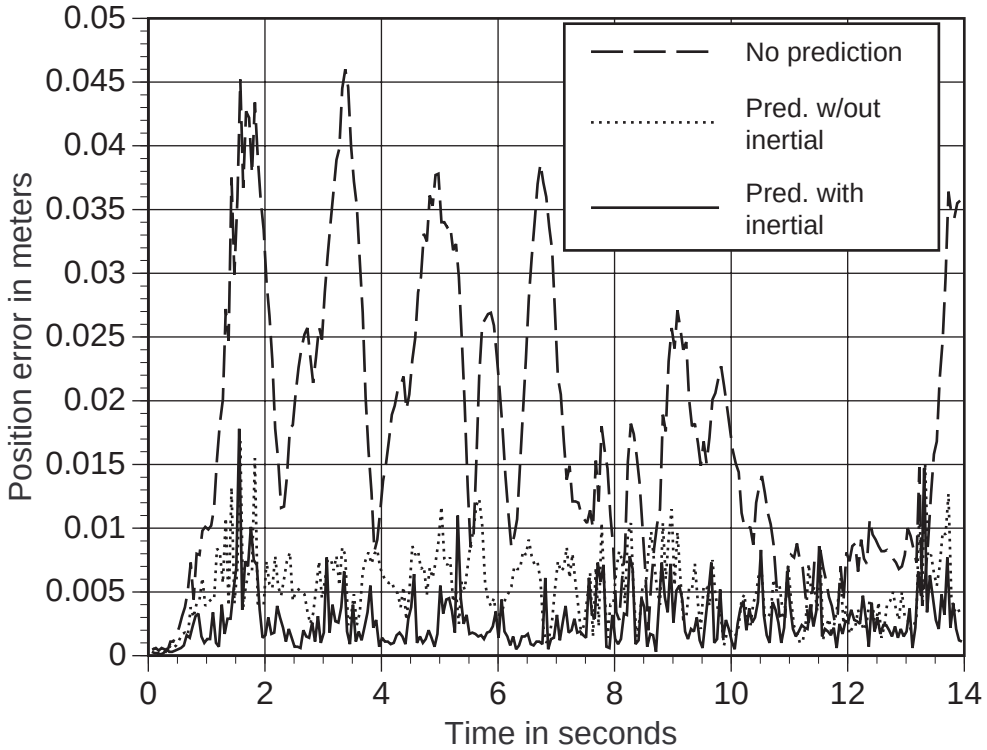


Figure 4.24: Position errors for Swing motion sequence

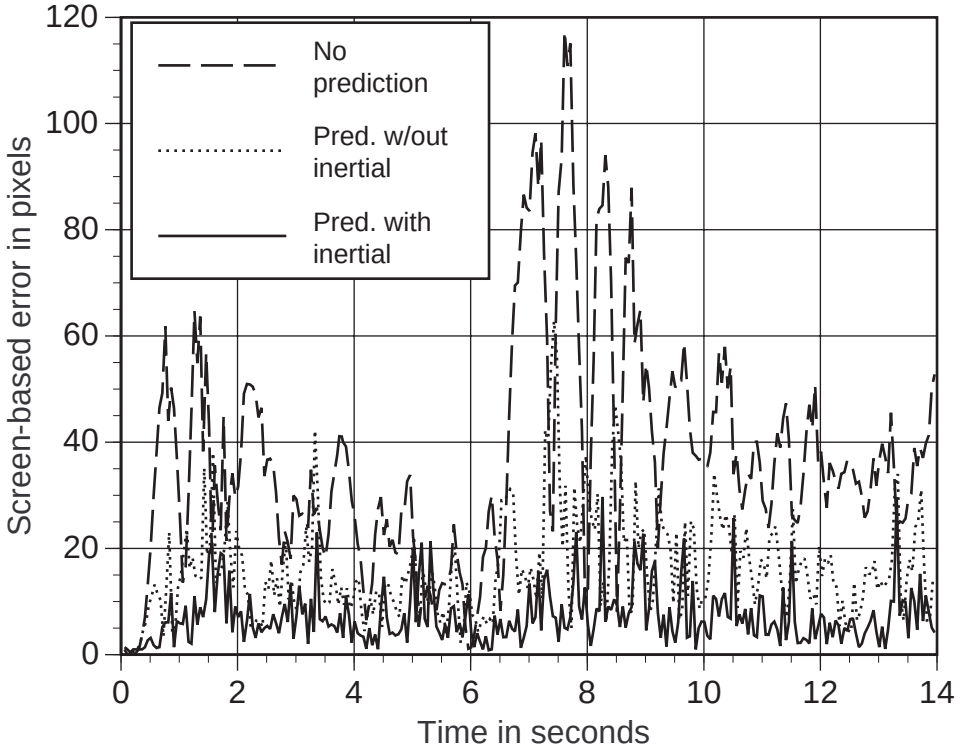


Figure 4.25: Screen errors for Swing motion sequence

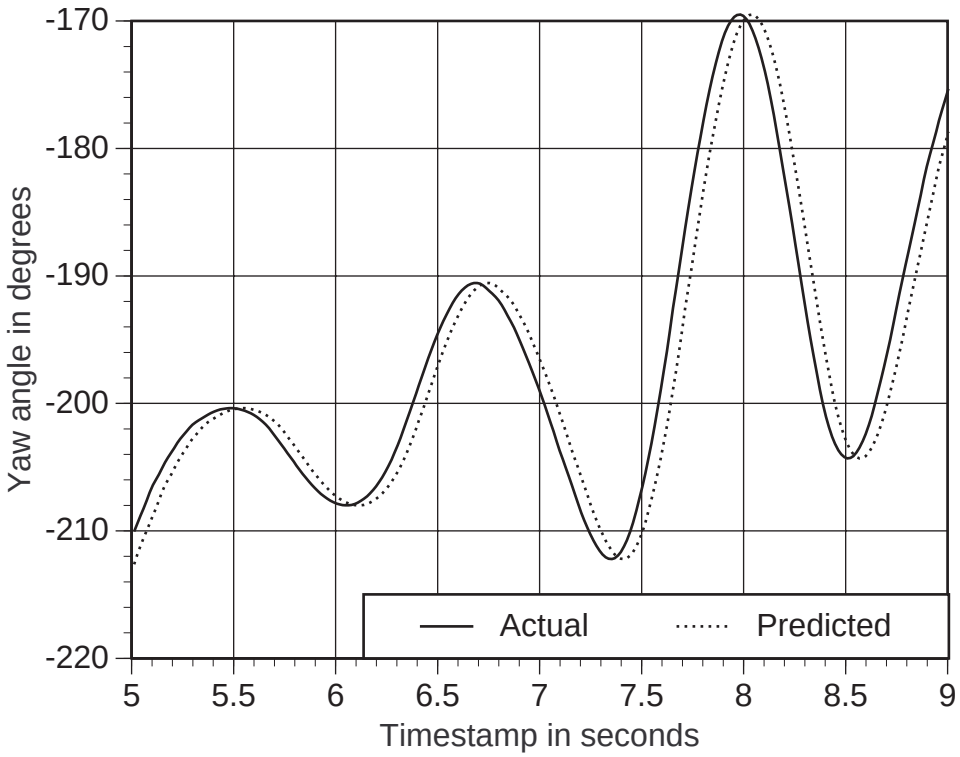


Figure 4.26: Yaw curve with no prediction

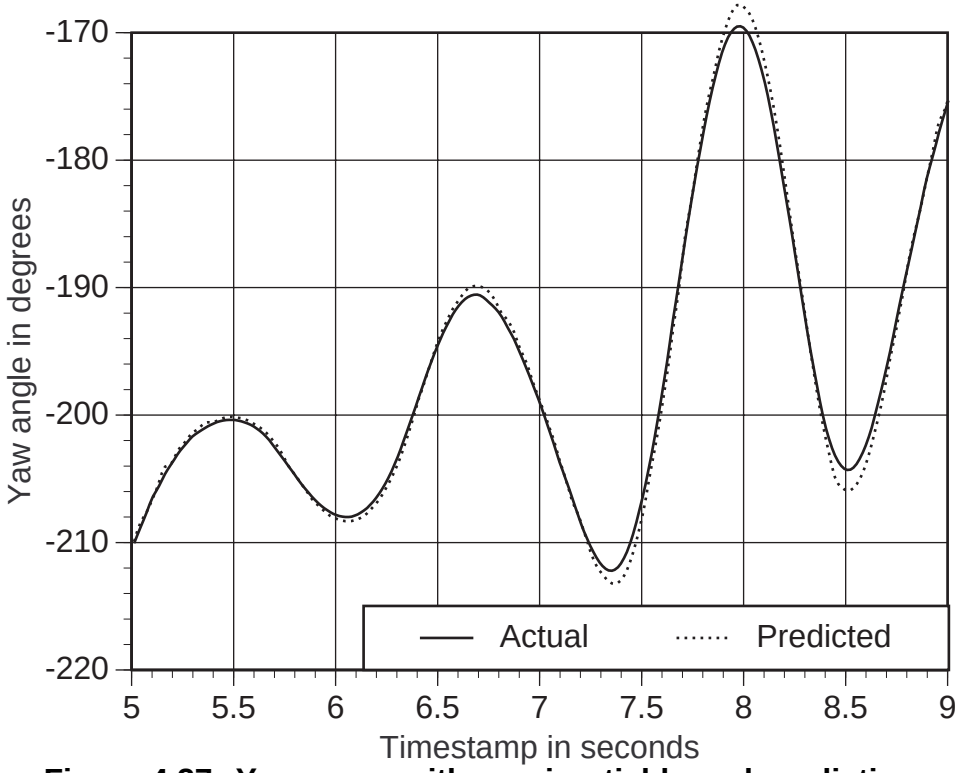


Figure 4.27: Yaw curve with non-inertial-based prediction

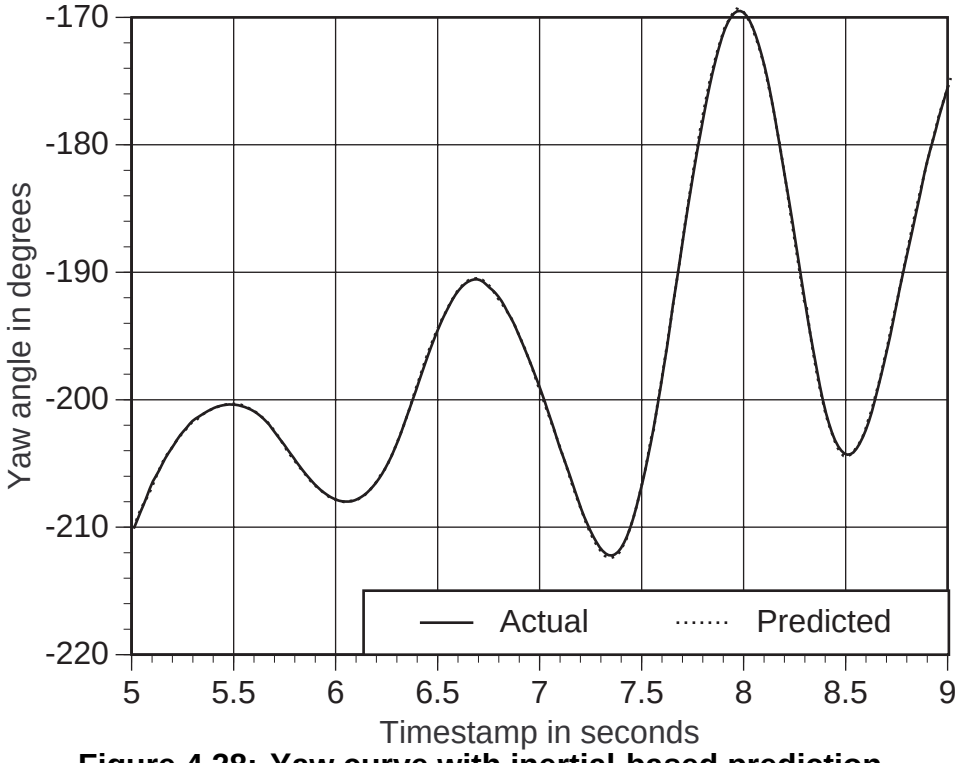


Figure 4.28: Yaw curve with inertial-based prediction

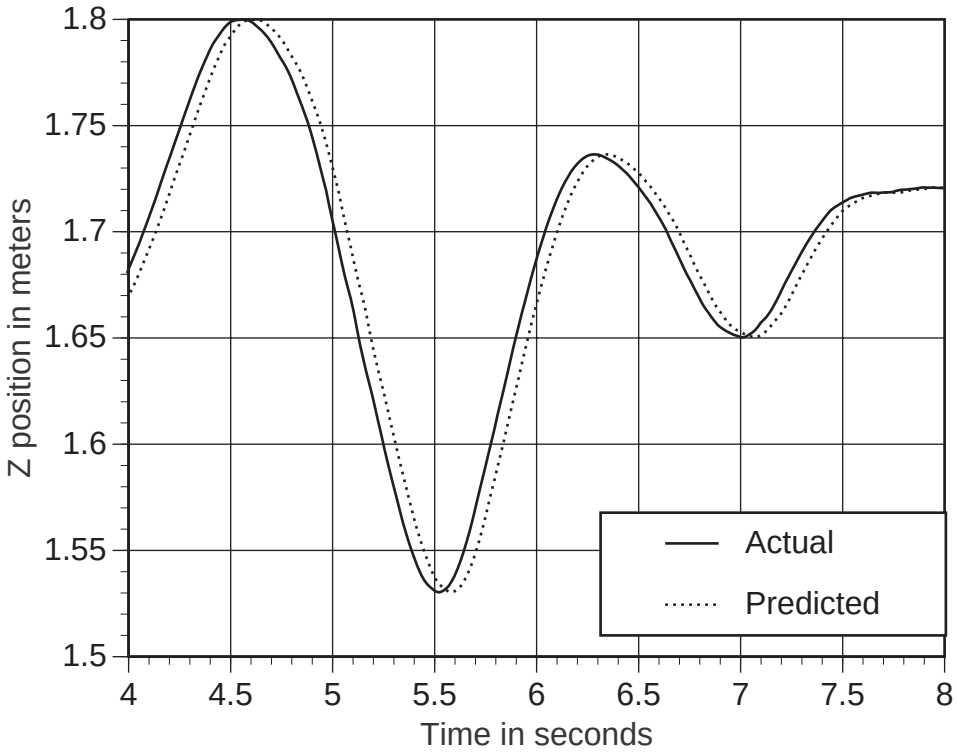


Figure 4.29: Z curve with no prediction

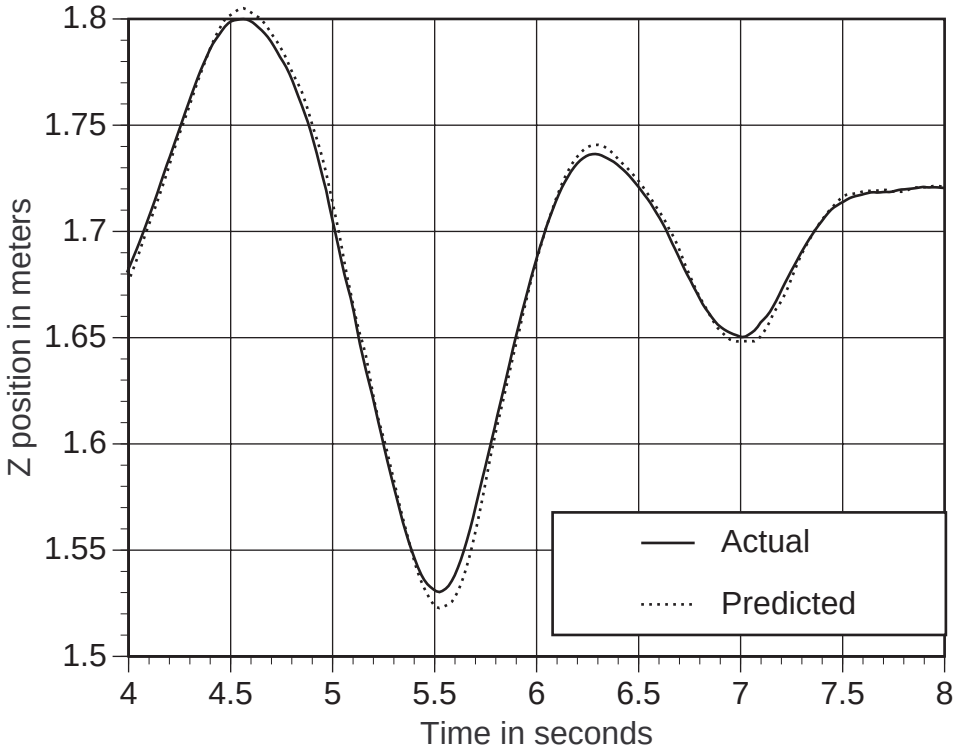


Figure 4.30: Z curve with non-inertial-based prediction

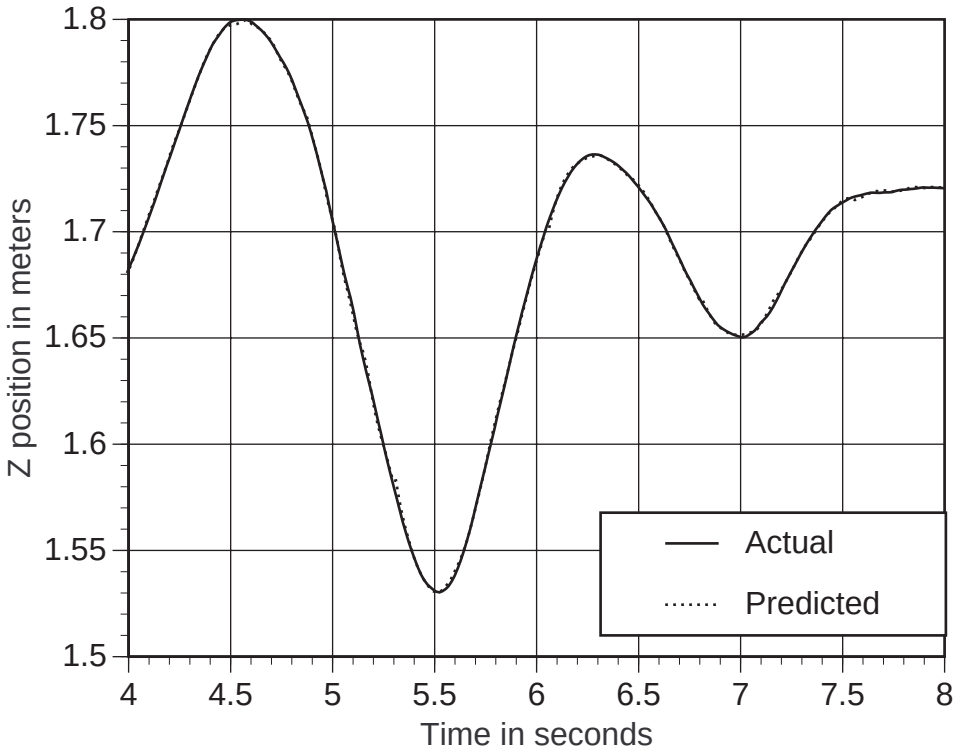


Figure 4.31: Z curve with inertial-based prediction

Another way to view the screen-based errors is in a *scatterplot* diagram. Imagine a point one meter in front of the user's right eye. This point is rigidly attached to the user's head, so that no matter how the user turns and moves his head, that point is always one meter in front of his right eye. Ideally, the projection of this imaginary point should always lie in the center of the field-of-view. However, system delays cause this projection to occur away from the center. I can take a motion sequence and plot where these projected points occur in screen space, for the cases of no prediction, non-inertial-based prediction, and inertial-based prediction. The distribution of these points indicates how much error the user sees. The wider the spread of points, the larger the error is. Note that this is similar to, but not the same as, the screen errors listed in Table 4.1. Those screen errors are based on a single, static World-space point (the corner of the crate), which is assumed to always be in the field-of-view. The scatterplot metric keeps the viewing distance constant and does not require that the user always look at a single point in World space, making it a more versatile metric.

The next four figures show scatterplots for the Rotation and Demo1 motion sequences. The prediction interval is a constant 80 ms for all the

graphs, on a hypothetical 512 by 512 display covering a 30 degree FOV. The points were evenly sampled out of the original sequence, with one-third of the points used in the Rotation graphs and one-twentieth of the points used in the Demo1 graphs. Figures 4.32 and 4.33 show scatterplots for the Rotation sequence, comparing no prediction, prediction without inertial sensors, and prediction with inertial sensors. The two figures are identical, except that Figure 4.33 is in color. Note that the yaw, pitch, and circling motions in the Rotation run appear as a broad horizontal band, a vertical band, and circular rings, respectively. This structure in the scatterplots is due to the highly structured testing motions. In contrast, the Demo1 sequence was taken from a first-time user running a demonstration application. It is the same sequence shown in Figures 4.1 through 4.4. Now the scatterplots in Figures 4.34 and 4.35 are more evenly distributed. Those two figures are identical, except Figure 4.35 is in color. For the Rotation sequence, inertial-based prediction reduces average errors by a factor of 3.5 over non-inertial-based prediction and by a factor of 9 over no prediction. For the Demo1 sequence, inertial-based prediction reduces average errors by a factor of 2 over non-inertial-based prediction and by a factor of 6 over no prediction. The difference between the two sequences is that the average rates of motion are higher in the Rotation sequences, so the factors are higher.

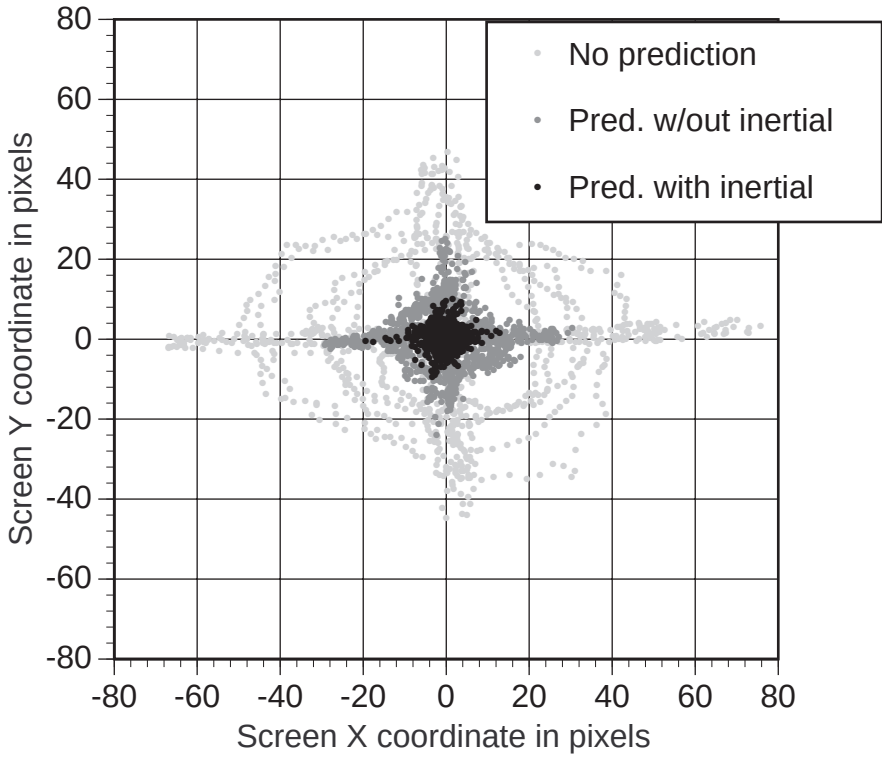


Figure 4.32: Rotation sequence: Scatterplots for no prediction, non-inertial-based prediction and inertial-based prediction (B/W)

Figure 4.33: Rotation sequence: Scatterplots for no prediction, non-inertial-based prediction and inertial-based prediction (Color)

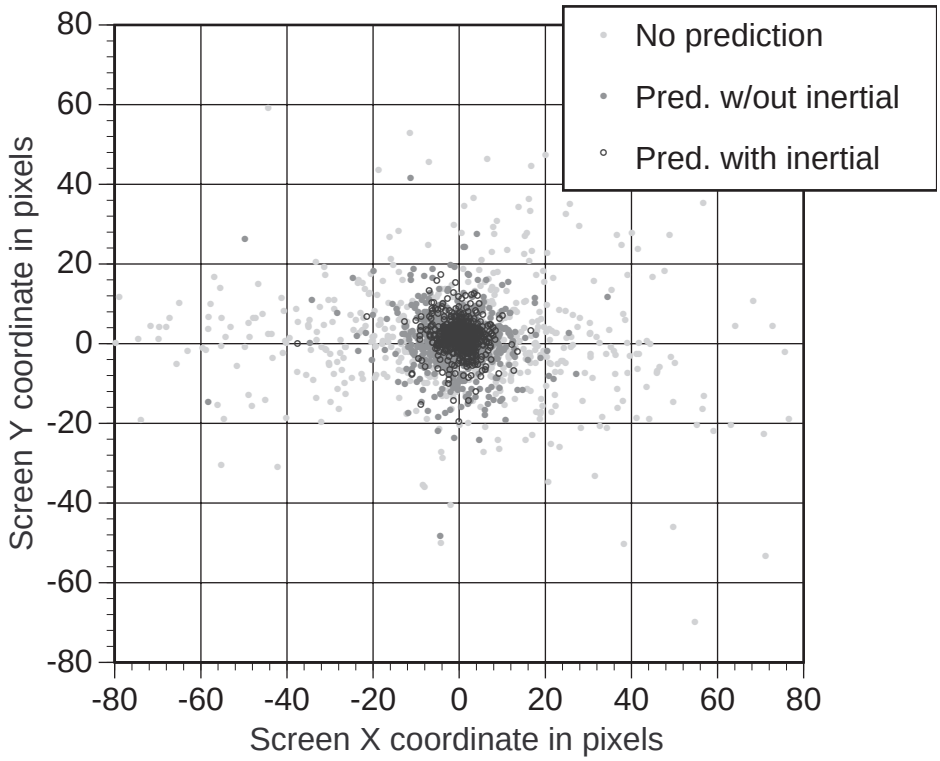


Figure 4.34: Demo1 sequence: Scatterplots for no prediction, non-inertial-based prediction and inertial-based prediction (B/W)

Figure 4.35: Demo1 sequence: Scatterplots for no prediction, non-inertial-based prediction and inertial-based prediction (Color)

One problem with using the objective error metrics is deciding what to use as the true, actual curves, because the readings reported by the head tracker are noisy and have small distortions in them. If compared directly against the predicted values, the errors in the head tracker signal will be counted as errors in the predictor. That is the case in the scatterplot diagrams shown above. Compensating for the distortions in the head tracker output is very difficult, but it is possible to reduce the noise. I chose to run the tracker positions and orientations through a lowpass filter with a 10 Hz cutoff frequency. Since the filtering occurs offline, I can use a noncausal lowpass filter that does not introduce any phase shifts into the signal [NASAksc]. The result is considered the actual signal that is compared against the predicted outputs for the results listed in Table 4.1.

How much do prediction errors grow as the prediction intervals increase? Figure 4.36 shows a typical result with the Rotation motion sequence. It shows how the average screen error changes for no prediction, prediction without inertial sensors, and prediction with inertial sensors, as the prediction interval changes from 25 ms to 200 ms in steps of 25 ms. While this figure is for one specific motion sequence, the results from other motion sequences and error metrics are quite similar to this one. This curve is another measure of how much inertial sensors help in the prediction task. While the "no prediction" curve grows linearly with time, the predicted curves grow at faster than linear rates. By the 500 ms mark, both types of predictions would be less accurate than doing no prediction at all. Thus, prediction is not effective at long prediction intervals.

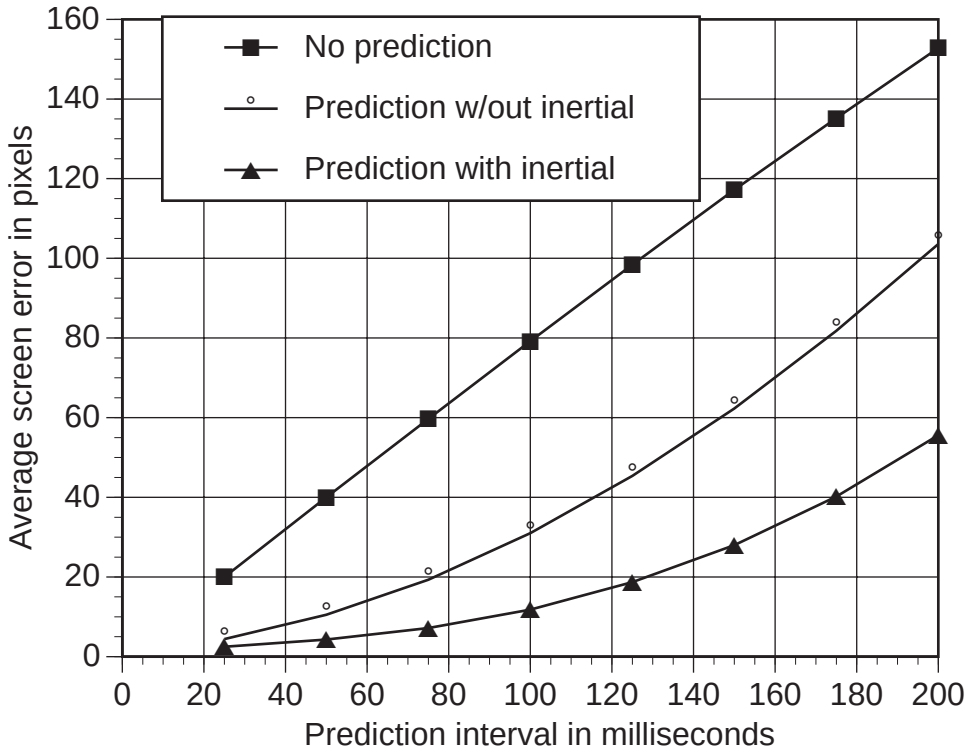


Figure 4.36: Average error versus prediction interval

Besides prediction errors, the jitter in the predicted signals also grows as the prediction interval increases. Jitter refers to noise-like high-frequency signals in the predicted outputs that are not representative of the original head motion. Jitter is disturbing to the viewer; it resembles motion that one might expect from a head that was trembling or shaking at a rapid rate. Figure 4.37 demonstrates what jitter looks like. This figure graphs a small segment of a pitch curve with an overlaid prediction generated at a prediction interval of 130 ms. The high-frequency oscillations in the predicted curve are jitter. Jitter occurs because the predictor amplifies the high-frequency components in the original signal. This is explained in the frequency-domain analysis performed in Chapter 6.

Removing jitter proved impossible because that adds too much delay to the predicted signal. Reducing jitter requires suppressing a range of frequencies. Any such method must examine the signal for a long enough time to identify those frequency components. Low frequencies require longer periods than high frequencies. This examination time causes a delay in the filtered signal. It turns out that the frequencies that must be suppressed are low enough that identifying them adds delay on the order of tens of

milliseconds. Recall that delay is what the predictor is trying to remove in the first place! I tried several approaches to reduce jitter, but anything that significantly reduced jitter also significantly hurt prediction accuracy. In practice, the only way to keep jitter within tolerable levels is to keep the prediction interval short.

Empirically, in my system it is best to keep system delays to ~80 ms or less. The actual system has a total end-to-end delay that usually varies between 50 and 70 ms. Thus, my predictor will not be useful in AR systems that have 200 ms of delay. To be effective, prediction must be used in conjunction with efforts to minimize system delay.

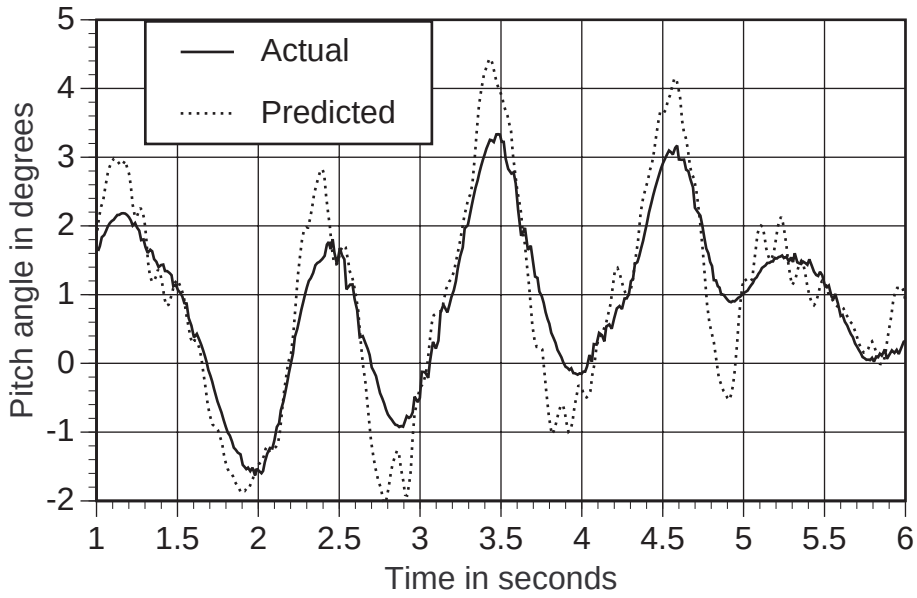


Figure 4.37: Jitter in predicted pitch curve

Making the theory agree with real data proved harder than expected. Defining a relationship, using that to generate simulated data, then running the data through an algorithm is not a difficult test. The data and the math will agree, because the equations used to generate the data are the same as the ones in the algorithm. Making an algorithm work on data collected from real sensors is a different story. For example, I struggled with several references that gave different definitions for the derivative of a quaternion, none of which matched my collected quaternions and omegas. I learned that this definition changes depending upon what direction of rotation the quaternion represents and what space the omegas are represented in. It took a while to find a source that agreed with my collected data [Chou92]. A significant amount of

my effort went into finding these relationships so that the math would work on collected data.

Simplifying the predictor by breaking it into four different Kalman filters makes the predictor much faster. Let *N* be the number of variables in the state vector **X** and *F* be the number of measurements in **Z**. Several matrix multiplications involving *N* by *N* and *N* by *F* matrices are required, and one *F* by *F* matrix inversion is needed. Therefore, the cost of operating a Kalman filter is a function of *F* cubed or *N* cubed, whichever is larger. On an i860, the three translation Kalman filters (*N* = 3) take less than one ms combined to process a new measurement, while the orientation filter (*N* = 10) requires nearly 10 ms. Putting everything into one filter (*N* = 19) makes the predictor too slow. The speed gains are important, because the prediction problem gets much harder with increasing system delay.

Using four separate Kalman filters also potentially hurts prediction accuracy, because that prevents modeling relationships between the separated terms. For example, the accelerometers detect angular acceleration and gravity, which could help the orientation filter. Also, any correlations between the orientation and translation terms cannot be accounted for. This is a potentially serious omission, because the origin of Tracker space does not lie along the center of rotation. That means that as the user rotates his head, the position of the tracker origin changes. These two motions are strongly correlated, so a more sophisticated predictor should exploit that.

4.6 Future directions

While my predictor is sufficient to support the thesis statement, more accurate predictors may be possible, especially if the predictor can be tuned to a specific application. Increasing prediction accuracy basically means building a more accurate motion model. This section shows how the models I use approximate motion and what might be done to improve them.

One way to examine models is to see how well they extrapolate velocity. If a predictor somehow knew exactly what the future velocities were

during the entire prediction interval, then the predictor could integrate those velocities to generate essentially perfect predictions. Deviations away from the correct future velocities result in prediction errors. I can graph how the non-inertial-based and the inertial-based predictors extrapolate velocity.

The non-inertial-based predictors have state vectors that only contain position and velocity terms. They usually assume that velocity is constant across the prediction interval. Figure 4.38 shows what this looks like. The extrapolated velocities, drawn as dashed lines, are overlaid against a 1-D angular velocity curve. This curve is the $\omega 2$ component of omega, representing the instantaneous angular velocity along the yaw direction in Tracker space. Note that the approximations to velocity are flat lines, and that they do not match the original curve well.

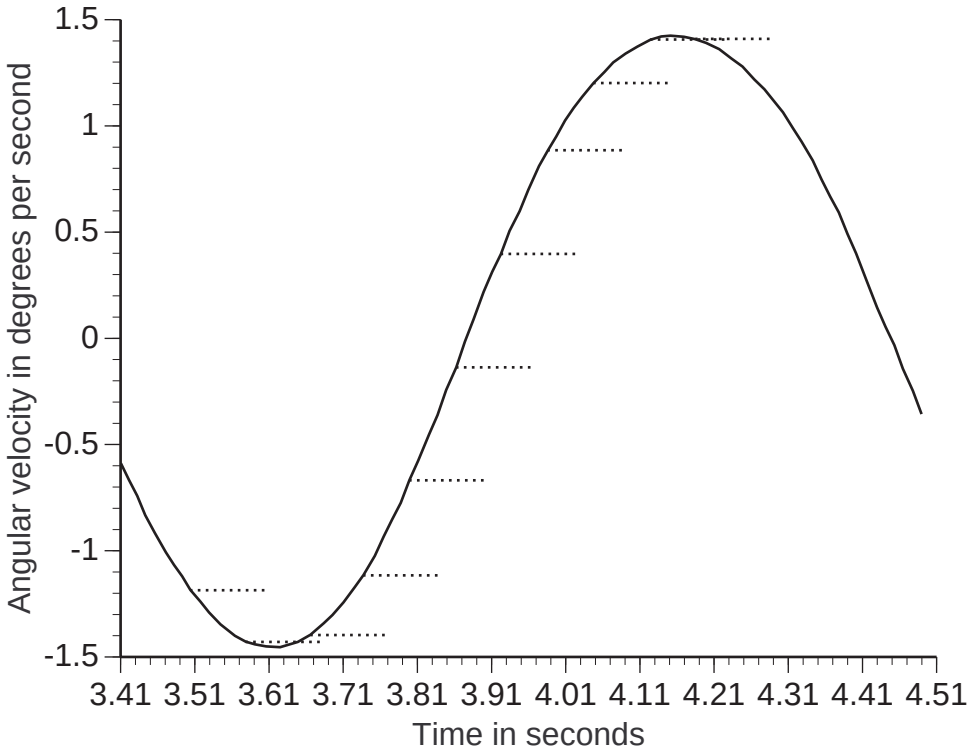


Figure 4.38: Constant predicted velocities

The inertial-based predictor includes acceleration in the state vector, making the velocity approximations more accurate. Including measured velocity or acceleration from the inertial sensors allows reliable estimates of acceleration to be included in the state vector. The motion model assumes that acceleration is constant across the prediction interval. Figure 4.39 shows

what these approximations look like. Again, selected extrapolated velocities are drawn as dashed lines on top of the original angular velocity curve. Note that the estimated velocities are now lines with constant slope. The match is better, but not always accurate. They match most closely when the curve can be adequately modeled as a straight line across the entire prediction interval.

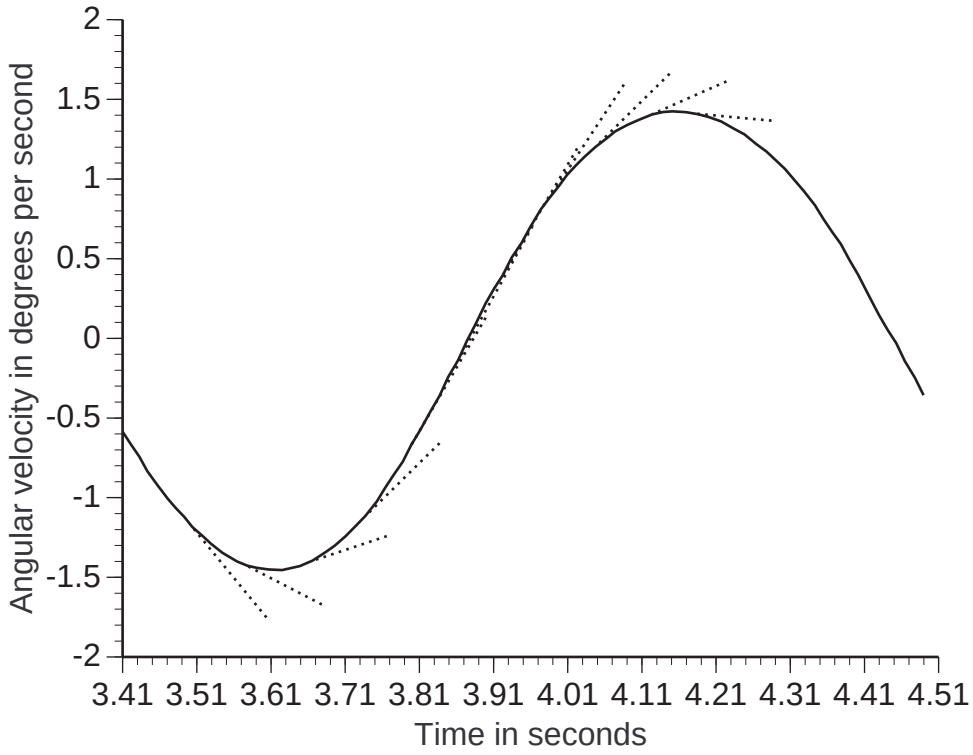


Figure 4.39: Linear predicted velocities

It is not practical to include more than one additional derivative in the state vector. Including more derivatives beyond acceleration in the state vector would allow quadratic, cubic, and higher-order curves to estimate future velocities. Much like including additional terms in a Taylor series, these derivatives theoretically should improve the velocity estimates. However, that is true only if accurate measurements or estimates of those higher derivatives are available. No sensors exist that directly detect derivatives above acceleration. In practice, it is only possible to estimate at most one derivative above what the sensors directly detect, because numerical differentiation is a noisy operation. Therefore, the most that can be added to the state vector is the derivative of acceleration (sometimes called *jerk*). Adding more terms also increases the amount of jitter in the predicted signal, so it is something to be carefully considered.

Instead of increasing the state vector, change the assumption that acceleration is constant across the prediction interval. This assumption is the result of setting the derivative of acceleration to zero. Figures 4.40 and 4.41 show some estimated acceleration curves for orientation and translation, based on captured data. Clearly, acceleration rarely stays constant. The Alberta model assumes acceleration is proportional to velocity with added white noise, but that model is no more accurate than the assumption that acceleration is constant. In fact, the acceleration curves change rapidly enough that I do not believe it possible to perfectly predict acceleration across the required prediction intervals of 50 to 80 ms. About the only hope is to modify the model of constant acceleration to something that is more accurate on average.

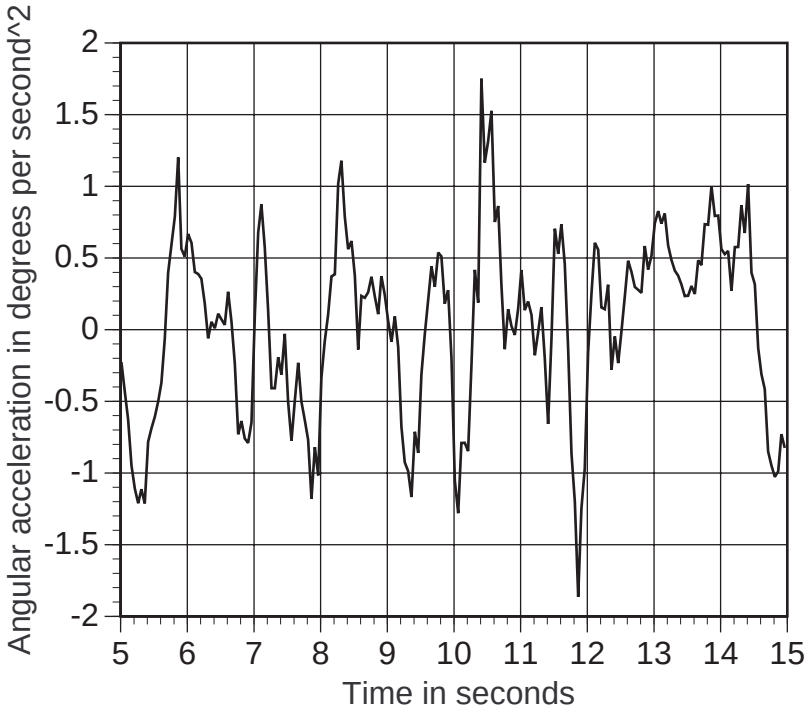


Figure 4.40: Estimated angular acceleration

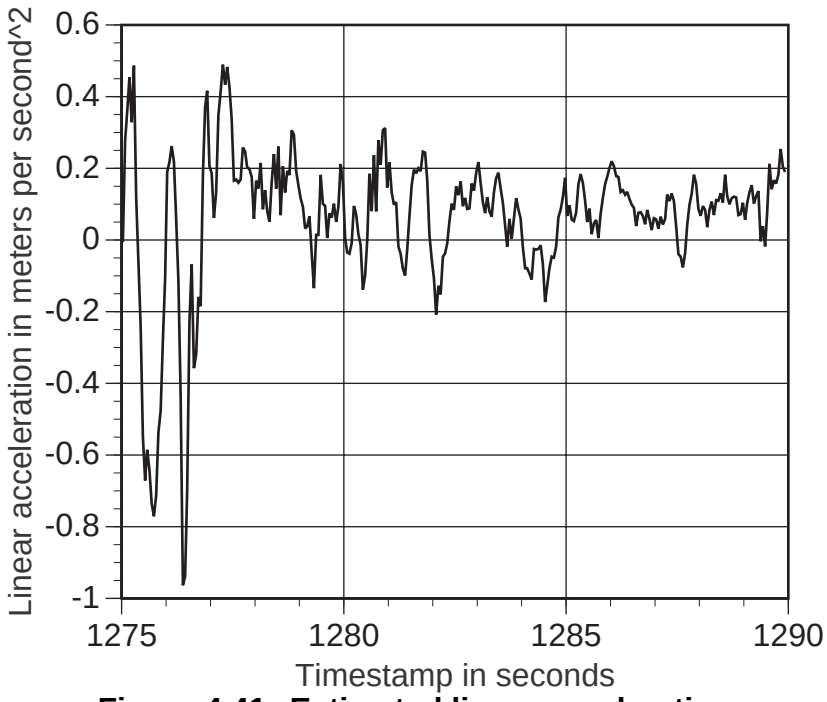


Figure 4.41: Estimated linear acceleration

A Bayesian approach might produce more accurate models for specific applications. The extrapolated acceleration is constant across the prediction interval because the predictor assumes that the probability that acceleration will increase is equally likely as the probability that it will decrease. Figure 4.42 shows what the range of possible accelerations looks like, based on this assumption. In this situation, the estimate that minimizes the expected mean-square error is simply the last known acceleration. That is, acceleration remains constant. The assumption of even probability distributions is essentially Fischerian. However, it may not be the case that the probability distribution of future accelerations is even, as the white noise model suggests. In that case, a colored noise model should be used which biases the expected direction of acceleration. This colored noise model must be statistically derived from recorded motion sequences, rather than high-level assumptions. The colored noise model can be thought of as a Bayesian prior that changes the estimate that produces the minimum expected mean-square error.

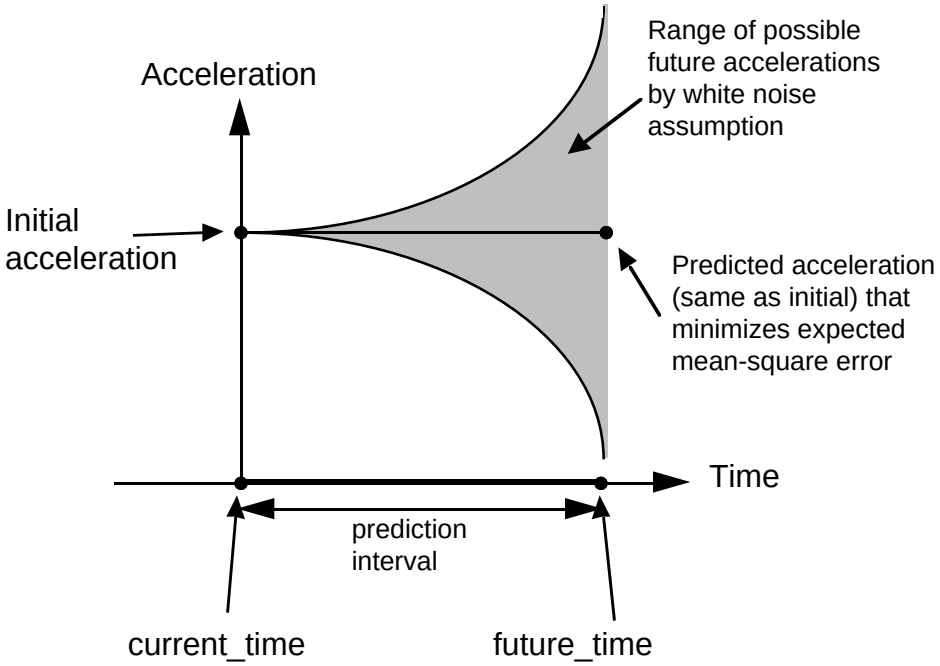


Figure 4.42: Why the acceleration estimate is constant

Since head motion is nonstationary, it may be possible to generate several models and switch between them adaptively. The framework for adaptive filters is straightforward [Magill65]. A common method is to run several Kalman filters in parallel, each with a different model, then choose one output or blend the various outputs. These are sometimes called Multiple Model or Generalized Likelihood techniques. The difficult part of adaptive filters is not in the adaptive framework; it is in building the models in the first place.

Tuning models too tightly to specific motions can be dangerous. When that happens, prediction will be highly accurate if the actual motion matches expectations. But if the actual motion is different from what the model expects, errors can become very large. In target-tracking applications this is sometimes referred to as "losing lock" on the target. Therefore, a tradeoff will exist between generalized models that have larger average errors versus more specialized models that have smaller average error but large peak errors. Because the characteristics of head motion can vary considerably, I investigated simpler, more general models that avoid large peak errors. However, the choice may depend upon the application being used.

Specializing prediction for a particular application has the potential to improve accuracy, at the cost of generality.

5. System and prediction details

Chapter 2 provided an abstract, high-level description of the Augmented Reality system. That was enough background to understand the static and dynamic registration techniques described in Chapters 3 and 4. However, understanding how the prediction routine described in Chapter 4 is actually implemented in the real-time system requires more detailed knowledge of the system than Chapter 2 provided. This chapter provides those details and describes the problems and lessons learned from implementing the predictor as part of an operational Augmented Reality system.

5.1 System details

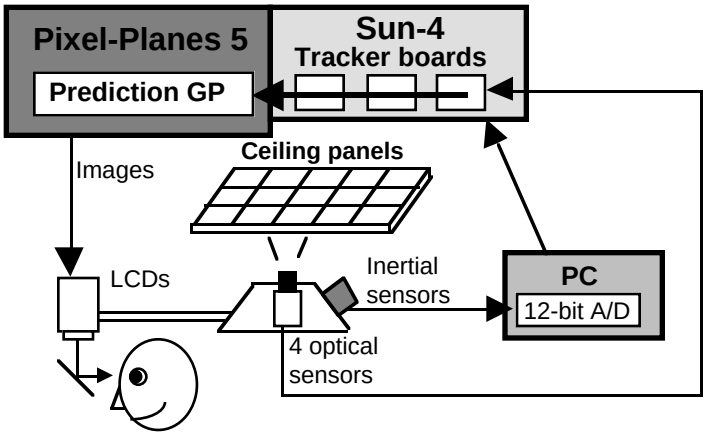


Figure 5.1: Overall system diagram

Figure 5.1 shows a high-level overall view of the system. This figure provides more detail than the conceptual system diagram in Figure 2.5. I now step through the basic system dataflow shown in Figure 5.1. Four optical sensors on the HMD view LEDs mounted in special ceiling panels above the user's head. The head-mounted inertial sensors are read by an A/D board in a personal computer. Both the optical sensor and inertial measurements are

sent into the three single-board computers that run the optoelectronic tracker. These computers determine the head position and orientation, given the measurements from the optical sensors. The computed head location and the inertial measurements are sent into Pixel-Planes 5, UNC's custom-built scene generator. Pixel-Planes 5 is a parallel machine, containing several Graphics Processors (GPs) in a MIMD arrangement. One of the GPs is reserved for the prediction computations. This prediction GP reads the head location and inertial measurements, predicts a future head location, and sends that to the other parts of Pixel-Planes 5 to generate a new set of images that is displayed in the HMD.

The next several subsections explain the individual components separately: the HMD, inertial sensors, optoelectronic tracker, Pixel-Planes 5, and the communication paths. Then this section ends by showing how all the parts connect together, in detail.

5.1.1 Optical see-through HMD

The optical see-through HMD was introduced in Chapter 3. Doug Holmgren built it using mostly off-the-shelf parts [Holmgren92], so that researchers at UNC would have a see-through HMD to experiment with until better ones arrived. Other see-through HMDs existed at UNC before Holmgren's, but none of those were as reliable, adjustable, and rigid. Figure 5.2 shows a front view of the HMD. Figures 3.8 and 3.9 show the HMD as I use it, equipped with optical and inertial sensors. The field-of-view in each eye is approximately 30 degrees. The optical combiners transmit about 30% of the light from the real environment. Each LCD display is a color Sony Watchman monitor, with 240 rows of 340 individual pixel elements. Since a triad of elements is required to represent one full-color pixel, the effective true horizontal resolution is about 113 pixels. Including the optical and inertial sensors, the HMD weighs over eight pounds.

Figure 5.2: Optical see-through HMD

5.1.2 Rate gyroscopes and linear accelerometers

The inertial sensors consist of three angular rate gyroscopes and three linear accelerometers. The three gyroscopes are mounted in a mutually orthogonal configuration. The three accelerometers are also mounted mutually orthogonally. The two clusters are shown in Figures 5.3 and 5.4. The small black boxes are the accelerometers, while the large silver metal disks recessed into the metal housing are the gyroscopes. The box that houses the gyroscopes measures 4.5" by 3.5" by 2". The total weight of all the inertial sensors and the mounting boxes is about one pound.

Figure 5.3: One view of gyroscopes and accelerometers

Figure 5.4: Another view of gyroscopes and accelerometers

The gyroscopes are Systron Donner QRS-11 units. Each detects the rate of angular rotation about one axis by using a pair of vibrating tuning forks that detect the Coriolis force. Each gyroscope reports angular rates within the

range of ±300 degrees per second by generating an output voltage in the range of ±2.5 Volts. The gyroscopes are purely analog devices that do not require any synchronization or clock signals; the voltage generated by a gyroscope represents the rate of rotation that it detects at that instant. These devices are quite accurate. For a ±100 degrees per second sensor, the manufacturer claims resolution of less than 0.002 degrees per second [Systron91] and a net integrated drift ranging between 1.4 and 8.3 degrees per hour [Garcia92].

The accelerometers are Lucas NovaSensor NAS-C026 linear accelerometers, reporting accelerations within the range of ± 2 *g*, where *g* represents the acceleration due to gravity at sea level. The accelerometers are micromachined cantilever beams that respond to linear acceleration, angular acceleration, and gravity along one axis. The measured acceleration is reported as a voltage in the 0-5 Volts range. Like the gyroscopes, the accelerometers are purely analog devices that do not use clocks or other synchronization signals.

The analog outputs of both the accelerometers and the gyroscopes are read by an Analog to Digital (A/D) conversion board in an Intel 80486 PC. The A/D board is the National Instruments AT-MIO-16D, which performs the A/D conversion to 12 bits of accuracy. This board does not have sample and hold capability, so the six signals are read sequentially. However, the board is capable of sampling a signal at 100,000 Hz, so the set of six signals can all be read within a few microseconds of each other, minimizing any errors due to the sequential sampling. Normally I set the A/D to sample each group of six signals at 200 Hz.

Maximizing the signal-to-noise ratio on the inertial sensors was a prime concern, especially since I have long analog lines running from the sensors to the A/D board in the 80486 PC. The sensors are plugged into an electronics box attached to a backpack that the user wears. This box provides power and reroutes the analog outputs so that only one cable is needed to carry all six signals back (Figures 5.5 and 5.6). The box converts the ±12 Volts from a remote power supply into the voltages that the inertial sensors require. The power lines for the accelerometers are biased by -2.5 Volts so that both the gyroscopes and accelerometers report outputs in the ±2.5 Volts range. The

gyroscopes require isolated power supplies, both on the positive and negative sources, or they produce noisy outputs due to a large negative power noise rejection ratio at a high frequency. Consequently, the electronics box contains six voltage regulators, two for each gyroscope (Motorola LM317M and LM337M). The cable from the box to the PC is about 25 feet long, which is a possible source of additional analog noise. Therefore, the A/D board runs in differential mode, and each signal and return pair has its own twisted-pair lines. The cable does not shield each twisted-pair individually, but it does provide outer shielding for all the wires. The cable is Belden 9509. Before the signals go into the A/D, they are put into a metal-shielded breakout box that sits next to the 80486 PC (Figure 5.7). Each signal is sent into an analog single-pole lowpass filter with 48 Hz cutoff frequency, then sent into the A/D board. The gyroscopes have a noise spike at 262 Hz, which appears to be an intrinsic property of all three gyroscopes, so using a notch filter for those devices might be a better solution. The 48 Hz lowpass filters add about 3 ms of delay to a 1 Hz input signal.

These steps appear to work well in controlling noise. When the inertial sensors are kept still, the digitized values reported by the 80486 PC change only in the least-significant bit, even when jostling the long cable.

Figure 5.5: External view of the electronics box

Figure 5.6: Internal view of the electronics box

Figure 5.7: A/D breakout board next to the PC

5.1.3 Optoelectronic head tracker

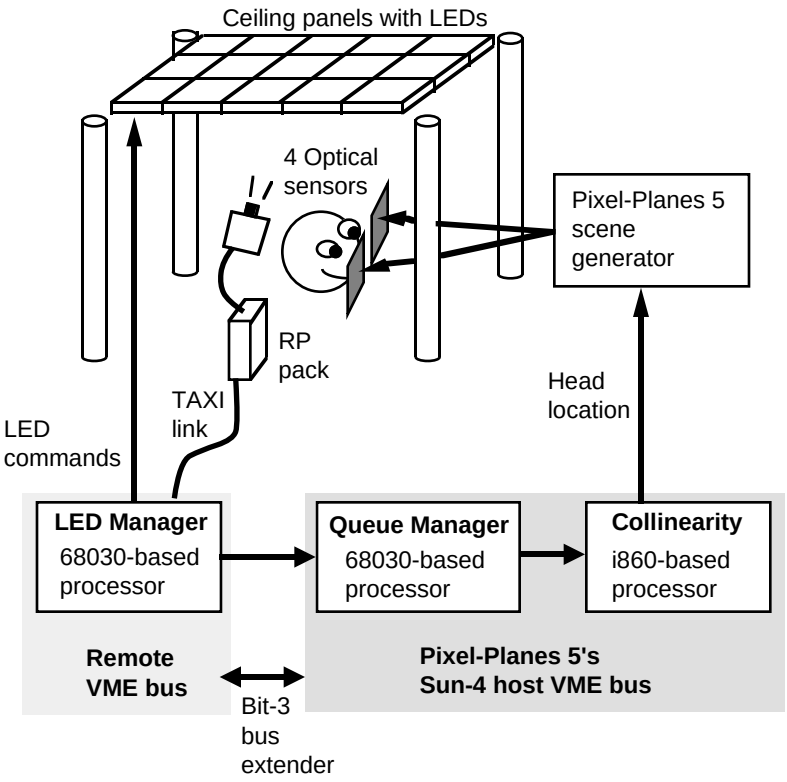


Figure 5.8: Optoelectronic tracker architecture

The basic operation and performance of the custom-built optoelectronic tracker were described in Chapter 3, but not the architectural details. Figure 5.8 shows the architecture of the optoelectronic tracker. Three single-board computers are used: two 68030 boards and one i860 board. One 68030 board resides in a standalone Sun-3 VME chassis that sits next to the ceiling superstructure. That board runs a program called "LED Manager," which is responsible for choosing the sets of LEDs to flash and reading the optical sensors. The readings reported by the sensors are the imaged 2-D photocoordinates of the viewed LEDs. These are digitized by A/D's in the "Remote Processor" (RP) pack, which the user wears to avoid sending analog signals across long distances. The digitized photocoordinates are sent to LED Manager by a TAXI-based datalink. LED Manager then sends the photocoordinates, along with the ID numbers of their associated LEDs, to the i860 processor. This communication is handled by the "Queue Manager," which runs on another 68030. The Queue Manager 68030 and the i860 board both reside on the VME bus of the Sun-4 host for Pixel-Planes 5.

The i860 runs "Collinearity," which is the mathematical routine that uses the imaged photocoordinates to compute the position and orientation of the head. The head locations are then sent to Pixel-Planes 5 for rendering. Usually this communication is done through shared memory to the Sun-4 host, which then passes the head locations to Pixel-Planes 5 [Azuma91] [Ward92].

5.1.4 Scene generator: Pixel-Planes 5

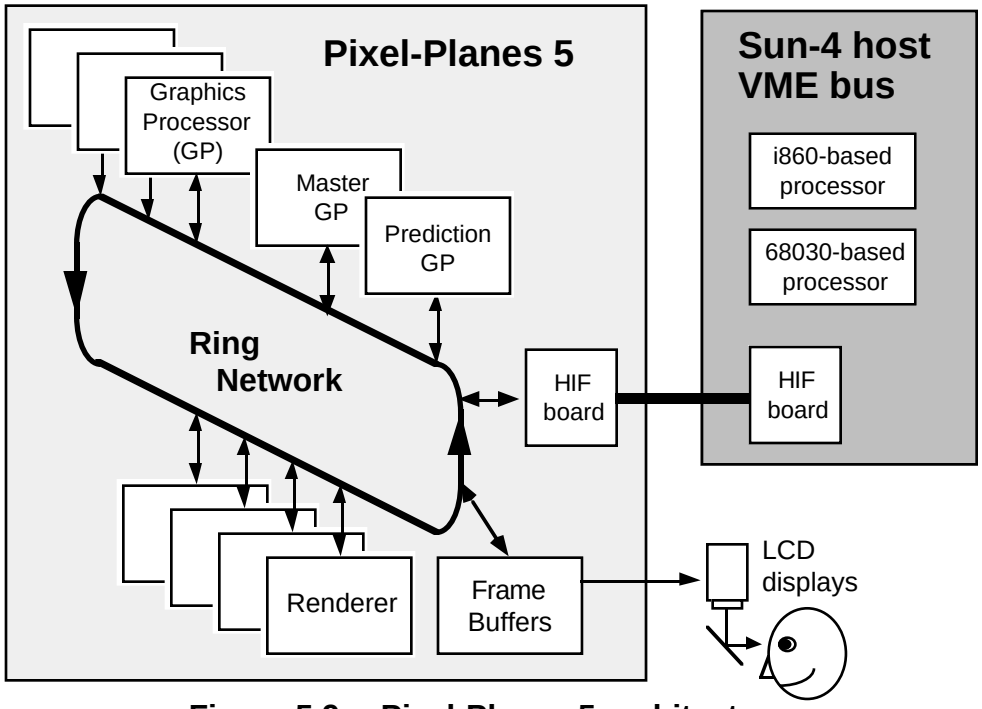


Figure 5.9: Pixel-Planes 5 architecture

Pixel-Planes 5 is a custom-built scene generator capable of rendering over two million polygons per second. The basic architecture is described in [Fuchs89]. What I describe here is how the machine is normally configured and used to render images. However, Pixel-Planes 5 is capable of running different rendering software, and I make use of an alternate set of software, described later in this section. This is possible because the heart of Pixel-Planes 5 is a 640 Mbyte per second token-ring network that links all the processors and I/O devices together. Figure 5.9 shows the basic configuration. Pixel-Planes 5 is a highly parallel machine, with both MIMD and SIMD components. The two main types of processor boards are the Graphics Processors (GPs) and the Renderers. Each GP has an Intel i860 CPU with 8 Mbytes of RAM and is responsible for the transformation steps in

the graphics pipeline. Pixel-Planes 5 has several GPs running in a MIMD parallel configuration. One GP, which is called the "Master GP," is responsible for synchronizing and controlling the other GPs. Transformed primitives (e.g., triangles) are sent from the GPs to the Renderer boards across the token-ring network. Each Renderer is a 128 by 128 SIMD array of pixel processors that rasterizes entire primitives in parallel. When each 128 by 128 subregion of the image is finished, the Renderer sends it across the token-ring network to the frame buffer, where the images are scanned out into the displays of the see-through HMD. Communication through the ring is mediated by a low-level operating system called the Ring Operating System (ROS).

The Sun-4 host communicates with Pixel-Planes 5 through a pair of Host InterFace (HIF) boards. One HIF board sits on the Sun-4's VME bus, and a matching board attaches to Pixel-Planes 5's token ring. Normally, the Sun-4 processor is responsible for sending the head locations to Pixel-Planes 5 through this HIF. However, I bypass the Sun-4 host by installing another pair of HIF boards that allow the Queue Manger 68030 to send the tracker outputs directly into Pixel-Planes 5, without going through the Sun-4 processor. This is an important capability, as Section 5.2 will show.

I use special low-latency rendering software with Pixel-Planes 5. The standard software is built to maximize throughput, but this also generates longer delays than one might expect. The lowest achievable lag with the standard software is about 55 ms [Mine93], even if only a single triangle is rendered. Most applications have much longer delays. However, Pixel-Planes 5 is flexible enough to support different rendering techniques. Marc Olano and Jon Cohen wrote a simpler renderer that sacrifices throughput to dramatically reduce latency [Cohen94]. This renderer, designed for interlaced NTSC displays, is synchronized to the vertical retrace signal and renders stereo images at a fixed 60 Hz rate, with 16.67 ms of delay. This reduction of 40 ms or more makes the prediction problem much easier.

5.1.5 Connections and communication paths

Figure 5.10 puts everything together by showing how the tracker, the inertial sensors, and the scene generator interconnect. The 80486 PC digitizes measurements from the inertial sensors at 200 Hz and sends them across a Bit-3 bus extender from the PC's ISA bus to the standalone VME chassis. A small amount of shared memory is installed in the standalone VME to hold these inertial measurements. Whenever the LED Manager 68030 collects a set of LED readings, it also reads the latest set of inertial readings by looking at the shared memory locations on the VME bus. The inertial data are grouped with the LED photocoordinates and shipped across another Bit-3 bus extender to the Sun-4's VME chassis. The data are sent to the i860, which turns the LED measurements into a head position and orientation. The Queue Manager then grabs the computed head location and its associated inertial measurements and sends those into Pixel-Planes 5 through a HIF, avoiding any use of the Sun-4 processor. Inside Pixel-Planes 5, I reserve one of the GP boards, calling it the "Prediction GP." This GP is not used to support graphics rendering. Instead, it receives the head location and inertial measurements from the Queue Manager and runs the prediction method described in Chapter 4 to generate a predicted head location. This is transformed into appropriate viewing matrices, which are sent across the token-ring network to the Master GP, which runs the low-latency rendering software and renders the appropriate images.

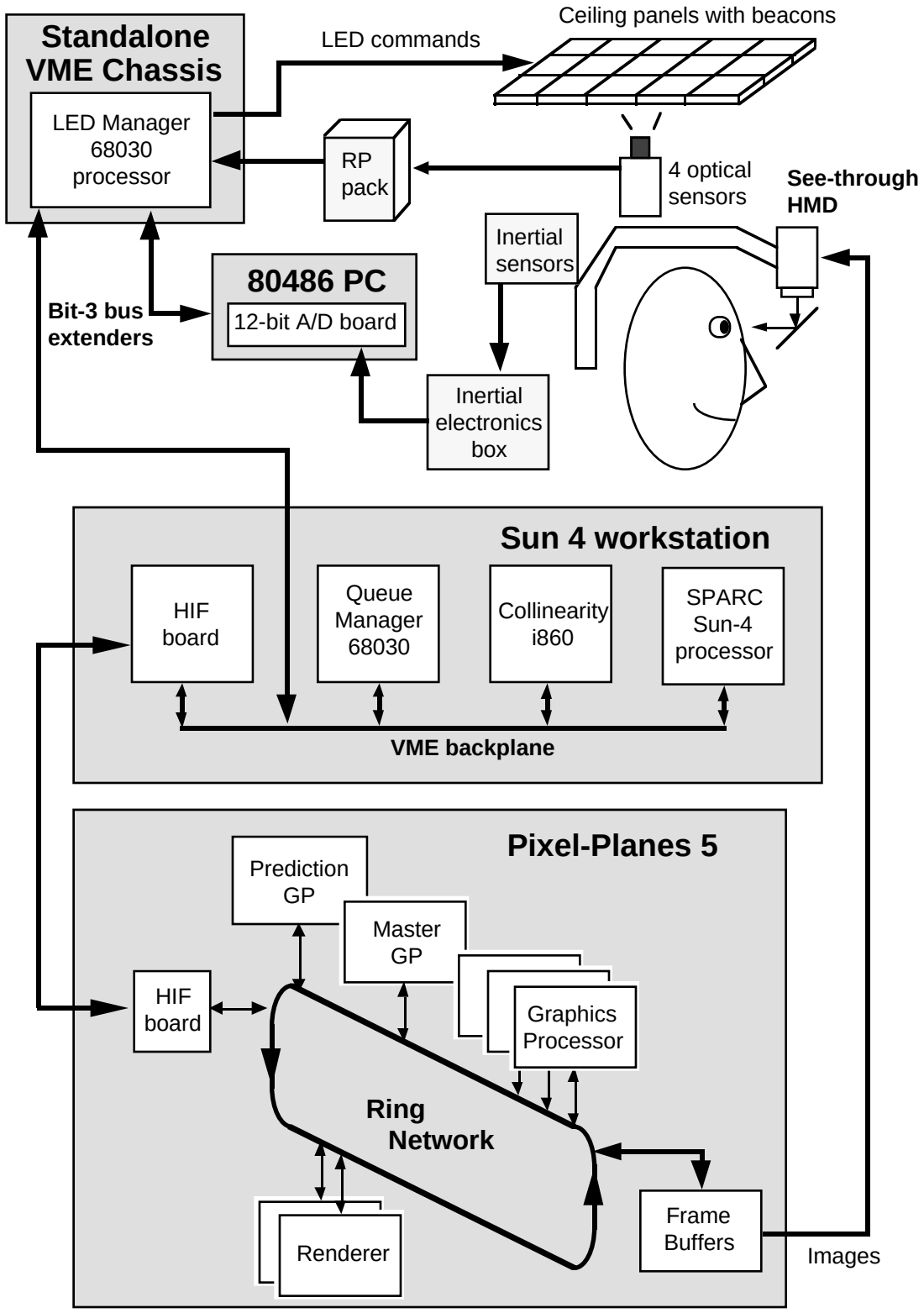


Figure 5.10: Dataflow diagram of entire system

Note that all the parts are tightly connected, using fast and dedicated communication paths. I avoided serial lines because they are too slow.

Using Ethernet or other networks shared with other users makes the system vulnerable to saturation and unpredictable delays caused by external users. Instead, communication occurs through shared memory, bus extenders, and Pixel-Planes 5's own token-ring network. Such tight connections are important for making prediction work, as the next section explains.

5.2 Timing details

The end-to-end system delay typically varies from 50-70 ms. The delay can be longer or shorter than that range, but almost all actual delays fall within that range. This delay is the sum of the tracker lag, the time it takes to run the predictor, the time it takes Pixel-Planes 5 to render the images, and other delays. The optoelectronic tracker uses 15-30 ms, the predictor consumes ~12 ms, and Pixel-Planes 5 requires 16.67 ms. The remaining lag comes from communication delays and the time spent waiting for synchronization. The variation in typical tracker delays is mostly due to the varying number of beacons viewed, as explained in Section 3.4. At 30 ms, the tracker accounts for about 43% of a 70 ms total delay, and Figure 4.36 shows what typical average errors result from such delays. Figure 5.11 shows what the actual end-to-end delays were during part of one motion sequence, as measured in the operational system. It demonstrates how the delays vary from iteration to iteration.

The delays sometimes appear to follow a recognizable pattern. For example, the first twenty iterations in Figure 5.11 seem to show a regular pattern in the prediction intervals. This pattern is broken up in later iterations. The regular pattern occurs because of the need to synchronize with the times when Pixel-Planes 5 is ready to render new images, as described later in this section. The patterns change because the delay in the tracking system also varies with time. However, if all the components took a constant amount of time, then the variable prediction intervals could be described by a simple model. Wloka provides an example of this with his "Orbit Models" [Wloka95].

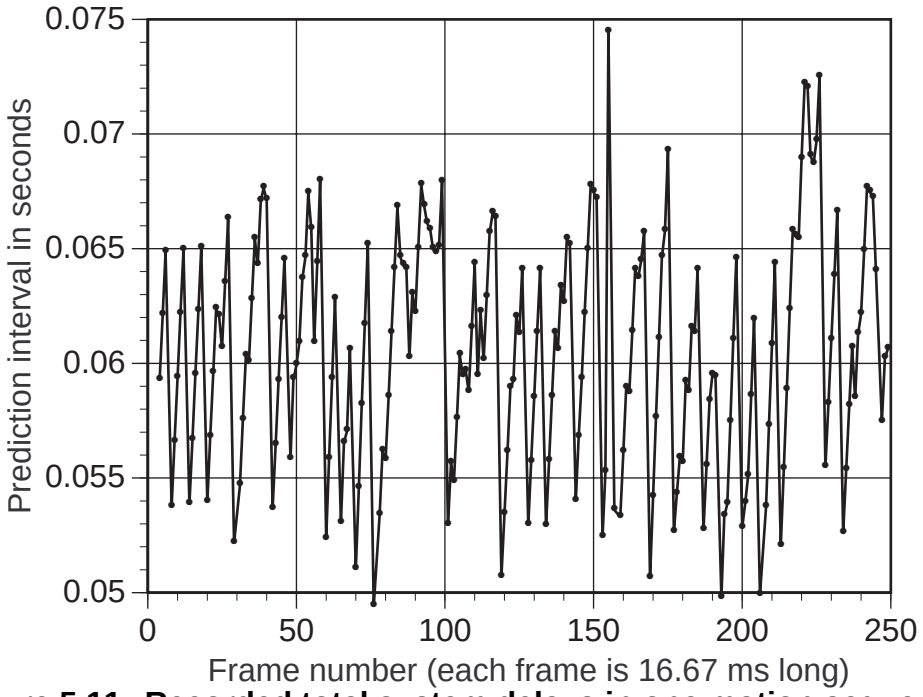


Figure 5.11: Recorded total system delays in one motion sequence

To perform accurate prediction, one must know how *far* to predict into the future. This is a truism, but one that has not been explicitly addressed. The prediction interval is constant only in systems that synchronize every component, with guaranteed performance specifications in each component. Some flight simulators have constant prediction intervals, but most commercial and academic systems do not. The result is that the prediction interval varies with time. Misjudging the prediction interval by as little as 10 ms can lead to visible misregistrations. At the moderate rotation rate of 50 degrees per second, 10 ms of unaccounted delay yields 0.5 degrees of error, which produces almost 9 mm of error for an object one meter away. Therefore, for prediction to be effective, the system must accurately predict how far to predict, before each prediction is performed.

This variable delay has serious ramifications to the system design. Enabling accurate estimation of the prediction interval for each rendering iteration requires accurate timestamps, removing all sources of unpredictable delays, and an accurate characterization of how long each rendering step takes.

1) Accurate timestamps: Accurate timestamps require synchronized clocks. Commercial trackers, such as those from Polhemus, Ascension, and

Logitech, do not supply timestamps with the measured locations. The optoelectronic tracker has a clock on the LED Manager 68030 board, so that can assign timestamps to the head tracker measurements. It also assigns timestamps to the inertial measurements sent from the PC. The GP boards inside Pixel-Planes 5 also have clocks. Both the 68030 clock and the GP clock have resolution of well under one millisecond. However, they do not have the same offset, and they run at slightly different rates, drifting by about 8.3 ms every three minutes. Therefore, the system synchronizes the clocks by sending a message from the Prediction GP to the LED Manager 68030 and back again. The round-trip communication time is less than one millisecond. Two synchronizations are done several minutes apart, to generate an estimate of the drift rate between the two clocks. This drift rate is used to compensate for the different clock speeds, and resynchronizations occur every three minutes or so to avoid long-term errors.

2) Removing unpredictable delays: Removing all sources of unpredictable delays means using real-time operating systems and fast, dedicated communication channels. Most academic VE and AR systems use Unix as the underlying operating system. However, standard versions of Unix are incompatible with real-time requirements, because Unix can swap out a user's process at any time for an unbounded amount of time. Even modifying the kernel to give maximum priority to one user's processes, which nearly starves all other users, does not solve the problem. When I tried collecting data on a Sun-4 running Unix with my processes granted highest priority, I would still regularly get pauses of 60-200 ms in the data stream during a three minute collection. The longest pause I ever saw, which was two seconds, occurred late at night when I was the only user on the machine! Therefore, my system does not use Unix except for the initial setup, which is not time-critical. It bypasses the Sun-4 host, as explained in Section 5.1.4, and it uses low-level operating systems: MS-DOS in the 80486 PC, VxWorks in the optoelectronic tracker boards, and ROS inside Pixel-Planes 5. While these operating systems are not strictly real time because they do not have guaranteed performance, they have proven sufficient in practice. The communication channels are fast, because they rely on shared memory, bus extenders, or the high-bandwidth token ring inside Pixel-Planes 5. These communication paths are not shared with other users, to avoid external

sources of delay. The main potential source of contention is accessing the shared memory. The Sun-4 processor also uses that for accessing the disk and other I/O devices. However, the Sun-4 processor accesses memory through a different bus, so I do not compete against that bandwidth. In practice, I have not observed any external source that saturates the bus. In contrast, many VE systems rely on using a LAN shared with several other users, any one of whom can easily monopolize the network and add unbounded amounts of delay.

3) Characterizing rendering delays: The simplest way to do this is to have a renderer that runs at constant rates, but the requirement is merely to know how long each rendering step takes. The low-latency renderer [Cohen94] that I use with Pixel-Planes 5 runs at a constant 60 Hz, synchronized to vertical retrace, so this characterization in my system is trivial. It is possible to estimate rendering delays for renderers that take variable amounts of time. This is a nontrivial problem, but work has been done in this area [Funkhouser93].

With these three properties integrated into the system, it is possible to accurately estimate prediction intervals. The total prediction interval is defined to start at the time when the tracker and inertial measurements are taken and end at the time when the frame buffer begins scanning out the images corresponding to those measurements (Figure 5.12). Later in this section I will discuss alternate definitions for the start and end time, but for now I will use this definition. Part of this interval is directly measurable. The Prediction GP can read the clock at the point labeled "Prediction distance must be set here" in Figure 5.12 and subtract that from the start timestamp to generate the measured portion of the interval. The remaining portion must be estimated. Why? The predictor requires an interval to predict into the future. The predictor must also be run before viewpoint computation and rendering. Therefore, the total prediction interval must be estimated immediately before running the predictor, requiring an estimate of the light-shaded interval in Figure 5.12.

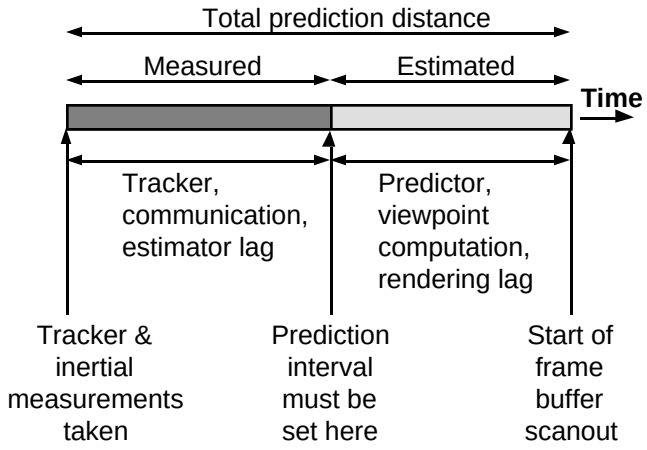
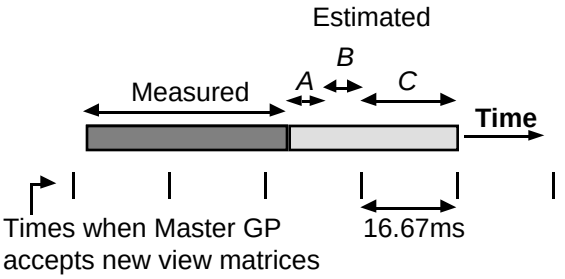


Figure 5.12: Components of the total prediction interval

Figure 5.13 shows how this estimate is generated. The key observation is that the low-latency renderer on Pixel-Planes 5 is synchronized to the vertical retrace signal, which occurs every 16.67 ms. Thus, the renderer is guaranteed to render fields at 60 Hz, whether or not the tracker can keep up with that rate. The estimated interval is made up of three components *A*, *B*, and *C*.



A = Time to run predictor and view matrix computation
B = Delay until Master GP accepts new matrices (which occurs at strict 16.67 ms intervals)
C = Time for Pixel-Planes 5 to compute new images

Figure 5.13: Components of the estimated interval

A, the time needed to run the predictor and compute the view matrices, is almost constant and can be measured empirically during trial runs. In practice, this is about 12 ms in my system.

B, the delay until the renderer accepts new matrices, is computed by finding the next rendering start point that is greater than *Measured* + *A*. The predictor knows when these starting points occur because it receives a constant stream of past starting points from the renderer. Since the starting

points are separated by a strict 16.67 ms, it is easy to determine all future starting points, given one point in the past.

C, the time it takes the renderer to rasterize the images and copy them to the frame buffer, is a constant 16.67 ms.

Figure 5.14 demonstrates the effectiveness of the prediction interval estimation. It graphs the same data shown in Figure 5.11, except that now the predicted intervals are overlaid on top of the actual intervals. Note that the two graphs coincide except for one point around frame number 225. This is more easily seen in Figure 5.15, which plots the difference between the actual and estimated intervals. When the prediction is incorrect, it is wrong by 16.67 ms. The predicted intervals were computed in real time during an actual motion sequence, and they were recorded along with the measured actual intervals.

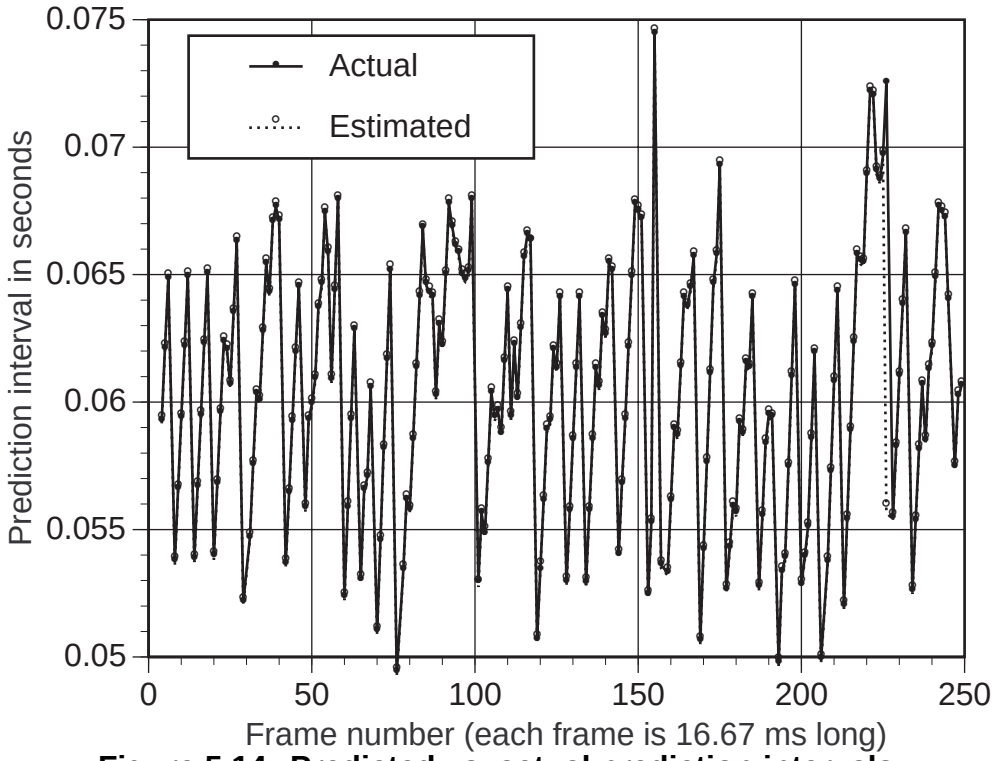


Figure 5.14: Predicted vs. actual prediction intervals

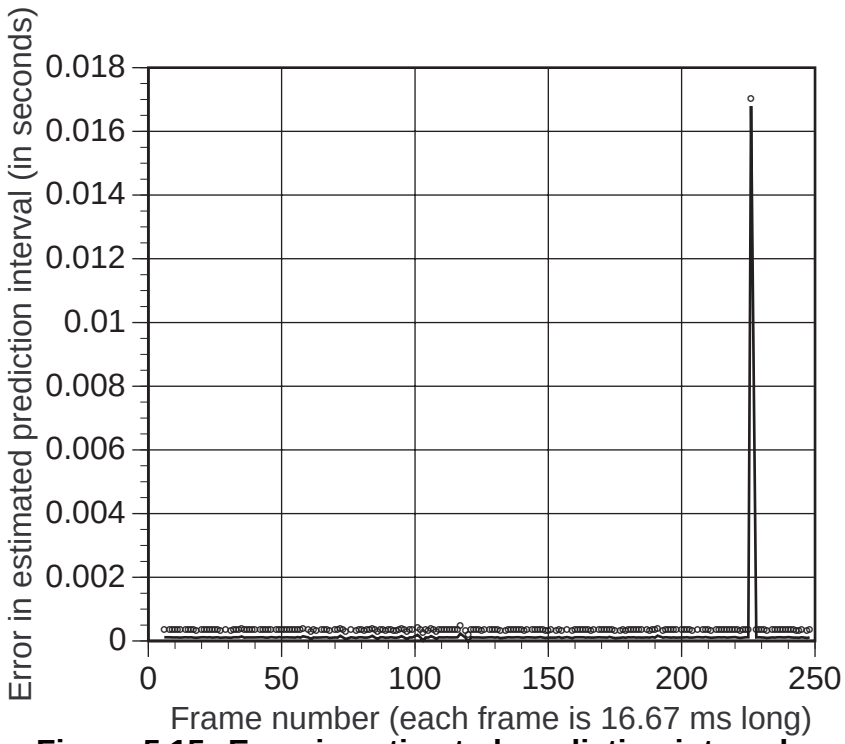


Figure 5.15: Error in estimated prediction intervals

The steps taken to accurately estimate the prediction intervals make the system difficult to build and debug, but they are necessary to achieve the desired performance. Unix is popular because it is a good development environment and has debugging tools. In contrast, developing and debugging code on several different real-time operating systems is painful. Tightly coupling the tracker and the scene generator reduces the flexibility of the system, making it difficult to change hosts, scene generators, trackers, or anything else. The integration in this system more closely resembles flight simulators than typical systems produced by academia, because flight simulators are often concerned with guaranteed update rates. A lesson from this work is that future AR systems that intend to use prediction will have to either synchronize their modules or account for the variable prediction intervals, either of which requires building a system with real-time performance in mind.

I conclude this section by discussing three details: accounting for the time difference between the tracker and inertial measurements, defining the start of the prediction interval, and defining the end of the prediction interval.

1) Time shift between inertial and tracker measurements: Each measurement packet sent to the Prediction GP consists of a position, an orientation, six inertial readings, and one timestamp for all the other values. In practice, however, the head tracker measurement is not taken at precisely the same time as the inertial readings, because they run asynchronously. The inertial readings are sampled and stored in a shared memory location in the standalone VME chassis at a constant 200 Hz. The tracker typically runs between 60-80 Hz. Each time the LED Manager collects a group of LEDs, it reads the most recent set of inertial data and groups the two together. To determine the average time difference between these two collection points, I collected a motion sequence and integrated the gyroscope readings. These should match the orientations recorded by the tracker. Powell's method [Press88] searches for the time difference between the two sets of measurements that generates the closest match between the integrated and actual orientations. This difference is about six milliseconds. To compensate for this in real time, a brief history of recent tracker measurements is kept, and I linearly interpolate back six milliseconds from the "present" to find a set of tracker measurements to associate with the inertial measurements.

2) Start of the prediction interval: The optoelectronic tracker usually takes over 10 ms to collect the set of LEDs used to compute the head location. Since 10 ms or more is a fairly large window, and the head may be moving during this time, it is not clear exactly what time the computed location actually represents. In practice, I assign the tracker timestamp to be the time before the first LED is sampled, but it could be argued that averaging the timestamps of when the first and last LEDs are sampled might be more accurate. Since the head may be in motion during the collection interval (at rates shown in Figures 1.7 - 1.9), the sampled LED photocoordinates may not correspond to any particular head location during that time interval, providing another source of distortion [Morris93]. This should be correctable, but the existing optoelectronic tracker software does not compensate for this.

3) End of the prediction interval: I define the end time to be the start of scanout. However, the middle of scanout (which adds 8.33 ms) or the end of scanout (which adds 16.67 ms) are other reasonable choices. The problem is complicated because the LCDs used in the HMD are difficult to characterize.

They are not as well behaved as typical CRT displays. For example, I do not know how long the images persist on the LCD displays. I chose the start of scanout to minimize the overall prediction interval. Also, when experimenting with end times in the actual system, choosing the start of scanout appeared to produce the smallest registration errors as seen inside the see-through HMD. This was a highly subjective measurement; the differences were not blatant.

5.3 Prediction method details

Chapter 4 did not adequately describe the inertial-based prediction method in two areas. First, it assumed that the angular rotation and linear acceleration information provided to the filter was in Tracker space, with units of radians per second and meters per second squared. How these values were extracted from the raw digitized voltages was not specified. Second, it did not say how the filter parameters were set. This section covers both of these details.

Throughout this section, ω represents omega, a 3 by 1 vector that specifies angular velocity.

5.3.1 Extracting angular velocity and linear acceleration

The Extended Kalman Filter (EKF) used to filter orientation requires the measured omega $\omega_{\mathbf{m}}$ to be in Tracker space, with each term in radians per second. Therefore, the raw digitized values from the gyroscopes must be converted into those values. Fortunately, this is straightforward. Use the biases and scales to convert the digitized readings from voltages to radians per second. Then rotate the vector from Gyroscope space to Tracker space. The exact details depend on how the gyroscopes are mounted on the HMD. Thus, the procedure described below works with my system but will require appropriate modifications for other configurations.

Let $\mathbf{G}_{\mathbf{raw}}$ be a 3 by 1 vector holding the raw digitized gyroscope values reported by the A/D board in the 80486 PC. Because these are 12-bit values, they fall in the range 0-4095, where 0 represents -2.5 Volts and 4095 is

+2.5 Volts. $\mathbf{G_{raw}}[0]$ is the value from gyroscope 0, $\mathbf{G_{raw}}[1]$ is from gyroscope 1, etc. Then $\mathbf{G_{raw}}$ is converted into $\mathbf{G_{radians}}$ as follows:

$$\mathbf{G_{radians}}[2] = \left(\frac{\pi}{180}\right)\left(\frac{-\mathbf{G_{scale}}[0]}{2047.5}\right)\left(\mathbf{G_{raw}}[0] - \mathbf{G_{offset}}[0]\right)$$

$$\mathbf{G_{radians}}[0] = \left(\frac{\pi}{180}\right)\left(\frac{\mathbf{G_{scale}}[1]}{2047.5}\right)\left(\mathbf{G_{raw}}[1] - \mathbf{G_{offset}}[1]\right)$$

$$\mathbf{G_{radians}}[1] = \left(\frac{\pi}{180}\right)\left(\frac{-\mathbf{G_{scale}}[2]}{2047.5}\right)\left(\mathbf{G_{raw}}[2] - \mathbf{G_{offset}}[2]\right)$$

$\mathbf{G_{scale}}$ is a 3 by 1 vector containing the scale values for the three gyroscopes, and $\mathbf{G_{offset}}$ is a 3 by 1 vector holding the bias values. These are some of the filter parameters determined in Section 5.3.2. The change in axes between $\mathbf{G_{raw}}$ and $\mathbf{G_{radians}}$ is due to the way the gyroscopes happen to be arranged in the gyro pack. For example, gyroscope #1 is oriented along the omega *X* axis (index 0), rather than the omega *Y* axis (index 1). This conversion will need modification for different gyro packs.

$\mathbf{G_{radians}}$ now holds omega with each term in radians per second. However, the angular rates were recorded in Gyroscope space (see Figure 2.3). They must be rotated into their equivalent values in Tracker space, which yields the desired result $\omega_{\mathbf{m}}$.

$$\omega_{\mathbf{m}} = \mathbf{Q_{gyro}} \bullet \mathbf{G_{radians}} \bullet \left(\mathbf{Q_{gyro}}\right)^{-1}$$

The two multiplications in the previous equation are quaternion multiplications, where $\mathbf{G_{radians}}$ is written as a quaternion with the *qw* term set to zero and the vector part set to the 3 by 1 vector. $\mathbf{Q_{gyro}}$ is a quaternion that rotates points and vectors from Gyroscope space to Tracker space, and that is determined by the procedures described in Section 5.3.2.

Recovering linear acceleration from the digitized accelerometer readings is more complicated. The accelerometers detect three things: 1) linear acceleration, 2) angular acceleration, and 3) gravity. Extracting linear acceleration from the accelerometer outputs requires estimating the other two terms with the other sensors. The fact that I have three 1-D accelerometers instead of one 3-D sensor further complicates matters, since the effect of angular acceleration varies with location on the HMD.

Each accelerometer is a cantilever beam that extends out over empty space, much like a tiny diving board [Lawrence92]. The beam consists of a mass at the end of a hinge. In the absence of motion or gravity, this beam extends out straight horizontally. If the device is held still in the presence of gravity, the beam sags down (Figure 5.16). As the user moves the accelerometer up and down, the beam moves up and down with respect to the rest of the device, due to inertia. The accelerometer electrically measures the height of the beam with respect to the rest of the device, and that voltage is what the sensor returns.

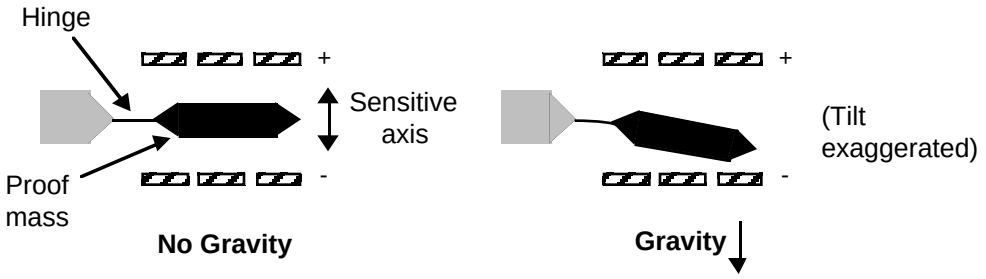


Figure 5.16: Accelerometers are tiny cantilever beams

Note two important properties of the accelerometer. To a first approximation, each accelerometer detects acceleration only along one direction in space and is insensitive to any motion perpendicular to its sensitive axis. Also, gravity sets the "bias point" of the accelerometer. That is, the value that the accelerometer reports when it is standing still depends upon its orientation with respect to the gravity vector.

Recovering linear acceleration from the accelerometers requires estimates of gravity and angular acceleration. These estimates are based on the state vector from the orientation EKF, which must be run in conjunction with the position Kalman filters. The estimate of the current head orientation specifies the gravity vector, and the state vector contains an estimate of angular acceleration. Removing gravity and angular acceleration recovers the desired linear acceleration. This recovery is a three step process:

- Step 1: Compensate for gravity by determining the bias points.
- Step 2: Change the acceleration into World space.
- Step 3: Remove the angular acceleration component.

Step 1) Gravity compensation: Gravity is assumed to point straight down in World space with an acceleration of 9.8 meters per second squared. Rotate this vector into Tracker space by using the estimated orientation of the tracker from the state vector of the orientation EKF. Then rotate the Tracker space vector into Accelerometer space (Figure 2.3) by using the fixed orientation between the accelerometer pack and the HMD. Change the Accelerometer-space values from meters per second squared to raw digitized values in the scale 0-4095 by using the offset and bias values. This determines the bias point set by gravity, which is subtracted from the raw digitized readings.

Let $\mathbf{A_{raw}}$ be the 3 by 1 vector containing the raw digitized accelerometer readings and $\mathbf{A_{biased}}$ be the vector with gravity compensation. $\mathbf{Grav_{world}}$, $\mathbf{Grav_{tracker}}$, and $\mathbf{Grav_{accel}}$ are the gravity vectors in World, Tracker and Accelerometer space, respectively. $\mathbf{Q}$ is the quaternion extracted from the EKF state vector that rotates points and vectors from Tracker space to World space. $\mathbf{Q_{accel}}$ is the quaternion that rotates points and vectors from Tracker space to Accelerometer space. $\mathbf{A_{scale}}$ and $\mathbf{A_{offset}}$ are the scale and bias values for the three accelerometers, converting raw digitized readings to meters per second squared. $\mathbf{Q_{accel}}$, $\mathbf{A_{scale}}$, and $\mathbf{A_{offset}}$ are all parameters to be determined by the procedures in Section 5.3.2. The following multiplications are quaternion multiplications, with $\mathbf{Grav_{world}}$ and $\mathbf{Grav_{tracker}}$ temporarily changed to quaternions with a zero *qw* component.

$$\mathbf{Grav_{world}} = \begin{bmatrix} 0 \\ 0 \\ -9.8 \end{bmatrix}$$

$$\mathbf{Grav_{tracker}} = \mathbf{Q}^{-1} \bullet \mathbf{Grav_{world}} \bullet \mathbf{Q}$$

$$\mathbf{Grav_{accel}} = \mathbf{Q_{accel}} \bullet \mathbf{Grav_{world}} \bullet \left(\mathbf{Q_{accel}}\right)^{-1}$$

The next few equations compute the desired $\mathbf{A_{biased}}$. The indices change between $\mathbf{Grav_{accel}}$ and $\mathbf{A_{biased}}$ due to the arrangement of the accelerometers on the HMD. For example, accelerometer #0 is along the *Y* axis (index 1) in Accelerometer space. Therefore, these equations will need modification for different systems.

$$\mathbf{A_{biased}}[0] = \mathbf{A_{raw}}[0] - \mathbf{A_{offset}}[0] + \left(\frac{1024}{\mathbf{A_{scale}}[0]}\right)\mathbf{Grav_{accel}}[1]$$

$$\mathbf{A_{biased}}[1] = \mathbf{A_{raw}}[1] - \mathbf{A_{offset}}[1] - \left(\frac{1024}{\mathbf{A_{scale}}[1]}\right)\mathbf{Grav_{accel}}[0]$$

$$\mathbf{A_{biased}}[2] = \mathbf{A_{raw}}[2] - \mathbf{A_{offset}}[2] + \left(\frac{1024}{\mathbf{A_{scale}}[2]}\right)\mathbf{Grav_{accel}}[2]$$

Step 2) Conversion to World space: Since the three accelerometers are mutually orthogonal, they form a 3-vector in Accelerometer space. Scaling the biased acceleration values converts them to meters per second squared. Then I rotate that vector into World space.

$\mathbf{A_{world}}$ is the desired acceleration in World space. $\mathbf{A_{temp1}}$ and $\mathbf{A_{temp2}}$ are intermediate variables. The multiplications that involve quaternion terms are quaternion multiplications, with vectors converted to quaternions as needed.

$$\mathbf{A_{temp1}}[0] = \left(\frac{\mathbf{A_{scale}}[1]}{1024}\right)\mathbf{A_{biased}}[1]$$

$$\mathbf{A_{temp1}}[1] = -\left(\frac{\mathbf{A_{scale}}[0]}{1024}\right)\mathbf{A_{biased}}[0]$$

$$\mathbf{A_{temp1}}[2] = -\left(\frac{\mathbf{A_{scale}}[2]}{1024}\right)\mathbf{A_{biased}}[2]$$

$$\mathbf{A_{temp2}} = \left(\mathbf{Q_{accel}}\right)^{-1} \bullet \mathbf{A_{temp1}} \bullet \mathbf{Q_{accel}}$$

$$\mathbf{A_{world}} = \mathbf{Q} \bullet \mathbf{A_{temp2}} \bullet \mathbf{Q}^{-1}$$

Step 3) Angular acceleration removal: The last step is to remove the angular acceleration components from $\mathbf{A_{world}}$, leaving the desired linear acceleration. I use the following formula from the kinematics of rigid bodies [Beer88] (Figure 5.17):

$$\mathbf{A_A} = \mathbf{A_B} - \dot{\omega} \times \mathbf{r} - \omega \times (\omega \times \mathbf{r})$$

where ω = omega, the angular velocity in Tracker space

$\dot{\omega}$ = angular acceleration in Tracker space

$\mathbf{A_A}$ = total acceleration at point **A**, origin of Tracker space

$\mathbf{A_B}$ = total acceleration at point **B**, location of an accelerometer

r = vector from point **A** to point **B**

x = vector cross product

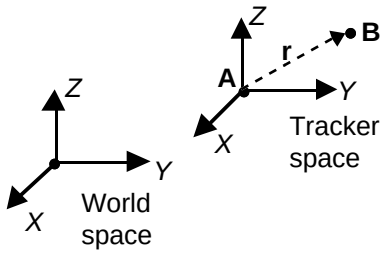


Figure 5.17: Definitions for rigid body kinematics formula

Note that this formula assumes that Tracker space shares the same orientation as World space, as shown in Figure 5.17. This is usually not the case. Thus, to use this formula, everything must be rotated into World space. A second problem comes from using three separate 1-D accelerometers. That means three different **r** vectors exist, one for each accelerometer. Call these **F0**, **F1**, and **F2**, as shown in Figure 5.18. Each vector results in a different contribution from angular acceleration, and each accelerometer only detects the component of acceleration that lies along its sensitive axis. Therefore, the basic kinematics formula requires some modification to use with this system.

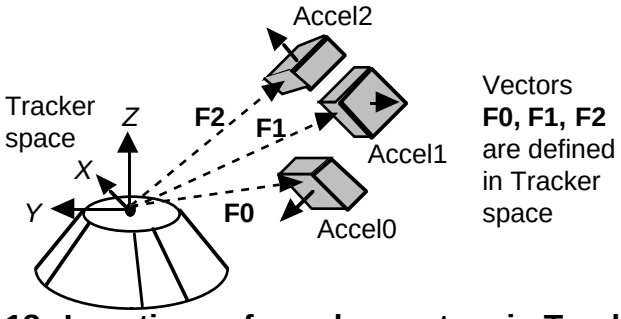


Figure 5.18: Locations of accelerometers in Tracker space

The basic idea is to start with $\mathbf{A_{world}}$ and subtract the three angular acceleration components detected by the three accelerometers. These can

be computed and subtracted individually because the three accelerometers are mutually orthogonal.

The first step is to rotate everything into World space. Let **U0**, **U1**, and **U2** be unit vectors in Accelerometer space that lie along the sensitive axes of accelerometer 0, accelerometer 1, and accelerometer 2, respectively. **U0**, **U1**, **U2**, ω and its derivative, **F0**, **F1**, and **F2** must all be rotated into World space. These vectors are temporarily converted into quaternions as needed for the rotation operations.

$$\mathbf{U0} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{U1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{U2} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{U0_{world}} = \mathbf{Q} \bullet \left(\mathbf{Q_{accel}}\right)^{-1} \bullet \mathbf{U0} \bullet \mathbf{Q_{accel}} \bullet \mathbf{Q}^{-1}$$

$$\mathbf{U1_{world}} = \mathbf{Q} \bullet \left(\mathbf{Q_{accel}}\right)^{-1} \bullet \mathbf{U1} \bullet \mathbf{Q_{accel}} \bullet \mathbf{Q}^{-1}$$

$$\mathbf{U2_{world}} = \mathbf{Q} \bullet \left(\mathbf{Q_{accel}}\right)^{-1} \bullet \mathbf{U2} \bullet \mathbf{Q_{accel}} \bullet \mathbf{Q}^{-1}$$

$$\mathbf{F0_{world}} = \mathbf{Q} \bullet \mathbf{F0} \bullet \mathbf{Q}^{-1}$$

$$\mathbf{F1_{world}} = \mathbf{Q} \bullet \mathbf{F1} \bullet \mathbf{Q}^{-1}$$

$$\mathbf{F2_{world}} = \mathbf{Q} \bullet \mathbf{F2} \bullet \mathbf{Q}^{-1}$$

$$\omega_{\mathbf{world}} = \mathbf{Q} \bullet \omega \bullet \mathbf{Q}^{-1}$$

$$\dot{\omega}_{\mathbf{world}} = \mathbf{Q} \bullet \dot{\omega} \bullet \mathbf{Q}^{-1}$$

Now for each accelerometer, compute the angular acceleration in World space at that accelerometer's location. Call these angular accelerations **V0**, **V1**, and **V2** for accelerometer 0, accelerometer 1, and accelerometer 2 respectively.

$$\mathbf{V0} = \dot{\omega}_{\mathbf{world}} \times \mathbf{F0_{world}} + \omega_{\mathbf{world}} \times \left(\omega_{\mathbf{world}} \times \mathbf{F0_{world}}\right)$$

$$\mathbf{V1} = \dot{\omega}_{\mathbf{world}} \times \mathbf{F1_{world}} + \omega_{\mathbf{world}} \times \left(\omega_{\mathbf{world}} \times \mathbf{F1_{world}}\right)$$

$$\mathbf{V2} = \dot{\omega}_{\mathbf{world}} \times \mathbf{F2_{world}} + \omega_{\mathbf{world}} \times \left(\omega_{\mathbf{world}} \times \mathbf{F2_{world}}\right)$$

Since each accelerometer detects acceleration only along its sensitive axis, the vectors **V0**, **V1**, and **V2** are not what are actually detected. Instead, the actual detected angular acceleration is the dot product of those vectors

with the unit vectors along the sensitive axes, in World space. Therefore, the desired linear acceleration $\mathbf{A}_{\mathbf{linear}}$ is the original World acceleration $\mathbf{A}_{\mathbf{world}}$ with the three angular acceleration contributions subtracted.

$$\mathbf{A}_{\mathbf{linear}} = \mathbf{A}_{\mathbf{world}} - \mathbf{V0} \cdot \mathbf{U0}_{\mathbf{world}} - \mathbf{V1} \cdot \mathbf{U1}_{\mathbf{world}} - \mathbf{V2} \cdot \mathbf{U2}_{\mathbf{world}}$$

I checked these equations by testing them on simulated data before trying them on real data. In generating the simulated datasets, I avoided using the motion equations listed above, especially the kinematics formula from [Beer88]. Generating simulated data by using the same equations that are being checked does not truly verify the equations, because the simulated data will match the equations by definition. Therefore, I instead wrote explicit equations that described the position of any point on the head at any time, given various control parameters. Differentiating the equations twice generates the accelerations.

A weakness of Step 3 is the need to estimate the derivative of omega, because the system lacks a sensor that explicitly measures it. Comparing the estimated angular acceleration from the EKF against perfect simulated data suggests that the estimated derivatives lag behind the true values by about 20 ms. Adding angular accelerometers to the HMD would eliminate this problem.

5.3.2 Parameter determination

The filter requires various parameters. They are listed here, both for the orientation EKF and the position Kalman filters:

Parameters for the orientation EKF:

$\mathbf{G}_{\mathbf{offset}}$ = 3 by 1 vector specifying biases for the three gyroscopes

$\mathbf{G}_{\mathbf{scale}}$ = 3 by 1 vector specifying scales for the three gyroscopes

$\mathbf{Q}_{\mathbf{gyro}}$ = quaternion that rotates points and vectors from Tracker space to Gyroscope space

$\mathbf{E}_{\mathbf{orient}}$ = 10 by 10 model covariance matrix

$\mathbf{R}_{\mathbf{orient}}$ = 7 by 7 measurement covariance matrix

Parameters for the three position Kalman filters:

$\mathbf{A_{offset}}$ = 3 by 1 vector specifying biases for the three accelerometers

$\mathbf{A_{scale}}$ = 3 by 1 vector specifying scales for the three accelerometers

$\mathbf{Q_{accel}}$ = quaternion that rotates points and vectors from Tracker space to Accelerometer space

F0, F1, F2 = 3 by 1 vectors from the origin of Tracker space to the locations of the three accelerometers, in Tracker space

$\mathbf{E_{tx}}$, $\mathbf{E_{ty}}$, $\mathbf{E_{tz}}$ = 3 by 3 model covariance matrices

$\mathbf{R_{tx}}$, $\mathbf{R_{ty}}$, $\mathbf{R_{tz}}$ = 2 by 2 measurement covariance matrices

It is desirable to develop techniques for measuring these parameters. In simulation, one can simply define most of the parameters, but in a real system they must match reality. Some of them can be directly measured or specified by mechanical means. For example, the HMD frame and inertial sensor mounting units could be built to precise mechanical specifications that place the inertial sensors at known locations. Alternately, one could build the frame, then measure the locations with a precise coordinate measuring machine. However, these approaches require access to specialized and expensive resources. Furthermore, the parameters might change with time if the HMD frame is not sufficiently rigid or if the system configuration changes. And some of the parameters, such as the model covariances, are not easily determined by mechanical means. Therefore, methods that determine these parameters without the use of specialized mechanical equipment would be useful.

I use nonlinear optimization and autocalibration techniques to set these parameters. Optimization techniques search for the best parameters that minimize a particular cost or error function. For example, the task might be to pick a path that a car should take to get from point **A** to point **B**. The optimal path depends on the cost function. The path that takes the least amount of time may not be the shortest path, and that may be different from the path that minimizes the cost, if tolls for bridges or roads are involved.

Autocalibration techniques are optimizers that control the search by applying geometric constraints to collected data. The geometric constraints

form the basis of comparison. The autocalibration routine computes a particular value or set of values in more than one way by using these geometrical relationships. Ideally, the results from each computation should be equal. The optimizer searches for the parameters that make the multiple estimates match most closely.

The basic ideas for optimizers and calibration routines are well known and have been studied extensively. However, each optimization and autocalibration problem has its own set of constraints. The difficult part is not the method itself; it is finding a set of constraints for a particular problem that works well on real data. That can be a nontrivial task, requiring creativity and experimentation. For example, Stefan Gottschalk and John F. Hughes developed an autocalibration technique to measure the locations of beacons in the optoelectronic tracker [Gottschalk93]. They had to try several different constraints and approaches before finding one that converged.

Optimization generally involves finding the minimum of a multidimensional function. This problem has been heavily studied. In practice, no method can guarantee finding a global minimum of a nonlinear multidimensional function in a finite amount of time, but heuristics exist that tend to perform well. I chose to use Powell's method [Press88] because it does not need the derivatives of the function, and because it is an iterative method that benefits from a good initial guess of the solution.

Determining gyroscope parameters: I developed two autocalibration strategies for measuring the gyroscope offsets, scales, and gyroscope pack orientation ($\mathbf{G}_{\mathbf{offset}}$, $\mathbf{G}_{\mathbf{scale}}$, and $\mathbf{Q}_{\mathbf{gyro}}$ respectively). Clearly, there are two basic approaches available for the geometric constraints. I can differentiate the orientation measurements taken by the tracker and compare those against the angular rates provided by the gyroscopes. Or I can go the other way and integrate the angular rates measured by the gyroscopes and compare those against the tracker orientations. Both approaches work.

For either method, the first step is to have the user wear the HMD and rotate her head. Both the inertial and tracker measurements are captured, generating a motion dataset. The autocalibration routines operate on this dataset offline to determine the parameters.

The first method differentiates the tracker orientations and compares those against the gyroscope measurements. I run an EKF similar to the one described in Chapter 4, except that it only takes tracker orientations as inputs and has seven variables in the state vector: four for the current orientation and three for the estimated angular rates. Numerical differentiation is an inherently noisy operation, but the EKF can perform this differentiation accurately, at the cost of adding a nearly constant amount of delay to the estimated signals. Therefore, the estimated angular rates must be temporally adjusted by some unknown timeshift before doing the comparison. I run the EKF once to generate the estimated angular rates. The gyroscope readings must be biased, scaled, and rotated in order to compare them against the estimated angular rates, as described in Section 5.3.1. Furthermore, the estimated omegas must be shifted in time by the unknown timeshift. Once those are done, the cost function compares the estimated angular rates against the gyroscope-measured angular rates at each recorded timestep. Omega is a 3 by 1 vector. The difference between an estimated omega and a gyroscope-measured omega is simply the distance between the two vectors. The cost function returns the sum of all the squared differences between the estimated and gyroscope-measured omegas at each recorded timestamp. Powell's method finds the best rotation, bias, scale, and timeshift parameters that minimize this cost function.

The timeshift is about 61 ms, which indicates that the rate gyroscopes help the prediction task by providing information that would otherwise take about 61 ms to determine. The timeshift depends on the EKF parameters, which control the tradeoff between the timeshift and the smoothness of the estimated velocity signals. Smooth estimates result in long delays (i.e., large timeshifts). But if the parameters are set to minimize the timeshift, the resulting estimates are very noisy. Neither extreme is useful for the prediction problem. Good estimates that are greatly delayed in time are not of much use, nor are promptly-provided estimates that are mostly noise. Empirically, the best compromise between smoothness and delay for this particular filter seems to be when the timeshift is 61 ms. This was determined by experimenting with the parameters and comparing the estimated velocities against the gyroscope-measured velocities. The way to avoid this tradeoff is

to directly measure the angular head velocities by using rate gyroscopes, instead of estimating velocities from the reported orientations.

The other approach integrates the gyroscope measurements and compares those against the quaternions reported by the tracker. Integrating noisy measurements over long periods leads to drift problems, because errors accumulate. Therefore, I take a slightly different approach. I run the same EKF described in Chapter 4 to estimate the orientation terms and its derivatives. However, I do not use the predictor described there. Instead, imagine that, somehow, the exact future angular velocities were available to the predictor. Then starting with the last known position and integrating those future velocities should yield perfect predictions. Of course, this is impossible in real time, but not in simulation. I use this "ideal noncausal predictor" that integrates the filter-estimated velocities from the gyroscopes. The filter-estimated velocities depend upon the bias, scale, and rotation parameters that I am trying to find. This prediction is performed at each recorded timestamp where sufficient "future" information is available. The difference between a pair of predicted and actual orientations is an angle, as defined in Section 4.5. The cost function returns the sum of the squared angles. Powell's method runs the EKF and recomputes the cost function for each set of parameters it tries. It finds the bias, scale, and rotation parameters that result in the best match.

The first method is faster than the second, although speed is not really a problem with either method. The slowest part is running an EKF across the entire motion sequence. The first approach only does this once, while the second must run an EKF every time Powell's method tries a new set of parameters. The first converges in a few hundred iterations, which takes less than one minute on a DECstation 5000. The second typically takes less than 10 minutes on the same machine.

Both methods produce results that are reasonably consistent with themselves and each other. Verifying accuracy is difficult, because I do not know what the true values are. Instead, I check for consistency. For example, on two different motion datasets, the first method produced gyro rotation parameters that differed by 0.6 degrees, offsets that were less than 2 counts apart (out of a scale of 4096), and scales that were within 2 degrees

per second of each other. Two different motion datasets, one analyzed by the first method and the other by the second, produced similar results. The rotation parameters were 0.5 degrees apart, offsets under 3 counts apart, and scales less than 4 degrees per second apart. The gyroscope scales stay close to 300 degrees per second, which is the manufacturer-specified scale.

Determining accelerometer parameters: I also developed two different methods of determining the accelerometer parameters. The first makes use of the fact that when the three accelerometers are still, they only detect gravity. The second tries to integrate the acceleration values twice and compare those against the tracker positions. In practice, the first method works, but the second does not. Furthermore, neither method can reliably measure the position vectors **F0**, **F1**, and **F2**.

The first method asks the user to place the HMD or tracker on a stand that can be tilted into different static orientations. I use an ordinary camera stand that has been modified to hold the 4-hat. At each static orientation, record the accelerometer readings and the HMD orientation reported by the tracker. About 20-30 different orientations suffice, and the collected measurements form a dataset that is processed offline.

The geometric constraint that each measurement shares is that the vector reported by the accelerometers must point straight down in World space, because gravity is the only force that affects the accelerometers in the static case. After biasing, scaling, and rotating the reported accelerations into World space, they should all point straight down at 9.8 meters per second squared. Powell's method searches for the offsets, scales, and rotation parameters that best satisfy this criterion.

Note that this approach cannot determine the *positions* of the accelerometers on the HMD (vectors **F0**, **F1**, and **F2**), because the positions only affect accelerometers in motion.

This approach yields reasonably consistent results. On two separate datasets, the rotation parameters were within 0.5 degrees, the offsets were within 3 counts (out of 4096), and the scales were within 0.02 meters per second squared.

This method is fast, converging in several hundred iterations requiring less than 10 seconds on a DECstation 5000.

In contrast, the second method is similar in flavor to the techniques used to compute the gyroscope parameters, but it does not work as well for the accelerometers as it does for the gyroscopes. When comparing accelerations with positions, three options are available: differentiate positions twice, differentiate positions once and integrate accelerations once, or integrate accelerations twice. The problem is that a combination of two integration or differentiation steps is needed, versus just one in the case of the gyroscopes. Two steps are difficult to achieve. While the Kalman filter seems capable of doing numerical differentiation once, two differentiation steps generate very noisy results. Integrating twice means that drift grows as a function of *t* squared instead of just *t*, where *t* is the integration time. Integrating once and differentiating once combines the two problems and is no better than the other two choices.

The second method tries integrating the accelerometers twice. Similar to the gyroscope case, I run the Kalman filters described in Chapter 4, then do "ideal noncausal prediction" by integrating the estimated linear accelerations. Theoretically, this should match the reported tracker positions. In practice, these ideally-predicted positions do not converge on a good match with the tracker positions. The result: the computed parameters are not consistent across different datasets. For example, the scales varied by over 0.6 meters per second squared. Because I could not get consistent results, I did not consider this method a success.

Another reason why the second method does not work is that the accelerometer outputs do not appear to correspond well with the tracker positions. Drift is lethal when double integrating the accelerometers for anything over 100 ms. But even with integration periods under 100 ms, the integrated readings do not match the tracker positions well. Errors typically exceed one cm. While double integration is clearly a major factor, another problem may be distortion in the positions reported by the optoelectronic tracker, which would prevent a close match with the accelerometer signals.

In contrast, the gyroscope outputs correspond well with the tracker orientations, which is why the gyroscope parameter calibration routines work. As an experiment, I recorded a long motion sequence of tracker and gyroscope data, then used Powell's method offline to search for the gyroscope parameters that yielded the best match between the integrated gyroscope measurements and the reported orientations. The integration took place across the entire duration of the motion sequence, which was four and a half minutes. Amazingly, Powell's method was able to find a set of parameters that resulted in an average error of 0.5 degrees between the integrated gyroscope readings and the reported orientations! Achieving such a close match across a long integration interval was a surprising result. It provides confidence in the accuracy of both the rate gyroscopes and the reported head orientations. This result also suggests that it might be possible to solely use the gyroscopes to track head orientation across short time intervals. This requires developing or applying alternate means of dynamically adjusting the gyroscope parameters, since Powell's method is not suitable for real-time searches.

I have not found a way to measure the positions of the accelerometers (the vectors **F0**, **F1**, and **F2**). The first method cannot detect those, and the second method simply was not sensitive to those values. It would converge to ridiculous distances, such as over one meter. In practice, all I do is measure these distances the best I can with a ruler.

The parameters that are most reliably determined appear to be the ones that the filter is most sensitive to, which are presumably also the ones that affect prediction accuracy the most.

Determining noise parameters: The Kalman filters require covariance matrices that indicate how much to trust the measurements and the motion model. These are matrices $\mathbf{E}_{\mathbf{orient}}$ and $\mathbf{R}_{\mathbf{orient}}$ for the orientation EKF and $\mathbf{E}_{\mathbf{tx}}$, $\mathbf{E}_{\mathbf{ty}}$, $\mathbf{E}_{\mathbf{tz}}$, $\mathbf{R}_{\mathbf{tx}}$, $\mathbf{R}_{\mathbf{ty}}$, and $\mathbf{R}_{\mathbf{tz}}$ for the position Kalman filters. The basic approach is simple. Collect several motion sequences of recorded head locations and inertial data. Offline, run the filters and predictors on that data, exactly as described in Chapter 4. Let the error be the difference between the predicted and actual positions and orientations. The cost function is the sum of the squared errors at each timestep. Pick the covariance matrices that

result in the lowest prediction error. Use Powell's method to find the matrices that yield this lowest error.

The main trick is determining which parameters to vary, because these matrices are large. $\mathbf{E}_{\mathbf{orient}}$ is 10 by 10, containing 100 parameters. $\mathbf{R}_{\mathbf{orient}}$ is 7 by 7, containing 49 parameters. $\mathbf{E}_{\mathbf{tx}}$, $\mathbf{E}_{\mathbf{ty}}$ and $\mathbf{E}_{\mathbf{tz}}$ have 9 parameters each, and $\mathbf{R}_{\mathbf{tx}}$, $\mathbf{R}_{\mathbf{ty}}$ and $\mathbf{R}_{\mathbf{tz}}$ have four each. Since the matrices should be symmetric, that cuts down the number of unique parameters in each type of matrix to 55, 28, 6 and 3, respectively. Even so, searching for 113 separate parameters takes too long.

To reduce the number of parameters to something more tractable, I make a number of simplifying assumptions. First, I assume that all cross-covariance terms are zero. That is, all non-diagonal terms in these matrices are declared to be zero. Next, certain terms along the diagonals are set equal to each other, if they are linked in some way. For example, all four terms in $\mathbf{R}_{\mathbf{orient}}$ that specify the noises in the measured quaternions are set equal to each other. Finally, I use the same **E** and **R** matrices for the *X*, *Y*, and *Z* translation directions. To summarize:

$$\mathbf{E}_{\mathbf{tx}} = \mathbf{E}_{\mathbf{ty}} = \mathbf{E}_{\mathbf{tz}}$$
$$\mathbf{R}_{\mathbf{tx}} = \mathbf{R}_{\mathbf{ty}} = \mathbf{R}_{\mathbf{tz}}$$

$$\mathbf{E}_{\mathbf{tx}} = \begin{bmatrix} E_{pos} & 0 & 0 \\ 0 & E_{vel} & 0 \\ 0 & 0 & E_{accel} \end{bmatrix}$$

$$\mathbf{R}_{\mathbf{tx}} = \begin{bmatrix} R_{pos} & 0 \\ 0 & R_{accel} \end{bmatrix}$$

$$\mathbf{E}_{\mathbf{orient}} = \begin{bmatrix} E_{measured} & & & & \mathbf{0} \\ & \ddots & & & \\ & & E_{measured} & & \\ & & & E_{nonmeasured} & \\ \mathbf{0} & & & & \ddots \\ & & & & & E_{nonmeasured} \end{bmatrix}$$

$$\mathbf{R}_{\mathbf{orient}} = \begin{bmatrix} R_{quat} & & & & \mathbf{0} \\ & \ddots & & & \\ & & R_{quat} & & \\ & & & R_{gyro} & \\ \mathbf{0} & & & & \ddots \\ & & & & & R_{gyro} \end{bmatrix}$$

This leaves nine scalar parameters to search for: R_{quat}, R_{gyro}, $E_{measured}$, $E_{nonmeasured}$, R_{pos}, R_{accel}, E_{pos}, E_{vel}, and E_{accel}. Theoretically, $E_{measured}$, E_{pos} and E_{vel} should be zero, but I include them because having a small amount of noise there helps filter stability. Nine parameters is a tractable number. Powell's method requires less than two hours on a DECstation 5000 to find these parameters. I search for the orientation parameters first, then the position parameters, because the position filtering and prediction require the orientation terms but not vice-versa.

How reasonable are these simplifying assumptions? If the measurements are truly independent, then setting all cross-covariances in the **R** matrices to zero is accurate. This may not be the case with measured quaternions, however, since all four terms of a quaternion are nonlinearly related. The gyroscope, accelerometer, and translation measurements are probably independent, though. Each individual sensor within each class (gyroscope, accelerometer, etc.) could have different variances, but any such differences appear to be small, based on looking at the signals generated when the sensors stand still. For the **E** matrices, I doubt that the nondiagonal

terms should differ much from zero for the simple motion model. The main values to worry about are E_{accel} and $E_{nonmeasured}$, which can be set to a wide range of values while still generating similar prediction accuracies. Therefore, allowing nondiagonal values near E_{accel} and $E_{nonmeasured}$ to be nonzero is probably not necessary; the filter is not that sensitive. Finally, the three translation terms probably should have their own independent **E** matrices. I would expect motion along the *Z* axis, which is the vertical direction in World space, to have different characteristics than motion along the *X* and *Y* axes, simply because users tend to spend more energy moving horizontally than vertically in our demonstration HMD applications.

The parameters are not tightly tuned to specific motion sequences. It turns out that Powell's method finds a broad "plateau" region in parameter space where the cost does not vary much as the parameters change. There is a broad range of parameters that seem to perform well, and these ranges do not vary dramatically from motion sequence to motion sequence. This may be due to the simplicity and generality of the filter and the predictor. More sophisticated models would expect specific types of motion and require more tightly-tuned parameters.

Letting the noise matrices change with time may yield more accurate predictions. Right now, the optimization techniques return constant values for the various **E** and **R** matrices. Constant covariance matrices imply a stationary situation, but head motion is not stationary. Therefore, more accurate estimations and predictions might result if **E** and **R** varied with time. As a simple experiment, I tried increasing values in the **R** matrices as the head velocity increased, in the belief that the tracker becomes less accurate at fast velocities. This experiment did not improve the predictor's accuracy, but perhaps more sophisticated ways of changing the **E** and **R** matrices could.

5.3.3 Miscellaneous details

The **P** matrix should always be symmetric, because it is a covariance matrix. If the state variable is **X**, the covariance between **X**[2] and **X**[5], which is stored in **P**[2, 5], is the same as the covariance between **X**[5] and **X**[2], which is stored in **P**[5, 2]. Numerical errors can cause **P** to become

nonsymmetric as the Kalman filter runs. Therefore, in my implementation I only operate on and retain the upper triangular part of **P**, which is sufficient to specify the entire **P** matrix.

6. Theoretical limits

The previous chapters described a prediction method and demonstrated its effectiveness in an operational Augmented Reality system. This evaluation, like the evaluation of virtually all other previous head-motion predictors, was empirical. The prediction method is run in real time or in simulation and the resulting errors are recorded. Therefore, no simple formulas were used to generate those error values. Without such formulas, it is not easy to tell how the prediction errors will change if the system parameters, such as the system delay or the input head motion, are modified. The lack of formulas makes it difficult to compare predictors against each other or to evaluate how well a predictor will work in a different system.

This chapter addresses the need for such formulas by characterizing the theoretical behavior of the predictor described in Chapter 4. Expressing a limit on how well *any* arbitrary predictor could do is an essentially intractable problem, because no algorithm exists that finds the best model to a set of data, as explained in Section 6.1. However, it is possible to analyze *specific* predictors by characterizing their behavior in the frequency domain. Section 6.2 provides a brief introduction to frequency-domain analysis. One previous work [Riner92] showed the performance of its prediction method in the frequency domain. This chapter builds upon that work by showing the performance of two other types of predictors in the frequency domain and exploring how the performance changes as the system parameters are modified. The first type is a 2nd-order polynomial used in many predictors. Section 6.3 shows the performance of the 2nd-order polynomial, under the assumption that perfect measurements are available. Of course, that assumption describes an idealized situation. In reality, predictors often use Kalman filters to smooth the noisy measurements and estimate state variables that are not directly measured. Section 6.4 provides formulas that specify the performance of three specific Kalman-filter-based predictors.

With the formulas provided by the frequency-domain analysis, I can describe how a predictor's performance changes with different system parameters. Section 6.5 shows how to do this with three specific examples. First, it quantifies the distribution and growth of errors as the prediction interval changes. Second, it shows how to estimate the spectrum of the predicted motion, given the spectrum of the original head motion. And third, it estimates the maximum time-domain error in the predicted signal, allowing designers to determine the maximum acceptable system delay given the maximum tolerable error.

Finally, Section 6.6 covers some implementation details that one should be aware of when performing a frequency-domain analysis on collected data. The techniques used in this chapter might be applied to any other linear predictor and could form a basis for comparing head-motion predictors.

Throughout this chapter, ω is angular frequency. That is, $\omega = 2\pi f$, where f is the frequency in Hertz. Also, j is the square root of —1.

6.1 Limits of arbitrary prediction

Since the inertial-based predictor described in Chapter 4 does not produce perfect predictions, the following question is opened: Is there a bound on how much improvement prediction can achieve, given the choice of any possible prediction method? That is, what is the largest improvement any predictor can hope to accomplish? Unfortunately, expressing a bound given the availability of any arbitrary predictor is basically an intractable problem, not because the predictors used may be nonlinear, but primarily because the head motion prediction problem is nonstationary and difficult to model perfectly.

Nonlinearity is not the main problem. Implementing optimal nonlinear predictors is rarely feasible in practice, because any optimal nonlinear predictor requires knowledge and consideration of the conditional probabilities between the state and *all* previous measurements, as mentioned on p. 261 of [Lewis86]. As Chang describes it, "the optimal (conditional mean) nonlinear

estimator cannot be realized with a finite-dimensional implementation, and consequently all practical nonlinear filters must be suboptimal" [Chang84]. However, it may be possible to represent head motion adequately with linear models. The Kalman filters used in Chapter 4 for the translation terms are linear. While the Extended Kalman Filter used for the orientation terms is nonlinear, Section 4.1 discussed how orientation might be linearized for prediction purposes. Furthermore, it is theoretically possible to derive nonlinear limits, even if those are not practical to implement.

The real problem is the combination of the following two properties: 1) Head-motion signals are nonstationary, and 2) No algorithm exists that can select the optimal model for a given situation. The combination is important, because perfect prediction is theoretically possible for bandlimited stationary signals. A *nonstationary* signal is one where the statistical properties of the signal, such as the mean and the covariances, change with time. For example, the user might keep still for a while, then suddenly start moving his head around quickly. The statistical properties will change drastically. A *stationary* signal is one whose statistical characteristics do not change with time. *Bandlimited* signals are ones that have no energy in the frequency domain beyond a specified frequency. Every bandlimited stationary signal will be periodic. If the predictor samples such a signal at several times the Nyquist rate, and these samples have no noise, and if the predictor has access to past measurements of the signal with no limits on how far back into the past one can draw upon, then it is possible to make a perfect prediction of any future value of the signal [Splettsösser82], as discussed in Section 4.1. However, head-motion signals are nonstationary, and any practical measurements of those signals include noise, so this theoretical result is not achievable in practice.

The problem of finding models and their associated parameters to match collected data is sometimes called the *system identification* problem, and no systematic technique exists for finding the optimal model for arbitrary data. What is an optimal model? Ideally, a model should be able to predict future values so well that the error between the predicted signal and the true signal has the characteristics of white noise. No other model can do better because no systematic information is left to be extracted. In essence, any

potentially better model must predict the remaining difference, which is white noise, and that cannot be done. Note that the definition of the Kalman filter assumes the existence of such a model, but it does not specify how to generate the model in the first place! In general, an optimal model does not exist. The question then becomes, what is the model that comes closest to the nonachievable optimal model? No algorithm exists that can determine this model. Regression techniques specify how to find parameters that achieve the best fit of a *specific model*, such as a line, a curve, or a sinusoid, to observed data. They cannot say which model to pick, nor can they find the best model out of the arbitrary, infinite space of potential models. A theoretical limit of how well the best predictor could possibly do on head-motion signals cannot be specified. In practice, system identification usually requires experimentation to find an acceptable model [Fleming89].

6.2 Frequency-domain analysis techniques

6.2.1 Introduction

Since it is not possible to find an upper bound for how well the best possible predictor performs, I instead focus on characterizing the properties of the predictors and filters described in Chapter 4. This characterization is provided by frequency-domain analysis techniques, which draw upon results from spectral analysis, linear systems theory, the Fourier transform, and the Z-transform. None of these techniques is new. My contribution comes from applying them to this particular problem and analyzing the results. Although this chapter assumes basic knowledge of these techniques, this section provides a brief introductory background. For details, see [Brown92] [Lewis86] [Oppenheim83] [Phillips90] [Priestley81].

The frequency domain can be introduced by comparing it against the *time domain*. A time-domain function $g(t)$ returns a value based upon the time t. Plotting the function values versus time shows how the magnitude of the function changes with time. However, this is not the only way to represent the function $g(t)$. It is possible to change the representation so that the function outputs values based on frequency, rather than time. For example, the function might be based upon angular (or rotational) frequency ω, so the

function values can be plotted versus frequency, showing the distribution of energy at various frequencies. A function represented in this manner is said to be in the *frequency domain*. By convention, time-domain functions are written with lower-case letters and frequency-domain functions are written with capital letters:

Time domain	Frequency domain
$g(t)$	$G(\omega)$

A time-domain representation and a frequency-domain representation are two different ways of looking at the same function. The representations are equivalent, so it is possible to convert from one representation to the other. In this context, the operation that computes the frequency-domain representation of a function, given a time-domain representation, is called a *transform*. An *inverse transform* performs the opposite operation, computing the time-domain representation from a frequency-domain representation.

The next two sections focus on two specific examples of transforms: the *Fourier Transform* and the *Z-Transform*. Both will be used later in this chapter. Section 6.3, which characterizes the 2nd-order polynomial predictor, uses the Fourier Transform. Section 6.4, which analyzes the Kalman-filter-based predictor, makes use of the Z-Transform.

6.2.2 The Fourier Transform

The Fourier Transform converts time-domain functions into a domain where the basis functions are of the form $e^{j\,\omega t}$, where e is 2.71828…, the base of the natural logarithm. Note that the exponent is purely imaginary, and the coefficient for each basis function is complex, in general. Complex numbers are represented in two equivalent forms, rectangular and polar:

Rectangular (real x, imaginary y):	$x + j\,y$
Polar (magnitude M, phase ϕ):	$M\,e^{j\phi}$

To convert between the two forms, use the following formulas:

Rectangular $\rightarrow$ Polar: $\quad M = \sqrt{x^2 + y^2}, \;\; \text{ø} = \tan^{-1}\left(\frac{y}{x}\right)$

Polar $\rightarrow$ Rectangular: $\quad x = M\cos(\text{ø}), \quad y = M\sin(\text{ø})$

The basis functions in the Fourier domain are sinusoids. To show why this is so, I will first derive some expressions that convert the Fourier-domain basis functions into sinusoids. First, take power series expansions of e^x, sin(x), and cos(x):

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^N}{N!} + \ldots$$
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

Then:

$$\begin{array}{rcccccccccccc} e^{jx} & = & 1 & + & jx & - & \frac{x^2}{2!} & - & j\frac{x^3}{3!} & + & \frac{x^4}{4!} & + & j\frac{x^5}{5!} & - & \ldots \\ j\sin(x) & = & & & jx & & & - & j\frac{x^3}{3!} & & & + & j\frac{x^5}{5!} & & \ldots \\ \cos(x) & = & 1 & & & - & \frac{x^2}{2!} & & & + & \frac{x^4}{4!} & & & - & \ldots \end{array}$$

Therefore:

$$e^{jx} = \cos(x) + j\sin(x)$$
$$e^{-jx} = \cos(-x) + j\sin(-x) = \cos(x) - j\sin(x)$$

Combine the previous two expressions and solve for the cosine term:

$$\cos(x) = \frac{e^{jx} + e^{-jx}}{2}$$

With this result, I can convert the $e^{j\omega t}$ Fourier-domain basis functions into sinusoids. First note that frequencies can be both positive and negative. Also, the time-domain values in the head-motion prediction problem are purely real. A basic property of the Fourier Transform is that if the time-domain values are purely real, then the complex coefficients in the equivalent Fourier-domain representation at every frequency pair ω and —ω are complex

conjugates. That is, the magnitudes of both complex coefficients are the same, and the phases are also the same except that one is negative and the other positive. Let the complex coefficient at frequency ω be A, where $A = M e^{j \phi}$. Then adding the two basis functions at the frequency pair ω and $-\omega$, along with their complex coefficients, yields the following:

$$\begin{aligned} & M e^{j \phi} e^{j \omega t} + M e^{-j \phi} e^{-j \omega t} \\ & = M \left(e^{j(\omega t + \phi)} + e^{-j(\omega t + \phi)} \right) \\ & = 2M \cos(\omega t + \phi) \end{aligned}$$

by using the previously derived expression for cosine. Thus, the sum of the two basis functions at each frequency pair is a sinusoid.

The Fourier transform applies to continuous signals or to discrete approximations of continuous signals. In the continuous case, integrating all the basis sinusoids yields the original time-domain function. In the discrete case, the integral is replaced with a summation. An important property is that it is easy to compute the derivatives of the function from the Fourier representation. Simply take the derivatives of all the basis sinusoids, evaluate that function at the specified time, and integrate (or sum) the results. In Section 6.3, I treat the signal as continuous. When actually implementing Fourier Transforms on a computer, I use a fast version of the Discrete Fourier Transform, called the *Fast Fourier Transform* (FFT). Table 6.1 lists some time-domain and Fourier-domain equivalents for the continuous Fourier Transform that I will use in Section 6.3.

	Time domain	Fourier domain
Linearity	$A g_1(t) + B g_2(t)$	$A G_1(\omega) + B G_2(\omega)$
Time shift	$g(t - a)$	$e^{-j \omega a} G(\omega)$
Differentiation	$\frac{dg(t)}{dt}$	$j \omega G(\omega)$

Table 6.1: Time and Fourier domain equivalents

6.2.3 The Z-Transform

The Z-Transform applies to discrete signals that are evenly spaced in time. It uses basis functions that form a power series with terms z^k, where k is an integer representing each discrete timestep. For example, take a discrete time signal $x(k)$, where k ranges from 0 to ∞. This signal takes the following values:

$$x(0) = 1,\ x(1) = 2,\ x(2) = 3,\ x(3) = 4,\ \text{etc.}$$

Then the equivalent Z-domain function $X(z)$ is:

$$X(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + \ \ldots$$

That is, the coefficients for the basis functions in the Z-domain are the time-domain values.

Matrices can also be converted into the Z-domain. If a matrix is composed of several time-domain functions, the Z-domain equivalent of that matrix is computed by changing each separate function into the Z-domain. For example, let the 2 by 2 matrix **R**(t) be:

$$\mathbf{R}(t) = \begin{bmatrix} r_{11}(t) & r_{12}(t) \\ r_{21}(t) & r_{22}(t) \end{bmatrix}$$

Then the Z-domain equivalent **R**(z) is:

$$\mathbf{R}(z) = \begin{bmatrix} R_{11}(z) & R_{12}(z) \\ R_{21}(z) & R_{22}(z) \end{bmatrix}$$

To plot a Z-domain function versus frequency, use the following substitution:

$$z = e^{j\,\omega T}$$

where T is the period of the evenly spaced time-domain values, in seconds.

Table 6.2 lists the other properties of the Z-Transform that I will use in Section 6.4.

	Time domain	Z-domain
Linearity	$A\,x_1(k)+B\,x_2(k)$	$A\,X_1(z)+B\,X_2(z)$
Time shift	$x(k+1)$, with $x(0)=0$	$z\,X(z)$
Sine function	$\sin(a\,t)$	$\frac{z\sin(aT)}{z^2-2z\cos(aT)+1}$
Cosine function	$\cos(a\,t)$	$\frac{z\,(z-\cos(aT))}{z^2-2z\cos(aT)+1}$

Table 6.2: Time and Z-domain equivalents

6.2.4 Assumptions

The analysis in this chapter makes three specific assumptions. It assumes that the predictors are linear, that head-motion is separable into several 1-D signals, and that the input signals are sampled at evenly spaced discrete intervals.

The first assumption is that the predictor is linear. A basic result of linear systems theory states that any sinusoidal input into a linear system results in an output of *another sinusoid of the same frequency* but with possibly different magnitude and phase. If the input is the sum of many different sinusoids (e.g., a Fourier-domain signal), then the output can be computed by taking each sinusoid, changing its magnitude and phase, then summing (or integrating) the resulting output sinusoids. This is due to the property of *linear superposition*. Thus, it is possible to completely characterize linear systems by describing how the magnitude and phase of input sinusoids transform to the output as a function of frequency. This characterization is called a *transfer function*, and that is what Sections 6.3 and 6.4 derive.

Transfer functions return the ratio of the magnitudes and the difference of the phases between the input and output sinusoids, as a function of frequency. Say that $C(\omega)$ is a transfer function from input $X(\omega)$ to output $Y(\omega)$:

$$C(\omega)=\frac{Y(\omega)}{X(\omega)}=\frac{M_y\,e^{j\,\phi_y}}{M_x\,e^{j\,\phi_x}}=\frac{M_y}{M_x}e^{j(\phi_y-\phi_x)}$$

Then $C(\omega)$ returns the following magnitude ratio and phase difference:

$$\text{Magnitude ratio} = \frac{M_y}{M_x}$$

$$\text{Phase difference} = \phi_y - \phi_x$$

The second assumption is that the predictor separates 6-D head motion into six 1-D signals, each of which is handled by a separate predictor. This makes the analysis simpler. Section 4.5 discussed why this assumption is generally valid for head motion and listed potential problems caused by this assumption.

The assumptions of linear predictors and separable signals are generally reasonable for the translation terms, but not necessarily for the orientation terms. In Chapter 4, translation is handled by three linear 1-D predictors. However, orientation is represented by quaternions, which are neither linear nor separable. Section 4.1 described a way of converting quaternions into three 1-D Euler angle curves that can be handled by three linear 1-D predictors. Although that method is not practical in real time, it works fine in non-real-time situations and is a way to apply the analysis from this chapter to orientation motion.

Finally, the third assumption is that the input signals are measured at evenly spaced discrete time intervals. Chapter 5 showed that this is not the case in the operating Augmented Reality system, but this assumption does not seriously change the properties of the filter or the predictor as long as the sampling is done significantly faster than the Nyquist rate, and it makes the analysis easier. The sampling period I use for the analysis is 5 ms, which is considerably faster than typical 15-30 ms period between measurements in the actual system.

6.2.5 Ideal predictor transfer function

Sections 6.3 and 6.4 compute transfer functions, showing how the predictors behave in the frequency domain. It would be useful to compare those transfer functions against an ideal. The ideal predictor is one that does nothing more than shift the original signal in time. While this is noncausal and

therefore not implementable in real time, it is useful as a basis for comparison. Let the original signal be $g(t)$ and the prediction interval be p seconds. Then the ideal predicted signal $h(t)$ is:

$$h(t) = g(t + p)$$

What does the transfer function for the ideal predictor look like? By the timeshift formula in Table 6.1, the magnitude is unchanged. Therefore, the magnitude ratio is one for all frequencies. However, the phase difference varies with frequency. Since this difference is ωp, the phase difference appears as a line of slope p. Note that the line changes as p changes. Figures 6.1 and 6.2 graph the ideal magnitude ratio and phase difference, respectively.

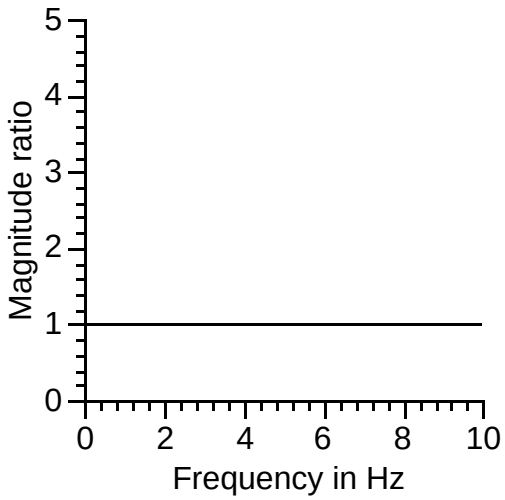


Figure 6.1: Magnitude ratio of ideal prediction transfer function

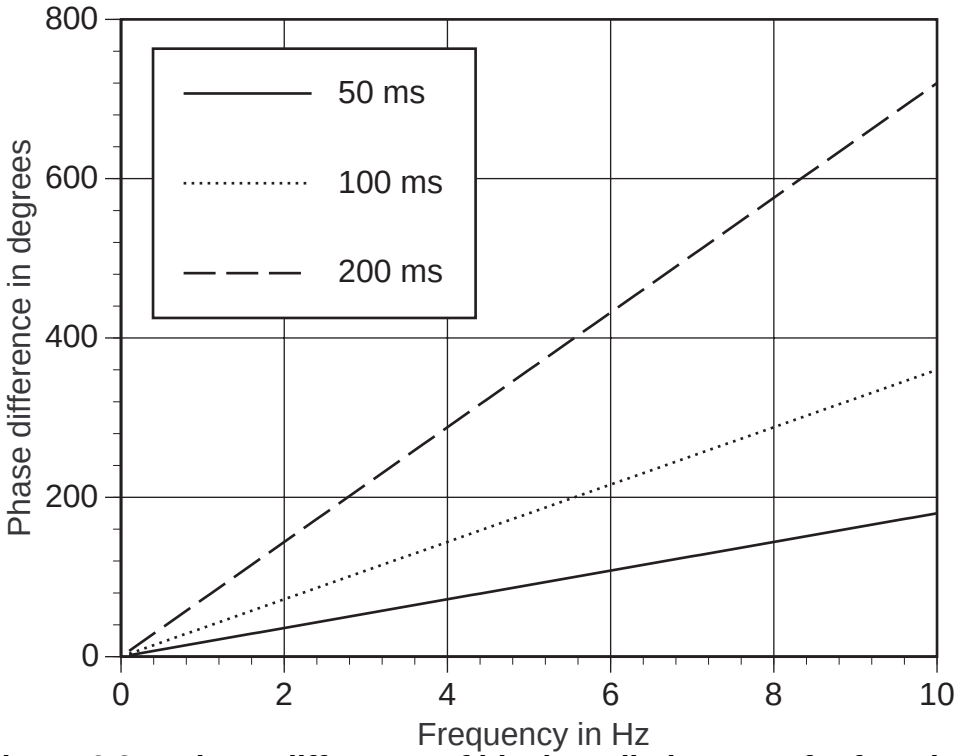


Figure 6.2: Phase difference of ideal prediction transfer function

6.2.6 RMS error metric

A problem with computing the error in the actual predictor transfer function, as compared to the ideal transfer function, is that transfer functions have two outputs: the magnitude ratio and the phase difference. Showing the error in just one or the other is insufficient. Both contribute to the overall error. At large magnitude ratios, the magnitude ratio difference dominates the error, but ignoring the phase difference is not wise at low magnitude ratios.

To capture the contributions of both magnitude and phase, I define a root-mean-square (RMS) error metric. At a particular angular frequency ω, let M be the magnitude of the actual transfer function and ϕ be the difference between the actual transfer function's phase difference and the ideal transfer function's phase difference. Then the RMS error at that frequency is defined as:

$$\sqrt{\frac{1}{T}\int_0^T \left[M\sin(\omega t+\phi)-\sin(\omega t)\right]^2 dt}$$

where T is the period of that frequency.

Another metric is to compute the error transfer function, which transforms the original signal to the error signal. The error signal is the difference between the predicted signal and the original signal. If $g(t)$ is the original signal and $h(t)$ is the predicted signal, recall that the ideal relationship is defined as:

$$h(t) = g(t+p)$$

Therefore, the error signal $e(t)$ is:

$$e(t) = h(t) - g(t+p)$$

Ideally, the error signal is always zero, but in practice that will not be the case.

Section 6.5.1 will use the RMS error metric to compare the accuracy of various predictors. Section 6.5.3 uses the magnitude ratio of the error transfer function to estimate the maximum time-domain error. The phase difference is not required for that estimate. Therefore, Sections 6.3 and 6.4 compute three formulas:

- The magnitude ratio of the prediction transfer function
- The phase difference of the prediction transfer function
- The magnitude ratio of the error transfer function

6.3 Analysis of 2nd-order polynomial predictor

In Section 4.4.2, I described the predictor used with the translation filters as a 2nd-order Taylor expansion of a function or the equation of a particle moving in a constant gravitational field. Let the original 1-D signal be $g(t)$ and the prediction interval be p seconds. Then the predicted signal $h(t)$ is computed as follows:

$$h(t) = g(t) + p\,g'(t) + \tfrac{1}{2}p^2\,g''(t)$$

The error $e(t)$ between the predicted signal and the actual signal is defined as:

$$e(t) = h(t) - g(t+p)$$
$$e(t) = g(t) + p\,g'(t) + \tfrac{1}{2}p^2\,g''(t) - g(t+p)$$

Now make some idealized assumptions:

- The position, velocity, and acceleration of the original signal are *perfectly known*.
- The signal is continuous, and all past values are known.

None of these assumptions is true in practice. Measurements are available only at discrete intervals, the measurements are noisy and distorted, and the system is able to measure only a subset of position, velocity, and acceleration. The more realistic case is analyzed in Section 6.4. This section makes these assumptions because they provide insight into how difficult the prediction problem is, even when perfect measurements are available. Polynomial-based predictors used with noisy and incomplete measurements will do worse than the ideal described here.

Now convert both the predicted signal and the error signal into the Fourier domain:

Time domain: $h(t) = g(t) + p\,g'(t) + \tfrac{1}{2}p^2\,g''(t)$

Fourier domain:
$$H(\omega) = G(\omega) + j\,\omega p\,G(\omega) + \tfrac{1}{2}(j\,\omega p)^2 G(\omega)$$
$$H(\omega) = \left(1 + j\,\omega p - (\omega p)^2\right)G(\omega)$$

Time domain: $e(t) = g(t) + p\,g'(t) + \tfrac{1}{2}p^2\,g''(t) - g(t+p)$

Fourier domain:
$$E(\omega) = G(\omega) + j\,\omega p G(\omega) + \tfrac{1}{2}(j\,\omega p)^2 G(\omega) - e^{j\,\omega p}\,G(\omega)$$
$$E(\omega) = \left(1 + j\,\omega p - \tfrac{1}{2}(\omega p)^2 - e^{j\,\omega p}\right)G(\omega)$$

With these Fourier-domain equations, I can compute the prediction transfer function, which transforms the original signal into the predicted signal, and the error transfer function, which transforms the original signal into the error signal. The next three sections derive the magnitude ratio and phase

difference for the prediction transfer function and the magnitude ratio for the error transfer function. Then Section 6.3.4 interprets the results.

6.3.1 Magnitude ratio of the 2nd-order prediction transfer function

The goal is to compute the magnitude of the predicted signal $H(\omega)$ and divide that by the magnitude of the original signal $G(\omega)$. This is expressed as a magnitude ratio:

$$\text{Magnitude ratio} = \frac{\|H(\omega)\|}{\|G(\omega)\|}$$

At any frequency ω, the value of $G(\omega)$ is a particular complex number. Call that complex number A, where:

$$A = x + j\,y$$

Now I can expand the expression for $H(\omega)$:

$$\begin{aligned} H(\omega) &= \left(1 + j\,\omega p - \tfrac{1}{2}(\omega p)^2\right) G(\omega) \\ &= \left(1 + j\,\omega p - \tfrac{1}{2}(\omega p)^2\right)(x + j\,y) \\ &= \left(x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2\right) + j\left(y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2\right) \end{aligned}$$

Then the squared magnitude of $H(\omega)$ is:

$$\|H(\omega)\|^2 = \left(x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2\right)^2 + \left(y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2\right)^2$$

Expanding the right-hand side of this equation and collecting terms yields:

$$\|H(\omega)\|^2 = \left(x^2 + y^2\right)\left(1 + \tfrac{1}{4}(\omega p)^4\right)$$

Now I can define the magnitude ratio:

$$\|G(\omega)\|^2 = x^2 + y^2$$

$$\frac{\|H(\omega)\|^2}{\|G(\omega)\|^2} = \frac{\left(x^2 + y^2\right)\left(1 + \tfrac{1}{4}(\omega p)^4\right)}{\left(x^2 + y^2\right)}$$

$$\frac{\|H(\omega)\|}{\|G(\omega)\|} = \sqrt{1 + \tfrac{1}{4}(\omega p)^4}$$

Note that when $p = 0$, the magnitude ratio is 1. That makes sense, because the predictor should just return the original signal when the prediction interval is zero.

Figure 6.3 graphs the magnitude ratio for three prediction intervals: 50 ms, 100 ms, and 200 ms. The ideal ratio is one at all frequencies, but the actual predictor magnifies high frequency components, even with perfect measurements of position, velocity, and acceleration.

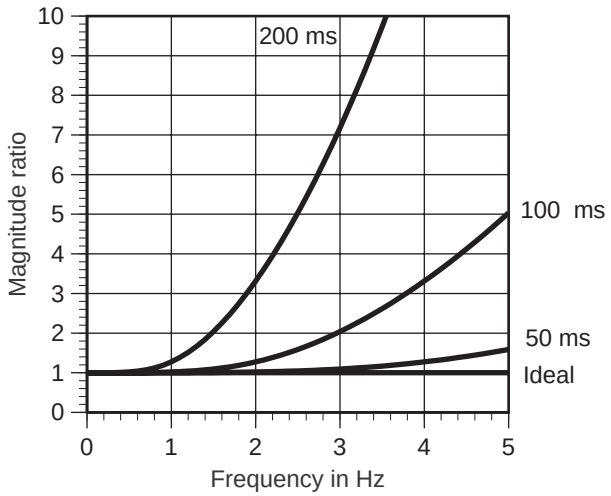


Figure 6.3: Magnitude ratio of 2nd-order prediction transfer function

6.3.2 Phase difference of the 2nd-order prediction transfer function

The goal is to find the difference between the phase of the original signal $G(\omega)$ and the predicted signal $H(\omega)$, at a particular frequency ω. Let α be the phase of $H(\omega)$ and ϕ be the phase of $G(\omega)$. Using the equation:

$$H(\omega) = \left(x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2\right) + j\left(y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2\right)$$

I can compute the phase of $H(\omega)$:

$$\alpha = \tan^{-1}\left(\frac{y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2}{x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2}\right)$$

Also, based on the definition of A, the phase of $G(\omega)$ is:

$$\phi = \tan^{-1}\left(\frac{y}{x}\right)$$

Now use the following trigonometric identity:

$$\tan(\alpha - \phi) = \frac{\tan(\alpha) - \tan(\phi)}{1 + \tan(\alpha)\tan(\phi)}$$

Substitute for $\tan(\alpha)$ and $\tan(\phi)$ in the right-hand side of the equation:

$$\tan(\alpha - \phi) = \frac{\dfrac{y + x\,\omega p - \frac{1}{2}y(\omega p)^2}{x - y\,\omega p - \frac{1}{2}x(\omega p)^2} - \dfrac{y}{x}}{1 + \left(\dfrac{y + x\,\omega p - \frac{1}{2}y(\omega p)^2}{x - y\,\omega p - \frac{1}{2}x(\omega p)^2}\right)\dfrac{y}{x}}$$

Simplify the right-hand side:

$$\tan(\alpha - \phi) = \frac{p\,\omega}{1 - \frac{1}{2}(p\,\omega)^2}$$

Therefore, the phase difference is:

$$\boxed{\alpha - \phi = \tan^{-1}\left(\frac{p\,\omega}{1 - \frac{1}{2}(p\,\omega)^2}\right)}$$

If the prediction interval p is zero, then the phase difference is zero, which makes sense.

Figure 6.4 plots the actual phase difference, along with the corresponding ideal phase difference, for three prediction intervals: 50 ms, 100 ms, and 200 ms. The actual phase differences match the ideal only at low frequencies, with the errors becoming larger at long prediction intervals or high frequencies. The actual phase differences asymptotically approach 180 degrees.

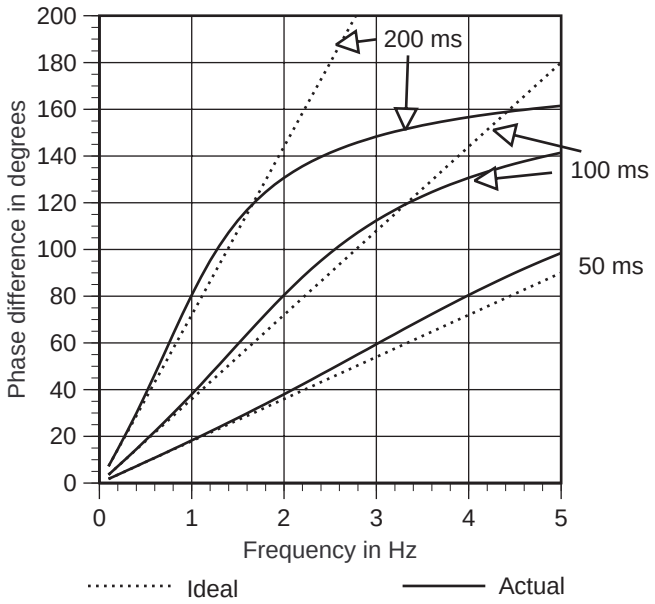


Figure 6.4: Phase difference of 2nd-order prediction transfer function

6.3.3 Magnitude ratio of the 2nd-order error transfer function

The goal is to compute the magnitude of the error signal $E(\omega)$ and divide that by the magnitude of the original signal $G(\omega)$. This forms the following magnitude ratio:

$$\text{Magnitude ratio} = \frac{\|E(\omega)\|}{\|G(\omega)\|}$$

As in Section 6.3.1, call A the complex number that is the value of $G(\omega)$ at a particular frequency ω, where:

$$A = x + j\,y$$

Now expand the expression for $E(\omega)$:

$$\begin{aligned} E(\omega) &= \left(1 + j\,\omega p - \tfrac{1}{2}(\omega p)^2 - e^{(j\,\omega p)}\right) G(\omega) \\ &= \left(1 + j\,\omega p - \tfrac{1}{2}(\omega p)^2 - \cos(\omega p) - j\sin(\omega p)\right)(x + j\,y) \\ &= \left(x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2 - x\cos(\omega p) + y\sin(\omega p)\right) \\ &\quad + j\left(y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2 - x\sin(\omega p) - y\cos(\omega p)\right) \end{aligned}$$

Then the squared magnitude of $E(\omega)$ is:

$$\|E(\omega)\|^2 = \left(x - y\,\omega p - \tfrac{1}{2}x(\omega p)^2 - x\cos(\omega p) + y\sin(\omega p)\right)^2 + \left(y + x\,\omega p - \tfrac{1}{2}y(\omega p)^2 - x\sin(\omega p) - y\cos(\omega p)\right)^2$$

By expanding the terms on the right-hand side and simplifying, this equation reduces to:

$$\|E(\omega)\|^2 = \left(2 + \tfrac{1}{4}(\omega p)^4 + \left((\omega p)^2 - 2\right)\cos(\omega p) - 2\omega p\sin(\omega p)\right)\left(x^2 + y^2\right)$$

Now I can define the magnitude ratio:

$$\|G(\omega)\|^2 = x^2 + y^2$$

$$\sqrt{\frac{\|E(\omega)\|^2}{\|G(\omega)\|^2}} = \sqrt{\frac{\left(2 + \tfrac{1}{4}(\omega p)^4 + \left((\omega p)^2 - 2\right)\cos(\omega p) - 2\omega p\sin(\omega p)\right)\left(x^2 + y^2\right)}{x^2 + y^2}}$$

$$\boxed{\frac{\|E(\omega)\|}{\|G(\omega)\|} = \sqrt{\left(2 + \tfrac{1}{4}(\omega p)^4 + \left((\omega p)^2 - 2\right)\cos(\omega p) - 2\omega p\sin(\omega p)\right)}}$$

If prediction interval $p = 0$, then this magnitude ratio goes to zero. That makes sense, because one expects no error with a prediction interval of zero.

6.3.4 Interpretation

The main result is that the magnitude of the predicted output grows rapidly as a function of both the prediction interval and the angular frequency of the original signal, which means it is important to keep the prediction interval small and avoid high-frequency signals. Figure 6.5 shows how the RMS error between the predicted and original signals changes with frequency, for three different prediction intervals. The error grows significantly as either the prediction interval *p* or the frequency of the original signal increases. The curve for the 50 ms prediction interval grows much more slowly than the curves representing the 100 ms and 200 ms prediction intervals. These graphs support the observation in Chapter 4 that this predictor must be combined with efforts to minimize system delays.

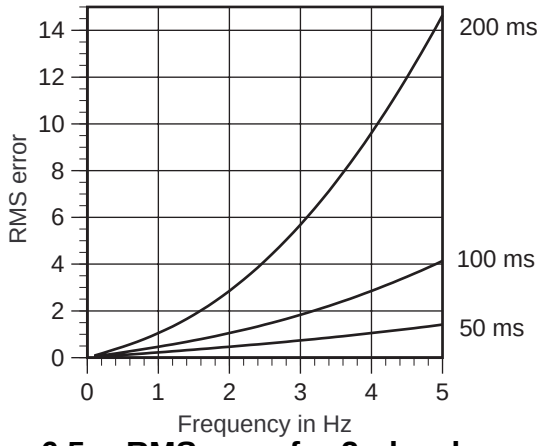


Figure 6.5: RMS error for 2nd-order predictor

Note the intimate relationship between *p* and ω in the formulas from Sections 6.3.1 through 6.3.3; they always occur together as ωp. This suggests a relationship between allowable bandwidth and the prediction interval. Halving the prediction distance means the signal can double in frequency while maintaining the same prediction performance. That is, bandwidth times the prediction interval yields a constant performance level.

Even if the prediction interval is small, the error will be large if high-frequency signals exist, even if those high-frequency signals have small magnitudes. This can be made clear with a simple example. Table 6.3 lists three sinusoids that form an original signal. If I set the prediction interval to 30 ms and run the predictor on that signal, the predicted outputs follow the actual curve fairly closely, as shown in Figure 6.6 for a small portion of the curve. The average error is 0.086, and the peak error is 0.160. Now declare the original signal to be the sum of the sinusoids listed in Table 6.4. The only difference between Tables 6.3 and 6.4 is the addition of one low-magnitude sinusoid at 60 Hz, which could represent a noise source. The original signals are almost exactly the same. However, adding the one extra sinusoid makes the prediction noisy and much less accurate, as shown in Figure 6.7. The average error is now 0.412 and the peak error 0.771.

Functions are of the form: $M \sin(2\pi f + \phi)$:

	Magnitude	Frequency (Hz)	Phase (radians)
Sine #1	5.0	1.0	0.5
Sine #2	3.0	2.0	-1.7
Sine #3	1.0	5.0	-0.3

Table 6.3: Three sinusoids

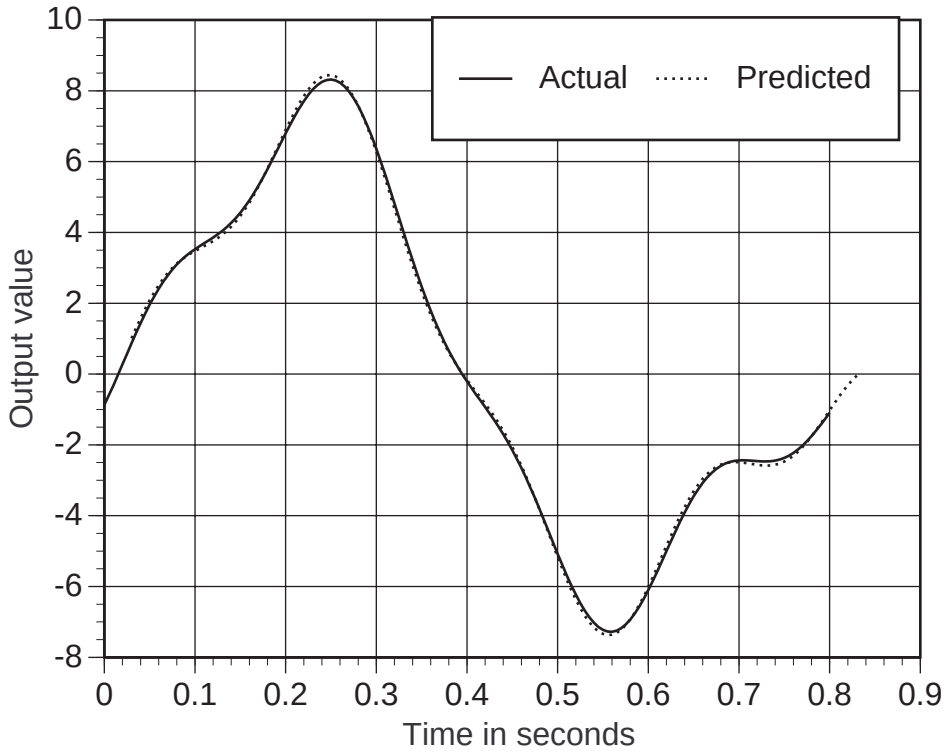


Figure 6.6: Portion of original and predicted signals from Table 6.3

Functions are of the form: $M \sin(2\pi f + \phi)$:

	Magnitude	Frequency (Hz)	Phase (radians)
Sine #1	5.0	1.0	0.5
Sine #2	3.0	2.0	-1.7
Sine #3	1.0	5.0	-0.3
Sine #4	0.01	60.0	0.2

Table 6.4: Four sinusoids

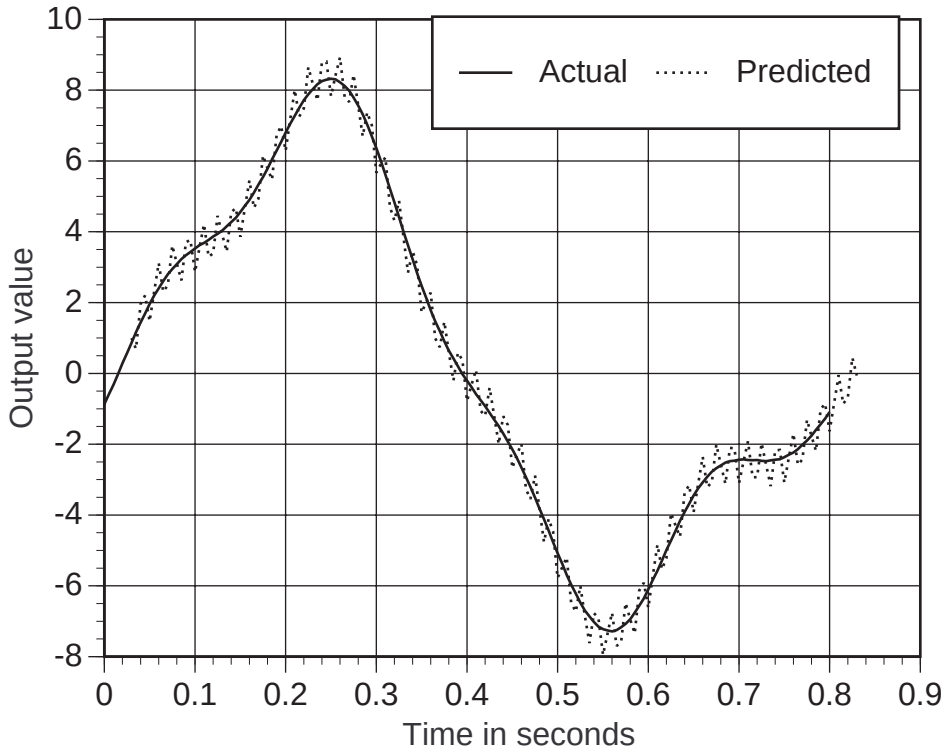


Figure 6.7: Portion of original and predicted signals from Table 6.4

This magnification of high-frequency signals shows up as "jitter" in the predicted signal, which was discussed in Section 4.5. This means that the original signal must not contain significant energy at high frequencies, or prediction will be ineffective. Section 6.5 will show what prediction does to the spectrum of a typical head-motion signal. This sensitivity to high-frequency components makes noise a problem, because white-noise processes contribute equally to low and high frequencies.

6.4 Analysis of Kalman-filter-based predictor

Section 6.3 assumed that the system had perfect, continuously-available measurements of position, velocity, and acceleration. This is an ideal situation that does not occur in reality. Instead, the measurements will be noisy and possibly distorted, and the system is rarely able to directly measure position and velocity and acceleration. Usually, the system is only able to measure a subset of those values, such as position only, or perhaps position and acceleration. Also, the measurements are not continuously sampled, but they are instead available only at discrete intervals.

Because of these limitations, Chapter 4 uses the 2nd-order polynomial predictor in combination with a Kalman filter. The Kalman filter provides estimates of position, velocity, and acceleration, based upon measurements and a motion model that may both be inaccurate. Depending on how the filter parameters are set, the Kalman filter can provide smoothed estimates, reducing the effect of high-frequency noise. Figure 6.8 shows a high-level dataflow diagram of how the two are combined.

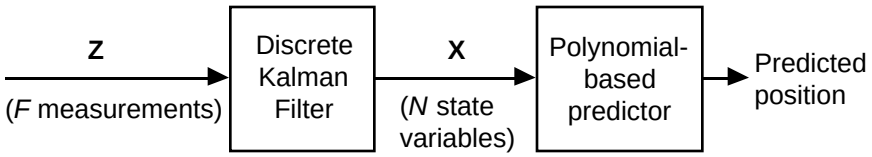


Figure 6.8: High-level dataflow for Kalman-filter-based predictor

This section analyzes the more realistic predictor. It is identical to the predictor used for the translation terms in Chapter 4, except it assumes that the measurements are sampled at evenly spaced discrete time intervals, as explained in Section 6.2.4. This allows the use of the Discrete Kalman Filter (DKF) for the analysis, which is simpler than the Continuous-Discrete Kalman Filter used in Chapter 4. In the case that the measurements arrive at evenly spaced intervals, the two filters can be set to generate exactly the same outputs (see Section 6.6.2 for details), and I use those techniques to set the DKF equations based upon the continuous-discrete equations. If the measurements do not arrive at evenly spaced intervals, the DKF can closely approximate the continuous case by setting the sampling period to a small value with respect to the periods between measurements, assigning the measurements to the closest discrete timestep, and running only time-update steps inside the filter when measurements are not available [Lewis86]. The DKF can also be applied to orientation if those signals are linearized by the method suggested in Section 4.1, instead of using the Extended Kalman Filter to do the linearization, which is what Chapter 4 does.

Because the signals are measured at evenly spaced time intervals, I use the Z-transform to convert the signals into the frequency domain. Therefore, the transfer functions must also be expressed in the Z-domain. Because the Kalman Filter uses multiple inputs and produces multiple outputs, its characterization in the frequency domain will be a *transfer matrix*, rather than a single transfer function that was the case in Section 6.3. The

transfer matrix is composed of individual transfer functions that show how each input is transformed to each output. For example, assume the system has three outputs **Y**(z) and two inputs **X**(z), for a total of six individual transfer functions. **Y**(z) is a 3 by 1 vector and **X**(z) is a 2 by 1 vector. Then the 3 by 2 transfer matrix **M**(z) is defined by $\mathbf{Y}(z) = \mathbf{M}(z)\mathbf{X}(z)$, where

$$\mathbf{Y}(z) = \begin{bmatrix} y_0(z) \\ y_1(z) \\ y_2(z) \end{bmatrix} \qquad \mathbf{X}(z) = \begin{bmatrix} x_0(z) \\ x_1(z) \end{bmatrix} \qquad \mathbf{M}(z) = \begin{bmatrix} \frac{y_0(z)}{x_0(z)} & \frac{y_0(z)}{x_1(z)} \\ \frac{y_1(z)}{x_0(z)} & \frac{y_1(z)}{x_1(z)} \\ \frac{y_2(z)}{x_0(z)} & \frac{y_2(z)}{x_1(z)} \end{bmatrix}$$

The contributions from every input signal must be combined to produce the true output signal. For example, computing the first output requires the following:

Let transfer function $G_0(z) = \dfrac{y_0(z)}{x_0(z)}$

Let transfer function $G_1(z) = \dfrac{y_0(z)}{x_1(z)}$

Then $y_0(z) = G_0(z)x_0(z) + G_1(z)x_1(z)$

Simply using the first measurement or the second measurement by itself is insufficient. This also means that examining each of the two transfer functions separately can be misleading, because they are both required to generate the output. Note that the transfer functions are not fractions; they are closer in form to partial derivatives, where all the partial derivatives are required to compute the total derivative. I have verified that computing signals in the frequency-domain in this fashion does produce the correct time-domain signals, by the methods described in Section 6.6.

The next section derives the transfer matrix for the Discrete Kalman Filter by itself. Then Section 6.4.2 derives the transfer matrix for the combination of the filter with the polynomial-based predictor. I investigate three specific configurations of the filter with the predictor: 1) only position is

measured, 2) position and velocity are measured, and 3) position and acceleration are measured. The first case is representative of most previous work, while the other two cases analyze what happens when inertial measurements are available. Sections 6.4.3 through 6.4.5 cover these three cases and graph the results.

6.4.1 The Discrete Kalman Filter transfer function

The Discrete Kalman Filter is similar to the Continuous-Discrete Kalman Filter described in Section 4.4.2, except that the time update step is now handled by a set of matrix operations instead of a numerical integrator. See Sections 4.2 and 4.4.2 for an introduction to the Kalman filter. In this section I list the equations and sketch the operation of the DKF, then derive the transfer matrix based on those equations. The basic model is:

$$\mathbf{X}(k+1) = \mathbf{A}(k)\mathbf{X}(k) + \mathbf{G}(k)w(k)$$
$$\mathbf{Z}(k) = \mathbf{H}(k)\mathbf{X}(k) + v(k)$$

In these equations, *k* is an integer representing the discrete timestep. Let *N* be the number of state variables, *F* be the number of measurements, and *L* be the number of model noise terms. Then the matrices are:

X(*k*) = *N* by *1* state variable
A(*k*) = *N* by *N* model
G(*k*) = *N* by *L* noise control
Z(*k*) = *F* by *1* measurements
H(*k*) = *F* by *N* measurements to state variable relationship
w(*k*) = white noise source with covariances **E**(*k*)
v(*k*) = white noise source with covariances **R**(*k*)
E(*k*) = *L* by *L* model covariance matrix
R(*k*) = *F* by *F* measurement covariance matrix

Two other matrices are used. These are **P**(*k*), which is an *N* by *N* state variable covariance matrix, and **K**(*k*), the *N* by *F* Kalman gain matrix. Note that **K** is a matrix, while *k* is an integer representing the timestep number. Also, **Z**(*k*) is a matrix, while *z* is the index for Z-domain representations.

To start the filter, set k=0 and set $\mathbf{X}(0)$ and $\mathbf{P}(0)$ to their initial values. Then at each new timestep k+1, there is a measurement $\mathbf{Z}(k+1)$. The discrete Kalman filter runs a *time update* step, followed by a *measurement update* step, to compute the new state variables $\mathbf{X}(k+1)$ and covariances $\mathbf{P}(k+1)$. These steps are repeated for every new measurement.

The time update step uses the following equations to update the state variables and covariance matrix, based on the model.

$$\mathbf{X}^{-}(k+1) = \mathbf{A}(k)\mathbf{X}(k)$$
$$\mathbf{P}^{-}(k+1) = \mathbf{A}(k)\mathbf{P}(k)\mathbf{A}(k)^{\mathrm{T}} + \mathbf{G}(k)\mathbf{E}(k)\mathbf{G}(k)^{\mathrm{T}}$$

The minus sign in the superscript for **X** and **P** indicates that this is a partial update of **X** and **P**.

To complete the update, run the measurement update step, which incorporates the new measurement $\mathbf{Z}(k+1)$ into **X**. The measurement update uses the following equations, which are the same as the measurement update from the Continuous-Discrete Kalman Filter listed in Section 4.4.2:

$$\mathbf{K}(k+1) = \mathbf{P}^{-}(k+1)\mathbf{H}(k+1)^{\mathrm{T}}\left[\mathbf{H}(k+1)\mathbf{P}^{-}(k+1)\mathbf{H}(k+1)^{\mathrm{T}} + \mathbf{R}(k+1)\right]^{-1}$$
$$\mathbf{P}(k+1) = \left[\mathbf{I} - \mathbf{K}(k+1)\mathbf{H}(k+1)\right]\mathbf{P}^{-}(k+1)$$
$$\mathbf{X}(k+1) = \mathbf{X}^{-}(k+1) + \mathbf{K}(k+1)\left[\mathbf{Z}(k+1) - \mathbf{H}(k+1)\mathbf{X}^{-}(k+1)\right]$$

Note that **I** is the N by N identity matrix.

Now that I have explained the basic operation of the discrete Kalman filter, I will point out three characteristics that are important for the frequency analysis. The first is that although all the matrices can vary with time, most do not. The models I use are simple and non-time-varying. **A**, **G**, **H**, **E**, and **R** are all constant matrices. Therefore, I can remove the k index from those matrices. The second characteristic to notice is that the values of **P** and **K**, which do vary with time, do not depend upon the measurements **Z**. That means it is possible to precompute those values for every timestep for any input data sequence. Finally, the third characteristic results from the combination of the first two: Matrices **P** and **K** eventually converge to constant values. In practice, this occurs quickly with my filters. Convergence occurs

with one to two seconds of data, for a 5 ms timestep. This means that in the steady-state mode, I can treat matrices **P** and **K** as constants.

I can now derive the transfer matrix for the steady-state Discrete Kalman Filter. This filter has only two time-varying matrices: **X**(*k*) and **Z**(*k*). When a new measurement **Z**(*k* + 1) arrives, the combined time update and measurement update steps are:

$$\mathbf{X}^{-}(k+1) = \mathbf{A}\,\mathbf{X}(k)$$
$$\mathbf{X}(k+1) = \mathbf{X}^{-}(k+1) + \mathbf{K}\left[\mathbf{Z}(k+1) - \mathbf{H}\,\mathbf{X}^{-}(k+1)\right]$$

Combine the two equations to get an expression for **X**(*k* + 1) in terms of **X**(*k*) and **Z**(*k* + 1):

$$\mathbf{X}(k+1) = \mathbf{A}\,\mathbf{X}(k) + \mathbf{K}\left[\mathbf{Z}(k+1) - \mathbf{H}\,\mathbf{A}\,\mathbf{X}(k)\right]$$
$$\mathbf{X}(k+1) = \left[\mathbf{A} - \mathbf{K}\,\mathbf{H}\,\mathbf{A}\right]\mathbf{X}(k) + \mathbf{K}\,\mathbf{Z}(k+1)$$

Now convert the equation into the Z-domain.

$$z\,\mathbf{X}(z) = \left[\mathbf{A} - \mathbf{K}\,\mathbf{H}\,\mathbf{A}\right]\mathbf{X}(z) + z\,\mathbf{K}\,\mathbf{Z}(z)$$
$$\left[z\,\mathbf{I} - \mathbf{A} + \mathbf{K}\,\mathbf{H}\,\mathbf{A}\right]\mathbf{X}(z) = z\,\mathbf{K}\,\mathbf{Z}(z)$$
$$\mathbf{X}(z) = \left[z\,\mathbf{I} - \mathbf{A} + \mathbf{K}\,\mathbf{H}\,\mathbf{A}\right]^{-1} z\,\mathbf{K}\,\mathbf{Z}(z)$$

Define **C**(*z*) to be the transfer matrix for the Discrete Kalman Filter. This specifies the relationship between the inputs, which are the measurements **Z**, and the outputs, which are the estimated state variables **X**.

$$\mathbf{X}(z) = \mathbf{C}(z)\,\mathbf{Z}(z)$$
where $\mathbf{C}(z) = \left[z\,\mathbf{I} - \mathbf{A} + \mathbf{K}\,\mathbf{H}\,\mathbf{A}\right]^{-1} z\,\mathbf{K}$

Note that **C**(*z*) is an *N* by *F* matrix. This matrix contains all the *N* by *F* transfer functions transforming each input to each output. For example, let *N* be 3 and *F* be 2. Then **X**(*z*) has three output signals in the Z-domain, and **Z**(*z*) has two input signals. I define them as follows:

$$\mathbf{X}(z) = \begin{bmatrix} x_0(z) \\ x_1(z) \\ x_2(z) \end{bmatrix} \qquad \mathbf{Z}(z) = \begin{bmatrix} z_0(z) \\ z_1(z) \end{bmatrix}$$

Then the 3 by 2 transfer matrix **C**(z) contains six transfer functions that specify the relationships between the individual input and output signals:

$$\mathbf{C}(z) = \begin{bmatrix} \frac{x_0(z)}{z_0(z)} & \frac{x_0(z)}{z_1(z)} \\ \frac{x_1(z)}{z_0(z)} & \frac{x_1(z)}{z_1(z)} \\ \frac{x_2(z)}{z_0(z)} & \frac{x_2(z)}{z_1(z)} \end{bmatrix}$$

Why don't noise matrices **E** and **R** appear in the final expression for **C**(z)? Don't they affect the transfer matrix? They do, and their effect is felt in the steady-state matrix **K**. To compute **C**(z), I have to run the DKF in simulation to determine what the steady-state **K** is. Different values of **E** and **R** will result in different steady-state **K** matrices.

I can plot the individual transfer functions inside **C**(z) against angular frequencies by the following procedure. For each angular frequency ω, set the complex number z to be $e^{j\omega T}$, where T is the period, in seconds, between consecutive measurements. Therefore:

$$z = e^{j\omega T} = \cos(\omega T) + j\sin(\omega T)$$

Note that z is a complex number, so the result is a matrix **C**(z) where each term is a complex number. The matrix routines must be able to multiply and invert matrices with complex components. The magnitude and phase of each complex number represent the magnitude ratio and phase difference for the transfer function that position in the matrix represents.

6.4.2 Transfer functions for combination of Kalman Filter and predictor

Figure 6.8 showed how the Discrete Kalman Filter is combined with the polynomial-based predictor. This combination has F measurements as inputs and one predicted position as the output. The goal is to generate a transfer function from the measured position to the predicted position, just as in Section 6.3. However, computing this transfer function is a problem if F is greater than one. Recall that the predicted position depends upon all the

inputs. If there are multiple inputs, say position and acceleration, then the transfer function from the measured position to the predicted position does not capture the contribution from the measured acceleration, so it is misleading to view that as the overall transfer function. Therefore, instead of an *F* by 1 matrix, the overall transfer matrix must be reduced to a 1 by 1 matrix to compute the desired transfer function.

Figure 6.9 shows how to do this. The overall 1 by 1 transfer matrix **O**(*z*) for the Discrete Kalman Filter combined with the polynomial-based prediction is the multiplication of three matrices. *G*(*z*) is the scalar input position signal. The measurement generator matrix **M**(*z*) transforms the position signal into the set of measurements required by the DKF, such as position and velocity. This assumes that position, velocity, and acceleration are perfectly matched. The DKF does not assume this, so it treats the inputs as completely separate signals. However, for the purpose of this analysis, I make this assumption so I can generate an overall transfer function from measured position to predicted position that captures the entire operation of the filter and the predictor. The outputs of the measurement generator are the measurements **Y**(*z*), which the DKF takes as input. Then the output of the DKF, the estimated states **X**(*z*), is sent into the predictor, which generates the predicted position *H*(*z*). The desired overall transfer function **O**(*z*) transforms the input position signal *G*(*z*) into the predicted position signal *H*(*z*).

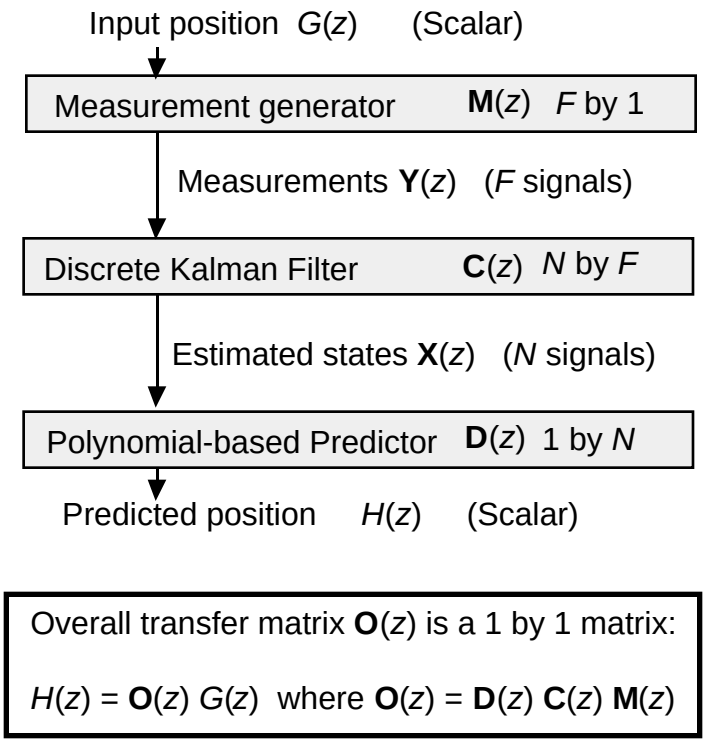


Figure 6.9: Transfer matrices for Kalman-Filter-based predictor

To compute **O**(*z*), I need expressions for **D**(*z*), **C**(*z*), and **M**(*z*). **C**(*z*) was computed in Section 6.4.1. **D**(*z*) is the polynomial predictor. Let *x*(*z*) be the estimated position, *v*(*z*) be the estimated velocity, *a*(*z*) be the estimated acceleration, and *p* be the prediction interval. If

$$\mathbf{X}(z) = \begin{bmatrix} x(z) \\ v(z) \\ a(z) \end{bmatrix}$$

then the corresponding **D**(*z*) is

$$\mathbf{D}(z) = \begin{bmatrix} 1 & p & \frac{1}{2}p^2 \end{bmatrix}$$

M(*z*) requires more work to compute. **M**(*z*) takes the original position signal *g*(*z*) and must be able to produce measured position $x_{measured}(z)$, measured velocity $v_{measured}(z)$, and measured acceleration $a_{measured}(z)$. Now $x_{measured}(z)$ is the same signal as *g*(*z*). Therefore, what remains is to compute the following two transfer functions:

$$\frac{v_{measured}(z)}{g(z)} \text{ and } \frac{a_{measured}(z)}{g(z)}$$

Computing these transfer functions is more difficult in the discrete case than in the continuous case. Differentiating position once or twice yields velocity and acceleration, respectively. In the Fourier domain, there is a simple relationship between a function and its derivative, which was used in Section 6.3. Unfortunately, no such relationship exists in the Z-domain. Therefore, I have to explicitly define the position, velocity, and acceleration signals, convert those into the Z-domain, then determine the Z-domain transfer function. Recall that the Fourier transform produces a set of sinusoids that, when added together, are equal to the original time-domain function. Therefore, at any particular angular frequency ω, the original signal is a sinusoid with unknown magnitude M and phase ϕ:

$$g(t) = M\sin(\omega t + \phi)$$

Then the corresponding measured velocity and acceleration signals are determined by taking the derivatives of $g(t)$:

$$v_{measured}(t) = \omega M\cos(\omega t + \phi)$$
$$a_{measured}(t) = -\omega^2 M\sin(\omega t + \phi)$$

Now use the following two trigonometric identities:

$$\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$$
$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$

Expand the three time-domain signals with those identities:

$$g(t) = M\left[\sin(\omega t)\cos(\phi) + \cos(\omega t)\sin(\phi)\right]$$
$$v_{measured}(t) = \omega M\left[\cos(\omega t)\cos(\phi) - \sin(\omega t)\sin(\phi)\right]$$
$$a_{measured}(t) = -\omega^2 M\left[\sin(\omega t)\cos(\phi) + \cos(\omega t)\sin(\phi)\right]$$

Convert these three signals into the Z-domain, using the substitutions listed in Table 6.2:

$$G(z) = M\left[\frac{z\sin(\omega T)\cos(ø) + z\left(z - \cos(\omega T)\right)\sin(ø)}{z^2 - 2z\cos(\omega T) + 1}\right]$$

$$_{measured}(z) = \omega M\left[\frac{z\left(z - \cos(\omega T)\right)\cos(ø) - z\sin(\omega T)\sin(ø)}{z^2 - 2z\cos(\omega T) + 1}\right]$$

$$A_{measured}(z) = -\omega^2 M\left[\frac{z\sin(\omega T)\cos(ø) + z\left(z - \cos(\omega T)\right)\sin(ø)}{z^2 - 2z\cos(\omega T) + 1}\right]$$

Clearly:

$$\frac{A_{measured}(z)}{G(z)} = -\omega^2$$

Now all that remains is to compute the other transfer function from position to measured velocity:

$$\frac{V_{measured}(z)}{G(z)} = \omega\left[\frac{z\left(z - \cos(\omega T)\right)\cos(ø) - z\sin(\omega T)\sin(ø)}{z\sin(\omega T)\cos(ø) + z\left(z - \cos(\omega T)\right)\sin(ø)}\right]$$

Cancel the z and substitute $z = e^{j\,\omega T} = \cos(\omega T) + j\sin(\omega T)$:

$$\frac{V_{measured}(z)}{G(z)} = \omega\left[\frac{\sin(\omega T)\cos(ø) - \sin(\omega T)\sin(ø)}{\sin(\omega T)\cos(ø) + j\sin(\omega T)\sin(ø)}\right]$$

$$\frac{V_{measured}(z)}{G(z)} = \omega\left[\frac{j\cos(ø) - \sin(ø)}{\cos(ø) + j\sin(ø)}\right] = j\,\omega\left[\frac{\cos(ø) + j\sin(ø)}{\cos(ø) + j\sin(ø)}\right]$$

$$\frac{V_{measured}(z)}{G(z)} = j\,\omega$$

Therefore, the three transfer functions used in **M**(z) are:

$$\frac{X_{measured}(z)}{G(z)} = 1 \qquad \frac{V_{measured}(z)}{G(z)} = j\,\omega \qquad \frac{A_{measured}(z)}{G(z)} = -\omega^2$$

Another transfer matrix that will be useful in Section 6.5.3 is the error transfer function, which transforms the original position signal to the error signal. Section 6.2.6 defined the error signal as:

$$e(t) = h(t) - g(t + p)$$

The goal is to derive the overall error transfer matrix **U**(z), which is defined as follows:

$$E(z) = \mathbf{U}(z)G(z)$$

Define

$$g_p(t) = g(t+p)$$

Since I already have an expression for $H(z)$, I must convert $g_p(t)$ into the Z-domain. Then I can change the entire error signal expression into the Z-domain and determine what **U**(z) is. Recall that for a particular frequency ω, the original signal $g(t) = M\sin(\omega t + \phi)$. Therefore:

$$g_p(t) = g(t+p) = M\sin\left(\omega(t+p)+\phi\right)$$
$$g_p(t) = M\sin\left(\omega t + \omega p + \phi\right)$$
$$g_p(t) = M\left[\sin(\omega t)\cos(\omega p+\phi) + \cos(\omega t)\sin(\omega p+\phi)\right]$$

Now convert $g_p(t)$ to the Z-domain:

$$G_p(z) = M\left[\frac{z\sin(\omega T)\cos(\omega p+\phi) + z\left(z-\cos(\omega T)\right)\sin(\omega p+\phi)}{z^2 - 2z\cos(\omega T)+1}\right]$$

Based on the previously-derived expression for $G(z)$, I can now derive the transfer function from $G(z)$ to $G_p(z)$:

$$\frac{G_p(z)}{G(z)} = \left[\frac{z\sin(\omega T)\cos(\omega p+\phi) + z\left(z-\cos(\omega T)\right)\sin(\omega p+\phi)}{z\sin(\omega T)\cos(\phi) + z\left(z-\cos(\omega T)\right)\sin(\phi)}\right]$$

Substitute $z = e^{j\omega T} = \cos(\omega T) + j\sin(\omega T)$:

$$\frac{G_p(z)}{G(z)} = \left[\frac{\cos(\omega p+\phi) + j\sin(\omega p+\phi)}{\cos(\phi)+j\sin(\phi)}\right] = \frac{e^{j(\omega p+\phi)}}{e^{j\phi}}$$

$$\frac{G_p(z)}{G(z)} = e^{j\omega p}$$

Now I can compute **U**(z) by converting the error signal expression into the Z-domain:

$$e(t) = h(t) - g_p(t)$$
$$E(z) = H(z) - G_p(z)$$
$$E(z) = \mathbf{O}(z)G(z) - \frac{G_p(z)}{G(z)}G(z)$$
$$E(z) = \left[\mathbf{O}(z) - \left[\frac{G_p(z)}{G(z)}\right]\right]G(z)$$

Therefore, the error transfer matrix **U**(*z*) is:

$$\mathbf{U}(z) = \mathbf{O}(z) - \left[e^{j\,\omega\,p}\right]$$
where $E(z) = \mathbf{U}(z)G(z)$

I have now derived the overall prediction and error transfer matrices for the Discrete Kalman Filter combined with the polynomial-based predictor. However, it is not obvious how these transfer matrices behave just from looking at the formulas. Therefore, the next three sections graph how the overall transfer matrices behave, as a function of frequency, for three specific examples:

- Case 1 (Section 6.4.3): Measured position
- Case 2 (Section 6.4.4): Measured position and velocity
- Case 3 (Section 6.4.5): Measured position and acceleration

Each section graphs the overall magnitude ratio and the phase difference from original position to predicted position. There are many other relationships in the transfer matrices that could be graphed, but I choose to focus on original position to predicted position because that is the crux of the prediction problem. Other relationships in the transfer matrices are briefly mentioned. The RMS errors for the three cases are shown in Section 6.5.1.

For each case, I specify the values of the various matrices **X**(*z*), **H**, **Z**(*z*), **A**, **K**, **M**(*z*) and **D**(*z*). These provide enough information to compute the transfer matrices **O**(*z*) and **U**(*z*) for each case. The results depend on the steady-state **K** matrix, which in turn depends on the noise parameters used to tune each Kalman Filter. For each case, I adjusted those parameters to perform a small amount of lowpass filtering on the last measurement in the

Z(z) matrix. Then I ran each DKF in simulation to determine the steady-state **K** matrices.

Throughout Sections 6.4.3 to 6.4.5, the following definitions apply:

$X(z)$ = Estimated position
$V(z)$ = Estimated velocity
$A(z)$ = Estimated acceleration
$X_{measured}(z)$ = Measured position
$V_{measured}(z)$ = Measured velocity
$A_{measured}(z)$ = Measured acceleration
T = Period, in seconds, between measurements (set to 5 ms)
p = Prediction interval, in seconds
N = The number of estimated state variables in $\mathbf{X}(z)$
F = The number of measurements in $\mathbf{Z}(z)$

6.4.3 Case 1: Measured position

This filter has measured position and estimated position and velocity:

$N = 2, F = 1$

$$\mathbf{X}(z) = \begin{bmatrix} X(z) \\ V(z) \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \mathbf{Z}(z) = \begin{bmatrix} X_{measured}(z) \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 0.568 \\ 41.967 \end{bmatrix}$$

$$\mathbf{M}(z) = \begin{bmatrix} 1 \end{bmatrix}, \quad \mathbf{D}(z) = \begin{bmatrix} 1 & p \end{bmatrix}$$

I tuned the parameters so that the estimated position matched the measured position closely, and the estimated velocity was a lowpassed version of the true, nonmeasured velocity. The estimated velocities were delayed by about 15 ms from the true velocities.

Acceleration is not included in **X**(z) because only measured positions are available. As previously mentioned in Section 5.3.2, estimating velocity and acceleration from measured positions requires numerical differentiation, an inherently noisy operation. Performing two numerical differentiation steps generates estimates that are too noisy or too delayed in time to be useful for

the prediction problem. Thus, without direct measurements of velocity or acceleration (which inertial sensors provide), the Case 1 filter only includes position and velocity in the state vector.

The DKF estimates velocity by combining numerical differentiation with lowpass filtering. Plotting the transfer function from measured position to estimated velocity shows that at low frequencies, the filter takes the derivative of the function. At high frequencies the filter changes to a lowpass filter, reducing the effect of noise at the cost of delaying the estimated velocities in time.

Figures 6.10 and 6.11 show the magnitude ratio and phase difference for $\mathbf{O}(z)$. Figure 6.10 plots the magnitude ratio for a prediction interval of 100 ms. The magnitude ratio becomes smaller at high frequencies because the polynomial predictor does not use acceleration:

$$h(t) = x(t) + p\,v(t)$$

However, the phase differences in Figure 6.12 do not match the ideal phase differences as closely as the 2nd-order predictor does with ideal measurements (Figure 6.4). This gives some indication of the penalty imposed by the lack of measured velocity and acceleration.

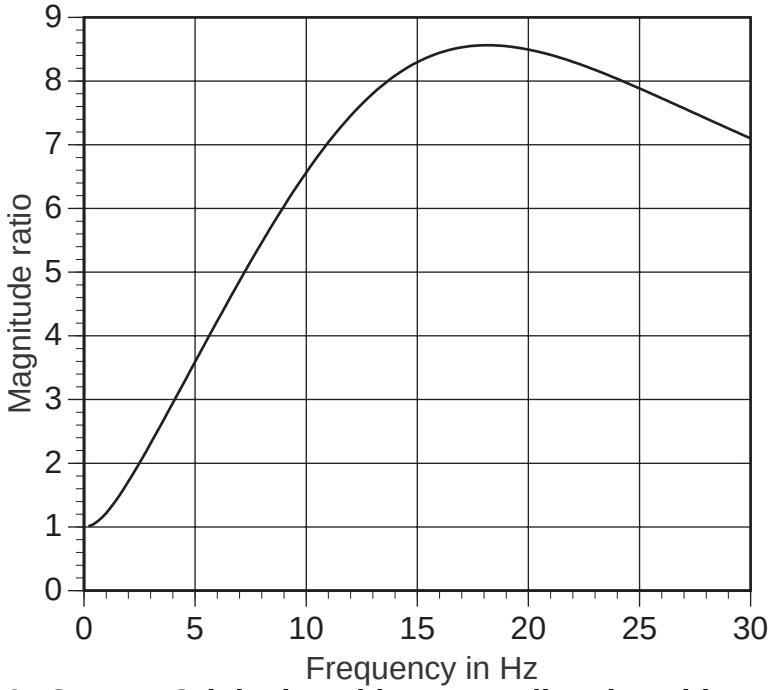


Figure 6.10: Case 1: Original position to predicted position magnitude ratio, for 100 ms prediction interval

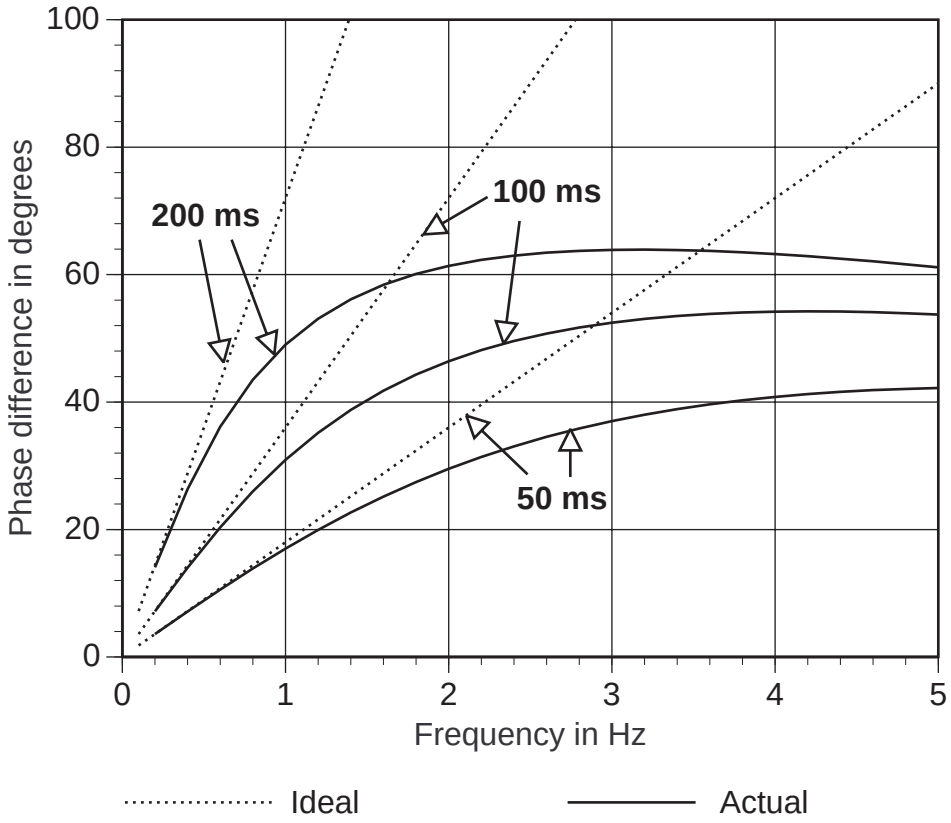


Figure 6.11: Case 1: Original position to predicted position phase differences

6.4.4 Case 2: Measured position and velocity

The Case 2 filter has measured position and velocity, with estimated position, velocity and acceleration:

$$N = 3,\ F = 2$$

$$\mathbf{X}(z) = \begin{bmatrix} X(z) \\ V(z) \\ A(z) \end{bmatrix},\ \mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},\ \mathbf{Z}(z) = \begin{bmatrix} X_{measured}(z) \\ V_{measured}(z) \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix},\ \mathbf{K} = \begin{bmatrix} 0.0576 & 0.0032 \\ 0.0034 & 0.568 \\ -0.0528 & 41.967 \end{bmatrix}$$

$$\mathbf{M}(z) = \begin{bmatrix} 1 \\ j\,\omega \end{bmatrix},\ \mathbf{D}(z) = \begin{bmatrix} 1 & p & \frac{1}{2}p^2 \end{bmatrix}$$

The parameters were tuned so the estimated position matched the measured position closely, the estimated velocity is a slightly lowpassed version of the measured velocity, and the estimated acceleration is a smoothed, delayed version of the true, unmeasured acceleration.

The DKF computes the estimated position, velocity, and acceleration in different ways, as described by the transfer functions from the measurements to those estimated state variables. At low frequencies, estimated position is basically the measured position, but at high frequencies, the filter estimates position by integrating the measured velocities. The estimated velocity is essentially the measured velocity; measured position has almost no effect. And estimated acceleration is computed by a combination of numerical differentiation and lowpass filtering at high frequencies, much as Case 1 estimated velocity from measured position.

Figures 6.12 and 6.13 show the magnitude ratio and phase differences for **O**(*z*), the overall transfer function from original position to the predicted position. The magnitude ratio in Figure 6.12 is for a prediction interval of 100 ms. Note that the magnitude ratio becomes much larger than the Case 1 ratio at high frequencies, because the predictor makes use of estimated acceleration. Figure 6.13 compares the transfer function's phase differences

against the ideal phase differences. They match more closely than in Case 1, shown in Figure 6.11.

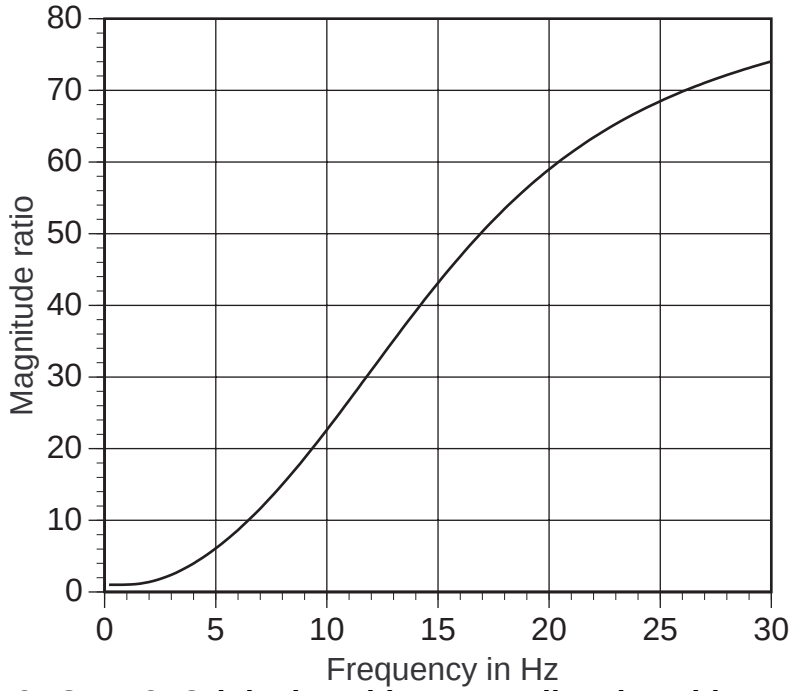


Figure 6.12: Case 2: Original position to predicted position magnitude ratio, for 100 ms prediction interval

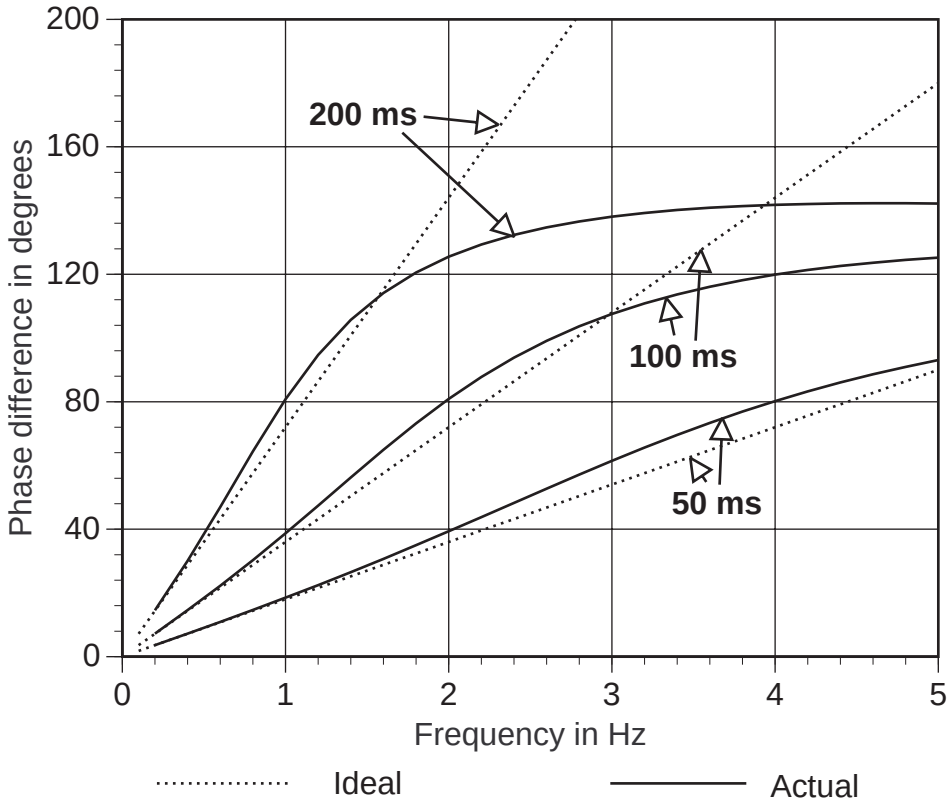


Figure 6.13: Case 2: Original position to predicted position phase differences

Since the predicted position depends on both the measured position and velocity, it is interesting to determine the relative contributions of each measurement to the predicted output. Let $M_{position}$ be the magnitude of the predicted position computed using only the measured position, with measured velocity set to zero. Similarly, $M_{velocity}$ uses only measured velocity, with measured position set to zero. Then I define the *relative contributions* of the two measurements as:

$$\text{Relative contribution of measured position} = \frac{M_{position}}{M_{position} + M_{velocity}}$$

$$\text{Relative contribution of measured velocity} = \frac{M_{velocity}}{M_{position} + M_{velocity}}$$

Figure 6.14 shows what these relative contributions look like. At low frequencies, measured position dominates, but at higher frequencies the Case 2 predictor relies on measured velocities instead.

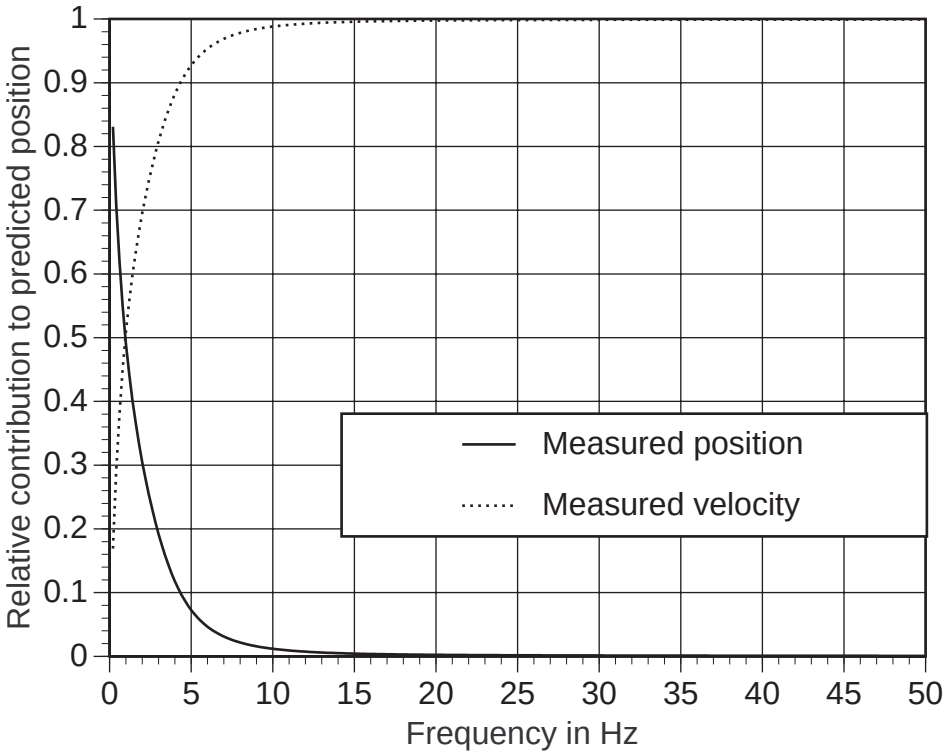


Figure 6.14: Case 2: Relative contribution of measured position and velocity to predicted position

6.4.5 Case 3: Measured position and acceleration

The Case 3 filter has measured position and acceleration, with estimated position, velocity and acceleration:

$N = 3, F = 2$

$$\mathbf{X}(z) = \begin{bmatrix} X(z) \\ V(z) \\ A(z) \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{Z}(z) = \begin{bmatrix} X_{measured}(z) \\ A_{measured}(z) \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 0.0807 & 0.000016 \\ 0.342 & 0.00345 \\ 0.0467 & 0.618 \end{bmatrix}$$

$$\mathbf{M}(z) = \begin{bmatrix} 1 \\ -\omega^2 \end{bmatrix}, \quad \mathbf{D}(z) = \begin{bmatrix} 1 & p & \frac{1}{2}p^2 \end{bmatrix}$$

The parameters were tuned so the estimated acceleration was a slightly lowpassed version of the measured acceleration, delayed by about 3 ms.

The transfer functions from the measurements to the estimated state variables describe how the DKF generates the estimates from the measurements. At low frequencies, estimated position is basically the measured position, but at high frequencies, the estimate is a blend of measured position and doubly-integrated acceleration. The estimated velocity is based on numerically-differentiated measured positions at low frequencies and integrated measured accelerations at high frequencies. Finally, estimated acceleration relies almost entirely on measured acceleration, with very little contribution from measured position.

Figures 6.15 and 6.16 show the magnitude ratio and phase difference for the Case 3 **O**(*z*) transfer matrix, from original position to predicted position. The magnitude ratio in Figure 6.15 was computed at a 100 ms prediction interval, and this ratio grows more rapidly than either the Case 1 or Case 2 magnitude ratios. Figure 6.16 compares the **O**(*z*) phase differences against the ideal phase differences for three prediction intervals.

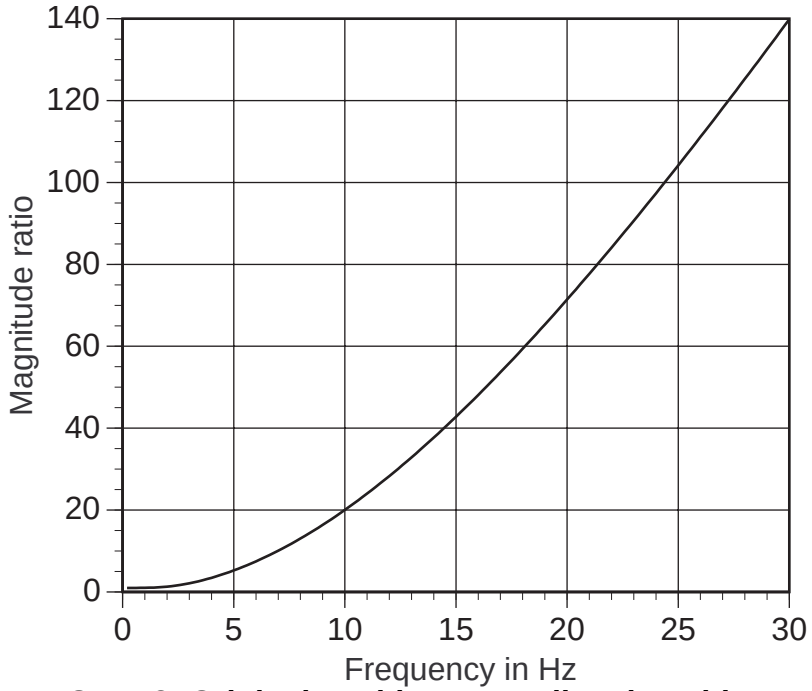


Figure 6.15: Case 3: Original position to predicted position magnitude ratio, for 100 ms prediction interval

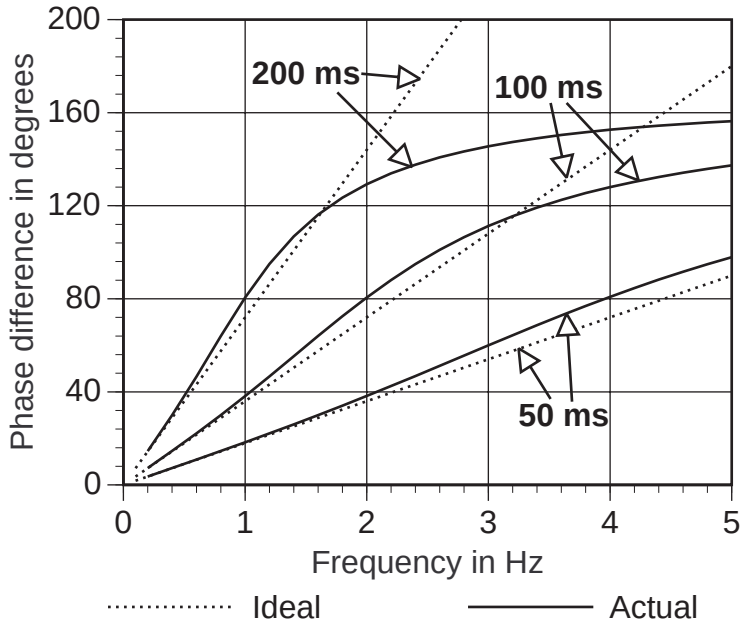


Figure 6.16: Case 3: Original position to predicted position phase differences

The relative contributions of measured position and acceleration to the predicted position, as defined in the previous section, are shown in Figure 6.17. At low frequencies, measured position is the largest contributor, but at higher frequencies the measured acceleration eventually dominates.

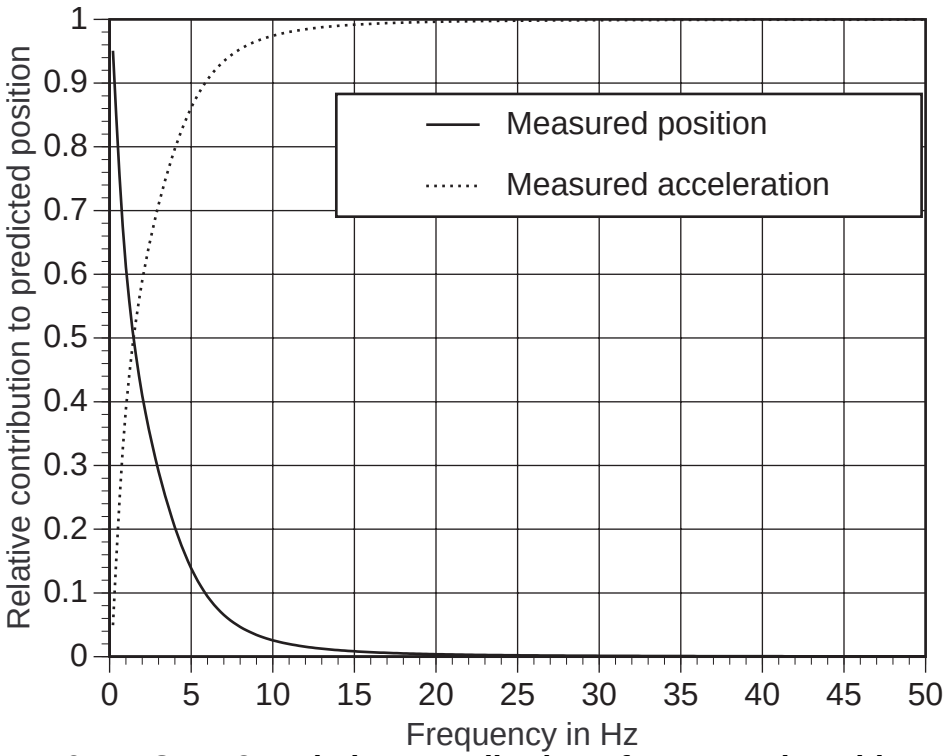


Figure 6.17: Case 3: Relative contribution of measured position and acceleration to predicted position

6.5 Exploring prediction parameter space

With the frequency-domain transfer functions, I can explore the behavior of the prediction routines as various parameters change. The main question: What happens as the prediction interval changes? Section 6.5.1 shows how the RMS errors for the predicted outputs change as the prediction interval varies, for the three cases described in the previous section. Even with the Kalman filter, it turns out that high-frequency components in the signal cannot be tolerated. Then Section 6.5.2 predicts the spectrum of the predicted signal, given the spectrum of the original signal. This shows what jitter looks like in the frequency domain and characterizes the predictor for different applications. Finally, Section 6.5.3 estimates the maximum time-domain error of a signal, given the frequency-domain spectrum of the original signal. This allows designers to determine the maximum acceptable system delay based upon a specification of maximum time-domain error.

Although this section focuses on specific analysis examples, the overall contribution is the frequency-domain framework in general, which permits the analysis of how these predictors will behave on any specified class of motion sequences and prediction intervals. Thus, by recording head motion sequences while operating a desired HMD application, other users can determine if these predictors will behave well on their applications and systems. If they do not behave well, the user can also determine how much the head motion or prediction interval must change to achieve the desired prediction performance.

6.5.1 Predicted position error versus prediction interval

How do the errors in the prediction signal change as a function of the prediction interval? The magnitude ratios for Case 1, 2 and 3 in Figures 6.10, 6.12 and 6.15 were for a single prediction interval of 100 ms. To show how the errors change as the prediction interval changes, I now plot the RMS error of all three cases for four different prediction intervals: 50 ms, 100 ms, 150 ms, and 200 ms. Recall that the RMS error, as defined in Section 6.2.6, measures the error based on both the magnitude ratio and the phase difference. Figures 6.18, 6.19 and 6.20 plot the RMS errors for Case 1, Case 2, and Case 3, respectively. Note that the basic trend is for the RMS errors to increase as the prediction interval increases.

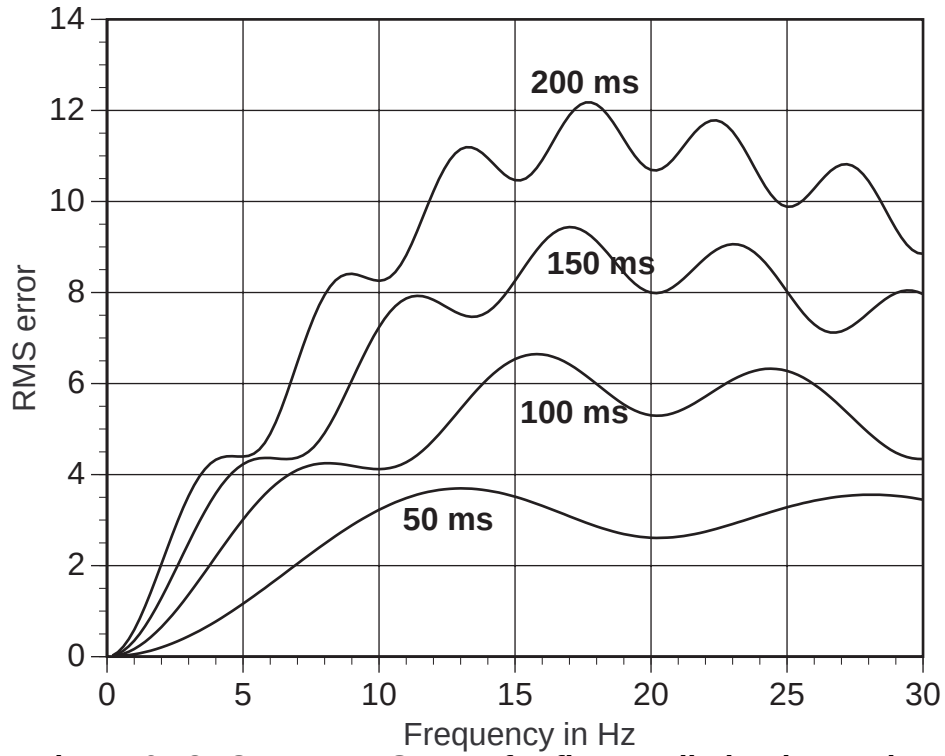


Figure 6.18: Case 1 RMS error for five prediction intervals

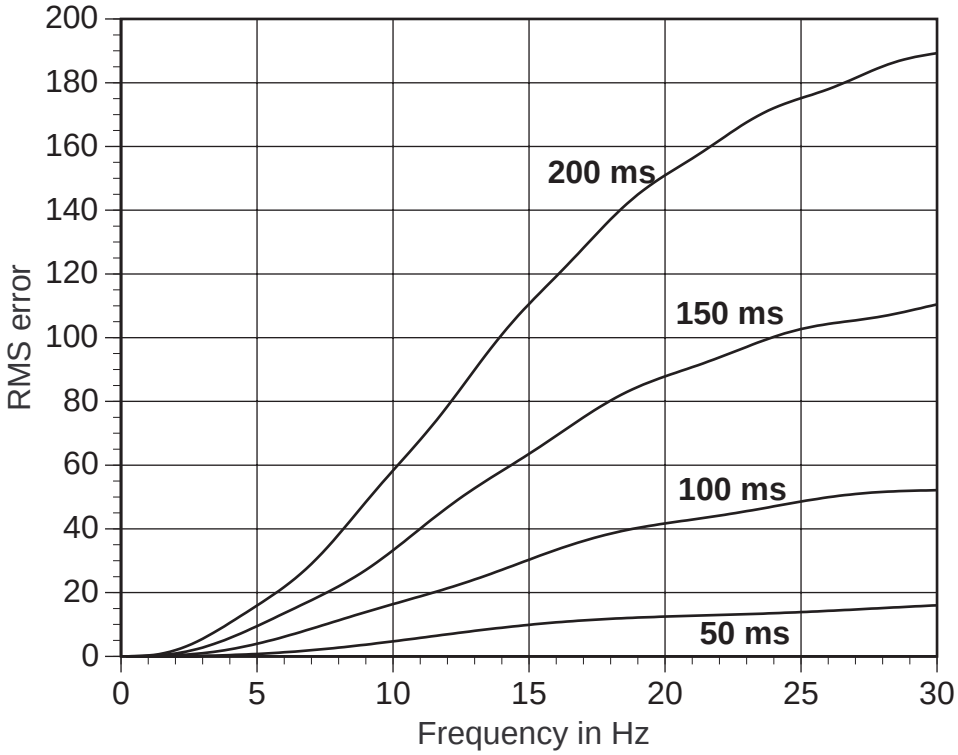


Figure 6.19: Case 2 RMS error for five prediction intervals

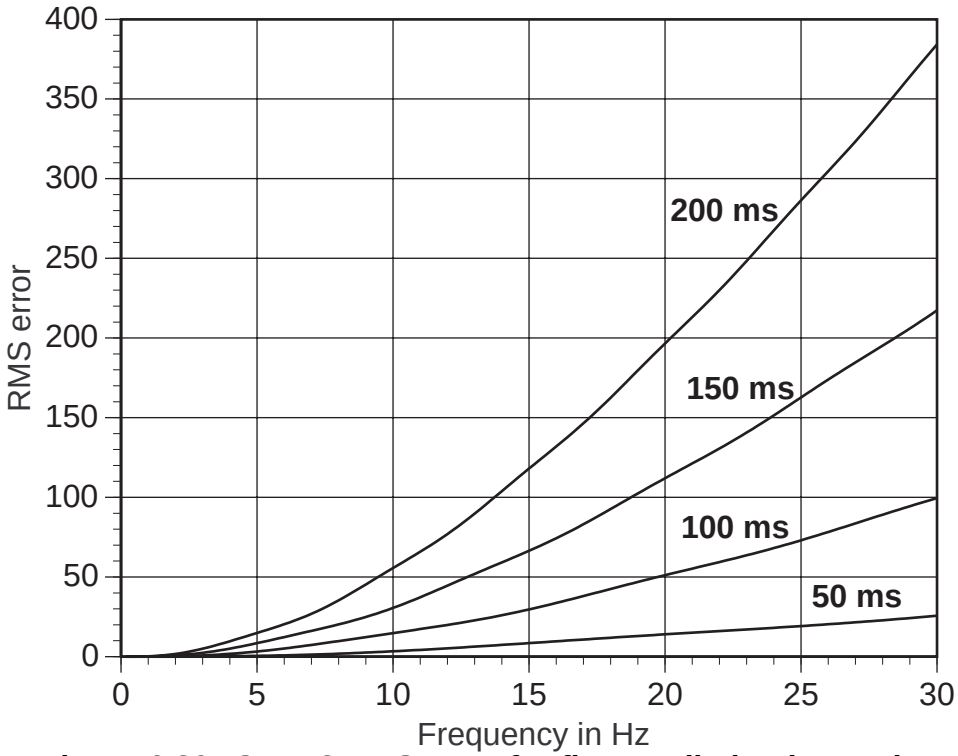


Figure 6.20: Case 3 RMS error for five prediction intervals

At first glance, these graphs appear to show that the non-inertial Case 1 predictor performs better than the inertial-based Case 2 and Case 3, even though Chapter 4 empirically demonstrated that the inertial-based prediction was more accurate. The RMS errors for Case 1 appear to be smaller than the errors for Case 2 and Case 3, which blow up much more rapidly with increasing frequency.

This apparent contradiction is resolved when I reduce the scale of the diagrams to focus on the low frequencies. Figure 6.21 shows the RMS errors of Case 1, Case 2 and Case 3 for a 50 ms prediction interval. It is true that the errors for Case 1 are lower at high frequencies. However, the inertial-based Case 2 and Case 3 produce lower errors than the non-inertial Case 1 at low frequencies, where most head-motion energy exists. In Figure 6.21, Case 2 is better for frequencies under ~7 Hz, and Case 3 is better for frequencies under ~9.5 Hz. I call these frequencies where Case 1 becomes better than the inertial-based predictors the *switchover* frequencies. If all the head-motion energy is contained below the switchover frequency, then the inertial-based predictors will be more accurate than the non-inertial-based.

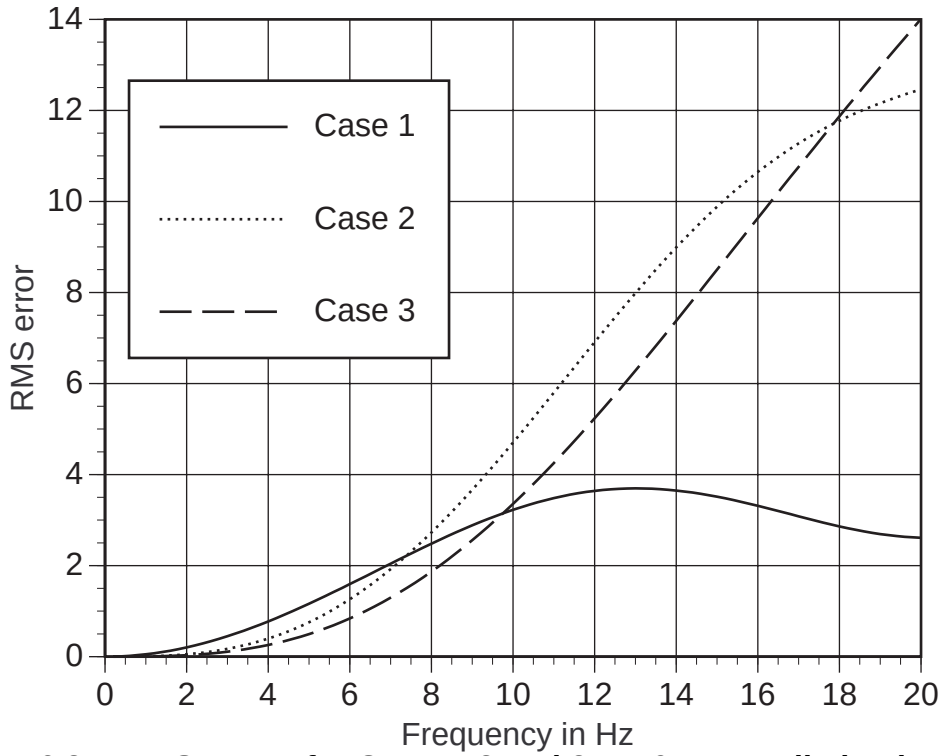


Figure 6.21: RMS errors for Case 1, 2 and 3 at 50 ms prediction interval

These switchover frequencies are reduced as the prediction interval increases. For example, Figure 6.22 plots the errors for the three cases at a prediction interval of 150 ms. Both axes are on a logarithmic scale to make the differences more apparent. Now the switchover frequency is around 2.5 Hz for both Case 2 and Case 3. This indicates that the effectiveness of the Case 2 and Case 3 predictors over Case 1 is degraded as the prediction interval increases. Increasing the prediction interval reduces the allowable signal bandwidth where using inertial-based prediction is better than just doing non-inertial prediction.

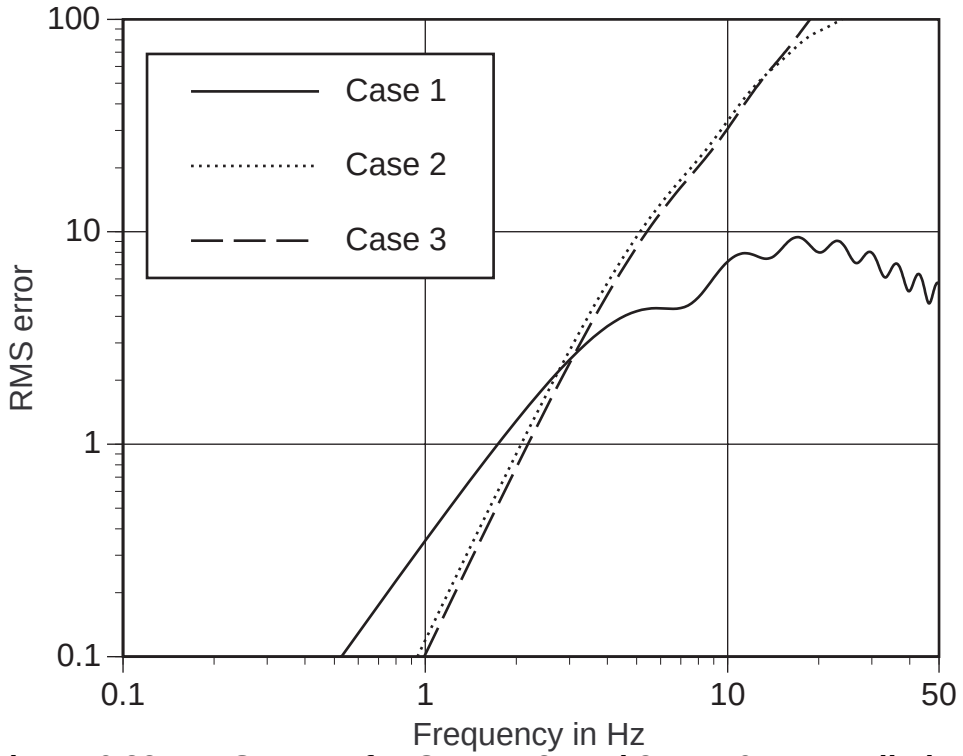


Figure 6.22: RMS errors for Case 1, 2, and 3 at 150 ms prediction interval

The reason the inertial-based predictor errors blow up more rapidly than the non-inertial based is that Case 2 and Case 3 use estimated acceleration in the predictor, while Case 1 only uses position and velocity. The inertial sensors make it possible to use estimated acceleration, since with them the filter avoids taking two numerical differentiation steps. The magnitude of acceleration grows by a factor of ω squared, while velocity only grows by a factor of ω. At high frequencies, this makes a large difference.

It is interesting that the Kalman filters, as tuned in Case 2 and Case 3, do not suppress the high-frequency components sufficiently quickly to prevent the errors from blowing up at high frequencies. The filters could be tuned to do so, but that would cause the estimated velocities and accelerations to be delayed significantly in time, and those delays would make it hard to do accurate prediction. The estimated velocities and accelerations must be close to the true ones if any derivative-based predictor is to perform well. Consequently, practical Kalman filter and predictor combinations will be vulnerable to signals with energy at high frequencies. If such signals are unavoidable, then it is best not to use estimated acceleration in the predictor.

Instead, use the inertial sensors to estimate only velocity and base the prediction on position and velocity. Figure 6.23 compares Case 1 against a Modified Case 2 at a 100 ms prediction interval, where Case 2 is modified to not use acceleration when computing the predicted output. This gives slightly more accurate results than the non-inertial Case 1 for frequencies under ~7 Hz, without magnifying high frequencies nearly as much as the normal Case 2 or Case 3 predictors do.

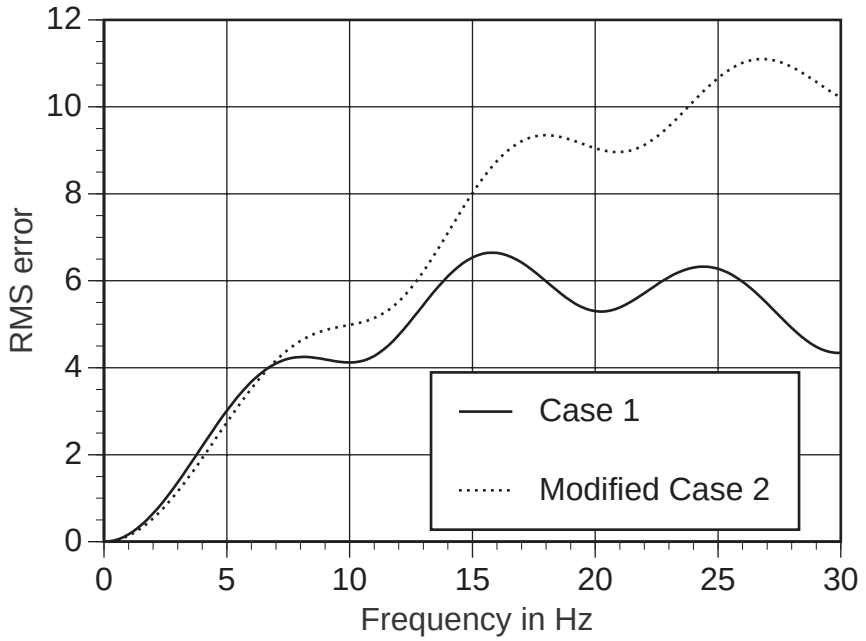


Figure 6.23: RMS errors for Case 1 and Modified Case 2 at 100 ms prediction interval

Another trend revealed by the figures is that Case 3 appears to be more accurate than Case 2, because Case 3 has better estimates of acceleration. The prediction accuracy depends on how accurate the filter estimates of position, velocity, and acceleration are. Case 3 does a better job of estimating the accelerations than Case 2 does. In Case 3, the Kalman filter estimates velocity from measured positions and accelerations. This is relatively easy, so the filter can perform these estimates accurately. However, in Case 2, acceleration must be estimated from measured positions and velocities. This requires numerically differentiating the velocity, yielding estimated accelerations that are either noisier or further delayed in time than the estimates in Case 3.

In the system I built, the position filters are aided by measured acceleration, but the orientation filters only receive measured angular velocity information, not angular acceleration. What these results suggest is that, theoretically, the orientation prediction could be more accurate with the use of angular accelerometers than with angular rate gyroscopes. In practice, this would depend on the accuracy of angular accelerometers and how easily they can be calibrated with the AR system.

6.5.2 Estimating spectra of predicted motion sequences

The prediction transfer functions can generate an estimate of the spectrum of the predicted signal, given the spectrum of the original signal. Multiply the input spectrum by the prediction magnitude ratio computed in Section 6.3.1 for the 2nd-order polynomial predictor or by the magnitude ratio of $\mathbf{O}(z)$ for the Kalman-filter-based predictor. Since different applications generate different input motion spectra, and a particular spectrum can represent an entire class of inputs, this technique can specify how a particular predictor will perform with a different application.

Figure 6.24 shows a specific example for the Case 1 predictor. It shows the spectrum of the user's *X* translation, so it is called the "Tx" sequence. This data comes from the Demo2 motion sequence mentioned in Chapter 4. The prediction interval was set to 100 ms. Figure 6.24 also shows the spectrum of the predicted signal. This spectrum was computed in two ways. First, it was estimated by the frequency-domain computations described in the previous paragraph. Second, I ran the Case 1 predictor in simulation, reading the original time-domain signal and generating the predicted time-domain signal. Then I used spectral-analysis techniques to estimate the spectrum of the predicted signal from the time-domain predicted signal. The two estimates of the spectrum of the predicted signal are virtually identical, as shown in Figure 6.24. This does not mean that either estimate is correct (see Section 6.6.3), but it does show that they corroborate each other.

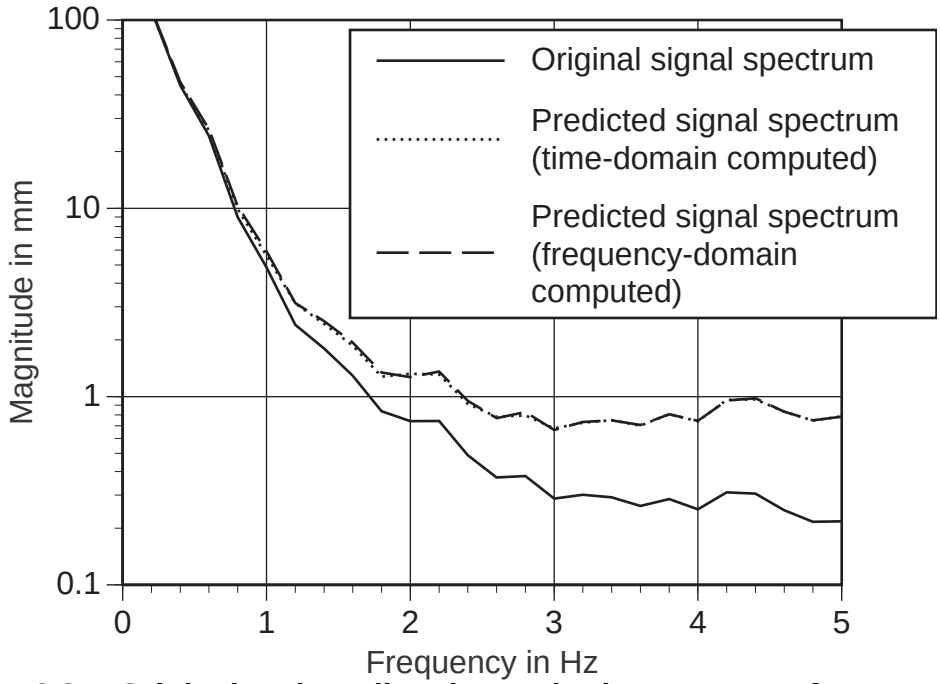


Figure 6.24: Original and predicted magnitude spectrums for Demo2 Tx sequence for 100 ms prediction interval

The spectrum of the predicted signal shows what jitter looks like in the frequency domain. Figure 6.25 uses the same data as Figure 6.24, except that it plots the predicted spectra for three different prediction intervals. As the prediction interval increases, the magnification of the predicted signal spectrum increases, especially at higher frequencies. These "humps" in the predicted spectra represent jitter. The magnified outputs at frequencies higher than typical head motion result in annoying "wiggles" in the time-domain that do not seem to correspond to the user's actual motion, as seen in Figure 4.37. This is a way to view the same effect in the frequency domain.

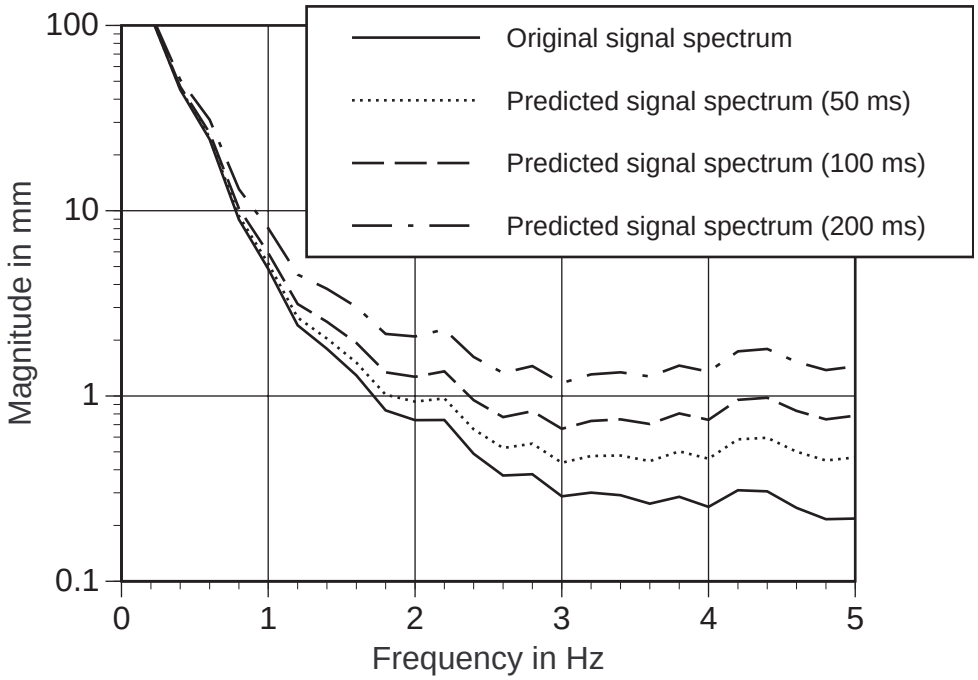


Figure 6.25: Predicted magnitude spectrums for Demo2 Tx sequence at three prediction intervals

6.5.3 Estimating the maximum time-domain error

The previous section showed what prediction error looks like in the frequency domain. However, it would also be useful to derive a theoretical expression for the error in the time domain. In particular, it would be useful to derive a bound on what the largest time-domain error could possibly be in the predicted signal, given the predictor, the motion sequence, and the prediction interval. For a *specific* motion sequence, this can be determined by running the filter in simulation, generating the predicted output, and searching for the maximum error. So instead, assume that the only information available about the motion sequence is the power spectrum. This is useful because spectra can represent entire classes of motion, rather than one specific sequence. No phase information is available. What is the maximum time-domain error?

To determine this, I need to transform the spectrum of the original signal to the spectrum of the error signal. This is done with the magnitude ratio of the error transfer function from Section 6.3.3 (for the 2nd-order polynomial predictor) or $\mathbf{U}(z)$ (for the Kalman-filter-based predictor). Multiplying the error magnitude ratio by the spectrum of the original signal

yields an estimate of the spectrum of the error signal. Now the problem reduces to finding the maximum time-domain value of the Fourier-domain error signal, where the magnitudes of that signal are known but the phases are not.

Clearly, one upper bound is to add the absolute value of all the magnitudes. Each magnitude is a coefficient for a sinusoid that ranges between 1 and —1. The values that each component sinusoid can generate therefore range between $\pm M$, where M is the magnitude of the complex coefficient. The time-domain signal is generated by summing the contributions from all the component sinusoids. The largest value that the sum of sinusoids can achieve is the sum of the maximum of each sinusoid, or the sum of the absolute value of all the magnitudes.

Unfortunately, that upper bound is also the smallest upper bound achievable, given the few restrictions that I can impose upon the error signal. The error signal is real, and that is the only restriction of any use. A real time-domain signal has a Fourier representation with even magnitudes and odd phases. The complex coefficients in the frequency domain occur in frequency pairs, at angular frequencies ω and $-\omega$. The constraints of even magnitudes and odd phases mean that the coefficients for the pair take the following form:

Coefficient at frequency $\omega \rightarrow$ Magnitude M, phase ϕ
Coefficient at frequency $-\omega \rightarrow$ Magnitude M, phase $-\phi$

It turns out that the two components added together yield the following sinusoid:

$$2M\cos(\omega t + \phi)$$

Now magnitude M can either be positive or negative, but I am free to choose the phase, as long as it is odd. If the magnitude is positive, then set phase to zero. This forces the magnitude to be positive M at time $t = 0$. What if magnitude M is negative? Then set one phase to π and the other to $-\pi$. This satisfies the oddness constraint. At time $t = 0$, $\cos(\pi) = \cos(-\pi) = -1$, which reverses the sign of the magnitude and forces it to be positive. Thus, I can pick phases such that the worst-case upper bound of summing all the magnitudes is actually achieved.

By using this procedure, a system designer could specify the maximum allowable time-domain error, and then determine the acceptable system delay that keeps errors below the specification. The power spectrum recorded from the desired application would be analyzed by the previous procedure, which produces an estimated maximum time-domain error. Unfortunately, this is only an estimate rather than a hard upper bound because of uncertainties in the power spectrum estimate and because the measured spectrum is an average for the entire signal and may not represent what happens at a particular subsection of the signal. Section 6.6.3 discusses these issues.

How closely do the estimated maximum bounds match the actual peak errors observed in a specific time-domain signal? Table 6.5 compares the estimated maximum against the actual observed peaks for the six signals in the Demo2 motion sequence. The Demo2 sequence, introduced in Chapter 4, was recorded from a first-time HMD user running a demonstration HMD application. The prediction interval was 100 ms. The actual observed peaks are usually within a factor of two of the estimated maximum bound, and they are often much closer. However, for the Ty signal, the actually-observed peak error was slightly larger than the estimated maximum! This demonstrates that the theoretical maximum is only an estimate, rather than a true upper bound.

Motion sequence name	Estimated maximum error	Actual peak error
Tx	137.2 mm	100.1 mm
Ty	153.7 mm	155.6 mm
Tz	69.2 mm	54.2 mm
Yaw	6.9 degrees	2.6 degrees
Pitch	8.9 degrees	5.2 degrees
Roll	13.1 degrees	11.7 degrees

Table 6.5: Estimated versus actual time-domain maxima for Demo1 sequence

6.6 Implementation details

To verify the theoretical results listed in this chapter, I implemented the frequency-domain equations and compared their results against those generated by their time-domain equivalents. By checking that the two match, I made certain that the frequency-domain analysis is based on correct equations. However, converting signals into the frequency domain and operating on them there requires paying attention to some details that I did not cover in my introduction in Section 6.2. This section discusses these details.

Figure 6.26 shows how the comparison of the time-domain and frequency-domain results is done. This figure shows the verification of **O**(*z*), but similar approaches are taken for the other transfer matrices.

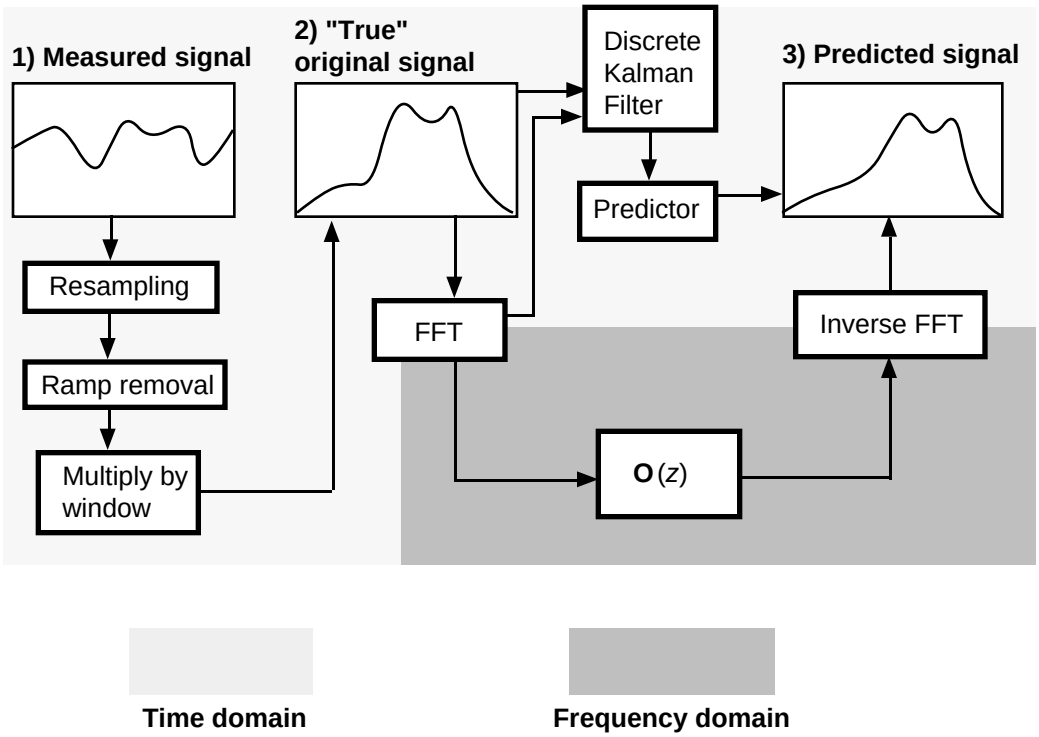


Figure 6.26: Verification of frequency-domain equations

I start with the graph labeled 1) Measured signal. This can either be simulated motion or a graph extracted from a motion sequence recorded from a user wearing the see-through HMD and walking around. Section 6.6.1 describes how this signal is sent through a lowpass filter, a ramp remover, and then multiplied by a spectral window. This results in graph #2: the "true"

original signal, which is suitable for conversion into the frequency-domain by an FFT. At this point, I compute graph #3, the predicted signal, in two ways. First, I convert the signal into the frequency domain via an FFT. Then I use the transfer matrix to change the signal in the frequency domain. The resulting signal is converted back into the time domain by an Inverse FFT. The second method is to do it in the time domain by running the signal through the combination of the Discrete Kalman Filter and the predictor. The complex coefficients for the sinusoidal basis functions computed by the FFT are used to determine the position, velocity, and acceleration of the time-domain signal, which are sent into the Kalman filter. That is why an arrow goes from the FFT to the Discrete Kalman Filter in Figure 6.26. Section 6.6.2 points out some details about the setup of the Discrete Kalman Filter. If the frequency-domain transfer matrix is correct, then the two computations of graph #3 should match. If they do not match, then the theoretical equations are not correct. The equations I tested resulted in virtually perfect matches except for the first few seconds, because the DKF does not run in steady-state mode, as Section 6.6.2 explains.

Some of the results in Section 6.5 require estimating the power spectrum of a signal. Section 6.6.3 discusses how this is done through spectral-analysis techniques and why it is an inherently imperfect operation.

6.6.1 Generating the "true" original signal

The measured signal requires processing before it is suitable for conversion into the frequency domain because the signal must be evenly sampled and because the Fourier Transform assumes that its input signal is periodic. That is, the endpoints of the signal should match, as well as the endpoints of all of its derivatives. Since the measured signal need not satisfy these conditions, it requires modification so that these conditions hold.

First, I resample the measured signal at a rate faster than the original, uneven sampling rate, using linear interpolation as required. Then I apply a noncausal lowpass filter that does not add any phase shift to the signal [NASAksc]. The lowpass filter removes artifacts generated by the linear interpolation that result in false high-frequency components in the frequency

domain. The filter also removes high frequencies in the original signal, which is important if I test the predictor without the Kalman filter.

Next, I fit a line to the curve. This line, or linear ramp, is then subtracted from the signal. The idea is to avoid large constant offsets that skew the magnitude of the signal at very low frequencies. Large constant signals are not interesting for the prediction problem because constant components are easily predicted.

Finally, I multiply the entire signal by a function that forces the initial and final values to match. It does this by forcing both endpoints to go smoothly to zero. This operation is called *windowing*. An example of a window function is shown in Figure 6.27, for a dataset that has 100 entries. Many different types of window functions exist; see [Harris78] for a thorough discussion. Windowing distorts the edges of the signal but keeps the middle part mostly intact. The Fourier Transform assumes that the input signal is periodic, so if the endpoints do not match smoothly, the Fourier Transform will place more energy into the high-frequency components in an attempt to make the endpoints match, even though these high-frequency components are not an accurate reflection of the original signal. Windowing greatly reduces such artifacts, which are called *leakage*, and they receive more discussion in Section 6.6.3.

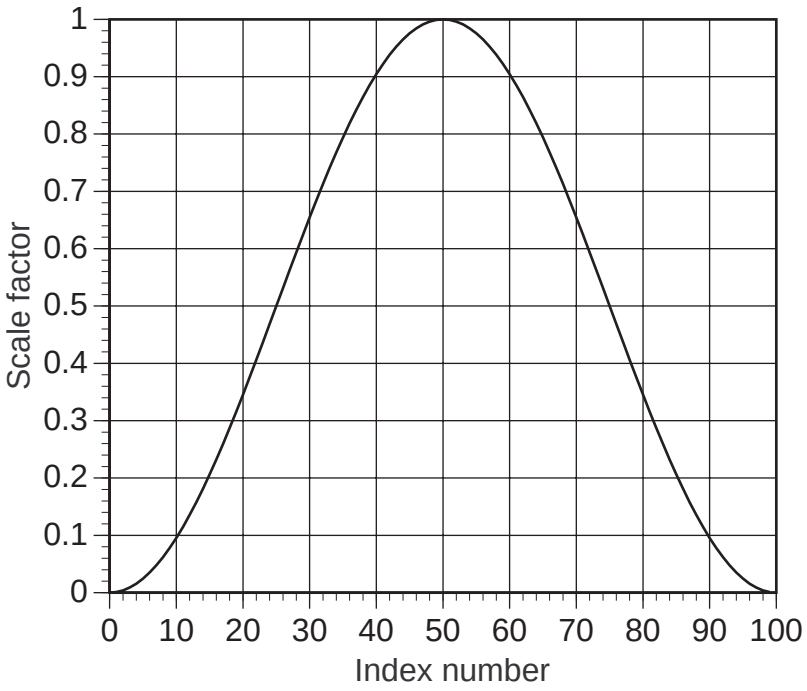


Figure 6.27: A sample window function

The resulting signal is suitable for conversion into the frequency domain and is considered the "true" original signal for comparison purposes. Note that it is now stationary overall, because I have forced it to be periodic, although different parts of the signal may exhibit differing properties. Also, the resulting signal may not look much like the measured signal, but it retains the important spectral characteristics of the measured in the middle section, and it can be converted into the frequency domain through a Fourier Transform without large amounts of artifacts.

6.6.2 Discrete Kalman Filter

The Discrete Kalman Filter has two details the should be pointed out. First, the time-domain filter does not run in steady-state mode. And second, the **A** and **E** matrices for the Discrete Kalman Filter are not the same as those for the Continuous-Discrete Kalman Filter.

The frequency-domain analysis assumes that the Discrete Kalman Filter operates in steady-state mode, when matrices **P** and **K** have reached their constant values. However, when I compute graph #3 in the time-domain, I do *not* run the Discrete Kalman Filter in steady-state mode,

because that introduces permanent delays that skew the output signal. The result is that the time-domain-computed graph #3 and the frequency-domain-computed graph #3 do not match perfectly during the first second or two of data, until the time-domain Kalman filter reaches steady-state. After that point, the match is exact.

The Discrete Kalman Filter has different **A** and **E** matrices from the Continuous-Discrete Kalman Filter I normally use in the actual real-time system. Careful examination of the model definitions in both filters shows that these matrices must be different. Since I want the Discrete Kalman Filter to match the performance of the Continuous-Discrete Kalman Filter, I must convert the matrices used in the continuous-discrete case (call these **A** and **E**) into equivalent matrices for the Discrete Kalman Filter (call these $\mathbf{A_d}$ and $\mathbf{E_d}$). The conversion formulas are listed on p. 83 of [Lewis86]:

$$\mathbf{A_d} = \mathbf{I} + \mathbf{A}T + \frac{(\mathbf{A}T)^2}{2!} + \ldots$$

$$\mathbf{E_d} = \mathbf{G}\mathbf{E}\mathbf{G}^{\mathrm{T}}T + \frac{\left(\mathbf{A}\mathbf{G}\mathbf{E}\mathbf{G}^{\mathrm{T}} + \mathbf{G}\mathbf{E}\mathbf{G}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\right)T^2}{2!} + \ldots$$

The *T* in these formulas is the period, in seconds, separating the evenly-spaced measurements read by the Discrete Kalman Filter. In practice, I ignore all terms with components above *T* squared because I run the filters with small values of *T* (e.g., 5 ms).

Note that **H** and **R** do not change, since the measurement step is discrete in both types of Kalman filters.

I verified the equivalence of the Discrete Kalman Filter and the Continuous-Discrete Kalman Filter by implementing both, using the above formulas, running them on the same input and confirming that the outputs of both filters match.

6.6.3 Spectral Analysis

Recovering the spectral properties from measured data is an estimation task that almost always returns answers that have some error. It is rare that spectral analysis will find the exact coefficients of the sinusoidal

functions that generated the time-domain signal. Parts of Section 6.5 depend upon estimating the *power spectrum* of an observed signal, which is a graph of the squared magnitudes at every frequency. This section briefly describes how the power spectrum is computed and why the estimate of the power spectrum has error. For details, please read [Press88] and [Harris78].

One way to compute a power spectrum is to run an FFT on a signal, then square the computed magnitudes at each frequency. However, the magnitudes estimated by this method are not very reliable. To understand why, first note that all computations occur in discrete space. Therefore, a discrete frequency of 10 Hz does not actually represent 10 Hz by itself. Instead, it represents a "bin" of frequencies that extends halfway to the previous discrete frequency and halfway to the next one (e.g., 9.8 Hz to 10.2 Hz). Ideally, the power at this discrete frequency should represent an average of the power of all the frequencies in this bin. Unfortunately, this is not the case. Instead, the result is a weighted average of a sinc function centered at that frequency. Because the sinc function does not fall off very quickly as the distance from the origin increases, values at frequencies far away from the bin contribute to the power estimate at that bin. This unwanted contribution is called *leakage*. Therefore, the estimate of the power at every frequency is unreliable, making the resulting power spectrum incorrect.

Two steps can be taken to improve the power spectrum estimates. First, rather than running the FFT once on the entire signal, run it several times on small sections of the signal, called chunks. These chunks should overlap each other halfway for optimum results [Childers78]. Second, multiplying each chunk by a spectral window before running the FFT reduces leakage. Averaging the results from all the chunks reduces the overall variance. Using both steps produces a smooth estimate of the power spectrum that represents an average of the entire signal, but it may not be representative of any one particular section of that signal.

Even with the use of a window and averaging, the power spectrum estimates will almost always have some error. Take the following example. Table 6.6 lists several sinusoids that are added together to form a time-domain signal. Since I know the original sinusoids and their magnitudes, I can graph those in Figure 6.28. Then Figure 6.29 shows the estimate of

those magnitudes generated from the time-domain version of the sinusoids listed in Table 6.6. The estimates were generated by a program that computed the power spectrum, using both windowing and averaging to reduce the variance. The square roots of the estimated power spectra are the estimated magnitudes. Note that the outputs are no longer spikes, but rather broader "hills." This is caused by leakage of energy from the true sinusoid functions into neighboring frequencies. The leakage means that the two separate peaks at 0.5 Hz and 0.6 Hz are no longer visible; they are instead merged into one.

Functions are of the form: $M \sin(2\pi f + \phi)$:

	Magnitude	Frequency (Hz)	Phase (radians)
Sine #1	10.0	0.5	0.2
Sine #2	5.0	0.6	0.2
Sine #3	3.0	2.0	0.9
Sine #4	2.0	5.05	—0.9
Sine #5	1.0	9.0	—0.2

Table 6.6: Original set of sinusoids

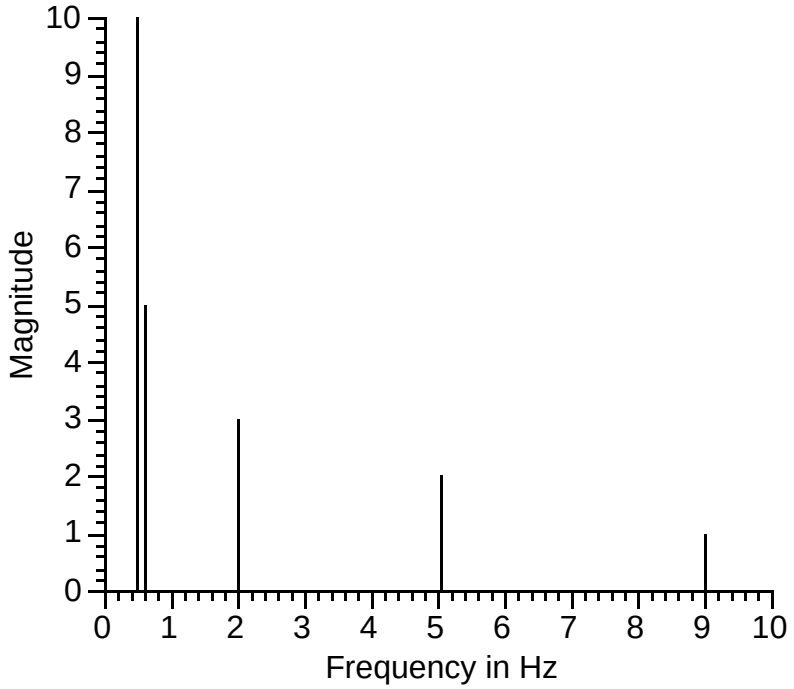


Figure 6.28: The true magnitudes of the sinusoids

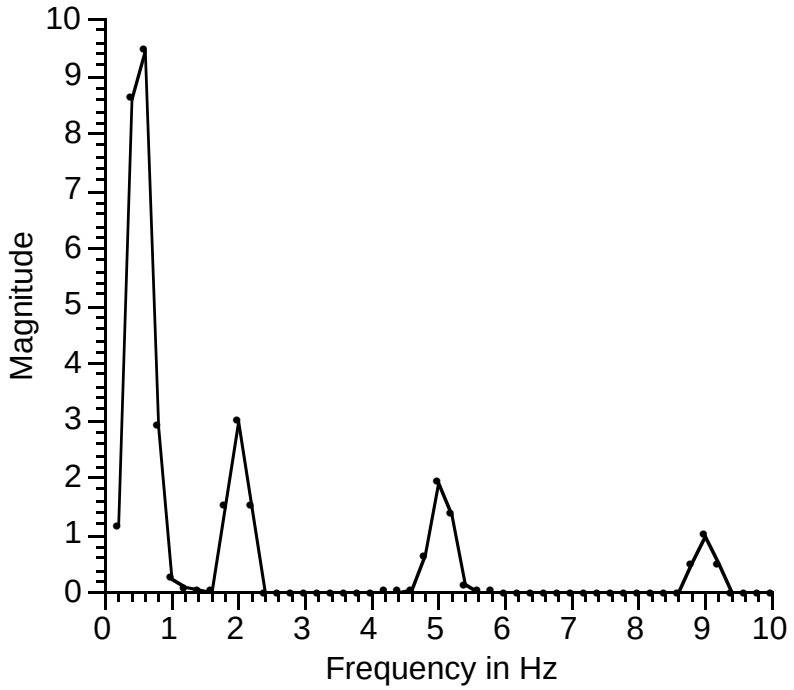


Figure 6.29: The estimated magnitudes of the sinusoids

Given finite amounts of measured data, there is always a tradeoff between the *variance* and the *bandwidth* of the power spectrum estimates, as expressed by Grenader's Uncertainty Principle [Priestley81]. This result is similar in tone to Heisenberg's Uncertainty Principle from quantum physics. The variance is the reliability of the estimates: smaller variance indicates more accurate estimates. The bandwidth specifies the resolution or the spacing between the discrete frequencies. One can get accurate power spectrum estimates with large frequency bins, or inaccurate estimates with small frequency bins, but not accurate estimates and small frequency bins simultaneously. The only way to improve both is to have longer data sequences, but this is often not practical. Because the head-motion data changes spectral characteristics with time, using longer data sequences may not be a wise choice. Therefore, in practice, power spectrum estimates always have some uncertainty and error.

For more examples of how the estimated power spectrum can differ from the theoretical power spectrum, please see [Preistley81] and [Jenkins68].

7. Future work

Section 1.5 summarized the contribution of this dissertation, which I briefly recount here. The main result is a demonstration of a prediction technique that reduces dynamic errors by a factor of five to ten over not doing any prediction at all and by a factor of two to three over doing prediction without inertial sensors. This demonstration is supported by static registration that is usually within ±5 mm. A frequency-domain analysis of the predictor specifies the conditions under which prediction is effective. Overall, users now perceive the virtual and real objects to be "sticking closely together" instead of "swimming around each other."

While this dissertation demonstrates improved registration between virtual and real objects, many applications demand even closer registration than what this work achieves. Still, the results shown suggest that the ideal of "no swimming" may be reached in the future. This section outlines some areas that require improvement and suggests future directions worth pursuing to achieve this goal.

Much work remains to further improve static registration. Although I have demonstrated static registration from a wide variety of view positions and directions, this registration is subject to several limitations:

1) It matches only one virtual object with one real object, where the real object is the calibration rig itself. While it should not be difficult to register multiple objects if the locations of the real objects are precisely known and the head tracker is accurate, this has not been demonstrated.
2) The optoelectronic tracker has several limitations. Because the tracker loses accuracy when few of the optical sensors on the HMD are aimed at the ceiling beacons, the HMD cannot move far away from the wooden frame, nor can the HMD tilt far from horizontal.

This limits the range of supportable viewpoints. Also, the optoelectronic tracker has subtle distortions that limit the registration accuracy achievable in this system.

3) The system is monocular. Registration is demonstrated for only one eye in the HMD. While the static registration procedure could be applied to both eyes, stereo displays involve additional issues like convergence that this work does not address.
4) No compensation is done for the optical distortion in the see-through HMD. Because the HMD has narrow field-of-view displays, this distortion is small, becoming a significant problem only toward the edges of the displays. Eliminating this error requires mapping the distortion, then predistorting the graphic images before displaying them [Robinett92c]. Doing this quickly requires specialized hardware or several fast processors.
5) The variance in the measured viewing parameters is too large. This means the user may have to repeat the registration task several times to get satisfactory registration. As see-through HMDs become lighter, performing the registration tasks should become easier, reducing the variance. Also, more research must be done in developing different methods for measuring viewing parameters that generate less variance.

More sophisticated prediction methods might further reduce dynamic registration errors. Adaptive methods that adjust to varying head motion deserve more exploration. Using a nonadaptive predictor is like trying to race a car at constant speed; slowing down on the curves and speeding up on the straight-aways will improve your time. This adaptation might take the form of using non-white noise models, varying the model and measurement covariance matrices, or running several different models in parallel and selecting between them. Analyzing head motion for recognizable patterns or high-level characteristics may aid prediction. If one can assume a specific task, then more accurate prediction may be possible because a more specific model can be built. Other researchers have begun looking for such patterns [Shaw92]. For example, Fitts' Law has been shown to apply to head motion [Andres89] [Jagacinski85]. The drawback of building a specific, tuned model

of head motion is that the predictor is likely to be less accurate when the actual head motion does not match the model's assumptions.

This dissertation has not dealt with video see-through HMDs, where a video camera provides a view of the real world and the graphics are combined with the digitized images of the real world. With this class of see-through HMD, standard camera calibration techniques could determine the viewing parameters. And since the computer has digitized images of what the user sees, it may be possible to use image processing or computer vision techniques to detect features in these images and use them to aid registration. Such feature detection must run in real time and is often difficult to make robust, but it may be possible in some applications. Another limitation of the technology is that the video camera and digitization hardware impose inherent delays on the user's view of the real world. Therefore, even if the graphics are perfectly matched with the digitized images, a problem remains: the latency in the video stream will cause the user to perceive *both* the real and virtual objects to be delayed in time. While this may not be bothersome for small delays, it is a major problem in the related area of telepresence systems and may not be easy to overcome.

Augmented Reality is an area ripe for psychophysical studies. How much lag can a user detect? How much registration error is detectable, when the head is moving? Besides questions on perception, psychological experiments that explore *performance* issues are also needed. How much does head-motion prediction improve user performance on a specific task? How much registration error is tolerable for a specific application before performance on that task degrades substantially? Is the allowable error larger while the user moves her head versus when she stands still?

Such studies could help guide the development of future head motion prediction routines. For example, assume some studies determined that at certain times during head motion, users are not sensitive to registration errors. Evidence exists that suggests this may be the case. For example, parts of the human visual system "shut down" during large saccades [Chekaluk90]. If errors during such motions do not matter, that makes the prediction task somewhat easier and may lead to different prediction strategies.

The biggest single obstacle to building effective Augmented Reality systems is the requirement of accurate, long-range sensors and trackers that report the locations of the user and the surrounding objects in the environment. Right now, scene generators exist that can generate images in real time with sufficient fidelity to be useful in many Augmented Reality applications. Because the virtual objects supplement, rather than replace, the real world, the graphic images do not have to be as realistic as those demanded by Virtual Environment applications. Right now, high-resolution see-through HMDs exist. However, existing sensor and tracker technologies are inadequate. Commercial trackers are aimed at the needs of Virtual Environments and motion-capture applications. *Motion capture* is the tracking of an actor's body parts to control a computer-animated character or for the analysis of an actor's movements. Compared to those two applications, Augmented Reality has much stricter accuracy requirements and demands larger working volumes. More work needs to be done to develop sensors and trackers that can meet these stringent requirements.

The optical see-through HMD shown in Figures 3.8 and 3.9 weighs over eight pounds. Future see-through HMDs must be much lighter and less bulky. Ideally, they should be similar to a pair of glasses that does not change its position with respect to the wearer's eyes even under rapid head motion.

REFERENCES

Albrecht89 — Albrecht, R. E. An Adaptive Digital filter to Predict Pilot Head Look Direction for Helmet-Mounted Displays. M.S. Thesis, Electrical Engineering, University of Dayton, Ohio (July 1989).

Andres89 — Andres, Robert O., and Kenny J. Hartung. Prediction of Head Movement Time Using Fitts' Law. *Human Factors 31*,6 (1989), 703-713.

Ascen89 — Ascension Technology Corporation. The Bird 6D Input Device (Burlington, Vermont, 1989).

Ascen91 — Ascension Technology Corporation. A Flock of Birds Product Description Sheet (Burlington, Vermont, April 1991).

Axt87 — Axt, Walter E. Evaluation of a Pilot's Line-of-Sight Using Ultrasonic Measurements and a Helmet Mounted Display. *Proceedings IEEE National Aerospace and Electronics Conference* (Dayton, OH, 18-22 May 1987), 921-927.

Azuma91 — Azuma, Ronald, and Mark Ward. Space-Resection by Collinearity: Mathematics Behind the Optical Ceiling Head-Tracker. UNC Chapel Hill Department of Computer Science technical report TR 91-048 (November 1991).

Azuma93 — Azuma, Ronald. Tracking Requirements for Augmented Reality. *Communications of the ACM 36*, 7 (July 1993), 50-51.

Azuma94 — Azuma, Ronald, and Gary Bishop. Improving Static and Dynamic Registration in a See-Through HMD. *Proceedings of SIGGRAPH '94* (Orlando, FL, 24-29 July 1994). In *Computer Graphics*, Annual Conference Series, 1994, 197-204.

Bajura92 — Bajura, Mike, Henry Fuchs, and Ryutarou Ohbuchi. Merging Virtual Reality with the Real World: Seeing Ultrasound Imagery within the Patient. *Proceedings of SIGGRAPH '92* (Chicago, IL, 26-31 July 1992). In *Computer Graphics 26*, 2 (July 1992), 203-210.

Beer88 — Beer, Ferdinand P. and E. Russell Johnston, Jr. Vector Mechanics for Engineers: Statics and Dynamics (5th edition). Mc-Graw Hill (1988).

Bejczy92 — Bejczy, Antal K., Won S. Kim, and Steven C. Venema. The Phantom Robot: Predictive Displays for Teleoperation with Time Delay. *NASA Tech Brief 16*, 7, item #104. From JPL new technology report NPO-18277/7794. (July 1992).

Berg83 — Berg, Russell F. Estimation and Prediction for Maneuvering Target Trajectories. *IEEE Transactions on Automatic Control AC-28*, 3 (March 1983), 294-304.

Blom84 — Blom, H. A. P. An Efficient Filter for Abruptly Changing Systems. *Proceedings of 23rd Conference on Decision and Control* (Las Vegas, NV, December 1984), 656-658.

Bolger87 — Bolger, Philip L. Tracking a Maneuvering Target Using Input Estimation. *IEEE Transactions on Aerospace and Electronic Systems AES-23*, 3 (May 1987), 298-310.

Brooks88 — Brooks Jr., Frederick P. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *Proceedings of SIGCHI '88* (Washington D.C., 15-19 May 1988), 1-11.

Brown92 — Brown, Robert Grover, and Patrick Y.C. Hwang. Introduction to Random Signal and Applied Kalman Filtering (2nd edition). John Wiley & Sons. (1992) ISBN 0-471-52573-1.

CAE86 — Flight Simulator Wide Field-of-View Helmet-Mounted Infinity Display System. Technical report AFHRL-85-59, Williams AFB, AZ: Operations Training Division Air Force Human Resources Laboratory (May 1986), 48-64.

Cardullo90 — Cardullo, Frank M., and Yorke J. Brown. Visual System Lags: The Problem, The Cause, The Cure. *Proceedings of IMAGE V Conference* (Phoenix, Arizona, 19-22 June 1990), 31-42.

Caudell92 — Caudell, Thomas P. and David W. Mizell. Augmented Reality: An Application of Heads-Up Display Technology to Manual Manufacturing Processes. *Proceedings of Hawaii International Conference on System Sciences* (January 1992), 659-669.

Chang84 — Chang, Chaw-Bing, and John Tabaczynski. Application of State Estimation to Target Tracking. *IEEE Transactions on Automatic Control AC-29*, 2 (February 1984), 98-109.

Chekaluk90 Chekaluk, Eugene. Visual Stimulus Input, Saccadic Suppression, and Detection of Information from the Postsaccade Scene. *Perception and Psychophysics 48*, 2 (August 1990), 135.

Childers78 Childers, Donald G. Modern Spectrum Analysis. IEEE Press (1978).

Chou92 Chou, Jack C. K. Quaternion Kinematic and Dynamic Differential Equations. *IEEE Transactions on Robotics and Automation 8*, 1 (February 1992), 53-64.

Cohen94 Cohen, Jonathan, and Marc Olano. Low Latency Rendering on Pixel-Planes 5. UNC Chapel Hill Department of Computer Science technical report TR94-028 (1994).

Cook88 Cook, Anthony. The Helmet-Mounted Visual System in Flight Simulation. *Proceedings Flight Simulation: Recent Developments in Technology and Use* (Royal Aeronautical Society, London, England, 12-13 April 1988), 214-232.

Crane84 Crane, D. Francis. The Effects of Time Delay in Man-Machine Control Systems: Implications for Design of Flight Simulator Visual-Display-Delay Compensation. *Proceedings of the IMAGE III conference* (Phoenix, AZ, 30 May - 1 June 1984), 331-343.

Cruz-Neira93 Cruz-Neira, Carolina, Daniel Sandin, and Thomas DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *Proceedings of SIGGRAPH '93* (Anaheim, CA, 1-6 August 1993). In *Computer Graphics*, Annual Conference Series, 1993, 135-142.

Deering92 Deering, Michael. High Resolution Virtual Reality. *Proceedings of SIGGRAPH '92* (Chicago, IL, 26-31 July 1992). In *Computer Graphics 26*, 2 (July 1992), 195-202.

Doenges85 Doenges, Peter K. Overview of Computer Image Generation in Visual Simulation. *SIGGRAPH '85 Course Notes #14 on High Performance Image Generation Systems* (San Francisco, CA, 22 July 1985).

Drascic93 Drascic, D., J.J. Grodski, P. Milgram, K. Ruffo, P. Wong, and S. Zhai. ARGOS: A Display System for Augmenting Reality. *Video Proceedings of INTERCHI '93: Human Factors in Computing Systems* (Amsterdam, the Netherlands, 24-29 April 1993).

Feiner93 Feiner, Steven, Blair MacIntyre, and Dorée Seligmann. Knowledge-based Augmented Reality. *Communications of the ACM 36*, 7 (July 1993), 52-62.

Ferrin91 Ferrin, Frank J. Survey of Helmet Tracking Technologies. *SPIE Vol. 1456 Large-Screen Projection, Avionic, and Helmet-Mounted Displays* (1991), 86-94.

Fitts54 Fitts, Paul M. The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. *Journal of Experimental Psychology 47*, 6 (June 1954), 381-391.

Fleming89 Fleming, Wendell H., chairman. Report of the Panel on Future Directions in Control Theory: a Mathematical Perspective. Society for Industrial and Applied Mathematics (1989).

Foley90 Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. Computer Graphics: Principles and Practice (2nd edition). Addison-Wesley (1990).

Friedmann92 Friedmann, Martin, Thad Starner, and Alex Pentland. Device Synchronization Using an Optimal Linear Filter. *Proceedings of 1992 Symposium on Interactive 3D Graphics* (Cambridge, MA, 29 March - 1 April 1992). A special issue of *Computer Graphics*, 57-62.

Fuchs89 Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Proceedings of SIGGRAPH '89* (Boston, MA, 31 July - 4 August 1989). In *Computer Graphics 23*, 3 (July 1989), 79-88.

Funkhouser93 Funkhouser, Thomas A. and Carlo H. Séquin. Adaptive Dislay Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH '93* (Anaheim, CA, 1-6 August 1993). In *Computer Graphics*, Annual Conference Series, 1993, 247-254.

Garcia92 Garcia, F. M. Rate Output Integration (Angle) vs. Time: Test Report of the Quartz Rate Sensor. Systron Donner Inertial Division engineering department technical report DO 92-23 (28 October 1992),10 pages.

Gottschalk93 Gottschalk, Stefan, and John F. Hughes. Autocalibration for Virtual Environments Tracking Hardware.

Proceedings of SIGGRAPH '93 (Anaheim, CA, 1-6 August 1993). In *Computer Graphics*, Annual Conference Series, 1993, 65-72.

Harris78 — Harris, Frederic J. On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proceedings of the IEEE 66*, 1 (January 1978), 51-83.

Holmgren92 — Holmgren, Douglas E. Design and Construction of a 30-Degree See-Through Head-Mounted Display. UNC Chapel Hill Department of Computer Science technical report TR 92-030 (July 1992), 4 pages.

Jagacinski85 — Jagacinski, Richard J., and Donald L. Monk. Fitts' Law in Two Dimensions with Hand and Head Movements. *Journal of Motor Behavior 17*, 1 (1985), 77-95.

Jain89 — Jain, Anil K. Fundamentals of Digital Image Processing. Prentice Hall (1989). ISBN 0-13-336165-9.

Janin93 — Janin, Adam L., David W. Mizell, and Thomas P. Caudell. Calibration of Head-Mounted Displays for Augmented Reality Applications. *Proceedings of IEEE VRAIS '93* (Seattle, WA, 18-22 September 1993), 246-255.

Jenkins68 — Jenkins, Gwilym M. and Donald G. Watts. Spectral Analysis and its Applications. Holden-Day (1968). Library of Congress number 67-13840.

Kalman60 — Kalman, R.E. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME, J. Basic Eng 82*, D (March 1960), 35-45.

Kalman61 — Kalman, R.E., and R.S. Bucy. New Results in Linear Filtering and Prediction Theory. *Transactions of the ASME, J. Basic Eng 83*, D (March 1961), 95-108.

Krueger92 — Krueger, Myron W. Simulation Versus Artificial Reality. *Proceedings of IMAGE VI Conference* (Scottsdale, AZ, 14-17 July 1992), 147-155.

Lawrence92 — Lawrence, Anthony. Modern Inertial Technology: Navigation, Guidance, and Control. Springer-Verlag (1992). ISBN 0-387-97868-2.

Lenz88 — Lenz, Reimar K. and Roger Y. Tsai. Techniques for Calibration of the Scale Factor and Image Center for High Accuracy 3-D Machine Vision Metrology. *IEEE Transactions of Pattern Analysis and Machine Intelligence 10*, 5 (September 1988), 713-720.

Lewis86 Lewis, Frank L. Optimal Estimation with an Introduction to Stochastic Control Theory. John Wiley & Sons, Inc. (1986). ISBN 0-471-83741-5.

Liang91 Liang, Jiandong, Chris Shaw, and Mark Green. On Temporal-Spatial Realism in the Virtual Reality Environment. *Proceedings of the 4th Annual ACM Symposium on User Interface Software & Technology* (Hilton Head, SC, 11-13 November 1991), 19-25.

List83 List, Uwe H. Nonlinear Prediction of Head Movements for Helmet-Mounted Displays. Technical report AFHRL-TP-83-45, William AFB, AZ: Operations Training Division Air Force Human Resources Laboratory (December 1983), 21 pages

Magill65 Magill, D. T. Optimal Adaptive Estimation of Sampled Stochastic Processes. *IEEE Transactions on Automatic Control AC-10*, 4 (October 1965), 434-439.

Maybeck79 Maybeck, Peter S. Stochastic Models, Estimation and Control, Volume 1. Academic Press (1979).

McFarland86 McFarland, Richard E. CGI Delay Compensation. NASA Technical Memorandum S6703 (1986) [N88-12932].

McFarland88 McFarland, Richard E. Transport Delay Compensation for Computer-Generated Imagery Systems. NASA Technical Memorandum 100084 (January 1988) [N88-17645].

Meyer88 Meyer, David E, Sylvan Kornblum, Richard A. Abrams, Charles E. Wright, and J.E. Keith Smith. Optimality in Human Motor Performance: Ideal Control of Rapid Aimed Movements. *Psychological Review 95*, 3 (1988), 340-370.

Mine93 Mine, Mark R. Characterization of End-to-End Delays in Head-Mounted Display Systems. UNC Chapel Hill Department of Computer Science technical report TR 93-001 (March 1993), 11 pages.

Montgomery90 Montgomery, Douglas C., Lynwood A. Johnson, and John S. Gardiner. Forecasting and Time Series Analysis (2nd edition). McGraw-Hill (1990) ISBN 0-07-042858-1.

Morris93 Morris, Ted and Max Donath. Using a Maximum Error Statistic to Evaluate Measurement Errors in 3D Position and Orientation Tracking Systems. *Presence 2*, 4 (Fall 1993), 314-343.

Mugler90 — Mulger, Dale H. Computationally Efficient Linear Prediction from Past Samples of a Band-Limited Signal and its Derivative. *IEEE Transactions of Information Theory 36*, 3 (May 1990), 589-596.

Murray85 — Murray, P. M. and B. Barber. Visual Display Research Tool. *AGARD Conference Proceedings No. 408 Flight Simulation*. (Cambridge, UK, 30 September - 3 October 1985).

NASAksc — Digital Low-Pass Filter Without Phase Shift. NASA Tech Briefs KSC-11471. John F. Kennedy Space Center, Florida.

Oppenheim83 — Oppenheim, Alan V. and Alan S. Willsky. Signals and Systems. Prentice-Hall (1983). ISBN 0-13-809731-3.

Oyama93 — Oyama, Eimei, Naoki Tsunemoto, Susumu Tachi, and Yasuyuki Inoue. Experimental Study on Remote Manipulation Using Virtual Reality. *Presence 2*, 2 (Spring 1993), 112-124.

Paley92 — Paley, W. Bradford. Head-Tracking Stereo Display: Experiments and Applications. *SPIE Vol. 1669 Stereoscopic Displays and Applications III* (San Jose, CA, 12 - 13 February 1992), 84-89.

Pausch92 — Pausch, Randy, Thomas Crea, and Matthew Conway. A Literature Survey for Virtual Environments: Military Flight Simulator Visual Systems and Simulator Sickness. *Presence 1*, 3 (Summer 1992), 344-363.

Peterson88 — Peterson, Barry W. and Frances J. Richmond. Control of Head Movement. Oxford University Press (1988). ISBN 0-19-504499-1.

Phillips90 — Phillips, Charles L., and H. Troy Nagle. Digital Control System Analysis and Design (2nd edition). Prentice-Hall (1990). ISBN 0-13-213596-5.

Press88 — Press, William H, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. Numerical Recipes in C. Cambridge University Press (1988). ISBN 0-521-35465-X.

Priestley81 — Priestley, M.B. Spectral Analysis and Time Series, Volume 1. Academic Press (1981). ISBN 0-12-564901-0.

Raab79 Rabb, Frederick H., Ernest B. Blood, Terry O. Steiner, and Herbert R. Jones. Magnetic Position and Orientation Tracking System. *IEEE Transactions on Aerospace and Electronic Systems AES-15* , 5 (September 1979), 709-718.

Radwin90 Radwin, Robert G., Gregg C. Vanderheiden, and Mei-Li Lin. A Method for Evaluating Head-Controlled Computer Input Using Fitts' Law. *Human Factors 32*, 4 (August 1990), 423-438.

Rebo88 Rebo, Robert. A Helmet-Mounted Virtual Environment Display System. M.S. Thesis, Air Force Institute of Technology (December 1988).

Regan94 Regan, Matthew, and Ronald Pose. Priority Rendering with a Virtual Reality Address Recalculation Pipeline. *Proceedings of SIGGRAPH '94* (Orlando, FL, 24-29 July 1994). In *Computer Graphics*, Annual Conference Series, 1994, 155-162.

Reisman90 Reisman, Ron. A Brief Introduction to the Art of Flight Simulation. *Virtuelle Welten, Ars Electronica* (Linz, Austria, 1990) 159-170.

Riner92 Riner, Bruce and Blair Browder. Design Guidelines for a Carrier-Based Training System. *Proceedings of IMAGE VI* (Scottsdale, AZ, July 14-17, 1992), 65-73.

Robinett92a Robinett, Warren. Synthetic Experience: A Proposed Taxonomy. *Presence 1*, 2 (Spring 1992), 229-247.

Robinett92b Robinett, Warren and Richard Holloway. Implementation of Flying, Scaling and Grabbing in Virtual Worlds. *Proceedings of 1992 Symposium on Interactive 3D Graphics* (Cambridge, MA, 29 March - 1 April 1992). A special issue of *Computer Graphics*, 189-192.

Robinett92c Robinett, Warren and Jannick Rolland. A Computational Model for the Stereoscopic Optics of a Head-Mounted Display. *Presence 1*, 1 (Winter 1992), 45-62.

Shaw92 Shaw, Chris and Jiandong Liang. An Experiment to Characterize Head Motion in VR and RR Using MR. *Proceedings of 1992 Western Computer Graphics Symposium* (Banff, Alberta, Canada, 6-8 April 1992), 99-101.

Shoemake89 Shoemake, Ken. Quaternion Calculus For Animation. *SIGGRAPH '89 Course Notes #23 on Math for*

SIGGRAPH (Boston, MA, 31 July - 4 August 1989), 187-205.

Sims94 Sims, Dave. New Realities in Aircraft Design and Manufacture. *IEEE Computer Graphics and Applications 14*, 2 (March 1994), 91.

Slama80 Slama, C.C, editor. Manual of Photogrammetry (4th edition). American Society of Photogrammetry (1980).

Smith84 Smith Jr., Bernard. R. Digital Head Tracking and Position Prediction for Helmet Mounted Visual Display Systems. *Proceedings of AIAA 22nd Aerospace Sciences Meeting*, (Reno, NV, 9-12 January 1984) [AIAA-84-0557].

So92 So, Richard H. Y. and Michael J. Griffin. Compensating Lags in Head-Coupled Displays Using Head Position Prediction and Image Deflection. *Journal of Aircraft 29*, 6 (November - December 1992), 1064-1068.

Sobiski87 Sobiski, Donald J., and Frank M. Cardullo. Predictive Compensation of Visual System Time Delays. *Proceedings of AIAA Flight Simulation Technologies Conference* (Monterey, CA), 59-70. [AIAA 87-2434].

Sorenson70 Sorenson, Harold W. Least-squares estimation: from Gauss to Kalman. *IEEE Spectrum* (July 1970), 63-68.

Splettsösser82 Splettstösser, W. On the Prediction of Band-Limited Signals from Past Samples. *Information Sciences 28* (1982), 115-130.

State94 State, Andrei, David T. Chen, Chris Tector, Andrew Brandt, Hong Chen, Ryutarou Ohbuchi, Mike Bajura and Henry Fuchs. Case Study: Obseving a Volume Rendered Fetus within a Pregnant Patient. *Proceedings of IEEE Visualization '94* (Washington D.C., 17-21 October 1994), 364-368.

Sutherland65 Sutherland, Ivan E. The Ultimate Display. *Proceedings of the IFIP Congress 2* (1965), 506-509.

Sutherland68 Sutherland, Ivan. A Head-Mounted Three Dimensional Display. *Fall Joint Computer Conference, AFIPS Conference Proceedings 33* (1968), 757-764.

Systron91 QRS-11 Summary Specifications Sheet. Systron Donner Inertial Division (9 July 1991).

Taubes94 Taubes, Gary. Surgery in Cyberspace. *Discover 15*, 12 (December 1994), 84-94.

Tugnait82 Tugnait, Jitendra K. Detection and Estimation for Abruptly Changing Systems. *Automatica 18*, 5 (September 1982), 607-615.

WangC90 Wang, Chu P., Lawrence Koved, and Semyon Dukach. Design for Interactive Performance in A Virtual Laboratory. *Proceedings of 1990 Symposium on Interactive 3D Graphics* (Snowbird, UT, 1990). In *Computer Graphics 24*, 2 (March 1990), 39-40.

Wanstall89 Wanstall, Brian. HUD on the Head for Combat Pilots. *Interavia 44* (April 1989), 334-338. [A89-39227].

Ward92 Ward, Mark, Ronald Azuma, Robert Bennett, Stefan Gottschalk, and Henry Fuchs. A Demonstrated Optical Tracker With Scalable Work Area for Head-Mounted Display Systems. *Proceedings of 1992 Symposium on Interactive 3D Graphics* (Cambridge, MA, 29 March - 1 April 1992). A special issue of *Computer Graphics*, 43-52.

Welch78 Welch, Robert B. Perceptual Modification: Adapting to Altered Sensory Environments. Academic Press (1978). ISBN 0-12-741850-4.

Welch86 Welch, Brian L., Ron V. Kruk, Jean J. Baribeau, Charles L. Schlef, Martin Shenker, and Paul E. Weissman. Flight Simulator: Wide-Field-Of-View Helmet-Mounted Infinity Display System, Air Force Human Resources Laboratory technical report AFHRL-TR-85-59 (May 1986), 48-60.

Wells87 Wells, Maxwell J. and Michael J. Griffin. A Review and Investigation of Aiming and Tracking Performance with Head-Mounted Sights. *IEEE Transactions on Systems, Man, and Cybernetics SMC-17*, 2 (March - April 1987), 210-221.

Wloka95 Wloka, Matthias M. Lag in Multiprocessor Virtual Reality. To be published in *Presence 4*, 1 (Winter 1995).