XPole: An Interactive, Graphical Signal Analysis and Filter Design Tool

Kennard White

May 21, 1993

Abstract

XPole is an interactive, graphical signal analysis and filter design tool for both continuous-time and discrete-time systems. It serves as an educational tool for learning about pole-zero plots, impulse responses, and frequency responses; as a filter design tool for rapidly designing filters; and as a framework for developing new algorithms. It uses the X Window System via `Tcl/Tk`.

Acknowledgements

This research was aided and supported by many people. My research advisor Prof. Edward Lee originally suggested the concept behind XPole, and provided support through out its development. Prof. Lee also used XPole in his EE123 classes, which provided valuable feedback. Mei-Tjng Huang worked extensively on the eem library, a major component of XPole. Phil Lapsley and Beorn Johnson for many useful discussions and comments on `Tcl/Tk` usage, user interfaces, and signal processing algorithms. Prof. Ousterhout's `Tcl/Tk` package is an integral part of XPole.

This work was supported by the Experimental Computing Facility (XCF) at U.C. Berkeley, the California Microelectronics Fellowship (1990-91), Sony Corp. and by NSF grant MIP-9201605.

Contents

1 XPole Overview **1**

1.1 Introduction 1

1.1.1 Organization 1

1.1.2 Previous Work 2

1.1.3 Purpose 2

1.1.4 Implementation 3

1.2 Capabilities 3

1.2.1 Signal Processing Concepts 4

1.2.2 Filter Design 4

1.2.3 Windowing 10

1.2.4 Scripting and Extensions 12

1.3 Framework 13

1.4 Future Plans 13

2 XPole User's Guide **15**

2.1 Introduction 15

2.1.1 About XPole 16

2.1.2 Acknowledgements 17

2.1.3 Help 17
2.2 Kernel 18
2.2.1 Filters 18
2.2.2 Pipes 19
2.3 Graphical User Interface 21
2.3.1 Pole-Zero Window 21
2.3.2 Frequency Domain Window 23
2.3.3 Time Domain Window 24
2.3.4 Complex Sequence Widget 24
2.3.5 Axis Widget 26
2.3.6 Filter Selection 28
2.3.7 Export Filter 28
2.4 Tcl/Tk 28
2.4.1 Menus 29
2.4.2 Entry Widget 30
2.4.3 Complex Label 30
2.4.4 Complex Entry 31
2.5 Design Algorithms 31
2.5.1 Standard Filter Design 31
2.6 Commands and Scripts 35
2.6.1 Command Line Options 35
2.6.2 Variables 35
2.6.3 Commands 35

3 XPole Programmer's Guide **37**

3.1 Introduction 37

3.2 Framework 37
3.2.1 Current Framework 37
3.2.2 Evolution 44
3.2.3 Future Directions 46
3.3 Libraries 47
3.4 Event Loop 48
3.5 EEM Library 48
3.5.1 Polynomial Arithmetic 48
3.5.2 Polynomial Factoring 49
3.5.3 Partial Fraction Expansion 49
3.5.4 Jacobian Elliptical Functions 49

Chapter 1

XPole Overview

1.1 Introduction

XPole is a interactive, graphical signal analysis and filter design tool for both continuous-time and discrete-time systems. It serves as an educational tool for learning about pole-zero plots, impulse responses, and frequency responses; as a filter design tool for rapidly designing filters; and as a framework for developing new algorithms.

XPole graphically displays various representations of a system including its pole-zero plot, impulse response, and frequency response. Each of these may be modified by the user while the other representations are interactively recalculated to correspond. In addition, mappings such as the bilinear transform and spectral transformations may link two pole-zero plots: user modification of one plot causes the other to be updated. Standard filter design algorithms are implemented, with each step of the algorithm visible to the user.

1.1.1 Organization

Chapter 1 describes the purpose of XPole and overviews its capabilities. Chapter 2 is the *XPole User's Guide*, and details how XPole is used and operated. It is aimed at the reader who wishes to use XPole to accomplish a specific purpose. Chapter 3 is the *XPole Programmer's Guide*, and details the implementation of XPole. It is of interest to anyone wishing to modify or extend XPole. It also discusses XPole's development, and explains approachs that were attempted and discarded.

1.1.2 Previous Work

Signal analysis and filter design has a long history. XPole does not implement any new system representations or new algorithms. Instead, it aims to encapsulate existing concepts and algorithms into a common framework. The graphical interface helps provide intuition about these concepts and exposes the fundamentals of these algorithms. Furthermore, the common framework allows concepts and algorithms to be used as building blocks.

There are many commercial filter design programs[1]. Some will display the pole-zero plots of designed filters. Most provide for tabular entry of poles and zeros and will display pole-zero plots of designed filters. A few allow graphical, interactive placement of poles and zeros much like XPole. However, all of these tools are closed systems with their emphasis on quickly and easily producing a filter realization given a filter specification. In contrast, XPole's emphasis is on illustrating concepts and providing an open framework that may be easily extended.

Most of the algorithms used in XPole are available in any signal processing textbook e.g., [Jac89] and [ZTF83]. A collection of historically useful algorithms was published by the IEEE in [Com79], and AT&T's netlib [NET] is an on-line source of many modern algorithms. However, the emphasis in both cases is on single-purpose algorithms with little effort towards integrating them into a single framework. Further, they provide little or no user interface, much less an interactive graphical system.

1.1.3 Purpose

XPole serves three roles:

Education

The primary purpose of XPole is to aid the student in understanding basic concepts in signal processing. Specifically, the tool demonstrates how different system representations such as pole-zero plots, frequency domain behavior, and time domain behavior relate to each other. The interactive, graphical nature of XPole allows the students to quickly and easily experiment with systems to gain an intuitive understanding of their underlying principles. Similarly, the operation of common filter design algorithms is visually exhibited, allowing the student to gain an understanding of the behavior of the algorithms.

[1] An extensive survey of commercial filter design tools may be found in [BLL93] from Berkeley Design Technology, Inc.

Filter design

XPole implements several standard filter design algorithms; thus it may be used by a system designer to automatically generate an appropriate filter meeting some specification. In addition to designing standard filters, non-standard, special-purpose filters may be manually designed.

Algorithms

XPole provides a framework for describing, manipulating, and displaying filters. Thus an algorithm designer may implement his algorithm within the framework, building on the pre-existing algorithms and the user interface provided by XPole. The immediate visual feedback provided by XPole's user interface can be particularly useful during algorithm development.

The filter design aspect of XPole is secondary since, as described above, there are many commercial filter design tools. However, the requirements of filter design algorithms are an important driving force in designing the framework. Filter design algorithms also provide an educational opportunity to demonstrate fundamental signal processing concepts.

1.1.4 Implementation

XPole is written in ANSI `C`-code (25,000 lines) and `Tcl/Tk`-code (8,000 lines), and runs under the X Window System. The `C`-code implements the XPole framework, most of the signal processing algorithms, and some custom `Tk` widgets (the kernel of the graphics interface). The `Tcl/Tk`-code implements some of the simple signal processing algorithms and most of the graphical user interface.

XPole currently runs on SparcStation and DECStation systems, and should run on any UNIX workstation where the X Window System and an ANSI `C` compiler is available. Complete source code is freely available from the author.

1.2 Capabilities

XPole is designed around *filters*: linear, time-invariant (LTI), single-input, single-output (SISO) systems.

It is important to note that mathematically there is no distinction between signals and systems. More precisely, a given signal is equivalent to an LTI SISO system whose impulse response is equal to the given signal. Thus *filter* should be read as either *system* or *signal*, as appropriate to the context.

A particular filter may be represented in several different ways; each such representation is called a *view*. In particular, a filter may be described by its pole-zero plot, its impulse response, and its Fourier transform.

1.2.1 Signal Processing Concepts

XPole may be used to explore the relationship between views of a filter. Figure 1.1 is a screen dump showing a simple two-pole discrete-time system. The upper-left shows the Z-plane with the two poles, the upper-right shows the magnitude and phase frequency response, and the bottom shows the impulse response. The user may use the mouse to move the poles around in the Z-plane, and the frequency and impulse responses will interactively update. Similarly, the user can see the effect of adding and deleting new poles and zeros by similar mouse manipulation.

Another simple concept demonstrated by XPole is the bilinear transform. Figure 1.2 is a screen dump showing an approximate low-pass filter. The upper-left shows the S-plane and the upper-right shows the Z-plane. The poles and zeros in each are bound according to the bilinear transform. The user may move any of the poles and zeros in either the S-plane or Z-plane, and its 'mate' under the bilinear transform will move appropriately. The bottom portion of the figure shows the frequency response of both the continuous-time (thin line) and discrete-time (thick) filters. Note that the discrete-time system is periodic while the continuous-time system is not. The user may also select a matched-z or impulse invariant transformation instead of bilinear. However, currently these only map from the S-plane to Z-plane (e.g., user changes to the Z-plane will not affect the S-plane).

1.2.2 Filter Design

IIR Filters

XPole supports IIR filter design using the menu shown in figure 1.3. The pole-zero plot and frequency response of the filter obtained using the options and parameters in figure 1.3 (6-th order elliptical band-pass) is shown in figure 1.4; the user may also view its impulse response. The user can change any of the options shown in figure 1.3, as well as any of the parameters, and the filter will be immediately redesigned and the plots updated accordingly. Through experimentation between different design methods and parameters, this allows the

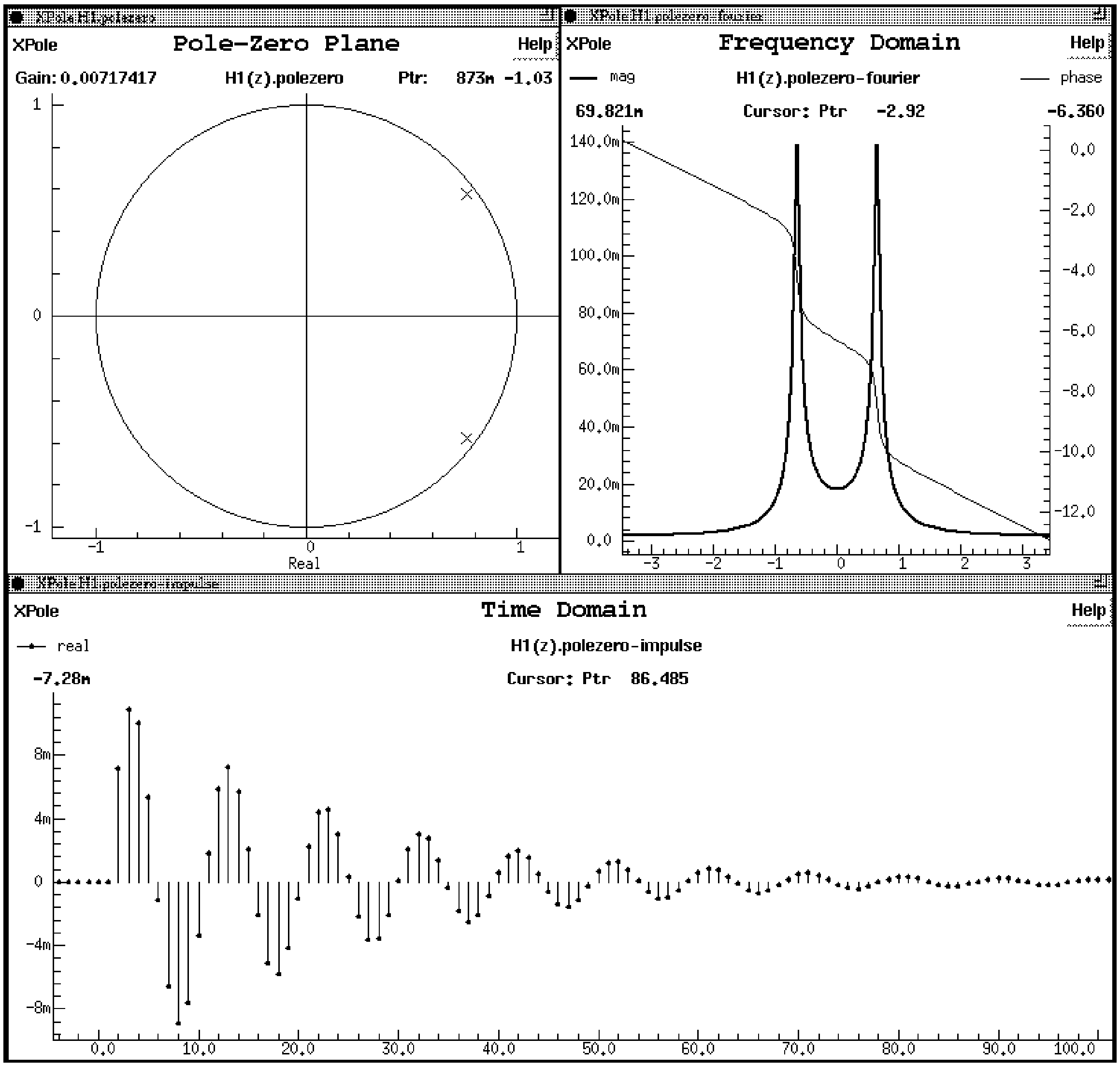

Figure 1.1: Simple Two Pole System. Upper-left is pole-zero plot, upper-right is frequency response, and bottom is impulse response.

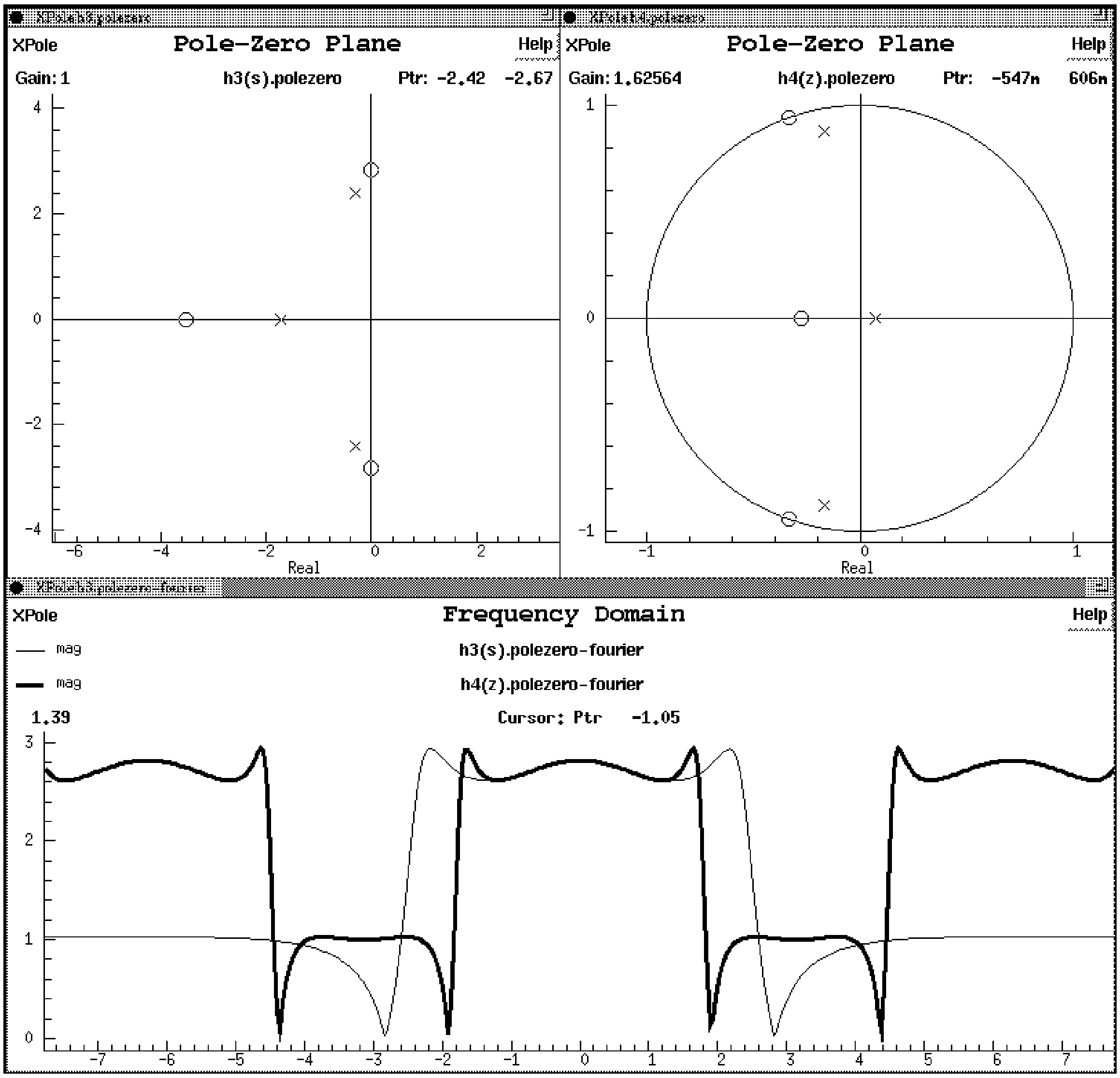

Figure 1.2: Bilinear Transform. The discrete-time filter in upper-right is the bilinear transform of the continuous-time filter in upper-left.

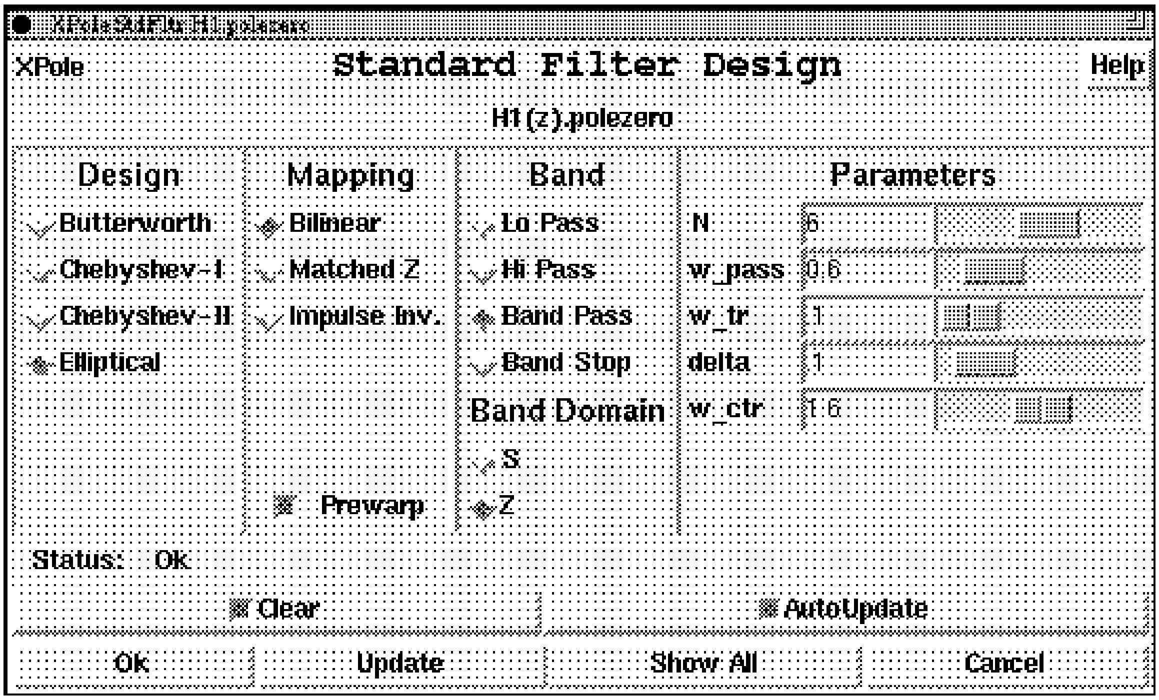

Figure 1.3: IIR Filter Design Menu

user to develop an intuitive understanding for the filter design process. For example, the left side of figure 1.5 shows the pole-zero plot of a Butterworth filter designed using the same parameters. The right side shows the frequency response of both filters on a logarithmic scale (thick line is Butterworth).

For educational uses, XPole will expose the details of the design process: the user may ask to see the intermediate filters that are used at each step of the process. In the example above (6-th order elliptical band-pass), this includes the original low-pass S-domain filter and the low-pass Z-domain filter. The pole-zero plot, frequency response and impulse response of all intermediate filters may be displayed. For example, figure 1.6 shows the pole-zero plots of the two intermediate filters for the previous example, as well as their impulse responses.

FIR Filters

Currently, XPole does not support FIR filter design. Algorithms for design by frequency sampling are currently being implemented, and will be followed by more sophisticated de-

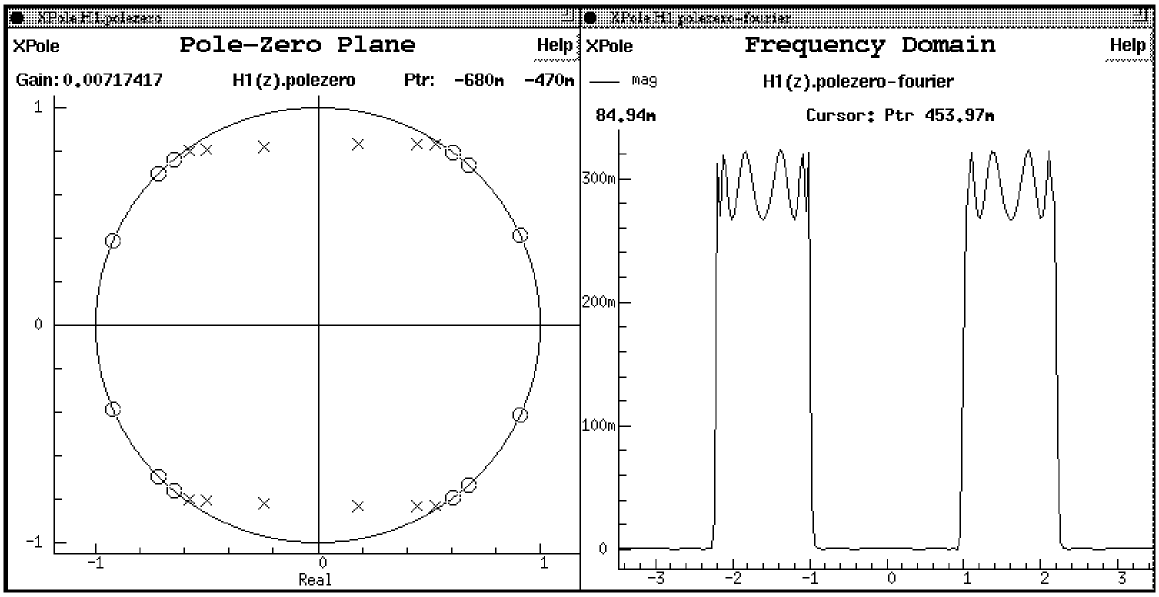

Figure 1.4: 6-th order Elliptical Band Pass Filter

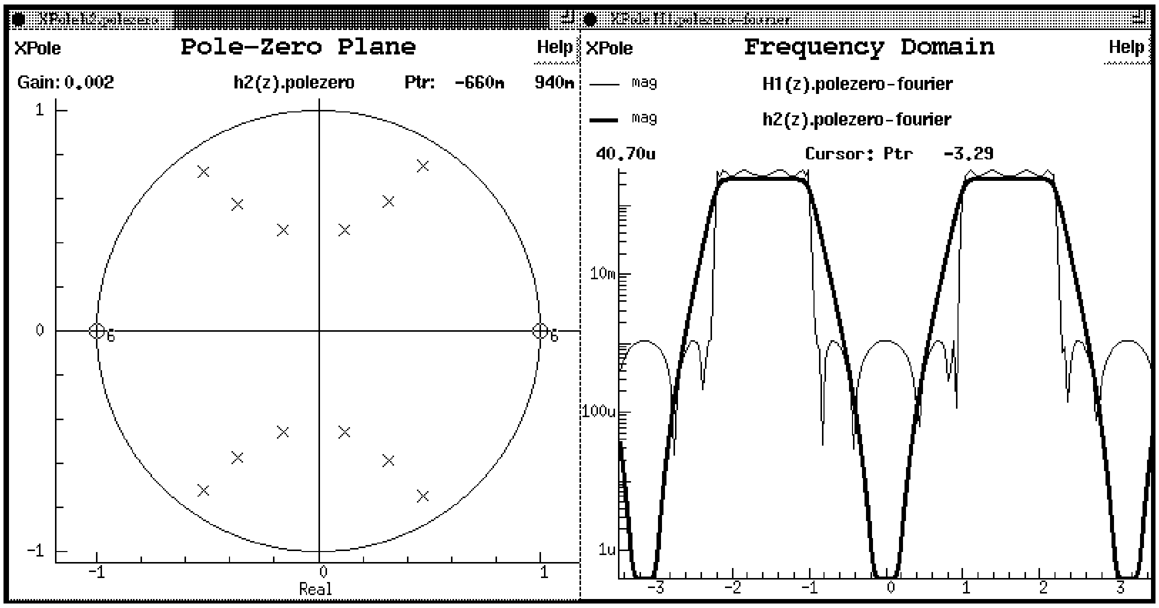

Figure 1.5: 6-th order Butterworth Band Pass Filter

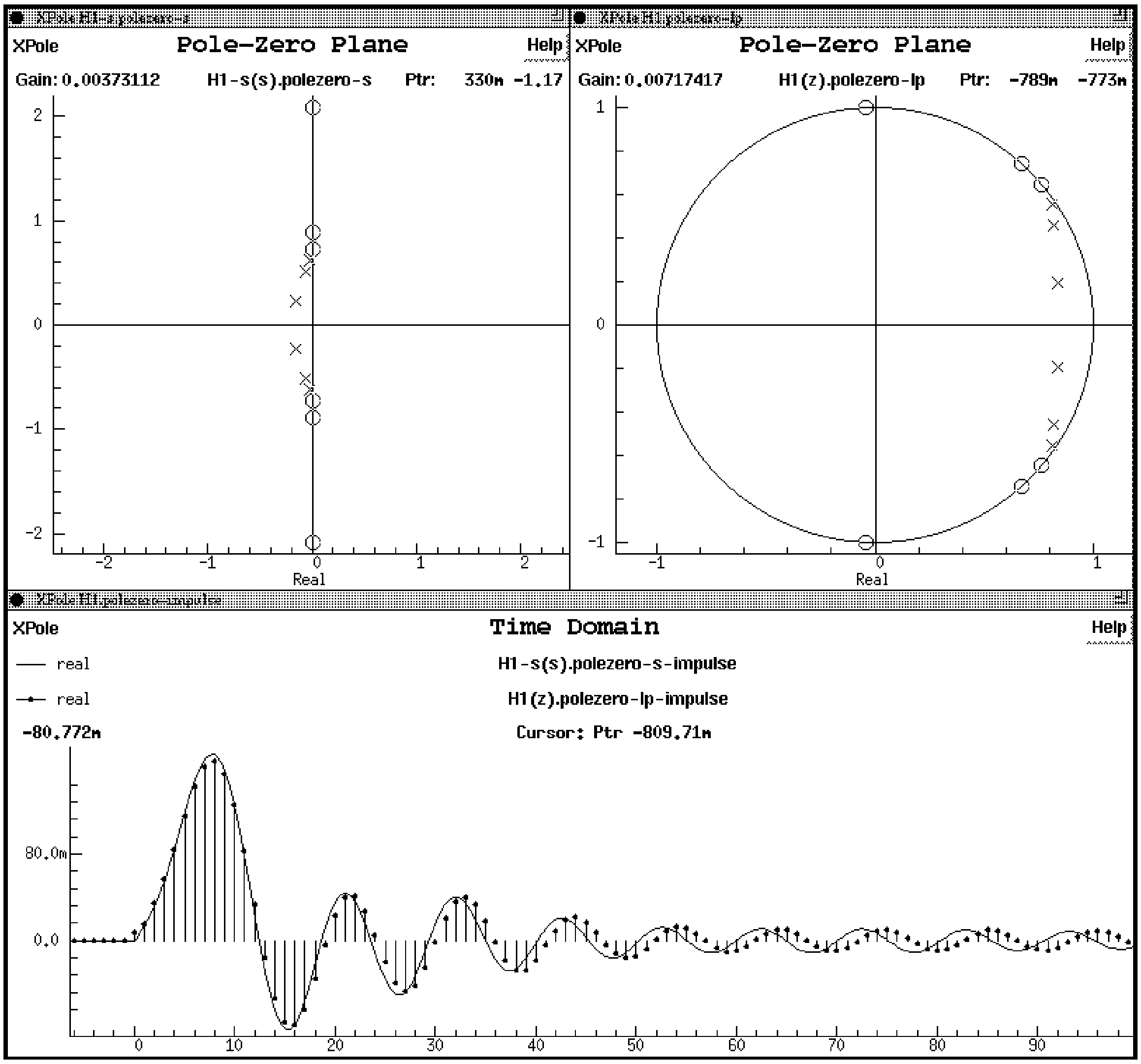

Figure 1.6: Intermediate Filters in the Design Process for a 6-th order Elliptical Band Pass Filter. Upper-left is a continuous-time, low-pass filter, and upper-right is discrete-time low-pass via the bilinear transform.

signs such as equiripple filters.

Note that IIR and FIR designs require different methodologies; in particular, IIR filters are designed by placing poles and zeros, while FIR filters and designed by calculating the desired impulse response (tap values). Which mathematically equivalent, in practice these are very different. For example, finding the zeros of a 256-tap FIR filter would require factoring a 256-degree polynomial, which simply cannot be done robustly (the problem is extremely ill-conditioned). Similarly, given an 256-zero FIR filter, calculating the impulse response requires multipling together 256 first-order polynomials. This leads to substantial numerical round-off noise, as well as dynamic range errors unless the zeros are well chosen.

Realization and Export

Typically, once a filter has been designed, an implementable realization must be found, and that realization must be exported to another application. XPole is designed to interface to the Ptolemy system developed at U.C. Berkeley; see [BHLM93]. XPole will find realizations for a variety of Ptolemy filter stars, and export them in a format suitable for use by Ptolemy. The realization can be communicated to the Ptolemy system either through files or through an inter-process communication (IPC) mechanism. The IPC mechanism is useful when XPole is being used to interactively control a filter running on a real-time system (via QDM[2]). The realization may also be viewed directly: figure 1.7 shows a biquad cascade realization. When the filter is modified, the realization will be interactively updated.

1.2.3 Windowing

Windowing is a common concept is signal processing, and occurs in spectral estimation and FIR filter design. XPole may be used to demonstrate the effect of windowing. Figure 1.8 shows a 10th order elliptical filter. On the left is the impulse response. The right side shows the frequency response: the thin line is the complete IIR filter, and the thick line is the result of applying a 16-sample rectangular window to the impulse response. Other windows, including the Hanning, Hamming, and Blackman windows, are available.

While windowing may be applied to standard IIR filters to obtain an FIR filter, such an FIR filter would be inferior to standard FIR designs. However, windowing will eventually be part of XPole's FIR design system.

[2]QDM is an debugger and monitor for real-time DSP systems; see [Lap91]

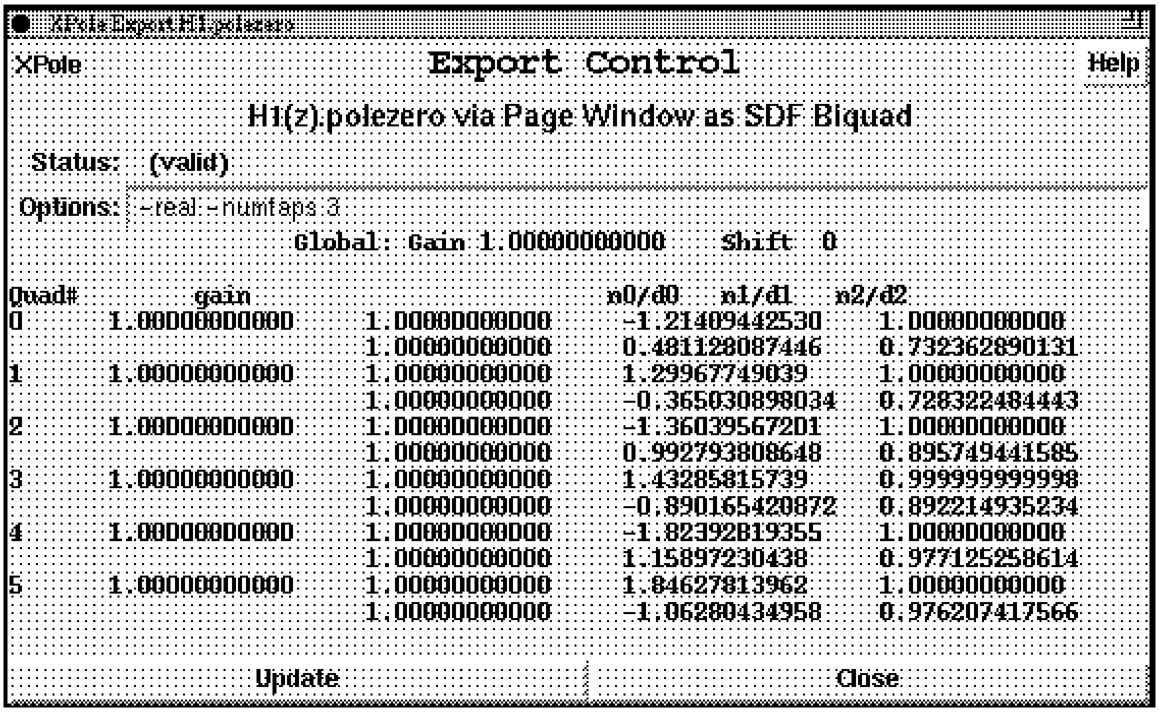

Figure 1.7: Elliptical Filter Biquad Cascade Realization

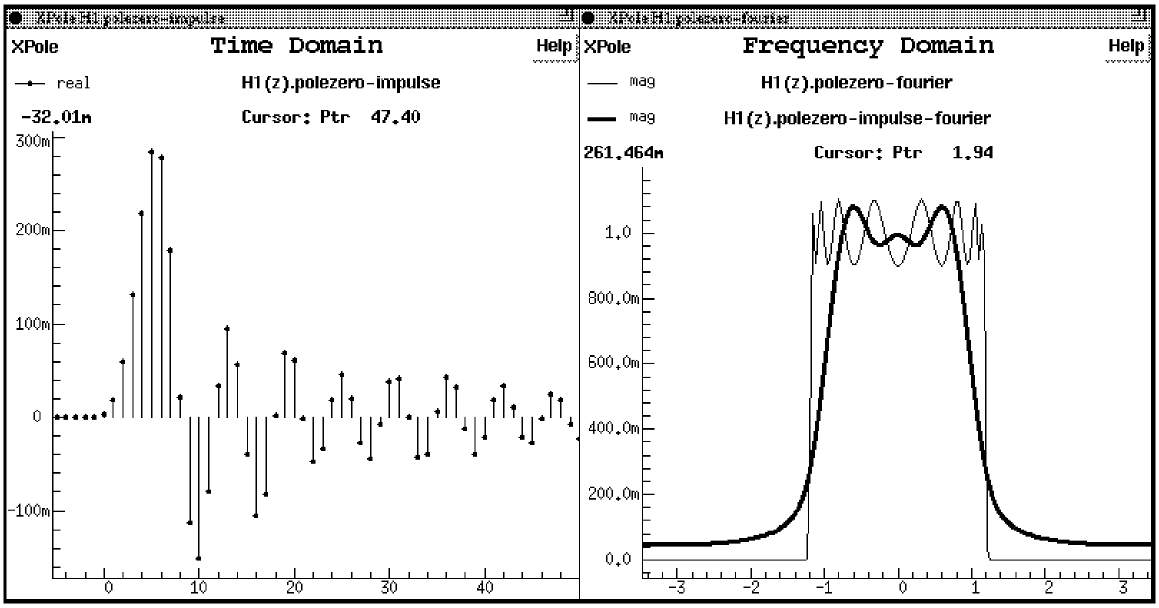

Figure 1.8: Effects of Windowing on 10-th Order Elliptical Filter

1.2.4 Scripting and Extensions

XPole uses Tool Command Language (`Tcl`) [Ous90] to provide a command and scripting language. This is an interpreted, complete language (providing procedures, variables, conditionals, and loops) and is extensible to provide custom commands for a particular application. For XPole, commands have been added so that all views may be monitored and modified. This allows views to be manipulated in `Tcl`-code rather than `C`-code. In fact, the simpler filter design algorithms and the spectral transformation algorithm are implemented in `Tcl`-code. This extensibility may be used in several different ways:

- The instructor may write a script in `Tcl` that configures XPole to demonstrate a particular signal processing concept of interest to students. The script may hide implementation details that are not important to convey to students.
- The algorithm designer may quickly express an algorithm in `Tcl` code, and immediately invoke and test the algorithm. This avoids time-consuming and tedious compile and link stages, since `Tcl` is interpreted.
- All XPole graphics are implemented using the X Window System `Tk` toolkit [Ous91], which is based on `Tcl`. Thus a programmer may easily add to or modify the user

interface by writing `Tcl` scripts. This requires no compilation nor linking.

Details on how to use `Tcl/Tk` and extensions are provided in the *XPole Programmer's Guide*.

1.3 Framework

In addition to the capabilities described above, XPole is intended to be easily extensible. To accomplish this goal, XPole is implemented as many independent modular components interacting through a common framework. This framework allows components of the system to be combined as building blocks to construct an arbitrarily large system. This allows an algorithm designer to add a new component (e.g., a new pole-zero transformation), and combine it with the existing algorithms. Further, the graphical user interface may be used without change. Similarly, a programmer may desire to represent or display filters in a new way; this new approach may be easily implemented and integrated with the existing system. The *XPole Programmer's Guide* explains the concept underlying the framework and how to use it.

1.4 Future Plans

XPole is a useful tool for learning basic signal processing concepts. However, it is still incomplete. FIR design capabilities would greatly improve its utility. Root-locus and Nyquist plots would be very useful for understanding classical control theory. All-pass/minimum-phase decompositions would be useful for communications. The framework must be extended to allow more complicated associations between several views. Finally, the user interface is not as clean as could be hoped, and is too complicated for casual users.

Chapter 2

XPole User's Guide

This is the XPole User's Guide: it explains what XPole is and how to operate XPole. Basic knowledge of signal processing terminology is assumed. This guide is available as both a printed TeX document and as an on-line Info file.

2.1 Introduction

XPole is a graphical signal analysis and filter design tool. It supports both continuous-time (S-domain) and discrete-time (Z-domain) filters. Synthesis of standard filters is also supported.

Due to its graphical nature, XPole is an useful tool for gaining an intuitive understanding of S- and Z- transforms, and for understanding the structure of standard filters such a Butterworth and elliptical designs.

XPole is not a replacement for numerical engines such as `Matlab`: explicit vector and matrix manipulation is not supported. XPole is also not a replacement for symbolic algebra systems such as `Maple`: the algorithms in XPole are intended to be quick and provide intuition, not to provide exact solutions with known error bounds.

XPole may augment, but not replace, commercial filter design tools such as `QEDesign` and `Monarch`. Commercial packages allow rapid design of a large variety of filter designs, whereas XPole supports a smaller set of standard filters and emphasizes understanding.

2.1.1 About XPole

XPole version 1.4.5 by Kennard White (based on tcl-6.6 and tk-3.1).

XPole and its related files are the work of Kennard White. XPole was written at the University of California, Berkeley with guidance by Prof. Edward A. Lee of the Electrical Engineering and Computer Science Department. Mei-Tjng Huang wrote portions of the numerical algorithms (see below) with guidance by Kennard White.

Development of XPole is or was supported by the following; however, their support is not an endorsement nor recommendation of XPole.

- The Experimental Computing Facility (XCF) at U.C. Berkeley
- The California Microelectronics Fellowship (1990-91)
- Sony Corp.
- NSF grant MIP-9201605.

For information, comments, code-contributions or bug reports please contact the author at:


```
phone:  510/643-6686
e-mail: kennard@ohm.eecs.Berkeley.EDU
s-mail: Kennard White, Box #115
        207 Cory Hall
        U.C. Berkeley
        Berkeley, CA 94720, USA
```


Other fun facts:

- The version number shown above is actually the version of this guide. Its possible that the software you are using is a different version. To see the actual version number, use the tcl command

  ```
  xpole version
  ```

- XPole source is organized into 4 packages (top, eem, tkgraph and xpole), with a total of 25k lines of C source code and 6k lines of Tcl/Tk scripts.

- Mei-Tjng Huang wrote portions of the eem library; in particular, she wrote initial versions of the poly, pfe, and locus modules, and wrote much of the jef module.

- XPole uses Tool Command Language (Tcl) as its command interface and scripting language, and Toolkit (Tk) for its graphical user interface. Both software packages are by Prof. Ousterhout at U.C. Berkeley. See section 2.4 [Tcl/Tk], page 28.

- Guide RCS: $Revision: 1.1 $, $Date: 93/05/18 12:11:25 $.

2.1.2 Acknowledgements

The concept behind XPole was originally suggested by Prof. Edward Lee in an undergraduate signal processing class during Spring '89. The first implementation was undertaken in Fall '90, and Prof. Lee has supported its development since that time. I am grateful for both the original idea and the support. I would also like to thank Prof. Lee for using XPole in the EE123 classes of Spring '92 and '93, and the classes themselves for providing useful feedback.

I thank Mei-Tjng Huang for her extensive work on the eem library, Phil Lapsley for discussions of Tcl/Tk usage, user interfaces and signal processing algorithms, and Beorn Johnson for countless discussions on user interfaces and data structures.

Prof. Ousterhout's Tcl/Tk package is an integral part of XPole; Tcl/Tk is an amazing package and I thank Prof. Ousterhout for developing it and making it available.

2.1.3 Help

XPole has context-sensitive help to guide you. Just press the nearest help button, and a help screen will appear. In most cases, pressing the `HELP` button will also work. You can also get help through the command line interface: just type "help [topic]", and a help screen will appear.

To learn how to use the help system, press the `?` key over the help screen.

The help system is implemented using an Info file with a texinfo file as the source document. See section *Texinfo* in Introduction, for details. The Info file is displayed on the screen using the tkinfo package. See Info file '`builtin`', node 'Top', for details.

2.2 Kernel

This chapter describes the core concepts, terminology and algorithms that are at the heart of XPole.

2.2.1 Filters

XPole is designed around *filters*: linear, time-invariant (LTI), single-input, single-output (SISO) systems.

It is important to note that mathematically there is no distinction between signals and systems; more precisely, a given signal is equivalent to an LTI SISO system whose impulse response is equal to the given signal. Thus "filter" should be read as either "system" or "signal", as appropriate to the context.

Currently, the sample rate of all discrete-time filters is assumed to be 1.0 sec. Hopefully this will be user-controllable in the future.

A particular filter may be expressed in several different ways; each such expression is called a *view* within XPole. In particular, a filter may be described by its poles and zeros (S- or Z-plane), its impulse response, and its Fourier transform. These views are described in greater detail below.

Algorithms may be used to calculate one view from another. In XPole, such algorithms are encapsulated in *pipes*, and are also described below.

Pole-Zero View

The *Pole-Zero* view describes the filter in the S- or Z-domain by its poles and zeros. Specifically, a Z-domain pole-zero view represents a filter as:

```
                  (z-a1)(z-a2) ... (z-aN)
       H(z) = G * -----------------------
                  (z-b1)(z-b2) ... (z-bM)
   where
       G is the complex gain
       a1,...,aN are the complex zeros
       b1,...,bM are the complex poles
```

The S-domain form is equivalent. The internal representation has special support for complex conjugate pairs, and for poles and zeros with multiplicity. A pole or zero is described by its location in the S or Z plane and its power (multiplicity or order). The location in the complex plane can be expressed by its real(r) and imaginary(i) parts, or by its magnitude(m) and phase(p).

This view may be displayed graphically, see section Polezero Win in Pole-Zero Window.

Internally, and pole or zero is called a *singularity*, or *silar* for short. XPole attempts to use '`pole-zero`' everywhere; however, sometimes you might see '`silar`'. The term singularity in this context is standard in complex analysis, see [MH87]. Essentially, poles and zeros are exactly symmetric in the two-sided Laurent expansion; both are termed singularities.

Impulse View

The *impulse* view is the impulse response of the filter; more precisely, it is samples of the output of the filter with an impulse function as input. For continuous-time filters, the sample locations are chosen based on the domain of the graph being displayed: a sample is calculated every two or three pixels. For discrete-time filters, the sample locations match the sample rate.

This view may be displayed graphically, see section Time Win in Time Window.

Fourier View

The *fourier* view is the Fourier transform of the filter; more precisely, it is samples of the Fourier transform of the impulse response of the filter. The sample locations are chosen based on the domains of the graph being displayed: a sample is calculated every two or three pixels.

This view may be displayed graphically, see section Freq Win in Frequency Window.

2.2.2 Pipes

Signal processing algorithms are accessed within XPole as *pipes*. A pipe maps one filter view into another filter view. The currently implemented pipes are described below.

Polezero Fourier Pipe

This pipe maps a pole-zero view into its Fourier transform view. In S-domain, the silar view provides a transfer function H(s); this is evaluated at s=jw to calculate the Fourier sequence. In Z-domain, the silar view provides H(z); this is evaluated with z along the unit circle.

Polezero Impulse Pipe

This pipe maps a pole-zero view into its impulse response view. This is implemented by performing a partial fraction expansion (PFE) of the transfer function H(s) or H(z). The impulse responses of the individual PFE terms are calculated and summed.

Caution: Partial fraction expansion, as implemented, is not numerically robust. In particular, poles with multiplicity may lead to considerable numerical error.

An alternative method for calculating the impulse response of a discrete-time filter is to synthesize a direct form or cascade form realization, and input an impulse into the synthesized filter. This method requires time-domain convolution for two-sided sequences, and requires calculating the impulse response starting at time 0. Also, this method only works for discrete-time filters, though it could yield an approximation for continuous-time filters. For these reasons, this method is not used by XPole.

Impulse Fourier Pipe

This pipe takes an impulse response, windows it, and computes the DTFT, DFT or FFT as appropriate to produce samples of the Fourier transform. You specify the window `w(n)` by its length N and its type: rectangle, Bartlett, Hanning, Hamming, Blackman and steep-blackman. See [Jac89, chap. 7] for details.

Fourier Impulse Pipe

This pipe does not exist yet. When it does, it will apply an inverse DTFT, DFT or FFT algorithm.

2.3 Graphical User Interface

The XPole interface consists of several windows. Below is a brief description of each.

Pole-Zero Plane This window plots the poles and zeros of the filter on either the S- or Z-plane. You may create, destroy and move poles and zeros on the plane, and instantly see the other windows update.

Frequency Domain This window plots the Fourier transform of the filter. You may view the real, imaginary, magnitude and phase of the response with a multitude of options.

Time Domain This window plots the impulse response of the filter. You may view the real, imaginary, magnitude and phase of the response with a multitude of options.

Export Filter This window allows a filter design to be exported to another application. A variety of modes and formats are available.

2.3.1 Pole-Zero Window

The pole-zero window displays a filter's pole-zero view (see section 2.2.1 [Polezero View], page 18); that is, it plots the poles and zeros of the filter on either the S- or Z-plane. You may create, destroy and move poles and zeros on the plane, and see the other windows update. Poles are shown as a cross (i.e., `X`) and zeros are shown as a circle. If a pole or zero has power (multiplicity) greater than 1, its power will be shown as a subscript. A pole or zero may be *selected* as described below. With the default color scheme, selected poles and zeros are colored green and unselected are white.

The gain `G` is displayed in the upper left corner, and the current position of the mouse pointer in the S or Z-plane is displayed in the upper right corner. Both the gain and position values may be displayed in rectangular or polar coordinates (see section 2.4.3 [Tk Complex Label], page 30).

The contents of the window may be manipulated using a combination of menus, keys, and mouse buttons as described below. In addition, the individual axes may be controlled (e.g., scale, range, labeling format, etc.; see see section 2.3.5 [Axis Widget], page 26).

Menu

To use the pole-zero window menu, press the left mouse button over the top center button that is labeled '`Pole-Zero Window`', and a menu will appear (see section 2.4.1 [Tk Menus], page 29). The entries are:

Conjugate Pairs When on, new poles and zeros will either be created exactly on the real line or in complex conjugate pairs. When off, poles and zeros may be placed anywhere. Filters whose poles and zeros are either on the real line or complex conjugate pairs have a real-valued impulse response.

New Zero, New Pole Create a new pole or zero. The pole or zero will be placed at the origin if invoked from the menu; it will be placed under the mouse pointer if invoked by a key binding.

Zoom Full, Zoom Out, Zoom In Zooming "full" changes the view so that the unit circle and all poles and zeros are visible, zooming "out" shows more of the plane, and zooming "in" shows less of the plane.

Edit Edit all currently selected poles and zeros. A small window will popup for each pole and zero to be edited. The window will allow you to change the location, power, and selection state. See section 2.3.1 [PolezeroWin EditPopup], page 23.

Delete Delete all currently selected poles and zeros.

Increment Power, Decrement Power Modify the power of all currently selected poles and zeros. Note that poles have negative power, thus to increase a pole's multiplicity, you must decrement its power. If the power becomes zero, the pole or zero will be deleted.

New Zero, New Pole Configure the pole-zero window so that when the middle mouse button (button #2) is pressed, a new pole or zero will be created under the mouse pointer.

Zoom Region Configure the pole-zero window so that the middle mouse button (button #2) may be used to zoom in on a region of the plot. This is the default mode and is described below.

Display Impulse, Display Fourier Display the impulse response or the Fourier transform of the filter. See section 2.2.2 [Polezero Impulse Pipe], page 20. See section 2.2.2 [Polezero Fourier Pipe], page 20.

Standard Design Use a standard filter design. See section 2.5.1 [Std Fltr Design], page 31.

Zap Window Destroy this window. The filter itself will not be affected.

Mouse Buttons

The mouse buttons have the following meanings in the pole-zero window:

left(B1) Select a group of poles and zeros. This unselects all poles and zeros and selects the pole or zero nearest to the mouse pointer. You can also drag out a region while holding the button down; in this case, all poles and zeros inside the dragged out region will be selected. Also, if the `SFT` (shift) key is depressed with the mouse button, the currently selected poles and zeros will remain selected.

middle(B2) The action of the middle mouse button depends on the current menu section. The default action is `Zoom Region`. In this mode, you drag out a region while holding the button down. The window will be zoomed in so that the selected region is displayed in a larger scale. The other two modes are `New Zero` and `New Pole`. In these two modes, a new pole or zero will be created under the mouse pointer when the button is pressed.

right(B3) Move (drag) the currently selected poles and zeros. The poles and zeros will track the mouse pointer movement while the button remains depressed. Note that the currently *selected* poles and zeros will be moved, *not* the pole or zero nearest the pointer.

Edit Popup Window

Pressing `e` or using the Pole-Zero Window menu entry '`Edit`' will bring up an edit popup window. The window will allow you to change the location, power, and selection state of a pole or zero. By default, the location is shown in rectangular form, but may be switched to polar form (see section 2.4.4 [Tk Complex Entry], page 31).

2.3.2 Frequency Domain Window

This window displays a fourier view (see section 2.2.1 [Fourier View], page 19); that is, it plots the Fourier transform of a filter. It is generally calculated from another view; see section 2.2.2 [Polezero Fourier Pipe], page 20, and section 2.2.2 [Impulse Fourier Pipe], page 20.

The x-axis is scaled in radians/second. The right y-axis is for phase graphs and is scaled in radians. The left y-axis is used for all non-phase graphs, including real-part, imaginary-part, and magnitude.

This window is a special case of the Complex Sequence Widget. See section 2.3.4 [CSeq Widget], page 24, for details of operation.

2.3.3 Time Domain Window

This window displays an impulse response view (see section 2.2.1 [Impulse View], page 19); that is, it plots the impulse response of a filter. It is generally calculated from another view; see section 2.2.2 [Polezero Impulse Pipe], page 20, and section 2.2.2 [Fourier Impulse Pipe], page 20.

The x-axis is scaled in seconds. The right y-axis is for phase graphs and is scaled in radians. The left y-axis is used for all non-phase graphs, including real-part, imaginary-part, and magnitude.

This window is a special case of the Complex Sequence Widget. See section 2.3.4 [CSeq Widget], page 24, for details of operation.

2.3.4 Complex Sequence Widget

This window displays a set of complex sequences. The sequence is composed of samples of a Fourier response, in the case of the fourier window (see section Freq Win in Frequency Window), or samples of an impulse response in the case of the impulse window (see section Time Win in Time Window). The window plots one or more such sequences.

A complex-valued sequence cannot itself be directly plotted; instead, its *components* (real-part, imaginary-part, magnitude, phase) may be. Each component is plotted using *lines* and *marks*. The lines joins the points of the sequence, while the marks indicate the sequence points themselves.

The complex sequence widget's legend is at the top of the window. The legend is used both to label the displayed components and to control their appearance (mark and line styles). See section 2.3.4 [CSeqWdg Legend], page 25.

The scale of the x-axis depends on the context. The right y-axis is used for phase components (in radians), and the left y-axis for all others. The individual axes may be controlled; in particular, linear/logarithmic, zooming and labeling modes are all controlled through the axis widget. To modify and of these parameters, bring up the axis widget's menu by pressing the left mouse button over the axis (where the numeric labels are, not the graphing area), and see section 2.3.5 [Axis Widget], page 26.

Pressing the left mouse button over the label '`Frequency Domain`' or '`Time Domain`' (at the top of the window), brings up the complex sequence widget's main menu with entries:

Add Sequence Specify an additional sequence to display in the window.

Zoom Full, Zoom Out, Zoom In Change how much of the sequences are visible. By default, the axis have '`Auto Full`' mode enabled, which will tend to interfere with attempts to manually control the zoom region.

Zap Window Destroy the window. The sequences themselves are not affected.

The middle mouse button may also be used to drag out a region; the selected region will be enlarged.

Note: Currently, there is no mechanism to plot Dirac (continuous-time) delta functions. (Discrete-time delta functions are plotted without difficulty).

Complex Sequence Widget Legend

The legend is used to both label the data plotted and to control how the data is displayed. Each sequence on the graph corresponds to a row in the legend; the sequence name appears in the middle of each row. Phase components are listed to the right of the sequence label and all other components are to the left. Each component is shown with its name and a sample line segment.

Pressing the left mouse button over the sequence label will bring up a menu with entries:

Rect, Polar Show the sequence in either rectangular (real & imaginary) or polar (magnitude & phase) form. These are short cuts to avoid using the checkbuttons described next.

Real, Imag, Mag, Phase, WrapPhase These checkbuttons allow you to select which components of the sequence to plot. '`WrapPhase`' has phase values from -PI to +PI, while '`Phase`' has had the 2 PI ambiguity removed.

Zap Remove the entire sequence from the graph. The sequence itself is not affected.

Pressing the left mouse button over a component label will bring up a menu with entries:

Line Style Choose how the line for the component will be drawn using the submenu:

- **Solid, Fence, None** Selects a variety of different line styles.
- **xor** Enables *xor* drawing. When off, component lines overlay each other in a random stacking order. When on, xor drawing mode will be used, leading to random colors where ever component lines intersect.
- **0-9** Selects the line width. A line width of 0 may behave strangely on some X servers.

Mark Style Choose how the mark for the component will be drawn using the submenu:

- **Circle, Dot, Box, Cross, None** Selects a variety of different mark styles.
- **1-9** Selects the line width (diameter).

Zap Remove the component from the graph.

2.3.5 Axis Widget

The axis widget displays the labels on an axis of a graph. It also provides an interface to allow you to modify how the data and labels are displayed. Pressing the left mouse button or the `m` key over the axis numbers brings up the axis menu with entries:

Auto Scale When enabled, the axis will automatically zoom in or out as required to show the entire picture.

Show Label When enabled, the textual label for the axis will be displayed.

Show Scrollbar When enabled, a scrollbar is placed adjacent to the axis. This scrollbar may be used to pan the graph.

Scale For more information on the transforms for the scales described below, see section 2.3.5 [AxisWdg Scale], page 28.

- **Linear** Use a linear mapping transform. Normally you will want to use '`Natural`' labeling mode (see below) when using a linear scale.

Log Use a logarithmic mapping transform. Clipping may be active for small values. Normally you will want to use '`dB-20`' labeling mode (see below) when using a logarithmic scale.

Dslog Use a double-sided logarithmic mapping transform. This is a logarithmic transform that allows negative values to be displayed.

Label Selects one of several labeling modes. Note that the labeling mode is independent of the scale; in particular, decibel labels may be used on a linear or log scale.

Natural Display the axis numeric labels as-is, without additional transform.

dB-10 Label the axis in power decibels (10*log10). Labels will be suffixed with '`d`'.

dB-20 Label the axis in signal decibels (20*log10). Labels will be suffixed with '`D`'.

PI Label the axis in multiples of PI. Labels will be suffixed with '`P`'.

Format Select how the labels are formatted.

Engineer Format the labels using engineering notation: the exponent is adjusted to be a multiple of 3, and the standard abbreviations (f,n,u,m,K,M,G) are used.

Scientific Format the labels using scientific notation.

Short Sci Format the labels using shortened scientific notation: this is the sprintf() format "%g".

Fixed Pt Format the labels in fixed point notation.

Ticks Select where the tick marks are displayed.

None Don't display any tick marks.

Inside Place the tick marks inside the picture area.

Outside Place the tick marks outside the picture area.

Set Lo, Set Mid, Set Hi, Set Clip When invoked a small dialog box will appear. You may enter the desired new lo limit, middle point, hi limit, or clipping threshold and press <Enter>. You may press <Escape> to abort the operation. The lo, middle or hi point may also be adjusted by typing a number in the lower, middle or upper third of the axis widget.

Zoom In, Zoom Out, Zoom Full Change the view area.

Help Display this description.

Dismiss Make the popup menu go away.

Axis Widget Scale

In logarithmic mode, it is not possible to display zero or negative values. In fact, very small positive numbers are difficult to display due to numeric problems. Such numbers occurs often in frequency responses, particularly when there are zeros on the unit circle (z-domain). To address this problem, the axis widget implements a *clipping threshold*: values less than this will either not be displayed, or will be displayed in a modified form. Specifically, when the axis's lo limit is less than the clipping threshold then a small portion of the axis edge near the bottom will be displayed in dark blue. This region corresponds to values between the lo limit and the threshold, and values in this region are mapped linearly, not logarithmicly. This allows you to see the full graph (by setting the lo limit to zero) while viewing the bulk of the graph logarithmicly.

To change the clipping threshold use the menu entry '`Set Clip`'.

2.3.6 Filter Selection

This dialog window lets you select an existing filter or create a new filter. Use the two listboxes to select an existing filter, or type the name of a new filter and/or view in the entry widget towards the bottom.

2.3.7 Export Filter

This dialog window lets you export a filter to a tool outside of XPole, typically Ptolemy.

2.4 Tcl/Tk

XPole uses *Tcl* (Tool Command Language) as its command language, and *Tk* (Took Kit) to implement its graphical user interface. Both are software packages from Prof. Ousterhout at U.C. Berkeley, and are documented elsewhere (see [Ous90] and [Ous91]).

Tcl is an interpreted, easily extensible language supporting variables, conditionals, looping constructs and procedures. Most users will never need to use the command language; it is primarily of interest to those wishing to extend XPole's capabilities. For example, most of the standard IIR filter designs are implemented entirely by Tcl code; a new filter design could be implemented in minutes. For more information, see the XPole Programmer's Guide.

Tk is an X11 toolkit based on Tcl. XPole's entire graphical user interface is implemented in Tk. While this user interface is relatively straight forward, an explanation of some parts may be helpful. The explanations below summarize standard operation of `Tcl/Tk` widgets; for more details see the official `Tcl/Tk` documentation. Some of the widgets have been slightly customized for XPole.

2.4.1 Menus

Pulldown menus are activated by pressing the left mouse button over the associated menu's *menubutton*. Normally the menubutton appears as a raised 3-D button. However, XPole may be configured so that some menubuttons will not be raised; instead, they will look like plain label. (This gives the windows a less cluttered and more attractive look.)

Once the menu is visible, it may be used in one of two ways:

- Release the mouse button over the menubutton. Then move the mouse pointer to the desired entry, then press and release the left mouse button. To dismiss the menu without selecting an entry, position the mouse pointer anywhere outside them menu and then press and release the left mouse button.

- Keep the mouse button depressed. Move the mouse pointer to the desired entry, and release the mouse button. To dismiss the menu without selecting an entry, move the mouse pointer outside the menu and release the button.

Menus may be "torn" off. Instead of pressing the left mouse button over the menubutton, press the middle mouse button. The menu will appear. Keeping the button depressed, drag the menu to a suitable location and release the button. The menu will remain in that location: it will not go away after you select an entry. To dismiss the menu once torn, press the left mouse button over the menubutton and dismiss the menu as usual.

The right side of some menus will list *accelerator* keys. Most accelerators are a single key; however, some are a two character sequence. Typing the accelerator key or sequence in the appropriate window will have the same action as selecting the corresponding menu entry. It is not always obvious which windows the accelerators are valid in: experimentation may be required.

2.4.2 Entry Widget

XPole allows entry of numbers and names using Tk `entry` widgets. There are many key bindings for a entry widget, consult a Tk manual for the details. A short summary:

Left Arrow Move cursor left one character.

Right Arrow Move cursor right one character.

Backspace, Ctrl-h, Delete Delete character to the left of cursor.

Ctrl-u Erase everything.

Ctrl-w Erase word to left of cursor.

Ctrl-d Erase all selected characters, if any.

Escape Restore widget to previous value and abort entry. (Not always active).

Return Indicates text entry is done. Perform context-dependent processing. (Not always active).

Many of the entry fields within XPole require you to press "Return" within the entry widget for the change to take affect. If you don't press Return, the change will be ignored and may be over-written later. However, some windows will automatically process the value when the mouse cursor leaves the entry window.

2.4.3 Complex Label

Some of the values displayed by XPole are complex numbers. Most such values are shown with a label to the left of the complex number. The format used to display the number may be controlled using either a menu or key bindings. Pressing the left mouse button over the label brings up a menu with the entries:

Rectangular The complex number is displayed with rectangular coordinates (real and imaginary parts). If the imaginary part is zero, it will sometimes be omitted.

Polar (rad) The complex number is displayed with polar coordinates (magnitude and phase), with the phase in radians. The number will be suffixed by `rad`.

Polar (deg) As above, but the phase is in degrees and the suffix is `deg`.

The accelerator keys shown on the menu are active over both the label and the complex number itself.

2.4.4 Complex Entry

You will occasionally need to provide XPole with a complex number: this is performed using a *Complex Entry* widget. This behaves much like the Complex Label widget (see section 2.4.3 [Tk Complex Label], page 30); in addition, you can also edit the complex value using methods described for the Entry widget (see section 2.4.2 [Tk Entry Widget], page 30).

2.5 Design Algorithms

This chapter describes design algorithms available in XPole. Note that many algorithms that transform views are implemented as pipes, and are described elsewhere (see section 2.2.2 [Pipes], page 19).

2.5.1 Standard Filter Design

XPole currently supports standard IIR filter designs for both continuous-time and discrete-time. To use the filter design algorithms, select '`Standard Design`' from a Pole-Zero Window's main menu (see section 2.3.1 [Polezero Win], page 21). XPole will design a continuous-time or discrete-time filter based on the domain of the pole zero window.

In designing a filter, there are typically three frequencies involved: the passband edge, the stopband edge, and the center frequency. In some cases you may explicitly chooses the passband edge, and in other cases the stopband edge according to what is natural for that filter. In the discussion below, all frequencies are in radians/sec. The '`w`' used in parameter names should be interpreted as omega.

The implementation in XPole is based on [Jac89, chap. 8] Also see [PB87, chap. 7].

Design Type

The following design types are supported:

Butterworth Designs a Butterworth filter. This is smooth everywhere (no ripples).

Chebyschev-I Designs a Chebyschev type 1 (or direct) filter, which has ripples in the passband and is smooth in the stopband The parameter *delta* specifies the passband ripple.

Chebyschev-II Designs a Chebyschev type 2 (or inverse) filter, which is smooth in the passband and has ripples in the stopband. The parameter *delta* specifies the stopband ripple. Also, the *w_pass* parameter actually specifies the stopband edge, not the passband edge.

Elliptical Designs an elliptical filter, which has ripples everywhere, but features the sharpest possible transition region between stop and pass band.

Mapping Type

Discrete time filters are implemented by first calculating an s-domain filter, and then mapping that to a z-domain filter. The mappings available are described below. Note that not all mappings are always meaningful.

Bilinear Uses a bilinear transform (fractional linear transform). This is a robust, algebraic, one-to-one transform: z=(2+Ts)/(2-Ts), where T is the sample period.

Matched Z Uses a "matched z" transform. This is a robust, algebraic transform: z=exp(sT). There is little theoretical rational for this transform, but it sometimes produces good filters.

Impulse Invariant The resulting z-domain filter will have an impulse response which is scaled samples of the s-domain impulse response: h_d(n)=T*h_c(nT). This is an extremely non-robust mapping. The implementation involves a partial fraction expansion and polynomial factoring; both algorithms introduce significant numerical noise. This mapping will often fail, yielding a nonsense filter. In particular, the algorithm sometimes believes that there are more zeros than there really are. This typically results in an extremely small gain and extra zeros. The parameter *n_zeros* can be used to help XPole when the algorithm fails.

Band Type

The band type of the filter may be chosen as listed below. Filters other than low-pass filters are implemented by first calculating a low-pass filter, and then applying a spectral transformation.

low-pass The pass band of the filter will extend from -*w_pass* to +*w_pass* rad/sec.

high-pass The pass band of the filter will range from *w_pass* to infinity (or PI/T). The stop band will be interior to the range from -*w_pass* to +*w_pass* rad/sec.

band-pass The pass band of the filter will extend from *w_ctr-w_pass* to *w_ctr+w_pass* rad/sec.

band-stop The pass band of the filter will range from 0 to *w_ctr-w_pass* and from *w_ctr+w_pass* to infinity (or PI/T). The stop band will be interior to the range from *w_ctr-w_pass* to *w_ctr+w_pass* rad/sec.

Band Domain

When designing a discrete time, bandpass, bandstop or highpass filter, the spectral transformation may be performed in either s-domain or z-domain. This is a complicated and subtle decision: the novice user should select z-domain and ignore this issue. The more advanced user may wish to explore this design choice in order to better understand the impact of the resulting filter. The spectral transformation may occur in:

z-domain A low-pass s-domain filter is designed, mapped into a low-pass z-domain filter, and then spectrally transformed in the z-domain. Any domain mapping may be used since it operates on a low-pass filter. Frequency warping occurs as baseband, which is relatively easy to compensate for (prewarping). However, the z-domain spectral transformation uses an approximation based on fractional linear transforms, and may cause unacceptable frequency warping.

s-domain A low-pass s-domain filter is designed, spectrally transformed in the s-domain, and then mapped to the z-domain. This means that the mapping will be operating on a non-low-pass filter, and thus the impulse invariant mapping will not work. Also, frequency warping induced by the mapping occurs at passband which is much more complicated than at baseband.

Parameters

The following parameters control the design of the filter. Note that not all parameters are always active, and the exact meaning of a parameter depends on context as described above.

N The order of the filter (the number of poles).

w_pass Roughly speaking, this specifies the width of the passband or stopband in rad/sec.

w_tr The width of the transition region (passband edge to stopband edge) in rad/sec.

w_ctr For band-pass and band-stop filters, this is the center frequency of the passband or stopband in rad/sec.

delta Specifies the amount of ripple in passband or stopband. For passband ripple, the frequency response will range from 1.0-*delta* to 1.0. For stopband ripple, the frequency response will range from 0.0 to *delta*.

n_zeros When using an impulse invariant mapping, this specifies the number of zeros in the resulting transfer function. By default this is -1, indicating that XPole should attempt to determine the appropriate number of zeros itself. This parameter allows manual override when XPole fails.

Dialog Window

The standard filter design dialog consists of a set of radio buttons to select designs, sliders to control parameters, and some buttons. The parameters may be changed by both moving the slider using the mouse, or by entering a number (see section 2.4.2 [Tk Entry Widget], page 30).

The following buttons are available:

Ok The filter is designed as currently specified, and the dialog will disappear.

Update The filter is redesigned with the current set of parameters.

Auto Update When enabled, any change to the radio buttons or parameters causes the filter to be redesigned. When disabled, the filter is no modified until '`Ok`' or '`Update`' is pressed.

Show All Windows will appear that display all intermediate filters, and their frequency and impulse response.

Cancel The dialog will disappear without having designed a window.

2.6 Commands and Scripts

This chapter describes XPole's command and script language. This language is Tcl/Tk (see section 2.4 [Tcl/Tk], page 28) with many extensions.

2.6.1 Command Line Options

The following options may be given on the command line when starting XPole.

-s Sets the default filter domain to be continuous-time (i.e., *Laplace*-transform, s-domain).

-z Sets the default filter domain to be discrete-time (i.e., *Z*-transform, z-domain).

-white XPole's graphs will be drawn with a white background and black foreground. Currently, dialog windows' colors will be unaffected.

-black Like the '`-white`' option, except with a black background and a white foreground.

-send Enable XPole to respond to Tk's `send` command. This will allow XPole to interact with other Tk-based tools, but creates a potential security threat.

-command cmd Execute the Tcl command '`cmd`' after completing all other initialization. Any number of pairs may be given.

2.6.2 Variables

This section describes the variables that can be used to control XPole.

xpole(xfdomain) This must be single character, either `"s"` or `"z"`. It indicates the default transform domain for filters that don't have an explicit domain.

2.6.3 Commands

This section describes the XPole commands. They are implemented as commands within a Tcl interpreter. Most of these commands are documented elsewhere as man pages; however, some of the more common commands are described here.

xpole version Returns the version number of the program. This version number is fixed at compile time.

Chapter 3

XPole Programmer's Guide

3.1 Introduction

This guide overviews of the key concepts in XPole and its overall structure, emphasizing knowledge required to modify or extend XPole. In particular, the XPole framework is described. It also discusses the development path of XPole, and explains approaches that were attempted and discarded.

3.2 Framework

The core of XPole is the framework used to describe filters and views, and how they relate. This framework has gone through considerable evolution. First the current framework is described, and then the evolutionary path is discussed. This later discussion is presented both to enumerate approaches that do not work, and to explain artifacts in the current implementation. Finally, ideas for future evolution are presented.

3.2.1 Current Framework

The XPole framework is properly viewed as a dynamic database: this includes both the data itself and the mechanisms used to access and modify the data. The data is organized into *filters* and *views*, while *agents* provide an access mechanism. In addition, *pipes* describe how different views in the database relate to each other. Each of these are described in more detail below.

Filters and Views

A filter is described by an XPFilter structure (below)[1]. A filter is primarily just a collection of views (the Views field) with little other substance.

```
struct _XPFilter {
    XPLabel         Lbl;            /* my name */
    char*           Desc;           /* dynamic, unfmt'd text desc */
    TOPPtrVector    Views;          /* my views */
};
```

Each view is described by an XPView structure. The view is the central object in the framework.

```
struct _XPView {
    XPLabel         Lbl;            /* my name */
    XPFilter*       PntFltr;        /* parent filter */
    XPXformDomain   XfDomain;       /* my domain, S or Z */
    double          SampleTime;     /* sampletime for Z domain */
    char*           Desc;           /* dynamic, unfmt'd text desc */
    XPAgent*        MasterAgent;    /* my master, if slave view */
    DLListHandle    PartList;       /* of XPViewPart */
};
```

Each view contains a set of *view parts* (the PartList field), described by an XPViewPart structure. Conceptually, each view part corresponds to a portion of the entire view; this will be made more precise in the next section.

```
struct _XPViewPart {
    XPView*         PntView;        /* the View which we belong to */
    char*           Name;           /* a name assoc. with part */
    TOPMask         Events;         /* pending events */
    XPViewSnap*     DfltVS;         /* default ViewSnap */
```

[1]The structures shown contain only the essential fields. Note that the discussion of most fields is deferred to later sections where the context is more appropriate.

```
    DLPtrListHandle AgentList;      /* attached agents */
    XPAgent*        RangeOwner;     /* who controls my range */
};
```

Each view part contains a *view snap* (XPViewSnap structure): it is the view snap that contains the actual data (e.g., pole-zero locations or sequence samples). Currently there is only one view snap per view part; however, future versions may allow multiple view snaps. A specific view class (e.g., polezero or fourier) is derived from these structures as described below:

Polezero View This describes a filter $H(s)$ or $H(z)$ by its poles and zeros. It is implemented by an XPSilarView structure (derived from XPView), and an XPSilarSnap structure (derived from XPViewSnap). Each silar snap contains a list of EEMSilar structures, each describing a single pole or zero.

Fourier View This describes a filter $H(s)$ or $H(z)$ by samples of its Fourier transform. It is implemented by an XPCplxSeqView structure (derived from XPView) and an XPCplxSeqSnap (derived from XPViewSnap). The later contains dynamic vectors to store the actual samples. Several vectors are supported; the most important are the Index vector which indicates the frequency of each sample, and the valRI vector which contains the samples themselves.

Impulse View This describes a filter $H(s)$ or $H(z)$ by samples of its impulse response. It is implemented exactly like the Fourier view, except that the Index vector indicates the time of each sample.

Agents

The filter, view, view part, and view snap structures provide a static database; this data must be displayed to and manipulated by the user. Agents (described by XPAgent structures) allow this. An agent is associated with exactly one view: the agent is notified whenever a change occurs to its view; specifically, a function associated with the agent is invoked. For example, each frequency window has an agent for each view that it is displaying. When one of these views changes, the agent is notified and the window draws an updated graph.

Pipes

A pipe (described by XPPipe structure) is used to relate several views together.

Unranged Pipes The simplest class of pipes consumes (reads) one view and produces (writes) another view. The consumed view is often called the *source* and the produced view is the *target*. For example, the `PolezeroMap` pipe reads the pole-zero view of an s-domain filter and produces a pole-zero view for an equivalent z-domain filter. The pipe has *parameters*: one such is the `maptype` that indicates which mathematical method should be used to convert from s-domain to z-domain: bilinear transform, matched-z transform, or impulse-invariant transform.

Another simple example is the `SpectralXform` pipe which reads the pole-zero view of a low pass filter and generates a pole-zero view of the corresponding high pass filter. The pipe has a variety of *parameters* including the cutoff frequency of the low pass filter and the desired cutoff frequency of the high pass filter. The parameters are required in order to define the mapping between the two views.

Within the XPole framework, whenever the source view of a pipe is modified, the target view is recalculated and updated based on the modified source view. The mechanism used to implement this is described later.

Note that some pipes implement invertible transformations and thus may be bi-directional; for example, the bilinear `PolezeroMap` is such a pipe. In this case, both the s-domain and z-domain views are both sources and targets: whenever either view is modified, the other view must be updated. Of course, care must be taken to prevent infinite recursion.

Ranged Pipes More complicated pipes map between different types of views; for example, the `PolezeroFourier` pipe consumes a pole-zero view and produces a Fourier view. The pole-zero view provides $H(s)$ or $H(z)$, and thus samples of the filter's Fourier transform may be computed by direct evaluation with $s = jw$ or $z = e^{jwT}$. However, the complete Fourier transform is an infinite amount of data and thus cannot be calculated nor stored. Instead, the pipe can only generate a finite number of samples of the Fourier transform. Thus the pipe must be told which samples to calculate.

Furthermore, there might be more than one consumer of the Fourier view: for example, a frequency window to display the data and an inverse FFT algorithm[2]. The frequency window requires samples at pixel-resolution over the range where the user is current zoomed, and the FFT algorithm requires equally spaced samples from 0 Hz to $1/T$ Hz. Thus each consumer may require a different set of samples, and the pipe must calculate each set of samples. This is the role of the view part. Each view part is owned by a consumer: the

[2]Note that an inverse FFT algorithm will generally be used as part of a larger filter synthesis process and not to display samples of the time-domain signal. For graphical display, an inverse DTFT will typically be used to allow for zooming and easy alignment to pixel resolution.

view part owner stores into the view part a specification of which samples are required, and the pipe calculates these samples and stores them into the view part. The pipe must do this for each view part associated with the target view.

Most Fourier and impulse views allow the required set of samples (in general, the *range*) to be specified. Such views allow multiple view parts and are considered *ranged*. In contrast, the pole-zero views described above can be completely expressed by a small amount of data: such a view requires only one view part (called the *primary part*) and is considered *unranged*. However, not all Fourier and impulse views are ranged; for example, an impulse response drawn by the user or a frequency response calculated by a fixed-order FFT will be unranged.

This leads to the following classification of views:

Master A master view is not the target of any pipe. Thus it is modified only due to explicit action by the user: it is not automatically updated from any other view. Such a view will have its `MasterAgent` field set `NULL`. Also, a master view is always unranged, and thus can have only one view part (its primary part).

Slave A slave view is the target of exactly one pipe (a view may not be the target of more than one pipe). The `MasterAgent` field references the pipe for which the view is a target. A slave view may be ranged or unranged.

Note that slave views may be directly modified by the user through an agent even though they are the target of a pipe. Typically any such user changes will be lost (over-written) next time the pipe updates the view. However, allowing the user to make changes is still useful. Also, if the pipe is bi-directional, the changes will be reflected elsewhere and won't be lost.

The filter 'data' of a view is stored in the view part (actually, in the view snap associated with the view part). Thus an agent is actually associated with a view part, not a view. In general, there may be multiple agents per view part. In the case of slave views, this means all associated agents are interested in the same range.

Cascading Pipes The true power of the framework is in cascading several pipes together: a pipe may produce a view that is consumed by another view which produces yet another view, and so on. Such a system forms a bipartite graph. Shown in Figure 3.1 is a complicated example of a such a graph. Views are shown as circles while pipes and agents are diamonds.

The 'origin' of the graph is the master view `polezero-lp`. This view is manipulated directly by the user via a pole-zero window. A `SpectalXform` pipe consumes this view and produces

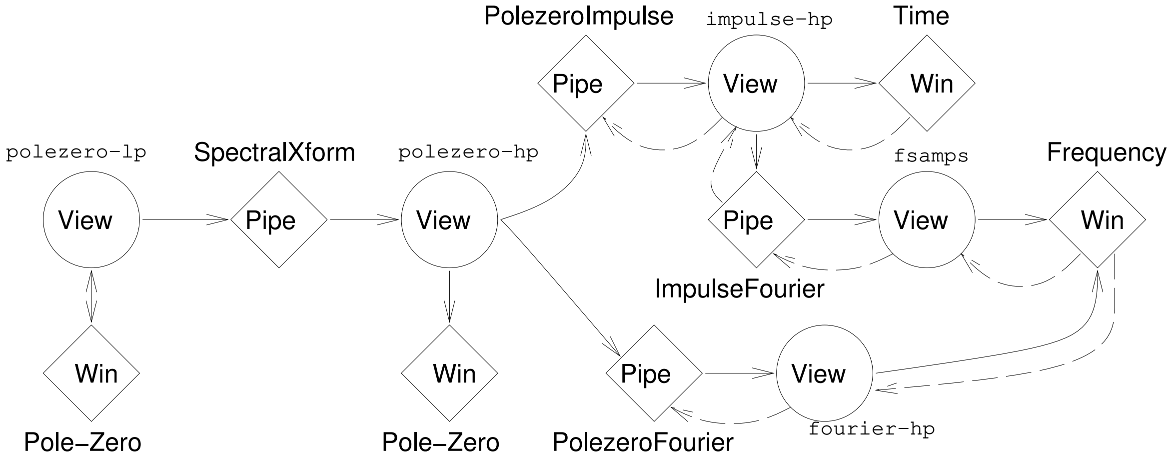

Figure 3.1: View-Agent Graph

a high-pass slave view, `polezero-hp`. This view is graphically displayed by another pole-zero window. The view is consumed by two pipes: the `PolezeroFourier` pipe calculates samples of the Fourier transform (view `fourier-hp`) while the `PolezeroImpulse` pipe calculates samples of the impulse response (view `impulse-hp`). These two views are displayed graphically in windows. The dashed lines going backwards indicate range information: note that pole-zero views do not require any range information. Finally, the `ImpulseFourier` pipe uses samples of the impulse response to compute samples of the Fourier transform (view `fsamps`), which is displayed in the same frequency window. Note that the two frequency views (`fourier-hp` and `fsamps`) will generally not be the same since the later uses a windowing function.

Observe that the view-agent graph may be rather complicated. It is in general a graph, not a tree. Some pipes are bi-directional, and thus the graph may have cycles that must be dynamically broken. Also, while not currently implemented, the framework design allows for pipes that interact with more than two views; for example, a minimum-phase/all-pass filter decomposition pipe would consume one view and produce two (and would be bi-directional).

Implementation Pipes are implemented using an agent for each view associated with the pipe. For example, the `PolezeroMap` pipe has two agents: one for its source view and one for its target view. Whenever the source view is modified the source agent (consumer agent) is notified. The pipe can then recalculate the target view and inform the framework that it has been modified using the target agents.

Updates

One of the key tasks of the framework is to notify interested agents whenever a view is changed. Due to the need to support unranged pipes efficiently, the mechanism used to perform updates is somewhat more complicated than implied above. The "update process" is broken into *notify events* and *update actions*. A notify event simply notifies an agent (or pipe) of a change, and recalculating a view is an update action. There are three types of notify events:

`XPVE_Param` A parameter on a pipe has changed. This requires recalculating all targets of the pipe. This event is issued to all source agents associated with the pipe.

`XPVE_Modify` A view has been modified in some way. For some views, the details as to what specifically changed is recorded within the view itself. This more detail information may be used to perform partial updates (e.g., if only one pole or zero has moved, then the pole-zero window need only redraw that one pole, not all of them).

`XPVE_Switch` A view has been significantly modified, and no detailed information as to what specifically changed is recorded within the view. This event is used instead of `XPVE_Modify` whenever the view has been sufficiently changed so that there would be no savings by using a partial recalculation.

An event is issued using the `xpViewNotify()` function (to notify every agent associated with a view), or the `xpAgentNotify()` function (to notify a specific agent). The event may be tagged as `XPNW_Now` or `XPNW_Idle`. In the later case, the event will not be dispatched until the program is idle. This allows multiple notify events to be collapsed into a single event.

When a pipe receives a notify event from one of its source agents, it issues a notify event as appropriate for all of its target agents; however, the target views are not recalculated at this time. In some cases the pipe will estimate some parameters of the target view; for example, the `PolezeroImpulse` will estimate the length of the "interesting" portion of the impulse response. Note that it can be efficiently estimated without computing the impulse response itself. Such estimates are stored into the target view (not the view parts).

Thus the notify events ripple "down" the view-agent graph through pipes until they reach a terminal agent (typically an agent associated with a window). That agent uses the estimated information (above) to update the range information in the view part that it owns. It will then call the `xpAgentGetSnap()` function. For master views no calculation is required; for slave views this function calls the `MasterAgent`'s `UpdateFunc` function (the master agent is part of a pipe). This function performs the update action, typically recalculating the view using the range information previously stored in the view part. Once this is complete, the terminal agent has an updated view, and can process the view (typically displaying the data graphically).

In the case of cascaded views, the notify events will ripple down as before. When the master agent's update function is called, it must invoke the update on its source agents using the `xpAgentGetSnap()` function. This occurs recursively, moving "up" the graph, until master views are encountered. In this fashion notify events ripple "down" the graph via `xpViewNotify()`, and update requests ripple "up" the graph via `xpAgentGetSnap()`.

3.2.2 Evolution

The current XPole framework has evolved from a very simplistic system. As discussed above the framework really consists of two components: the data structures used to represent views, and the mechanisms used to interrelate those views. As described below, the initial versions did not distinguish between these two components; as the system evolved the views (data) became strongly isolated from the update calculations (code). This is perhaps the most important characteristic of the current system.

Agents

The first version of the framework supported filter structures that had single, fixed entries for the singularity view and Fourier view. (Impulse views were not supported). There was no support for trajectories nor view parts: only one singularity view and one Fourier view per filter, and only one view part per view. The dependency relationship between the two views was hard-coded, and thus required no agents nor pipes.

While this was adequate for the relatively simple task of allowing the user to manipulate the singularity view and to display the Fourier view, more sophisticated tasks were impossible (e.g., maintaining low-pass and high-pass forms of the same filter, or exporting a filter realization to another application). Thus in the next version, each filter supported a list of arbitrary views (at the time, views were called *data modules*). All views were dependent on the singularity view, and needed to be updated whenever it changed. This update was invoked through callback function pointer in each view.

This allowed any number of views to be derived from the primary singularity view, but it did not allow for chaining of views (e.g., low-pass singularity to high-pass singularity to fourier). To allow for chaining, each source view maintained a dependency list of target views that required updating whenever the source view changed. This allowed for arbitrary acyclic digraphs of views.

At this point the view itself and the mechanism used to update it were identical. This did not allow for updating a view in an arbitrary way. Eliminating this limitation led to agents and pipes much as they are used currently: an agent registers with a view to be notified whenever there is a change. A pipe uses an agent to know when its source changes, and updates its target view accordingly. This provides a desirable isolation between views and pipes: any view may notify any agent of a change, and any agent may operate on any view of the appropriate class.

View Hierarchy

As described above, early versions of the framework supported multiple views within each filter. This evolved in two directions: support of time-series through *trajectories*, and support for multiple view parts through view parts.

View Parts As discussed before, slave views may have multiple parts. The need for slave views was not originally envisioned. Originally there was effectively only one view part per view. Problems were discovered whenever two agents attempted to control the range parameters of a slave view (e.g., when two windows are attempting to display different portions of the same Fourier view). One of the two agents would 'win' and the view would have the winning agent's parameters. The solution to this problem was to create multiple view parts within each view, and allow each agent to control the range parameters its own view part.

Trajectory Trajectories are a failed concept that has been completely removed from the current version.

A trajectory is a set of views that change together over some index set, typically time. For example, a trajectory might consists of the impulse response (taps) of an adaptive filter as it converges over time. The trajectory might also include the corresponding singularity view and Fourier view for each set of taps. Another example would be a trajectory of root-locus pole-zero locations for various loop-gain values. The trajectory might also include Fourier and impulse views for each loop-gain value.

Each filter maintains a set of trajectories, and each trajectory consists of a set of *traj snaps*, one for each index value. Each traj snap contains a set of views. The goal of the trajectory concept is to extract maintenance and manipulation of the index value into a higher level so that each individual view does not need to deal with the trajectory index. This also allows a set of windows displaying views of the same trajectory to advance along the trajectory index together by simply changing the 'current' index of the trajectory.

While the purpose of trajectories is desirable, the concept and implementation failed for a variety of reasons:

- Considering the first example above, it would be appropriate to compute and store the singularity view corresponding to each set of taps since it is static and a function of only the taps. In contrast, it is not sensible to store the Fourier view for each set of taps since it depends on what portion(s) the user is currently viewing.
- Most algorithms for computing a root-locus calculate different branches at different loop-gain values. Thus a single trajectory index value cannot adequately describe an entire singularity view. That is, the root locus must be described as a set of branches with the loop-gain ranging within each branch, and not as a set of singularity views with each singularity view describing a single loop-gain value.
- Most modes of use for XPole do not involve time-series. Thus the large effort required to maintain the sophisticated trajectory system was not worth the gain.

Thus while support for time-series is useful and appropriate within XPole, it must be implemented *inside* each view, in a view-specific way, and not outside each view. As a result, most of the `C`- and `Tcl`- code support for trajectories has been removed.

3.2.3 Future Directions

The framework is not yet complete. Some deficiencies and potential solutions are discussed below. More minor improvements and technically complicated details are discussed in the code itself or in the '`ToDo`' file.

Currently the sample time `T` and the transform domain (`s` or `z`) is defined by the filter and not the view. Both of these should be defined by the `view` instead, with the `filter` only providing the initial, default values. Much of the `C` code already supports this, but the `Tcl` commands require modification.

The framework needs to support time-sequences of views, but using an approach other than trajectories. Watching the evolution of an adaptive filter, or calculating a root-locus would be useful applications of time-sequences.

The framework current provides for delayed recalculation, so that multiple updates to the same view may be collapsed into a single update and consequent recalculation of associated views. The model for when and where notify and update delay occurs needs to be examined.

Other items too simple or too complicated to discuss here are listed in the file '`ToDo`' in the XPole source code directory.

3.3 Libraries

The XPole application is organized into a set of libraries. These are described below, followed by a discussion of how these are combined together to form the final application.

The `top` library provides low-level programming support that is common to any large application. This includes memory management, error handling, linked lists, dynamic arrays, porting support, and event loops. It also provides some specialized support for `Tcl/Tk`.

The `eem` library implements mathematical and signal processing algorithms. This includes complex-valued arithmetic, polynomial and rational polynomial arithmetic, partial fraction expansion, Jacobian elliptical functions, and pole-zero representation support.

The `tkgraph` library provides `Tk`-based graphics widgets. Widgets includes the pole-zero window and sequence window. These widgets are independent of the framework and are intended to be general purpose.[3] The core of the widgets are implemented in `C` code, with much of the overall structure and behavior implemented in `Tcl` code.

The `xpole` library implements the XPole framework. This includes filters, views, view parts, agents, and pipes. The implementation builds upon the types defined by the `eem` library, and pipes use `eem` algorithms in their operation.

These libraries are combined together with associated 'glue' code to yield the XPole application. Specifically, views in the framework must be bound to widgets in the `tkgraph` library. In addition, a user interface is required to allow the user to control the application: this is implemented entirely in `Tcl` code.

[3] Indeed, the `qdm` application uses the sequence widget.

3.4 Event Loop

XPole uses an *event loop*. While common in many large systems and most graphical programming, the event loop concept may be unfamiliar to the casual programmer. The basic concept is that after initialization, the program spins forever in the event loop (e.g., an infinite `for` or `while` loop). Inside the loop, the program looks for events including keyboard or mouse activity. When an event occurs, an *callback* function is invoked. This function will typically performs some computation, updates internal state, and outputs a result (either textually or graphically). The function the returns back to the event loop.

Most XPole programmers will never directly use the event loop nor be aware of its existence: it is only visible to the graphics widgets and the framework core. However, it is important to note that functions must never block (e.g., spin or sleep waiting for user input). Instead, user input must be recorded as state, and computation must not be initiated until all user input is available.

3.5 EEM Library

Most of the algorithms used by XPole are implemented by the `eem` library. These are described below in varying degrees of detail.

3.5.1 Polynomial Arithmetic

The library supports arithmetic on complex polynomial and on complex rational polynomials (a ratio of two polynomials). Common operations such as addition, subtraction, multiplication and differentiation are supported. More sophisticated operations such as binomial expansion are also supported. The implementation is direct and relatively naive. A common problem is determining the correct "order" of the polynomial: some operation may result in a highest order term that is almost, but not quite, exactly zero. A formal solution to this problem would require estimating error bounds for all the operations; such estimates is current beyond the scope of this library. Ideally these functions would be replaced by a commercial quality library such as the Cephes Mathematical Library (see [NET, netlib/cephes]). However, such libraries are either not freely redistributable or not yet complete.

3.5.2 Polynomial Factoring

Several algorithms require factoring a polynomial to find its roots (typically yielding the zeros or poles of a filter). Factoring in XPole is performed by the Jenkins and Traub method (`CPOLY`, ACM Algorithm 419). A Fortran version of this algorithm appears in [JT72] and is available on-line from netlib [NET, netlib/toms/419]. This code was translated to C using `f2c` [NET, netlib/f2c] and slightly modified.

Polynomial factoring is a hard problem without simple solutions. The above algorithm was chosen because of its reputation as a robust and accurate algorithm for high order polynomials. Many other algorithms exist [Mad73] [Moo76]; it would be worthwhile to explore their relative accuracy and speed. In particular, companion matrix based algorithms might work well. Also, the chosen algorithm factors a complex polynomial. For cases where the polynomial is known to be real, there could be significant speed and accuracy gains by using a real polynomial factoring algorithm such as [Jen75].

3.5.3 Partial Fraction Expansion

Several algorithms require expanding a filter described by its poles and zeros into a sum of partial fractions (PFE form). XPole uses a simplistic method based on Heaviside's Expansion Theorem [ZTF83, pp. 186-195]. This method solves for the PFE coefficients by evaluating a modified version of the transfer function $H(s)$ or $H(z)$. For first order poles this works well, but for higher order poles it requires expanding the transfer function into a rational polynomial and differentiating. This leads to large numerical errors.

Other algorithms for performing PFE should be examined, including symbolic differentiation and state-space formulations.

3.5.4 Jacobian Elliptical Functions

The construction of elliptical filters is rather complex and involves the use of Jacobian Elliptical Functions (JEF). The construction process and many properties of JEFs are explained in the source code `eem/jef.c`. The implementation is based on [GR69], [Com79], [AS65, Chap. 16 and 17] and [Jon82], with the last providing an excellent formulation and description of the construction.

Bibliography

[AS65] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions.* Dover Publications, New York, 1965.

[BHLM93] Joe Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, forthcoming special issue on *Simulation Software Development*, 1993.

[BLL93] Jeff Bier, Phil Lapsley, and Edward Lee. *Design Tools and Methodologies for DSP Systems.* Forward Concepts, 1993.

[Com79] ASSP DSP Committee. *Programs For Dig. Sig. Proc.* IEEE Press, 1979.

[GR69] Bernard Gold and Charles M. Rader. *Digital Processing Of Signals.* McGraw-Hill (Lincoln Labs), 1969.

[Jac89] Leland B. Jackson. *Digital Filters and Signal Processing.* Kluwer Academic Publishers, second edition, 1989.

[Jen75] M. A. Jenkins. Algorithm 493: Zeros of a real polynomial. *ACM Transactions on Mathematical Software*, 1(2):178–189, June 1975.

[Jon82] Mark T. Jong. *Methods of Disc. Sig. and Sys. Anal.* McGraw-Hill, 1982.

[JT72] M. A. Jenkins and J. F. Traub. Algorithm 419: Zeros of a complex polynomial. *Communications of the ACM*, 15(2):97–99, February 1972.

[Lap91] Philip D. Lapsley. Host interface and debugging of dataflow dsp systems. Master's thesis, University of California at Berkeley, 1991.

[Mad73] Kaj Madsen. A root-finding algorithm based on newton's method. *BIT*, 13:71–75, 1973.

[MH87] Jerrold E. Marsden and Michael J. Hoffman. *Basic Complex Analysys.* W. H. Freeman and Company, second edition, 1987.

[Moo76] J. B. Moore. A consistently rapid algorithm for solving polynomial equations. *Journal of the Institute of Mathematics and Its Applications (IMA)*, 17:99–110, 1976.

[NET] Anonymous ftp to research.att.com. AT&T.

[Ous90] John K. Ousterhout. Tcl: An embeddable command language. In *1990 Winter USENIX Conference Proceedings*, 1990.

[Ous91] John K. Ousterhout. An X11 toolkit based on the tcl language. In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[PB87] T. W. Parks and C. S. Burrus. *Digital Filter Design.* John Wiley & Sons, Inc., New York, 1987.

[ZTF83] Rodger E. Ziemer, William H. Tranter, and D. Ronald Fannin. *Signals and Systems: Continuous and Discrete.* Macmillan Publishing Co., New York, 1983.