Interactive Laboratory for Digital Signal Processing in iOS Devices

by

Jinru Liu

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2011 by the
Graduate Supervisory Committee:

Andreas Spanias, Chair
Kostas Tsakalis
Gang Qian

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

The demand for handheld portable computing in education, business and research has resulted in advanced mobile devices with powerful processors and large multi-touch screens. Such devices are capable of handling tasks of moderate computational complexity such as word processing, complex Internet transactions, and even human motion analysis. Apples iOS devices, including the iPhone, iPod touch and the latest in the family - the iPad, are among the well-known and widely used mobile devices today. Their advanced multi-touch interface and improved processing power can be exploited for engineering and STEM demonstrations. Moreover, these devices have become a part of everyday student life. Hence, the design of exciting mobile applications and software represents a great opportunity to build student interest and enthusiasm in science and engineering.

This thesis presents the design and implementation of a portable interactive signal processing simulation software on the iOS platform. The iOS-based object-oriented application is called i-JDSP and is based on the award winning Java-DSP concept. It is implemented in Objective-C and C as a native Cocoa Touch application that can be run on any iOS device. i-JDSP offers basic signal processing simulation functions such as Fast Fourier Transform, filtering, spectral analysis on a compact and convenient graphical user interface and provides a very compelling multi-touch programming experience. Built-in modules also demonstrate concepts such as the Pole-Zero Placement. i-JDSP also incorporates sound capture and playback options that can be used in near real-time analysis of speech and audio signals. All simulations can be visually established by forming interactive block diagrams through multi-touch and drag-and-drop. Computations are performed on the mobile device when necessary, making the block diagram execution fast. Furthermore, the extensive support for user interactivity provides scope for improved learning.

The results of i-JDSP assessment among senior undergraduate and first year

graduate students revealed that the software created a significant positive impact and increased the students interest and motivation and in understanding basic DSP concepts.

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my advisor, Professor Andreas Spanias, for his invaluable guidance and support. His patience, enthusiasm and vision helped me achieve more than I ever thought possible.

Next, I would like to express my sincere appreciation to Professors Kostas Tsakalis and Gang Qian for being in my supervisory committee. Their assistance is greatly appreciated.

Many thanks to Professor Antonia Papandreou-Suppappola for her great assistance in evaluating the application in her undergraduate class.

I would also like to thank the School of Arts, Media and Engineering for providing me with the research assistantship that supported me while I completed this work.

My colleagues and friends in the School of Arts, Media and Engineering and the Electrical Engineering department deserve my many thanks for giving me guidance, inspirations and creating a fun and challenging work environment. Special thanks is due Brandon Mechtley, Alex Fink, Gordon Wichern, Shawn Cook, Daniel Roth, Mahesh Banavar, Karthikeyan Natesan Ramamurthy, Jayaraman Jayaraman Thiagarajan Shuang Hu and Xue Zhang.

Finally, I would like to thank my parents Xinghua Liu and Saiying Chu for always being there and helping me get to where I am today.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1

1 INTRODUCTION 1

1.1 State of the Art in Mobile Technology 1

1.2 Motivation 3

1.2.1 Smartphones in Education 3

1.2.2 J-DSP and i-JDSP on the iOS 4

1.3 Challenges and Problems 6

1.4 Overview and Organization 7

2 USER INTERFACE 8

2.1 User Experience Overview 8

2.2 Aesthetic Integrity 9

2.3 Consistency 9

2.4 Direct Manipulation 10

2.4.1 Moving 12

2.4.2 Deleting 12

2.4.3 Editing 13

2.4.4 Establishing Connections 13

2.5 Feedback and Defensive Programming 14

3 SYSTEM ARCHITECTURE 17

3.1 Design Overview 17

3.2 Application Architecture 18

3.2.1 Hybrid Implementation 19

3.2.2 The Model-View-Controller Paradigm 21

Chapter Page

3.2.3 The Delegation Pattern . . . 23

4 SYSTEM IMPLEMENTATION . . . 27

4.1 Overview . . . 27

4.2 Fundamental DSP Functions . . . 28

4.2.1 Signal Generator . . . 29

4.2.2 Fast Fourier Transform (FFT) . . . 31

4.2.3 Filter . . . 32

4.2.4 Filter Coefficients . . . 33

4.2.5 Frequency Response . . . 34

4.2.6 Plot . . . 35

4.2.7 Adder . . . 37

4.2.8 PZ Plot . . . 38

4.2.9 PZ2Coef . . . 39

4.2.10 PZ Placement . . . 41

4.2.11 MIDI . . . 42

4.2.12 DTMF . . . 44

4.2.13 Sound Recorder . . . 45

4.2.14 Sound Player . . . 46

5 EXERCISES, SIMULATIONS AND EVALUATIONS . . . 49

5.1 Simulation Examples . . . 49

5.1.1 FFT Simulation . . . 49

5.1.2 Filter Response Simulation . . . 50

5.1.3 Pole-Zero Simulation . . . 52

5.2 User Experiment and Evaluation . . . 53

5.2.1 User Experiment Design . . . 53

5.2.2 Response Results and Analysis . . . 54

6 CONCLUSION AND FUTURE WORK . . . 57

Chapter Page

6.1 Conclusion 57

6.2 Future Work 57

REFERENCES 58

APPENDIX 61

A ADDING NEW FUNCTION BLOCKS TO I-JDSP 63

A.1 Creating the function class as a model 63

A.2 Generating the associated view controller (if needed) 65

A.3 Telling i-JDSP how the function block should be drawn 70

A.4 Registering the function block in i-JDSP 71

B I-JDSP LABORATORY ON DIGITAL FILTER BASICS 74

B.1 General information on i-JDSP 74

B.2 Getting started - generating and analyzing signals 74

B.3 Generating signals 76

B.4 Performing simulations 79

B.5 Visualizing results graphically 80

B.6 Spectral estimation of MIDI and DTMF signals 82

B.6.1 Working with the MIDI signal generator 82

B.6.2 DTMF generator 84

C LABORATORY EXERCISES USING I-JDSP 87

C.1 Digital filter simulations 87

C.2 The z-transform and frequency response 98

C.3 The discrete and the fast Fourier transform 116

LIST OF TABLES

Table Page

1.1 Latest iOS system hardware configuration. 2

2.1 Gesture recognizers used in i-JDSP. 12

3.1 Performance comparison between C Array and NSArray. 21

5.1 Response results on general usage information. 54

5.2 Response results on execise information. 56

B.1 Table for recording measured MIDI tone periods and frequencies. 84

B.2 Table for recording measured DTMF waveform frequencies. 85

LIST OF FIGURES

Figure Page

1.1 The iOS family in 2010 (from left to right: iPad, iPod touch, iPhone 4, iPhone 3GS). 2

2.1 Main interface for drawing in i-JDSP. 10

2.2 The use of pickerview in the FFT block. 11

2.3 Interface for editing function parameters. 11

2.4 Typical parameter editing view in i-JDSP. 13

2.5 Feedback on Pins when user taps. 15

2.6 The use of ActionSheet in i-JDSP. 16

2.7 The use of AlertView in i-JDSP. 16

3.1 User's Mental Model of i-JDSP. 18

3.2 Model–View–Controller design pattern relationship. 19

3.3 i-JDSP application flow diagram. 20

3.4 Inheritance UML diagram of the models. 22

3.5 Basic block layout. Pin 0, 1: inputs for time domain signals; Pin 4, 5: outputs for time domain signals; Pin 2: input for frequency domain signal; Pin 3: output for frequency domain signal. 23

3.6 A delegation example in tableview. 25

3.7 Nested delegation use in i-JDSP. 26

4.1 UML diagram of the Part class. 29

4.2 UML diagram of the SignalGen class. 30

4.3 The Signal Generator block. 30

4.4 UML diagram of the FFT class. 31

4.5 The FFT block. 32

4.6 The Filter block. 32

4.7 UML diagram of the Filter class. 33

4.8 UML diagram of the FilterCoeff class. 33

Figure Page

4.9 The Filter Coefficients block. 34

4.10 UML diagram of the FreqResp class. 34

4.11 The Frequency Response block. 35

4.12 The actionsheet in the Frequency Response block. 36

4.13 UML diagram of the Plot class. 36

4.14 The plot block. 37

4.15 The adder block. 37

4.16 UML diagram of the Adder class. 37

4.17 The PZ Plot block. 38

4.18 UML diagram of the PZ Plot class. 38

4.19 PZ Plot output when coefficients a0 = 1, a1 = 0.2, b0 = 1. 39

4.20 The PZ2Coef block. 39

4.21 UML diagram of the PZ2Coef class. 40

4.22 Interface for placing the poles and zeros in the PZ2Coef block. 40

4.23 The PZ Placement block. 41

4.24 UML diagram of the PoleZero class. 41

4.25 Interface of the PZ Placement block. 42

4.26 The MIDI block. 42

4.27 UML diagram of the MIDI class. 43

4.28 Interface of the MIDI block. 43

4.29 The DTMF block. 44

4.30 UML diagram of the DTMF class. 44

4.31 Interface of the DTMF block. 45

4.32 The Sound Recorder block. 45

4.33 UML diagram of the SoundRec class. 46

4.34 Interface of the Sound Recorder block. 47

4.35 The Sound Player block. 47

Figure Page

4.36 UML diagram of the SoundPlayer class. 48

5.1 i-JDSP simulation for calculating the FFT of a rectangular signal. 49

5.2 Output of the FFT demo when FFT size = 256. 50

5.3 Output of the FFT demo when FFT size = 64. 50

5.4 i-JDSP simulation for filtering and frequency response. 51

5.5 Filtering of a rectangular signal in i-JDSP with filter coefficients: a = [1, 0.9], b = [1, 3]. 51

5.6 i-JDSP simulation for pole-zero plot and frequency response. 52

5.7 Simulation results of a tenth-order system. Lower left: frequency response. Lower right: pole-zero plot. 53

B.1 i-JDSP simulation environment. In the figure, 'A': Add button (to add function blocks); 'B': Trash button (to delete all the blocks); 'C': Simulation diagram; 'D': Connection pin. 75

B.2 Lab-1 working with i-JDSP. 76

B.3 The signal generator (*Sig.Gen.*) dialog window in i-JDSP. 77

B.4 The editing dialog window in i-JDSP. 77

B.5 The picker menu for selecting different signal types in i-JDSP. 78

B.6 The coefficient panel. 80

B.7 A i-JDSP simulation diagram. 81

B.8 Sinusoid plus noise simulation. 82

B.9 The MIDI simulation diagram. 83

B.10 The DTMF simulation diagram. 84

C.1 Digital filter simulation diagram. 87

C.2 Exercise 1 part a impulse response. 87

C.3 Exercise 1 part b unit step response. 88

C.4 Exercise 1 part c(1) frequency response (magnitude). 88

C.5 Exercise 1 part c(2) frequency response (phase). 88

Figure Page

C.6 Exercise 1 part d(1) frequency response (magnitude). 89
C.7 Exercise 1 part d(2) frequency response (phase). 89
C.8 Exercise 1 part e unit step response. 89
C.9 Exercise 2 part a impulse response. 90
C.10 Exercise 2 part b(1) frequency response (magnitude). 90
C.11 Exercise 2 part b(2) frequency response (phase). 90
C.12 Exercise 2 part c(1) frequency response (magnitude). 91
C.13 Exercise 2 part c(2) frequency response (phase). 91
C.14 Exercise 3 part a impulse response. 92
C.15 Exercise 3 part b frequency response (magnitude). 92
C.16 Exercise 3 part b frequency response (phase). 93
C.17 Exercise 3 part c frequency response (magnitude). 93
C.18 Exercise 3 part c frequency response (phase). 93
C.19 Exercise 3 part d impulse response. 94
C.20 Exercise 4 part a impulse response. 94
C.21 Exercise 5 part a frequency response (magnitude). 95
C.22 Exercise 5 part a frequency response (phase). 95
C.23 Exercise 5 part b simulation diagram. 95
C.24 Exercise 5 part b output of the filter. 96
C.25 Exercise 6 part a frequency response (magnitude). 96
C.26 Exercise 6 part a frequency response (phase). 96
C.27 Exercise 6 part b simulation diagram. 97
C.28 Exercise 6 part b output of the filter. 97
C.29 Exercise 1 simulation diagram. 98
C.30 Exercise 1 part b impulse response. 98
C.31 Exercise 1 part c frequency response (magnitude). 99
C.32 Exercise 1 part c frequency response (phase). 99

Figure Page

C.33 Exercise 1 part c poles and zeros. 99

C.34 Exercise 1 part e poles and zeros. 100

C.35 Exercise 2 part b frequency response (magnitude). 100

C.36 Exercise 2 part b frequency response (phase). 100

C.37 Exercise 2 part b poles and zeros. 101

C.38 Exercise 3 part a frequency response (magnitude). 101

C.39 Exercise 3 part a frequency response (phase). 101

C.40 Exercise 3 part a poles and zeros. 102

C.41 Exercise 3 part b output of the system when $\Omega_1 = \pi/2$. 102

C.42 Exercise 4 part a phase response. 102

C.43 Exercise 4 part b poles and zeros. 103

C.44 Exercise 5 part b impulse response of $H_1(z)$. 104

C.45 Exercise 5 part b frequency response (magnitude) of $H_1(z)$. 104

C.46 Exercise 5 part b frequency response (phase) of $H_1(z)$. 105

C.47 Exercise 5 part b impulse response of $H_2(z)$. 105

C.48 Exercise 5 part b frequency response (magnitude) of $H_2(z)$. 105

C.49 Exercise 5 part b frequency response (phase) of $H_2(z)$. 106

C.50 Exercise 6 part a simulation diagram. 106

C.51 Exercise 6 part a output of the system. 106

C.52 Exercise 6 part b simulation diagram. 107

C.53 Exercise 6 part b output of the system. 107

C.54 Exercise 6 part c simulation diagram. 107

C.55 Exercise 6 part c output of the system. 108

C.56 Exercise 6 part d simulation diagram. 108

C.57 Exercise 7 simulation diagram. 109

C.58 Exercise 7 part a frequency response (magnitude) of $H_1(z)$. 109

C.59 Exercise 7 part a frequency response (phase) of $H_1(z)$. 110

Figure Page

C.60 Exercise 7 part a poles and zeros of $H_1(z)$. 110

C.61 Exercise 7 part b frequency response (magnitude) of $H_2(z)$. 110

C.62 Exercise 7 part b frequency response (phase) of $H_2(z)$. 111

C.63 Exercise 7 part b poles and zeros of $H_2(z)$. 111

C.64 Exercise 7 part c frequency response (magnitude) of $H_3(z)$. 111

C.65 Exercise 7 part c frequency response (phase) of $H_3(z)$. 112

C.66 Exercise 7 part c poles and zeros of $H_3(z)$. 112

C.67 Exercise 8 part b impulse response when $r = 0.96$. 112

C.68 Exercise 8 part c frequency response (magnitude) when $r = 0.96$. 113

C.69 Exercise 8 part c frequency response (phase) when $r = 0.96$. 113

C.70 Exercise 9 part a when $\Omega_c = \pi/3$. 113

C.71 Exercise 10 simulation diagram. 114

C.72 Exercise 10 part a poles and zeros. 114

C.73 Exercise 10 part b frequency response (magnitude). 114

C.74 Exercise 10 part b frequency response (phase). 115

C.75 Exercise 1 simulation diagram. 116

C.76 Exercise 1 part a FFT output of $x_1(n)$. 116

C.77 Exercise 2 simulation diagram. 117

C.78 Exercise 2 part a FFT output. 117

C.79 Exercise 2 part b FFT output. 117

C.80 Exercise 2 part c FFT output when $freq = \pi/11$. 118

C.81 Exercise 2 part c FFT output when $freq = \pi/11$. 118

C.82 Exercise 3 simulation diagram. 118

C.83 Exercise 3 part a FFT output using a rectangular window. 119

C.84 Exercise 3 part b FFT output using a Hamming window. 119

Chapter 1

INTRODUCTION

1.1 State of the Art in Mobile Technology

The rapid development in mobile technology has had a profound impact in personal computing. Growth in demand for advanced mobile devices has resulted in powerful processors, abundant memory, large multi-touch screens [1] and more. Although small in size, these handheld devices can be considered as personal pocket computers (PPC) that are capable of performing advanced applications such as word processing, complex Internet transactions, and even human motion analysis [2] on specific platforms. At the time of writing, there are five major platforms or Operating Systems (OS) on the market – iOS, Android, Windows Mobile, Symbian, and BlackBerry. According to a recent survey conducted in October 2010 by the Nielsen Company [3], Apple's iOS and the Android OS were tied for "most desired" operating systems on mobile platforms. The survey also showed that of those in the age group of 18-24, 35.9% chose iOS to be the next desired operating system, while 32.0% showed interest in Android.

Apple Inc. introduced its first iPhone in 2007, featuring the world's first touch-screen and novel multi-touch experience enabled by iOS. iOS, whose latest version is iOS4 (formerly known as iPhone OS), is developed by Apple Inc. as its mobile operating system [4]. It is derived from Mac OS X (the operating system for Macintosh computers), and is a Unix-like operating system by nature. Apple later expanded the mobile product line by introducing iPod touch and iPad to the iOS family (Figure 1.1), in 2007 and 2010 respectively. All these devices provide a similar multi-touch user experience and run the same operating system. Some differences in core system configurations of the latest models are shown in Table 1.1. But despite some of the differences in size and storage, these devices are equipped with the same powerful processor – the Apple A4. The Apple A4 chip is based on the ARM processor architecture and contains an ARM Cortex-A8 CPU core paired with a PowerVR SGX 535 graphics

processor with an emphasis on power efficiency.



Figure 1.1: The iOS family in 2010 (from left to right: iPad, iPod touch, iPhone 4, iPhone 3GS).

Table 1.1: Latest iOS system hardware configuration.

	iPad	iPhone	iPod touch
Display	9.7 in(diagonal)	3.5 in	3.5 in
Processor	1 GHz Apple A4	1 GHz Apple A4	1 GHz Apple A4
Memory	256 MB DRAM	512 MB DRAM	256 MB DRAM
Storage	16, 32 and 64 GB	16 and 32 GB	8, 32 and 64 GB
Dimensions	9.56×7.47×0.5 in	4.5×2.31×0.37 in	4.4×2.3×0.28 in

The iOS platform also offers a powerful software development kit (SDK) [5] for developers and a convenient on-line App Store [6] that makes the distribution of the applications easier. The iOS SDK was initially released in February 2008 to develop native applications for iOS. It consists of four major software frameworks, Cocoa Touch, Media, Core Services, and OS X Kernel, written in Objective-C and C. It also contains the iPhone Simulator, a program that can be used to simulate the iPhone/iPad on their computers. As of October 20, 2010, there are over 300,000 third-party applications officially available on the App Store, with over 7 billion total downloads according to the 2010 Apple Special Event [7].

1.2 Motivation

1.2.1 Smartphones in Education

Currently available mobile devices go against the traditional notion that educational applications cannot be delivered efficiently using the mobile platform. They have brought in a paradigm shift in the use of technology in an educational setting.

In 2004, researchers Paul Anderson and Adam Blackwood reviewed the technical situation and trends with regard to mobile devices in their JISC Techwatch report [8]. The report showed positive future development in education with a significant number of opportunities to enhance learning, administration and research. It also presented new challenges for the management and support of university infrastructure. Researchers Andreas Holzinger and his colleagues expanded the study further by experimenting with the Mobile Interactive Learning Objects (MILOs) which are used within a Mobile Learning Engine (MLE) runing on mobile phones [9]. Moreover, smartphones were also used in a recent study to facilitate communication and improve social skills of children with severe autism spectrum disorder [10].

Due to the aforementioned advanced hardware configurations, powerful SDK, convenient on-line App Store, and popularity among young people, the iOS family has become an ideal platform for engineering and STEM education demonstrations.

There are already dozens of fun and successful educational applications available, especially for the elementary age children. Popular categories are children's books, language learning, elementary mathematics, and science. Two examples of the top downloaded applications are AlgebraPrep [11], a mathematics app that allows users to practice tests, get answers, and even watch instructional videos, and another app – The Elements [12], a visual 3-D exploration of the elements that builds the universe. These novel applications combine the power of multimedia and multi-touch screen to address long standing concerns with young children: 1) short average learning session and 2) passive learning.

For high school students or above, however, the choice of these kind of educational applications are limited, especially in the field of engineering. MATLAB is a numerical computing environment that is widely used across academia and industry in the field of engineering, science, and economics. Its iOS version – MATLAB Mobile [13], only recently became available in the App Store. It features a traditional command line interface and allows users to type commands, execute scripts and view the corresponding results. However, this application requires a running instance of MATLAB on a remote machine and does not support sophisticated User Interfaces (UI). More importantly, this implies that the application cannot be used without Internet connectivity. Furthermore, rendering of the MATLAB figures relies heavily on the Internet connection, since figures are not generated on the mobile device but simply transferred from the MATLAB instance running on the remote machine.

1.2.2 J-DSP and i-JDSP on the iOS

The field of Digital Signal Processing (DSP) has occupied researchers and students because of its enabling nature, modern applications and the need for sophisticated digital implementations [14]. DSP algorithms have been used in the processing of a wide range of signals such as speech, audio, image, video, radar, sonar, biomedical and seismic data. In undergraduate engineering education signal processing courses involve complex mathematical concepts and in some cases students have difficulty making the connections to real-world problems and applications. Although several software components and web courses have been designed for education, they are expensive and often lack interactive capabilities.

To address these problems, a web-based, platform-independent simulation environment, Java Digital Signal Processing (J-DSP), has been developed at Arizona State University (ASU), and is being successfully used for a virtual laboratory in both high school and college level settings [15]. J-DSP was built from the ground up in Java to provide free and universal access to an array of functions that can be used for re-

search and education [16]. Several interactive laboratories have been developed and successfully used in undergraduate courses [17, 18]. In addition, several toolboxes have been developed for image processing [19], control systems [20], genomics [21], time-frequency analysis [22], analog/digital communications [23], earth system signal processing [24] and RF amplifier linearization [25]. Furthermore, hardware interfaces to TI DSK [26] and Crossbow sensor motes [27] have also been developed. The interfaces to MATLAB [28] and LabVIEW [29] allow synergies with other simulation environments.

J-DSP is implemented as a Java applet [30] that can be run in any browser with Internet connection and thus is platform independent. It offers highly intuitive graphical user interface (GUI) that all its functions are represented by graphical blocks and can be manipulated by mouse inputs. However, some off-line students and educators have been left out because of the lack of the internet connection. Therefore, the need of a portable interactive signal processing laboratory that does not require the internet while offers the similar user experience is very real.

In this paper, I present the design and implementation of an interactive signal processing simulation environment that offers an intuitive, all-graphical programming experience on the iOS platform. The proposed application executes functions on the iOS device directly and hence does not require Internet connectivity. Users have unlimited access to the software. Furthermore, the extensive support for user interactivity provides scope for improved learning. This iOS-based object-oriented application is called i-JDSP and is based on the J-DSP software. The current version of i-JDSP offers functions for FFT, filtering and spectral analysis through an easy-to-use GUI and provides a very compelling multi-touch programming experience. Built-in modules also demonstrate concepts such as MIDI, DTMF and sound capture and playback. All simulations can be visually established by forming interactive block diagrams through multi-touch and drag-and-drop procedures. It is important to note that the computa-

tions are performed on the mobile device when required, making the block diagram execution fast.

1.3 Challenges and Problems

Moving one application from one platform to another means more than just rewrite the code in another language. Serious consideration must be taken to ensure the look and feel of the original one. In addition, sometimes alternative measurements are necessary to be able to adapt to the new platform. As mentioned before, even though J-DSP is cross-platform, it still requires an internet browser and the Java Virtual Machine (JVM) [31] to run. Due to the limitations in both hardware and software of the iOS devices, we need to address the following problems when designing and implementing the system:

- **Choice of Language:** Since Java Applet is not supported by the iOS, we need to use C-like languages. In current SDK documentation, Objective-C, C, and C++ codes are allowed to be run on the iOS.

- **Performance:** Although iOS devices are equipped with advanced processors, they are still mobile devices and cannot be treated like today's powerful desktops and laptops. In our case, all signal processing algorithms require real-time processing and some of them are relatively complex. Thus, code optimization is a major concern in the implementation.

- **User Interface:** As mentioned earlier, the multi-touch screen is one of the most exciting features of the iOS platform. Instead of the traditional mouse input used in the J-DSP, the user's fingers become the main agent between the user and the application, which creates many possibilities when designing the user interface. Moreover, the size limitation of the screen, especially on the iPhone and iPod touch, requires a complete new user interface design while still preserves the intuitive feel of the current on-line laboratory.

1.4 Overview and Organization

In this chapter, I discussed the latest development in mobile devices that changes the way people think of mobile computing, with a special focus on the Apple iOS family. I also reviewed some of the recent trends and possibilities by employing smartphones in education to enhance learning, administration and research. Then I shifted our focus to the realm of DSP and came to the conclusion that the need of a portable DSP laboratory on the iOS platform is very real.

To address this need, I proposed a software solution on the iOS platform called i-JDSP. i-JDSP is developed based off the previous J-DSP on-line laboratory that provides similar graphical programming experience. I have also outlined the challenges and problems when porting to the iOS platform due to the nature of differences in both hardware and software.

In Chapter 2, I will describe the process of designing the User Interface (UI) in i-JDSP that ultimately defines the overall programming experience over a multi-touch mobile device. Chapter 3 presents the system architecture involving the choice of language, class hierarchy and several design paradigms. Chapter 4 fully details the process of implementing each DSP function in the current i-JDSP version. In Chapter 5, I will demonstrate and evaluate some exercises of simulation using i-JDSP. Finally, Chapter 6 concludes by summarizing the work done and presents directions for future development and improvements. Appendix A includes a set of instructions for future developers to add new functionality to i-JDSP. In Appendix B, introductory exercise on the iPhone/iPad of using i-JDSP is provided. Appendix C provides the solutions to some of the exercises taken from the *Digital Signal Processing An Interactive Approach* book using i-JDSP.

Chapter 2

USER INTERFACE

2.1 User Experience Overview

In addition to supporting advanced technical configurations, modern day handheld devices provide sophisticated Human Computer Interaction (HCI) interfaces, with the aid of multitouch screens. A good user interface design should comply with the basic HCI principles, which are typically based on the way people usually think and work. Lack of intuitive interfaces often results in applications that are convoluted, illogical and unattractive to use. However, an elegant intuitive user interface will immensely improve the usability of any application and enhance user experience.

The fundamental idea of User Experience (UX) considers several aspects regarding how a person feels about using a system. Introduced by Donald Norman in mid 1990s [32], UX emphasizes the experiential, effective, meaningful and valuable aspects of HCI. It also addresses some of the practical aspects such as utility and efficiency of a system.

Many factors can influence user experience with a system, such as the system properties, users state and context [33]. In our scenario, we intend to design a simulation framework to perform laboratory exercises in an undergraduate signal processing course. Hence, our goal is to build an all-graphical interactive signal processing laboratory environment for mobile devices. The most important challenge is to make the system as intuitive and friendly as possible. Because of the limitation of the screen size in iOS devices, regular desktop view hierarchy becomes impossible. Furthermore, while multiple view controllers will serve well in the purpose of presenting multiple views to the users, we need to design a more efficient way to navigate through the different views. Fortunately, the iOS SDK provides us with UINavigationController, a convenient class to manage a stack of UIView controllers.

A navigation controller object manages the currently displayed screens using

the navigation stack. The root view controller is placed at the bottom of this stack, which in our case is the main canvas for placing the function blocks. At the top of the navigation stack is the view controller currently being displayed. Users can access any view from the view hierarchy in the same way as a typical stack. Pushing a new view controller object onto the stack causes its corresponding view to be displayed on the screen. Users can also pop out a view controller, which results in the new view controller from the stack to be displayed on the current screen. However, this is usually performed by pushing the "Back" button on the navigation bar.

2.2 Aesthetic Integrity

The on-line signal processing laboratory, J-DSP, already provides the users with intuitive drag-and-drop and all graphical programming experience. The underlying architecture of J-DSP has been adapted for use in iOS devices, by exploiting the multi-touch capabilities. When the user first opens the i-JDSP app, a simple and instructive interface is displayed, that illustrates the layout of functions in i-JDSP and shows a set of instructions for using the program , as shown in Figure 2.1. The set of instructions presented to the user includes: a) Tapping the + button adds a new function block to the drawing space;b) Clicking the Trash button clears the drawing space; c) Press and long hold deletes a single block; d) Changing parameters of any function or viewing results can be performed by double tapping on functions and e) Output and input pins need to be tapped to establish connections. In addition, the overall color theme is consistent with the one used in J-DSP to preserve the aesthetic integrity, which also provides a familiar environment for the users to work with.

2.3 Consistency

Consistency of the interface design allows the users to transfer their experience and skills from one application to another. This implies that in addition to being compatible with the iOS, we have retained the look and feel of the online version of the J-DSP software as much as possible. Making the application consistent allows us to include the

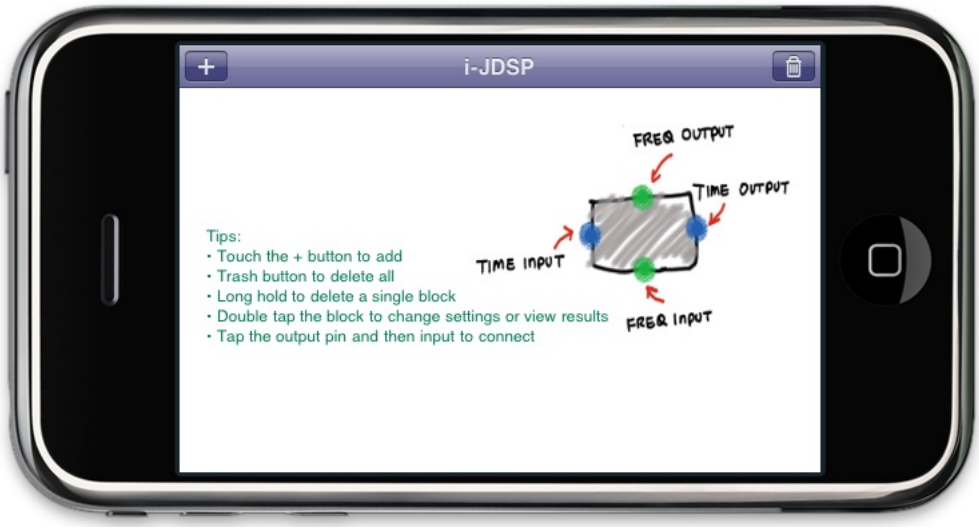


Figure 2.1: Main interface for drawing in i-JDSP.

well understood standards and paradigms of J-DSP that users are already comfortable with.

Apple iOS SDK supports a wide range of convenient user interface controls, views such as tableviews for listing items and pickerviews for selecting items and icons. For example, to select a possible FFT size, a pickerview is presented to the user, as shown in Figure 2.2. Since users are already familiar with this kind of interaction from their previous experience with other applications, it is highly likely that they will find it very easy to adapt to this new environment.

Another example is illustrated in Figure 2.3, where users can specify the Time Shift by typing the desired value by editing the parameter of the function block parameter using the built-in numeric keypad.

2.4 Direct Manipulation

The most important merit of a multi-touch screen is that it provides the user with a personal experience by employing the finger as an input device instead of the mouse. In fact, using a mouse poses an unnatural disconnection between the motion of the users hand and the intended action onscreen [34]. Furthermore, the improved user



Figure 2.2: The use of pickerview in the FFT block.



Figure 2.3: Interface for editing function parameters.

experience, as a result of direct interaction with the system, has been the foremost reason for the increased popularity of smartphone devices . The architecture of i-JDSP targets to provide better user control by allowing direct manipulation of functions and parameters whenever possible.

Understanding the finger as an input device is the key to building successful

applications in an iOS. In i-JDSP, since all function blocks are presented graphically to the user, gesture recognition algorithms are used to interpret the different gesture inputs from the user. For example, similar to moving the function block by a mouse, users can move the block around by the panning (dragging) gesture. Moreover, double tap on any function block results in different actions to be executed depending on the function type, which are also consistent with the on-line J-DSP. Specifically, double tap on the Plot block opens up a plot of the output signal; double tap on a Frequency Response block brings up an action dialog where the user chooses either to view the magnitude or the phase response; double tap on other blocks, such as the Signal Generator, results in a detailed parameter listing of that block and the user can change the current settings of the parameters. And finally, to delete a single block, the long press gesture recognizer is used, which keeps track of the time of the taps. Table 2.1 illustrates the actions on various visual objects with different gestures.

Table 2.1: Gesture recognizers used in i-JDSP.

User gestures	on Object	Task
Single tap	Pin	Establish connections between two piins
Double tap	Function block	Show plot, choose action, edit parameters
Panning	Function block	Move function blocks around
Long press	Function block	Delete a single block

2.4.1 Moving

The position of a function block in a block diagram can be changed by simply touching the block and dragging it to the desired new location. Users can do this to make room for later blocks on the screen even if the block is already connected to other blocks.

2.4.2 Deleting

To delete a single block, users need to touch and hold the block for 3 seconds. When the popup menu appears, select “Delete” and the block and its associated connections

will be removed from the screen. To delete all the blocks on the screen, tap the "Trash" button on the right of the navigation bar.

2.4.3 Editing

To edit or update the parameters of the block, double tap on the block and an edit view with a list of configurable parameters will appear. Selecting any row of the list will allow the user to edit the value of the selected parameter. Figure 2.4 shows a typical edit view. For blocks that do not have configurable parameters, such as the plot block and the sound player block, double tapping such block will result in displaying results or music playback.



Figure 2.4: Typical parameter editing view in i-JDSP.

2.4.4 Establishing Connections

Basic DSP systems can be simulated by connecting function blocks together. To connect blocks, tap the output pin of the parent block first and then the input pin of the child block. Connections are allowed only if the two blocks are compatible. Two blocks are compatible only if the output of one of the blocks (parent block) generates the same data type as that of the input of the other block (child block). Data types are organized

into time domain signals, frequency domain signals and filter coefficients. Once the connection is successfully established, a red line will be drawn between the two blocks to indicate the connection. Otherwise, an error message will be presented to the user, which will be discussed in detail in the next section.

2.5 Feedback and Defensive Programming

Feedback plays an important role in the user interface design. An application without any feedback is almost impossible to use. The built-in iOS UI elements respond to every user action with some perceptible change. For example, list items are highlighted briefly when users tap them and the navigation controller uses animations to alert the user when the view is being pushed or popped from the stack. In other words, the application needs to provide ample feedback to help the users track the change .

When designing custom UI elements, the feedback becomes more important since users are not familiar with the them in the first place. i-JDSP enables the user to establish visual connections between the function blocks by single tapping the corresponding pins. However, due to the small dimensions of the pins, users are more likely to tap the wrong element or have to aim very carefully before the tap. Therefore, a custom feedback is crucial here to inform the user when the tap is successful. A viable solution is illustrated in Figure 2.5. When the user successfully taps the pin, the pin itself scales up to indicate that it has been correctly selected.

Using minimal number of controls and intuitive instructions, sometimes, does not necessarily guarantee that the users will not face any problems with the application. Defensive programming by definition is a form of defensive design intended to ensure the continuing function of an application in spite of unforeseeable usage and user inputs [35].

Although i-JDSP provides a convenient graphical environment for the users to utilize the power of the multi-touch screen, it also poses a problem when users provide unexpected inputs. These inputs should be by all means suitably handled by the ap-

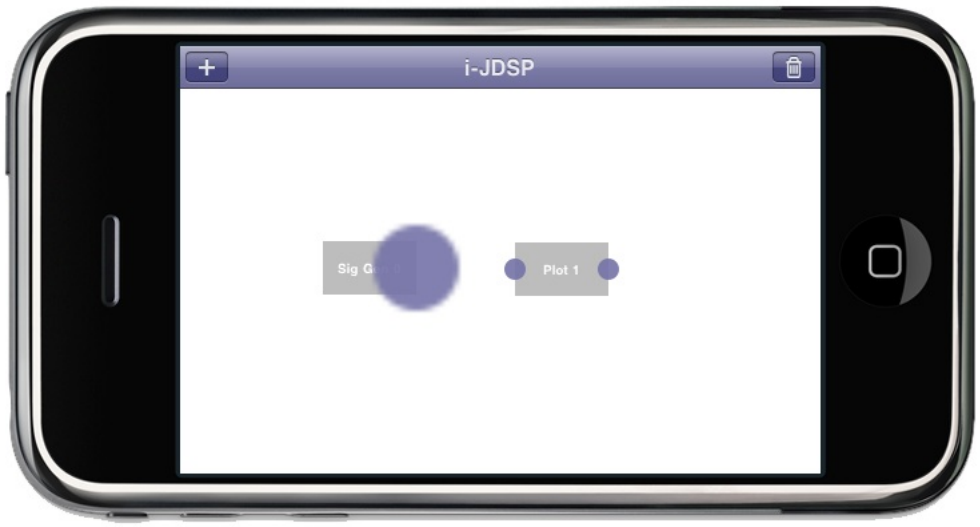


Figure 2.5: Feedback on Pins when user taps.

plication, otherwise they usually will result in erroneous outputs or can even crash the entire application. Informing the user, while handling the problems inside the code, is often a viable solution in this situation. For example, when users accidentally click a function block in the tableview listing that does not require parameter editing, the block will not be drawn immediately to the canvas, instead, the application will prompt an action sheet dialogue asking the users if they are sure about the action, as shown in Figure 2.6. If the user clicks "Yes", the block will be added to the main canvas, otherwise, the application will ignore this action.

Another means of defensive feedback in i-JDSP is by the use of AlertViews. The connection between the two function blocks is established by connecting the output pin of the parent block and the input of the child block. When the user tries to do this, the expected behavior will be to tap the output pin first and then the input pin second. However, this may not be the case. Possible scenarios could be the user tried to connect input pin to output or accidentally clicked the output pin twice. These aforementioned scenarios are handled in i-JDSP as illustrated in Figure 2.7.

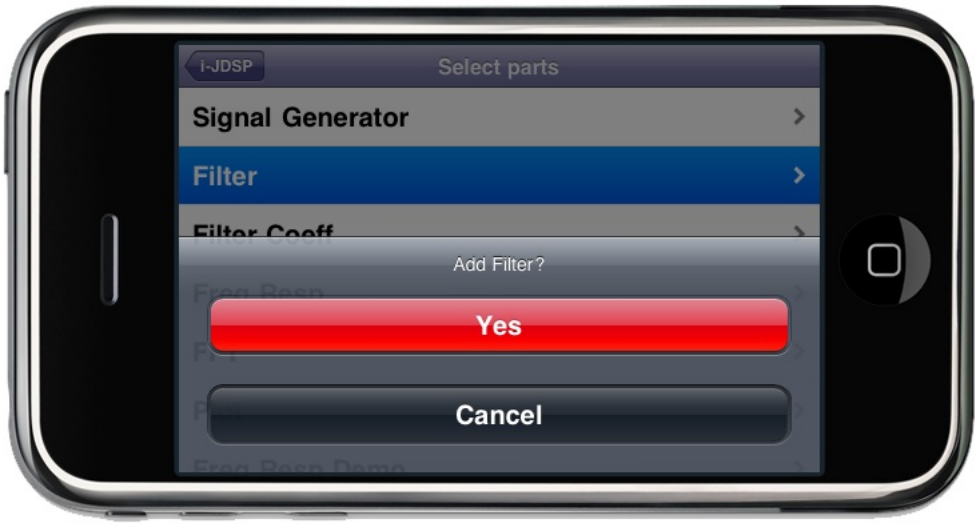


Figure 2.6: The use of ActionSheet in i-JDSP.



Figure 2.7: The use of AlertView in i-JDSP.

Chapter 3

SYSTEM ARCHITECTURE

3.1 Design Overview

The underlying architecture of the proposed iOS application is based on the existing version of the online J-DSP software. It is built using the Xcode IDE [36] and Interface Builder and developed as a native Cocoa Touch [37] application. The Cocoa Touch API is a collection of frameworks that drive the iOS and share many proven patterns found on the Mac, but with a special focus on touch-based interfaces and optimized for the iOS platform.

To address the all the challenges and problems mentioned in Section 1.3 (Chapter 1), we need to take into account several features of the iOS platform while designing the system architecture before actually writing the code.

The Users Mental Model in [38] describes the usability of a software application and its functionality as a combination of, users experience with other related software, and with computers in general. In our case, the user has prior experience with the conceptual model of using J-DSP to perform signal processing laboratories. An iOS application that ignores the users mental model and does not meet some of the important expectations of the user would be difficult and even unpleasant to use.

Figure 3.1 shows the user's mental model in detail for i-JDSP. In the application, there will be three major groups of objects:

- **DSP Function Blocks (Model):** Function objects for DSP, such as signal generator, FFT, filter.
- **Views (View):** View objects are used for representing function objects and any other drawings in the application.
- **View Controllers (Controller):** View controllers are responsible for the coordination between the models and views when receiving user inputs.

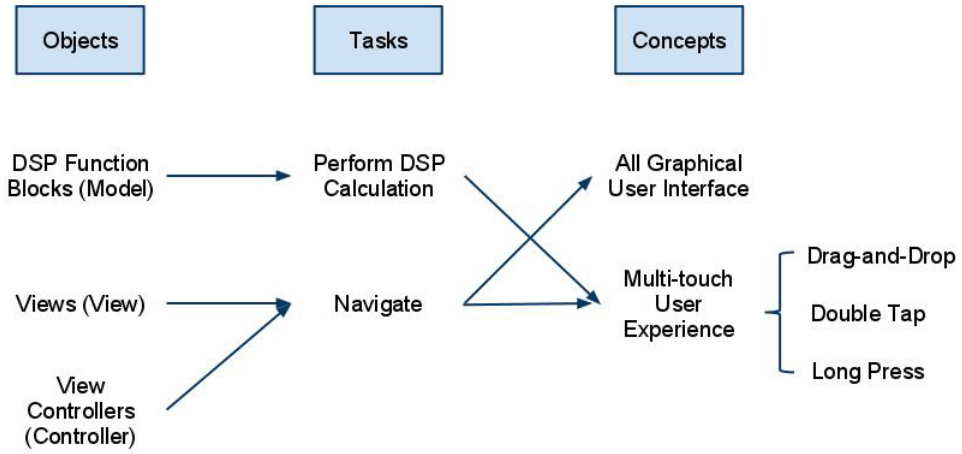


Figure 3.1: User's Mental Model of i-JDSP.

The ModelViewController (MVC) is a classic software architectural pattern that the iOS platform uses strictly to maintain multiple views on the same data model. The models are primarily used to maintain data, the views for displaying all or a portion of the data, and the controllers are used to handle events that affect the models or views. The MVC relationship in i-JDSP is illustrated in Figure 3.2. MVC is central to efficient design of a native Cocoa application on iOS. The benefits of adopting this design pattern are numerous. Many objects tend to be reusable. Moreover, applications based on the MVC design are easily extensible and scalable. Therefore, we will strictly comply to the MVC design pattern in our implementation as much as possible to achieve efficiency, robustness, and scalability.

3.2 Application Architecture

As mentioned earlier in Chapter 2, i-JDSP is a navigation-based application due to the limited screen size. The navigation controller takes care of the view stack and displays only one view at a time according to the user inputs, which, as a result, keeps the users focused on the current view. Each view has the minimal number of controls and necessary instructions in order to make the function immediately apparent to the users.

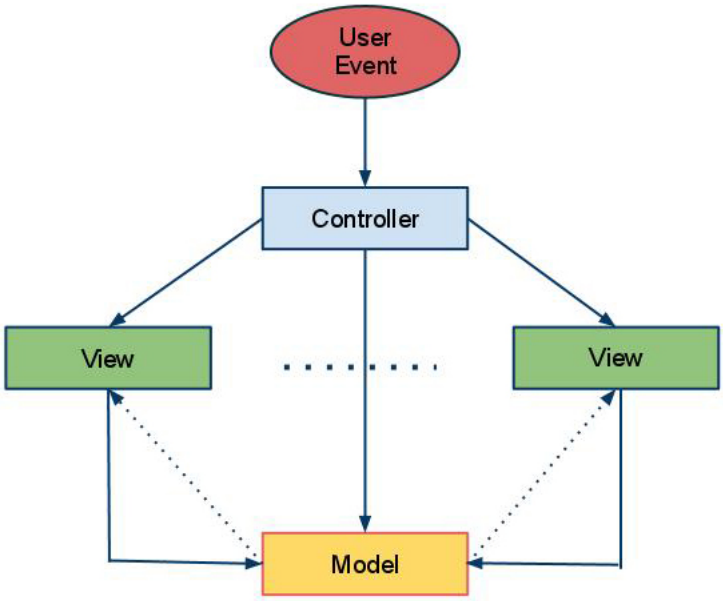


Figure 3.2: Model–View–Controller design pattern relationship.

Additionally, the users are always provided with a back button in order to retrace their steps. A detailed workflow to use the application is shown in Figure 3.3.

3.2.1 Hybrid Implementation

The Objective-C language is a reflective, object-oriented programming language defined as a set of extensions to the standard ANSI C language [39]. It was created primarily by Brad Cox and Tom Love in the early 1980s at their company called Stepstone. It is the primary language used for Apples Cocoa API, and therefore it is the main language to use for development in the iOS. In fact, most of the frameworks in the Cocoa Touch API such as the UIKit and the MapKit are implemented using Objective-C, and they are optimized for the iOS devices.

However, because the Objective-C language adds an object-oriented layer to the standard C language, it is a little slower than C in terms of performance. One major performance overhead in Objective-C is object allocation: every object allocated

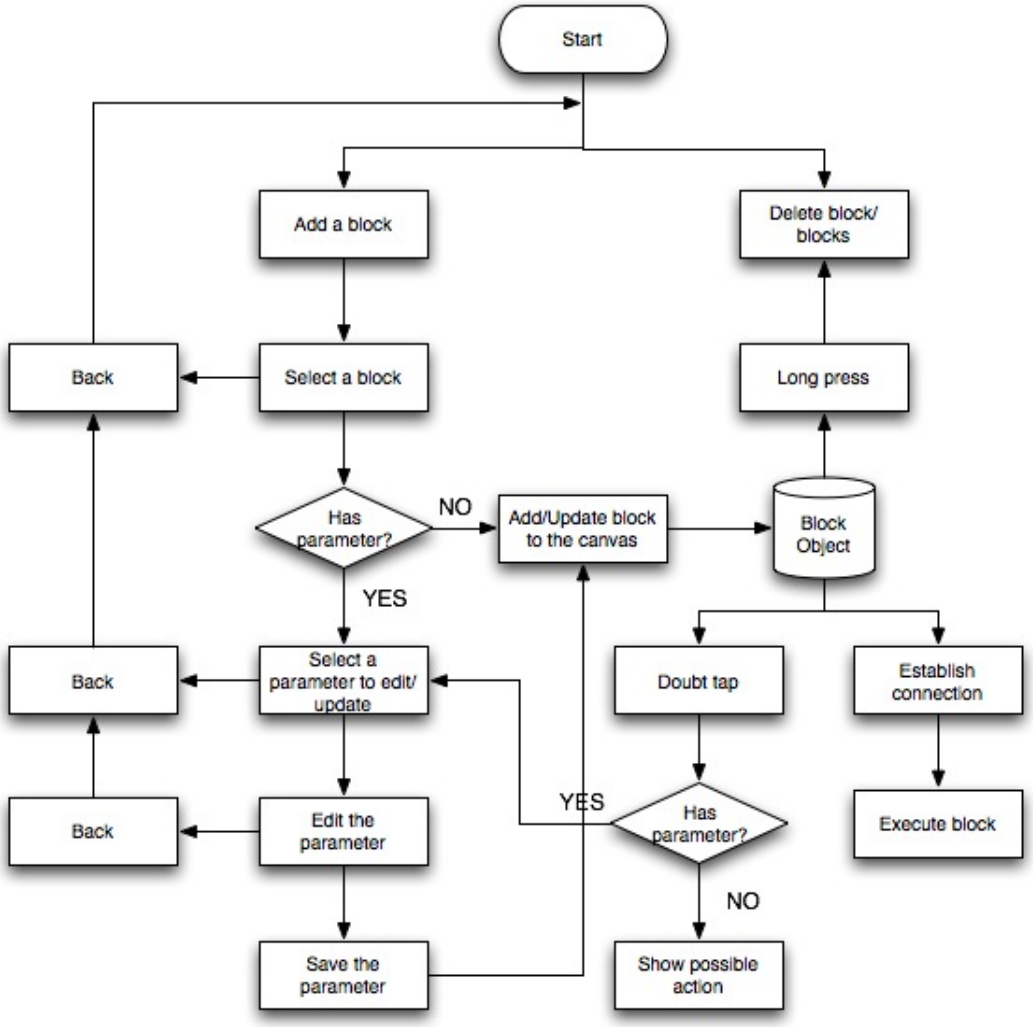


Figure 3.3: i-JDSP application flow diagram.

requires a call to alloc() or equivalent. It is perfectly fine if you call this alloc() function in small number. However, for applications like i-JDSP that require real-time signal processing of a large array of numbers, this becomes a problem. Table 3.1 compares the performance of the for loop iteration on a C array and an NSArray, using floats in both cases. The test is conducted on an iPhone 3G which is one of our target devices.

Luckily, the effectiveness of Objective-C allows us to use C when we need to. In our case, this means I need to implement all the functions and instance variables that related to number series in standard C arrays instead of Objective-C arrays (NSArray).

Table 3.1: Performance comparison between C Array and NSArray.

Number of elements	C Array iteration (ms)	NSArray iteration (ms)	Ratio
100	0.0194	0.113	5.825
200	0.0362	0.273	7.541
400	0.0721	0.663	9.200
5,000	0.756	9.12	12.06
20,000	3.27	34.2	10.46

More specifically, C arrays are wrapped as structures to be used in objects as instance variables to utilize the power of object-oriented programming and the performance boost of standard C arrays.

3.2.2 The Model-View-Controller Paradigm

Due to the fact that our i-JDSP deals with multiple views on the iOS based on user inputs and different DSP function blocks, it will benefit a lot from the MVC design pattern. I discussed a little bit about the MVC design pattern earlier in Section 3.1. In this section, I will describe in detail how MVC is used in our design.

- **Model:** Basic DSP functions such as signal generator, FFT, and filter are treated as models in i-JDSP. They are implemented as Objective-C classes and are derived from a base class named Part. The Part class is essentially inherited from NSObject – the root of the entire Objective-C class hierarchy. Figure 3.4 illustrates the Unified Modeling Language (UML) diagram of such relationship in the example of FFT.

  Please note that instance variables: parents and children are pointers to NSMutableArray class that later allow us to establish block flow relationships. In addition, data, buffer, signal, and fftData are also instance variables that are defined as custom C structures to hold standard C arrays for data storage and communication. The benefit of this approach is that not only it achieves improved performance as discussed in Section 3.2.1, but it also avoids the need to ex-

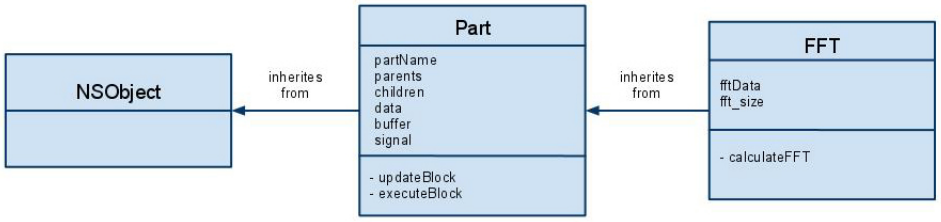


Figure 3.4: Inheritance UML diagram of the models.

plicitly allocate or free memory for these instance variables. However, because Objective-C does not automatically provide setter/getter methods for standard C arrays itself, we need to write our own if we want to access these values from outside of the class.

- **View:** The view renders the model into a form suitable for interaction, typically a user interface element. In i-JDSP, anything visible on the screen is essentially a view. In fact, all these visual elements are derived from one single class in Objective-C called UIView. UIView is the parent class for display objects on the iOS. It defines a rectangular area and the interfaces for managing the content in that area. Generally speaking, it provides a convenient way to layout simple interface elements on the screen by subclassing UIView.

  To represent the aforementioned DSP function models in a graphical environment like i-JDSP, we use PartView, a custom class inherited from UIView. This class basically designs the DSP function as a rectangular block, with a label inside rendering the title and a number of circular dots around depending on the function. Figure 3.5 shows the basic layout of a function block. In this case, the block has six corresponding dots (or pins). The number of pins are determined in run time according to the function type it represents, due to the dynamic nature of Objective-C.

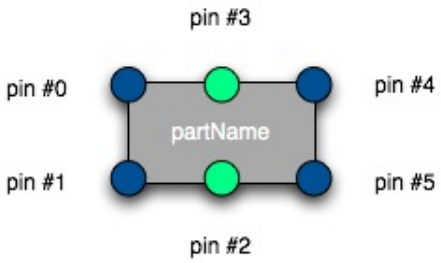


Figure 3.5: Basic block layout. Pin 0, 1: inputs for time domain signals; Pin 4, 5: outputs for time domain signals; Pin 2: input for frequency domain signal; Pin 3: output for frequency domain signal.

The pins are actually rendered by another custom view class named PinView. This simple but necessary class has a pinNum property to distinguish each other in the PartView and a belongToPartName property that refers to the PartView it belongs to. It also knows how to handle user input events, such as tap and double tap, which will be discussed in detail in the next section.

- **Controller:** In an iOS application, the controller is typically paired with a view. Thus, most often we will be dealing with view controllers. The view controller is responsible for receiving the user's input and acting accordingly. It acts as the coordinator between the MVC components.

  i-JDSP is built entirely as a navigation based iOS application. It has a RootViewController class that handles the main drawing canvas, a table view controller for presenting the list of all the possible DSP functions, a detailed view controller for displaying the parameters of the selected function, an attribute editor controller for editing the parameters, and finally a plot view controller for plotting the output signals.

3.2.3 The Delegation Pattern

According to Apple's official definition [40]: delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another

object. The delegating object keeps a reference to the other object, the delegate, and sends a message to it at the appropriate time. The delegation pattern is one of the fundamental abstraction patterns in object-oriented software design that many major programming languages employ.

Objective-C and the Cocoa Touch framework use the delegation pattern very heavily. The delegating object is typically a model class, and the delegate is typically a custom controller object. To make delegation work, the delegating class first needs to declare a protocol for the delegate class to conform to. A protocol is essentially a declaration of some methods that can be implemented by any class. Conforming to the protocol means the delegate class promises to implement some or all the methods declared in the protocol.

A simple example can be seen in Figure 3.6. In this example, an instance of the UITableView class declares a protocol called UITableViewDelegate, among whose method is didSelectRowAtIndexPath:. It then asks the PartsTableViewController to be its delegate. When user clicks one of the rows in the tableview, the tableview object sends didSelectRowAtIndexPath: message to its delegate – the PartsTableViewController – to ask if the current row has been selected. The delegate returns a Boolean value, thereby controlling the behavior of the tableview object.

A more complex example is the use of the nested delegation pattern in i-JDSP as shown in Figure 3.7. In order to edit the parameters of a function block before it is being drawn on the main canvas (the RootViewController), we need to somehow inform the RootViewController when the save button has been clicked in the AttributeEditor. Nested delegation allows us do just that by populating the action caused in the AttributeEditor back to the RootViewController through the navigation controller's view hierarchy. The whole process is explained as follows. First, the AttributeEditor declares the saved: forKey: method in a protocol named AttributeSavedDelegate. The instance of PartDetailViewController conforms to that protocol by implementing

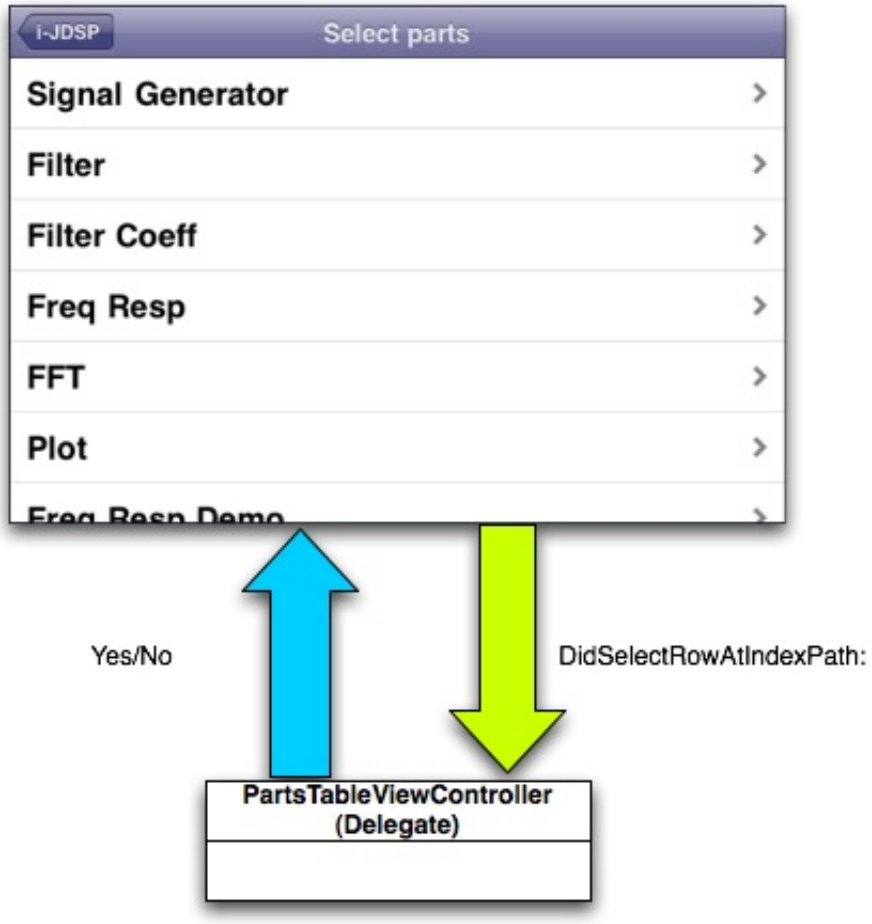


Figure 3.6: A delegation example in tableview.

the saved: forKey: method and thus acts as the AttributeEditor's delegate. It then declares a method called partAdded: in a protocol named PartAddedDelegate. Next, the instance of PartsTableViewController adopts the PartAddedDelegate protocol and declares a new method drawPart: in a protocol called DrawPartDelegate. At the same time, it asks RootViewController to be its delegate. Later, the instance of RootViewController conforms to the DrawPartDelegate and gets notified when the save button is clicked in the previous AttributeEditor. And finally, the function block with the new set of parameters is rendered on the main canvas.

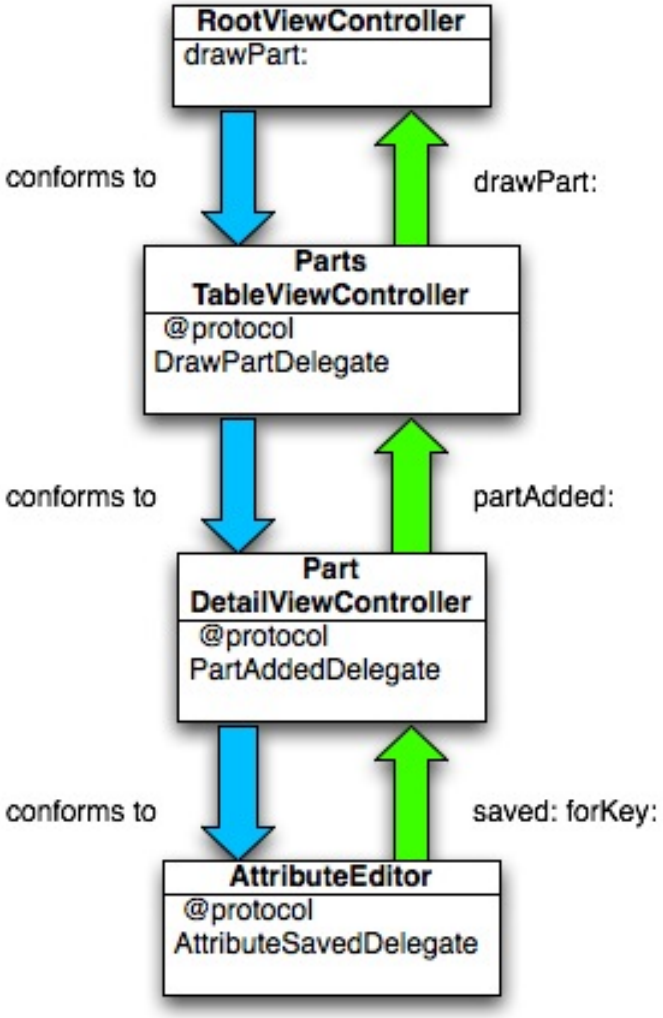


Figure 3.7: Nested delegation use in i-JDSP.

Chapter 4

SYSTEM IMPLEMENTATION

4.1 Overview

With regards to the implementation, i-JDSP uses all the design points mentioned in the previous chapters to offer a user-friendly environment through the combination of Objective-C, C, and the iOS SDK. Its multi-touch and all graphical user interface make using i-JDSP a pleasant and intuitive experience. Moreover, the only DSP functions that users need to select and connect with are the underlying GUI elements. i-JDSP currently has the following DSP functions implemented:

- **Signal Generator**
- **Fast Fourier Transform (FFT)**
- **Filter**
- **Filter Coefficient**
- **Frequency Response**
- **Plot**
- **Adder**
- **PZ Plot**
- **PZ2Coef**
- **PZ Placement**
- **MIDI**
- **DTMF**
- **Sound Recorder**

- **Sound Player**

All the above functions are represented in i-JDSP as graphical blocks that are also capable of handing user gesture inputs. Selecting and establishing individual blocks can be done simply by drag-and-drop and panning. By connecting blocks together, a variety of DSP systems can be simulated. Block parameters or plotting options can be edited through double tapping the current block, allowing the user to change the previous settings and see the results without adding a new one. Individual block can be deleted along with its established connections if any by touching and hold (long press). Theoretically, i-JDSP supports unlimited number of function blocks and multiple blocks with the same function. However, this is not practical since the screen space is limited.

4.2 Fundamental DSP Functions

Inheritance is one the most important features of object-oriented programming. It is a way to compartmentalize and reuse code by creating classes (derived classes) based on previously defined classes (super classes). By doing this, the derived classes automatically inherit attributes (or instance variables) and behavior (methods) of the super classes. As a result, the use of inheritance greatly reduces the size of the application and therefore improves the quality, stability and scalability.

In i-JDSP, all the function blocks are inherited from a super class named Part. The Part class, as shown in Figure 4.1, has six instance variables that will be shared in all the derived classes:

- **partName:** A unique NSString value for identifying each block.
- **parents:** A mutable array that can hold up to three parent blocks.
- **children:** A mutable array that can hold up to three child blocks.
- **data:** A custom structure that stores six double type arrays with 3×256 capacity.

- **buffer:** A custom structure that stores six double type arrays with 3×256 capacity.
- **signal:** Stores the signal type and signal size information for each of the data and buffer channels.

Part also provides the following methods:

- **setter/getter:** Setter and getter methods for accessing the values in the custom structures.
- **updateBlock:** This method is the same for all the function blocks. Basically, when either a connection is established or the parameters are updated, it iterates through the children list and executes them.
- **executeBlock:** Executes the block. This method is meant to be overridden by the derived class.

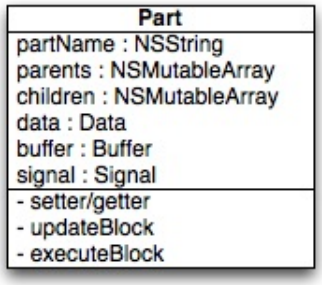


Figure 4.1: UML diagram of the Part class.

We will discuss the implementation of these fundamental DSP functions shortly. These basic functions can be combined to build more complex algorithm simulations.

4.2.1 Signal Generator

The Signal Generator block implements a suite of basic deterministic signals (rectangular, triangular, delta, sinusoid, sinc and random) with configurable parameters, which

later can be easily extended to user specified signals and real-life signals such as speech, music and other time-domain signals. A maximum of 256 time domain samples can be generated. Signals can also be periodic and shifted with a maximum shift of 256 samples.

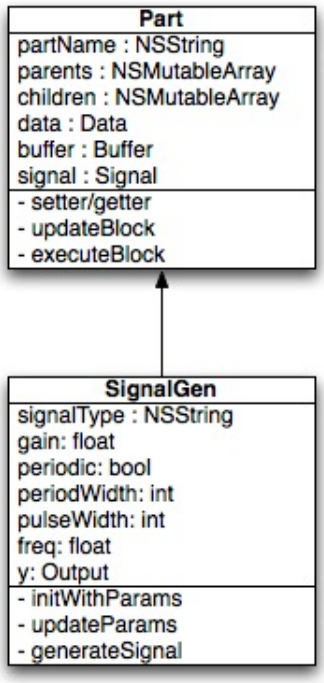


Figure 4.2: UML diagram of the SignalGen class.



Figure 4.3: The Signal Generator block.

As in Figure 4.2, we can see that the SignalGen class is inherited from the Part class with a few of added instance variables and methods. Note that instance variable *y* is a custom structure that holds up to 256 doubles for the actual output. The rest of the

instance variables are signal parameters that can be configured to form desired signals. The Signal Generator block acts as the signal source for later processing; thus, it has only one pin, that is, the time domain output (pin 4), as shown in Figure 4.3. It can be connected to any block that processes time domain signals.

4.2.2 Fast Fourier Transform (FFT)

The FFT block computes the discrete Fourier transform (DFT), as denoted in Equation 4.1, of the input signal using the Cooley-Tukey algorithm [41]. The Cooley-Tukey algorithm is a common fast Fourier transform algorithm that uses a "divide and conquer" approach to break down a DFT of any composite size, N, into many smaller DFTs. As a result, this method leads to $O(NlogN)$ complexity while the normal DFT algorithm requires $O(N^2)$ operations.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi nk}{N}} \tag{4.1}$$

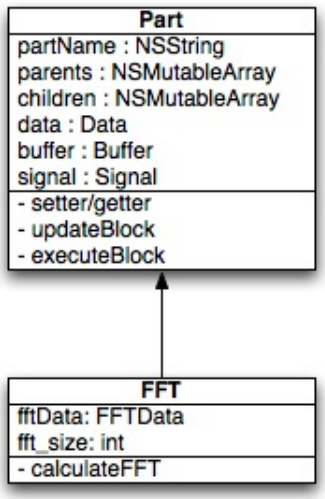


Figure 4.4: UML diagram of the FFT class.

The instance variable fft_size, shown in Figure 4.4 can be edited by the user to determine the FFT size. Possible values that are powers of two include 8, 16, 32, 64,



Figure 4.5: The FFT block.

128, and 256. The higher the value, the better the FFT resolution. The FFT block takes in a signal and computes the FFT in both magnitude and phase. Thus, it has two pins, one for time-domain input and another for output, as shown in Figure 4.5.

4.2.3 Filter

The Filter block acts as a digital filter that performs mathematical operations on the incoming signal to reduce or enhance certain aspects of that signal. More specifically, it filters the input signal based on the provided numerator and denominator coefficients. The characterization of a filter is determined by its transfer function which is defined in Equation 4.2 in the Z domain:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_M z^{-M}} \tag{4.2}$$

where a and b are the coefficients of the filter. The order of the filter is the greater of N or M.

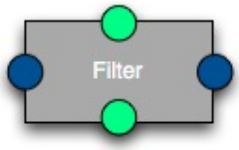


Figure 4.6: The Filter block.

The Filter block has four pins (Figure 4.6) to be able to work in both time domain and frequency domain. In addition, as seen in Figure 4.7, filterData is a custom

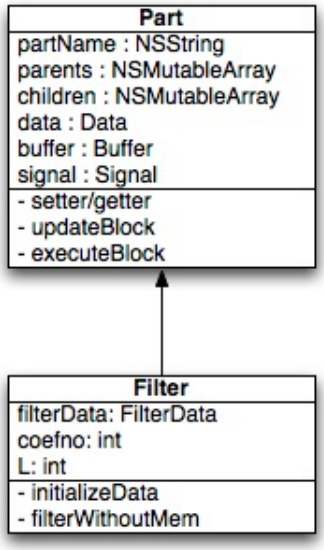


Figure 4.7: UML diagram of the Filter class.

structure that stores the coefficients as well as the input/output data. L denotes the input signal length; coefno is the number of coefficients.

4.2.4 Filter Coefficients

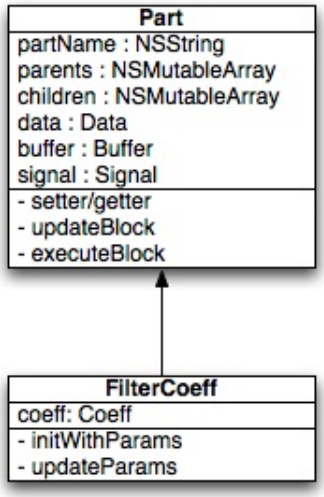


Figure 4.8: UML diagram of the FilterCoeff class.



Figure 4.9: The Filter Coefficients block.

The Filter Coefficients block simply sets the a_i and b_i in Equation 4.2 to determine the characteristics of the filter. A maximum of 11 coefficients can be used. As we can see from Figure 4.8, the FilterCoeff class stores the coefficients in a structure named coeff as an instance variable. Users can change and edit the coefficients at any time by double tapping the block. When updated, the block outputs the filter coefficients through pin 3 to the filter, as shown in Figure 4.9.

4.2.5 Frequency Response

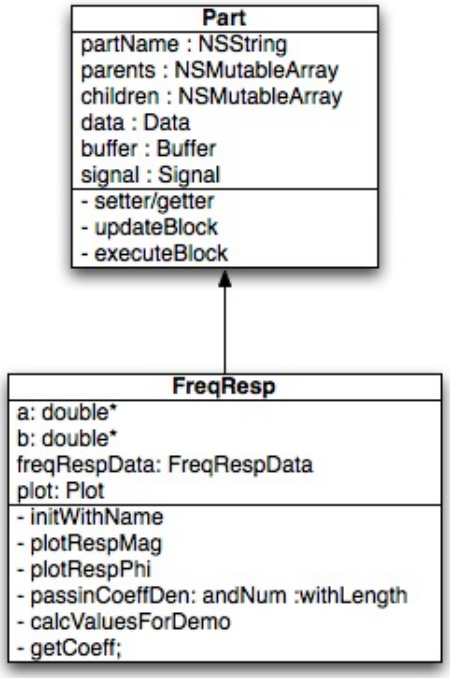


Figure 4.10: UML diagram of the FreqResp class.

The Frequency Response block measures the system's output spectrum in response to an input signal. It represents the output by the magnitude of the system's response, measured in both linear amplitude and decibels (dB), and the phase, measured in radians, versus frequency. Since a system's frequency response is the Fourier Transform of its impulse response, and the impulse response is then determined by the transfer function, the Frequency Response block needs to work with the Filter block as well as the Filter Coefficients block.

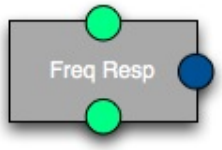


Figure 4.11: The Frequency Response block.

From the UML diagram in Figure 4.10, instance variables *a* and *b* are used for getting the filter coefficients; freqRespData is a custom structure that stores the calculated response output; plot is an instance of the Plot class that is for plotting the output figure. When users double tap the Frequency Response block (in Figure 4.11), an action sheet will be presented to allow users to choose between the phase response and the magnitude response (in Figure 4.12). Users also have the option to view the output figure in both linear and dB measurements.

4.2.6 Plot

The Plot block, as the name suggests, simply outputs the data in a figure. It accepts users' double tap input and stores the received output data from its parent block in a custom structure named plotOutput as an instance variable shown (in Figure 4.13). It also has another NSString instance variable, scale, to track the plotting option (linear or dB) based on users' input. Moreover, the plot block utilizes CorePlot, an open source 2D visualization framework on iOS, for plotting and manipulating the graph.



Figure 4.12: The actionsheet in the Frequency Response block.

The instance variable plotView directly interacts with an instance of CPXYGraph – a 2D graph class in the CorePlot [42] framework to visualize the output data as well as handle user input such as moving the graph. The plot block only has two pins (input and output), both in the time domain, as shown in Figure 4.14.

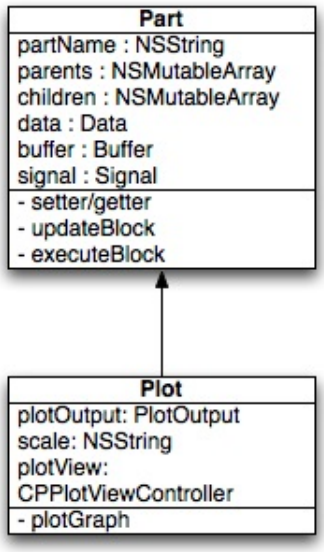


Figure 4.13: UML diagram of the Plot class.



Figure 4.14: The plot block.

4.2.7 Adder



Figure 4.15: The adder block.

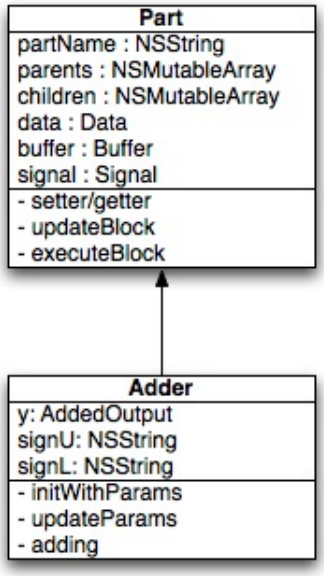


Figure 4.16: UML diagram of the Adder class.

The Adder block simply takes two time domain signals and adds or subtracts them depending on the $+/-$ sign. Users can change the input signal signs tracked by

two instance variables signU and signL by double tapping the block. The block holds the added data in a custom structure named AddedOutput as an instance variable, *y*, as seen in Figure 4.16, and outputs the data through pin 4 when executed.

4.2.8 PZ Plot



Figure 4.17: The PZ Plot block.

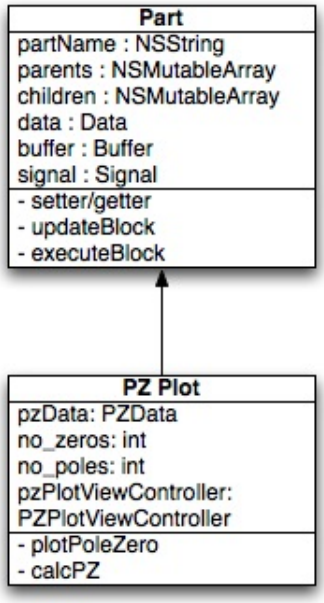


Figure 4.18: UML diagram of the PZ Plot class.

The PZ Plot block calculates and displays the poles and zeros of a system transfer function in the z-plane. Figure 4.17 shows the layout of the PZ Plot block. It accepts the filter coefficients as its input through pin 2.

When connected, it calculates the *x* and *y* positions of the poles and zeros in the z-plane and stores them in a structure named pzData (Figure 4.18). Users can then

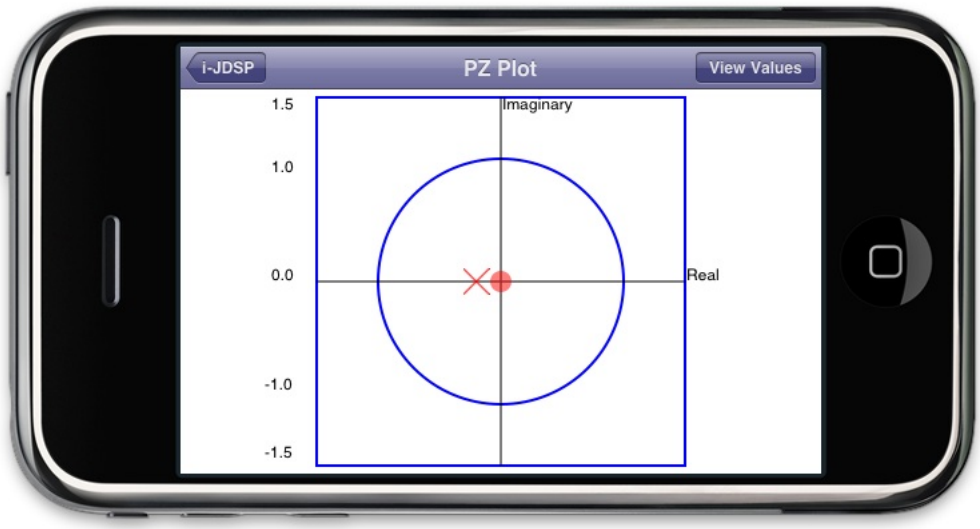


Figure 4.19: PZ Plot output when coefficients a0 = 1, a1 = 0.2, b0 = 1.

double click the block to see the resulting graph controlled by pzPlotViewController, as shown in Figure 4.19.

4.2.9 PZ2Coef

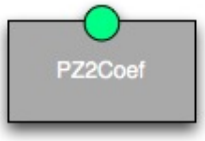


Figure 4.20: The PZ2Coef block.

The PZ2Coef block allows the user to place poles and zeros of a filter graphically in the Z-plane and outputs the filter coefficients through pin 3 (Figure 4.20). It stores the calculated coefficient data in an instance variable called coefData. Also, notice that in the class UML diagram in Figure 4.21, the instance variable polezeroPart is of class PoleZero, which we will talk about in detail in the later section that is responsible for describing the calculation of the coefficients.

When the user double taps the PZ2Coef block, an interface similar to PZ Plot

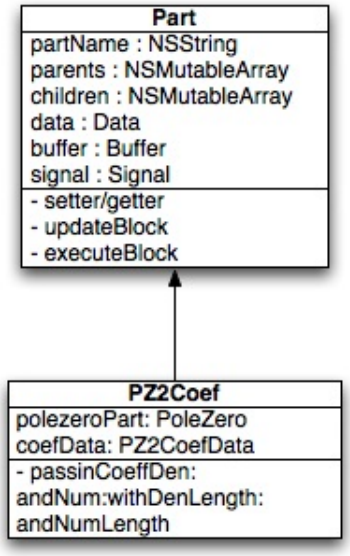


Figure 4.21: UML diagram of the PZ2Coef class.

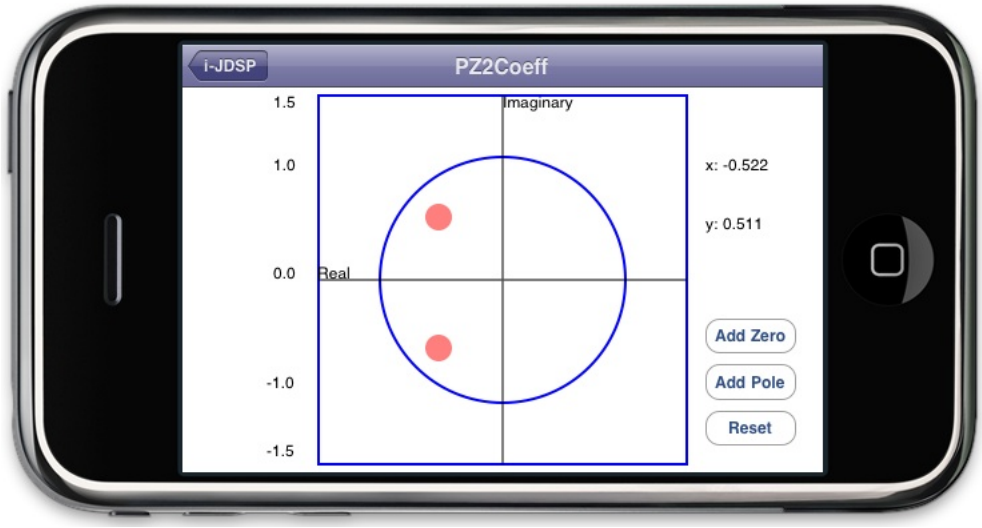


Figure 4.22: Interface for placing the poles and zeros in the PZ2Coef block.

is presented (Figure 4.22). Rather then viewing the poles and zeros, the user can add poles or zeros to the Z-plane by clicking the corresponding button. Once added, the poles and zeros can be dragged to the desired position. In the current version of the software, a maximum of 10 poles and zeros are currently supported. Moreover, the

poles and zeros are added as conjugate pairs to the Z-plane.

4.2.10 PZ Placement



Figure 4.23: The PZ Placement block.

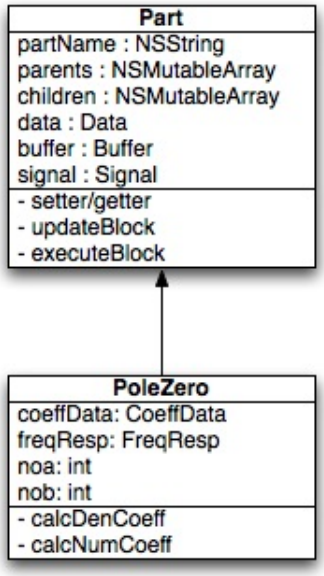


Figure 4.24: UML diagram of the PoleZero class.

The PZ Placement block enables the user to place poles and zeros graphically in the Z-plane and analyze the corresponding frequency response. This block has no input or output pins (Figure 4.23) as it operates on itself. As may be observed from Figure 4.24, the PoleZero class has an instance variable, freqResp, an instance of class FreqResp, that calculates the frequency response based on the poles and zeros. It also has a structure named coeffData to store the coefficient data that feeds the freqResp, as well as two integers (noa, nob) to keep track of the number of coefficients.

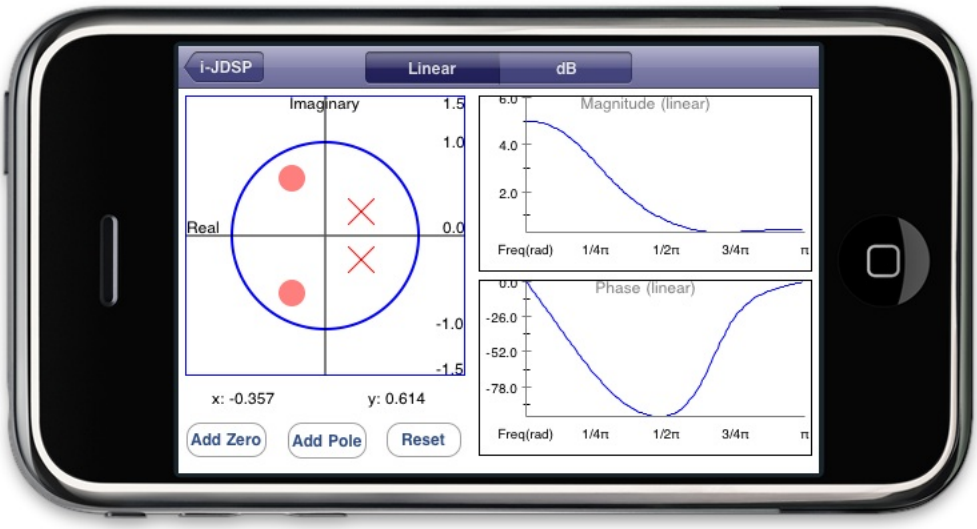


Figure 4.25: Interface of the PZ Placement block.

In Figure 4.25, we can see the interface when the user double clicks the block. On the left, there is a familiar interface that allows the user to add poles and zeros to the Z-plane. On the right, the frequency response, measured in both linear and decibel (dB) scales, and the phase, measured in radians, are displayed. Moving the poles and zeros dynamically updates the magnitude and phase response of the corresponding filter. Also note that the poles and zeros are added to the Z-plane as conjugate pairs and currently a maximum of 10 poles and zeros are supported.

4.2.11 MIDI



Figure 4.26: The MIDI block.

The MIDI block generates Musical Instrument Digital Interface (MIDI) sounds at frequencies described by the MIDI standard. It can generate MIDI tones of length

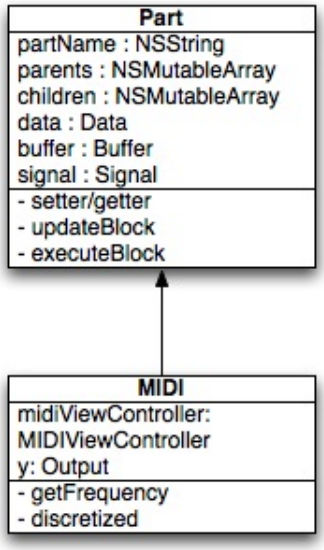


Figure 4.27: UML diagram of the MIDI class.

256 samples with a sampling frequency of 8kHz and output the signal through pin 4 (Figure 4.26). This block can be used along with the FFT block in i-JDSP to analyze the spectrum of the MIDI tones.



Figure 4.28: Interface of the MIDI block.

Seen in the UML diagram in Figure 4.27, the MIDI class has an instance vari-

able midiViewController that provides the MIDI interface. When double clicked, the block brings up a simple piano keyboard interface, as shown in Figure 4.28. The user can tap any keys and use the FFT block to relate musical and MIDI tones to frequencies in Hz.

4.2.12 DTMF



Figure 4.29: The DTMF block.

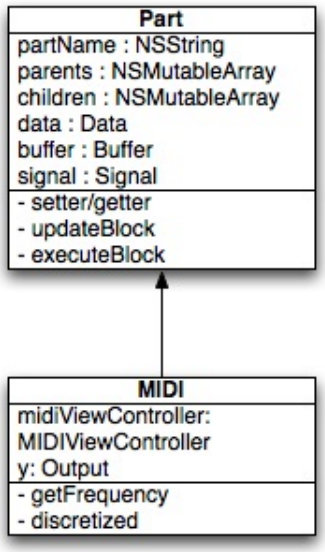


Figure 4.30: UML diagram of the DTMF class.

The DTMF block generates Dual Tone Multi-Frequency (DTMF) sounds that are used in cellular and wireline telephones. A dual tone consists of two frequency components that are generated upon dialing a number. The DTMF encoded telephone

numbers are then transmitted to a telephone switch which detects them using an appropriate detection system.



Figure 4.31: Interface of the DTMF block.

Similar to the MIDI block, the DTMF block can generate DTMF tones of length 256 samples with a sampling frequency of 8kHz and outputs the signal through pin 4 (Figure 4.29). Shown in Figure 4.30, freq1 and freq2 are used to track the two frequency components, *y* stores the generated signal, and dtmfViewController simulates a telephone dialing interface, as illustrated in Figure 4.31. Users can also use the FFT block to analyze the spectrum of the DTMF tones.

4.2.13 Sound Recorder

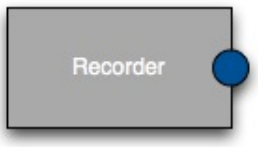


Figure 4.32: The Sound Recorder block.

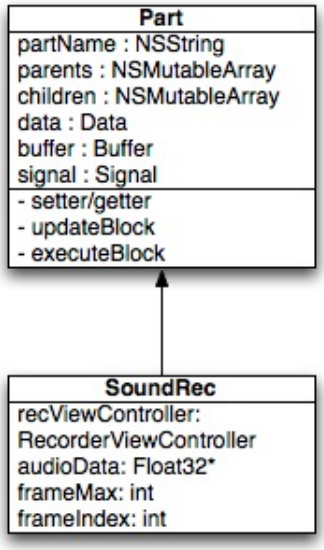


Figure 4.33: UML diagram of the SoundRec class.

The Sound Recorder block provides a simple interface for audio recording and outputs the audio data for later processing (Figure 4.32), utilizing the built-in microphones in the iOS devices. In the current version of i-JDSP, it supports a maximum recording time of 10 seconds of WAV format audio with a sampling frequency of 44.1kHz.

When the user double taps the block, the recorder interface is displayed (Figure 4.34). i-JDSP uses the AVFoundation framework for recording and the CoreAudio framework for reading the raw audio data. The audio data is converted to type Float32 and stored in an array pointed to by the instance variable audioData (Figure 4.33). It is important to note that in order to work with other function blocks that only support processing 256 samples of data at a time, users have to click the "Parse Data" button after recording to process the whole data.

4.2.14 Sound Player

The Sound Player block, as illustrated in Figure 4.35, is used for signal playback. It receives the signal through pin 0. If the signal is more than 256 samples, it appends the

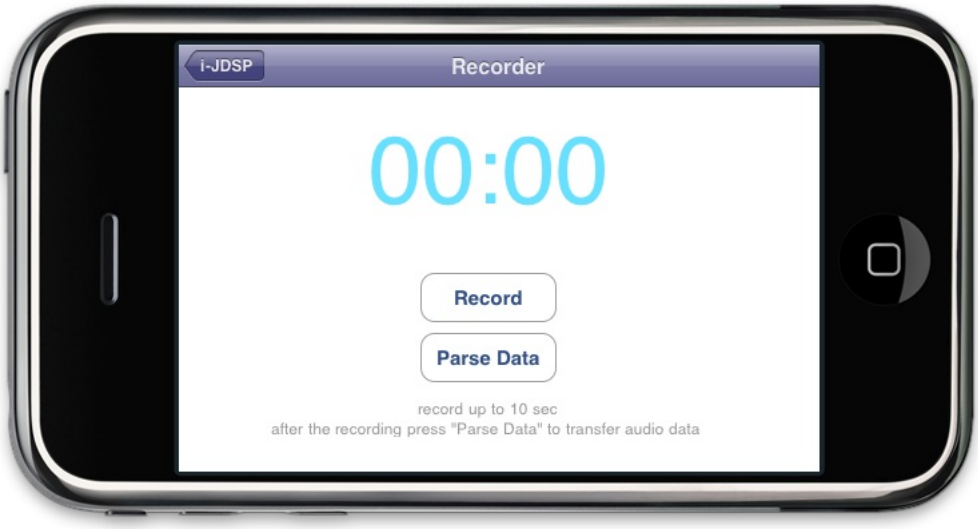


Figure 4.34: Interface of the Sound Recorder block.



Figure 4.35: The Sound Player block.

data and stores it inside the instance variable receivedData (Figure 4.36).

The sound player does not have an associated interface. When the user double taps the block, it writes the received data to a WAV file with a sampling frequency of 44100Hz and starts the playback through the built-in speakers.

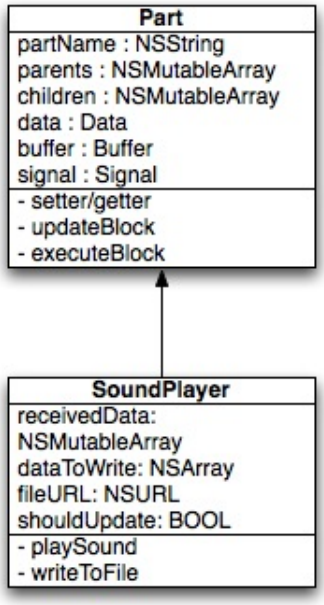


Figure 4.36: UML diagram of the SoundPlayer class.

Chapter 5

EXERCISES, SIMULATIONS AND EVALUATIONS

5.1 Simulation Examples

In this section, we demonstrate some of the common use cases that could be used to solve DSP problems when combining the basic DSP function blocks described in the previous chapter.

5.1.1 FFT Simulation

The FFT simulation (in Figure 5.1) consists of three function blocks: Signal Generator block, FFT block, and Plot block. The Signal Generator block outputs the user specified signal to the FFT block where the FFT is calculated. Then the calculated value is shown by the Plot block.

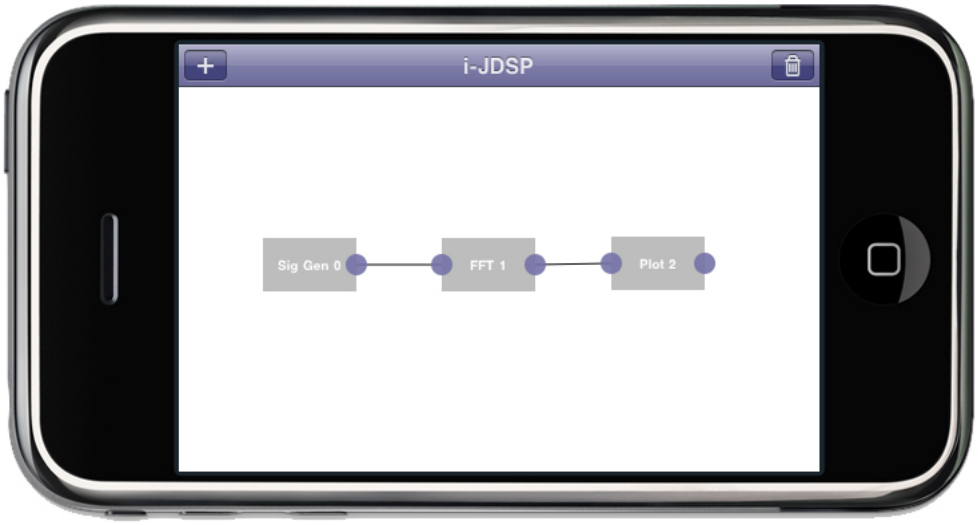


Figure 5.1: i-JDSP simulation for calculating the FFT of a rectangular signal.

Figure 5.2 shows the output of the FFT calculation when users double tap the Plot block. Users will easily discover the symmetric nature of FFT. Moreover, when users alter the FFT block setting, they will also notice the resolution difference of the two outputs due to the change in the FFT size. Users then can try various signal and FFT setting combinations and see the difference in the output figure.

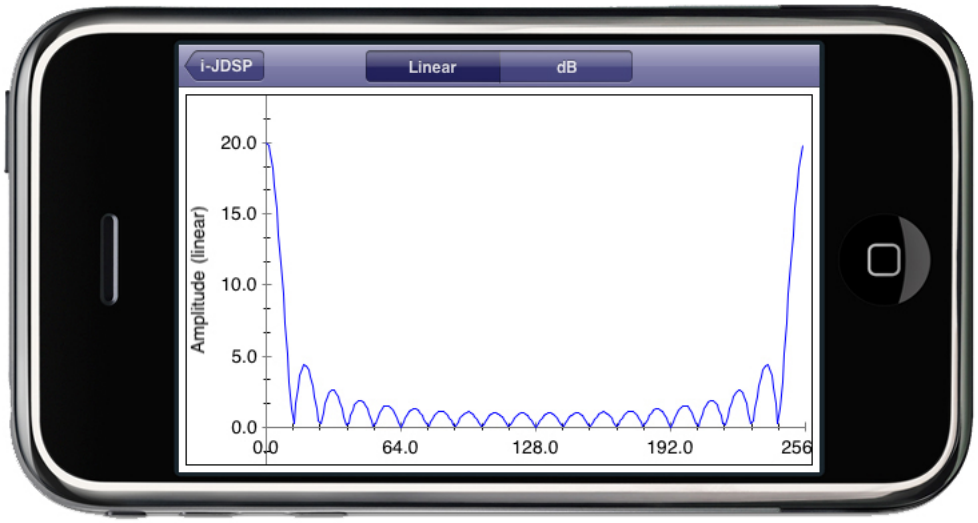


Figure 5.2: Output of the FFT demo when FFT size = 256.

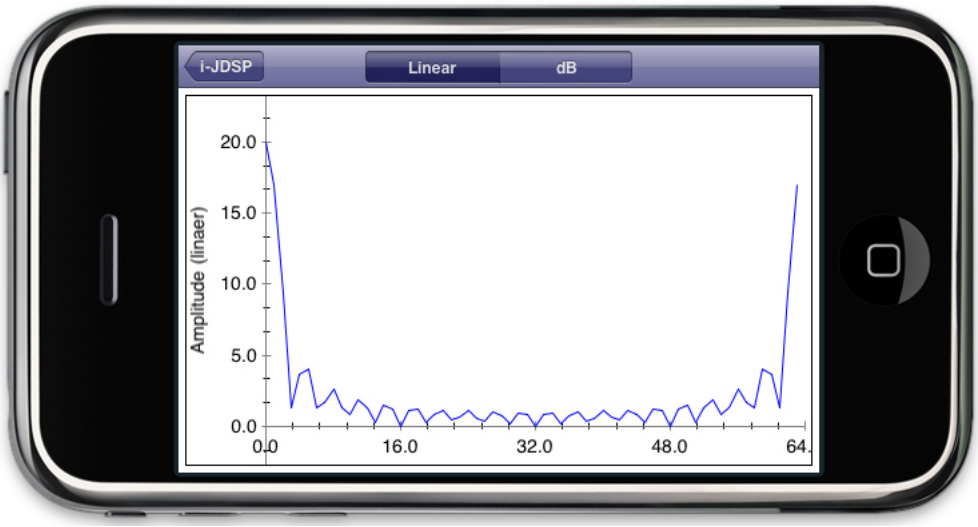


Figure 5.3: Output of the FFT demo when FFT size = 64.

5.1.2 Filter Response Simulation

The filter coefficients and response simulation involves four function blocks: Signal Generator, Filter, Filter Coefficients, and Frequency Response, as shown in Figure 5.4.

The Signal Generator block inputs a user defined time domain signal through the filter. The filter then performs the filtering with the coefficients which the filter

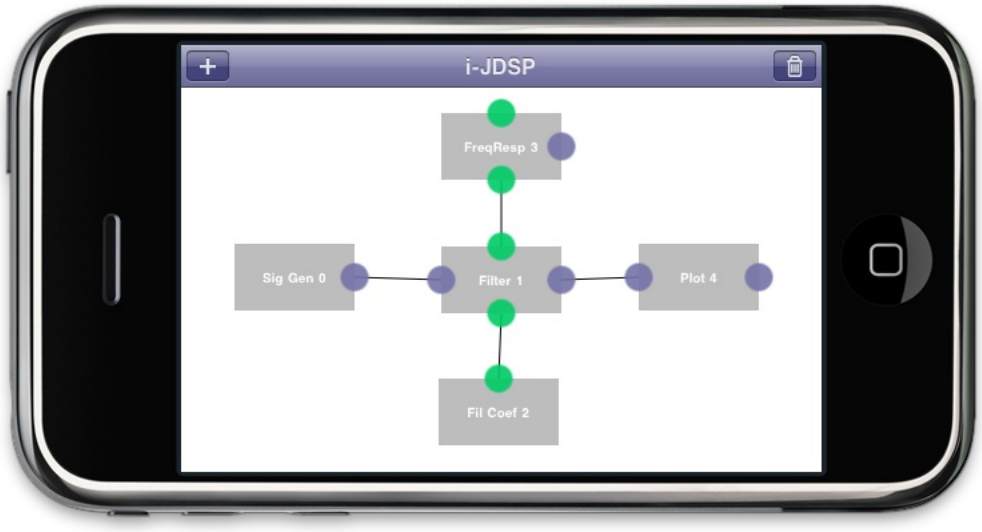


Figure 5.4: i-JDSP simulation for filtering and frequency response.

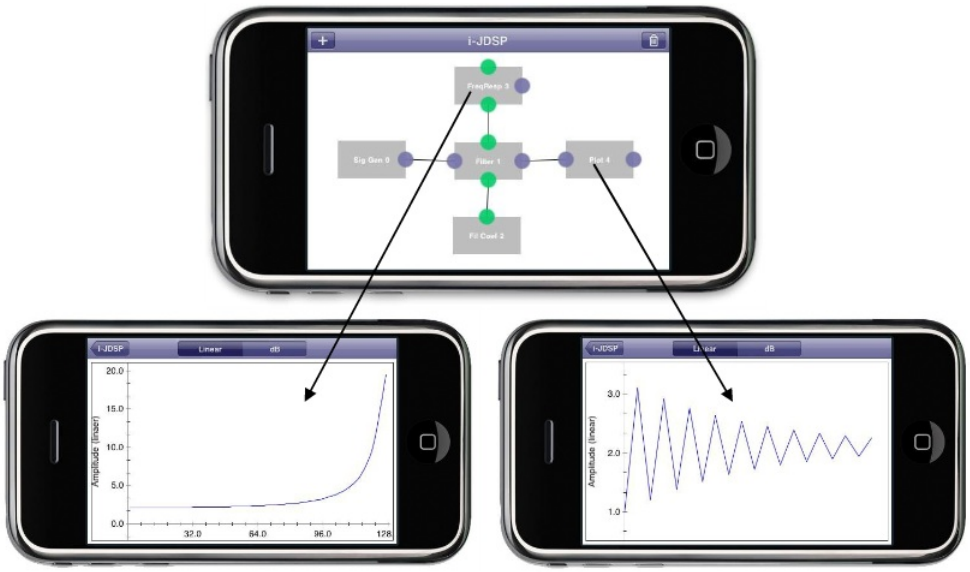


Figure 5.5: Filtering of a rectangular signal in i-JDSP with filter coefficients: a = [1, 0.9], b = [1, 3].

coefficients block supplies. Finally, the frequency response block plots the frequency response. Users can choose to view the frequency response in magnitude or phase by selecting the corresponding pop-up menu. When the user specifies the filter coefficients to the filter, the input signal gets filtered. And by double tapping the corresponding

block, the filtered signal (right) and the frequency response (left) are shown in Figure 5.5.

5.1.3 Pole-Zero Simulation

The pole-zero simulation is useful for providing pole-zero plot of the system and relating the locations of the poles and zeros with its frequency response. To see an example, imagine a tenth-order system whose transfer function is given below:

$$H(z) = \frac{1}{1 - 0.9z^{-10}} \tag{5.1}$$

simulate the system using the following function blocks: Signal Generator, Filter, Filter Coefficients, PZ Plot and Frequency Response, as illustrated in Figure 5.6.

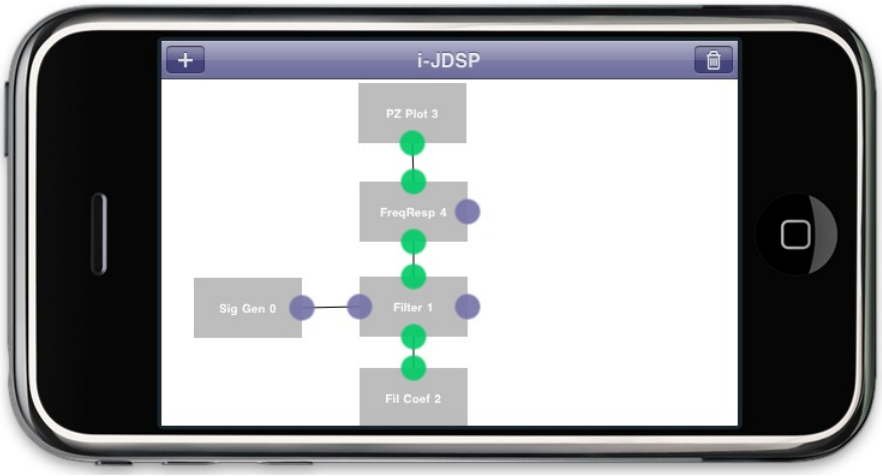


Figure 5.6: i-JDSP simulation for pole-zero plot and frequency response.

In Figure 5.7, the simulation results are displayed. Users can clearly see that the poles are distributed evenly inside the unit circle. Further looking into the frequency response, users will realize the unique characterization of the system. It is a special type of filter that is common for speech processing applications; specifically it is used for speech synthesis in digital cellular phones [43].

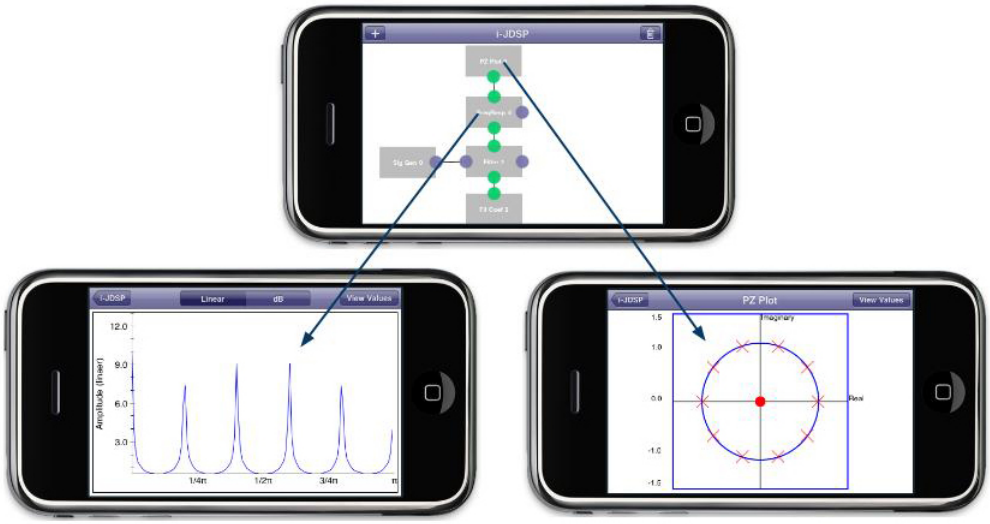


Figure 5.7: Simulation results of a tenth-order system. Lower left: frequency response. Lower right: pole-zero plot.

5.2 User Experiment and Evaluation

5.2.1 User Experiment Design

The capabilities of the i-JDSP environment allow students to experiment with basic DSP concepts on their mobile devices without the need of desktops and Internet connectivity. The user experiment was designed in order to evaluate the usability and merits of the new mobile interface in comparison to the existing desktop version of J-DSP. Furthermore, the overall student interest in using iOS devices to perform laboratories is assessed. The experiment was conducted on a group of student volunteers from the EEE 407 undergraduate DSP class. The survey was comprised of two components: choice response statements and open response statements. Participants were first asked to play with the application for a while and complete the first half of the choice response statements which collects the general usage information about the i-JDSP environment. Then the participants were asked to solve two simple lab exercises using i-JDSP and finish the exercise questionnaire in the second half of the choice response statements. Finally, since the group of students had previous J-DSP lab experience, the open re-

sponse statements part focused on the comparison between the two environments.

5.2.2 Response Results and Analysis

There were 36 participants in the assessment. The results regarding the general usage information from the first half of the choice response statements are as follows:

Table 5.1: Response results on general usage information.

Questions	**Response Percentage (%)**
Do you like the idea of an iPhone/iPad-based DSP app such as i-JDSP?	
Yes	97%
No	3%
Do you think it is useful to do DSP simulations on the iPad/iPhone?	
Strongly agree	25%
Agree	70%
Neutral	5%
Disagree	0%
Strongly disagree	0%
If you have used J-DSP, can you relate your previous experience to i-JDSP easily?	
Yes	100%
No	0%
How long did it take to get familiar with the basics of the i-JDSP environment?	
5 minutes or less	89%
Half an hour	11%
An hour or more	0%
The menu in i-JDSP is intuitive and easy to navigate through?	
Very easy	47%
Easy	33%
Neutral	14%
Difficult	6%
Very difficult	0%
Will you install the app if it is available free in the app store?	
Definitely yes!	47%
Yes	47%
Neutral	6%
No	0%
No way!	0%

The general usage information questionnaire consists of a special set of questions that assess the ergonomics and the intuitiveness of the i-JDSP application. From Table, almost all the students responded that they liked the idea of i-JDSP. 95% of the students agreed that it is useful to DSP simulations on the iPad/iPhone. For the participants who have used J-DSP before, they can easily relate their previous experience to the new environment and 89% spent less than 5 minutes to get familiar with the application. Furthermore, 80% found it easy to navigate the application using the menu in i-JDSP while only 6% found it difficult. The responses also indicate most of the participants are inclined to install the application on their own devices when i-JDSP is made available in the app store.

The questionnaire of the second part of the assessment is focused on evaluating effectiveness and usability when the participants are asked to solve specific exercise problems. The response results are revealed in Table 5.2.

The student responses revealed that the GUI in i-JDSP were received positively. Most of the students found working with the blocks using the multi-touch technology easy and intuitive even for first time users. In fact, 91% of the students agreed that setting up the simulation on the iPad/iPhone for the exercise was pretty easy. After doing the exercise using i-JDSP, 92% of the students responded that they felt more comfortable checking the relationship of the location of poles and zeros with the frequency response of a system. Moreover, there were 36% of the students found out something they didn't know before when doing the exercise on i-JDSP.

Table 5.2: Response results on execise information.

Questions	**Response Percentage (%)**
Working with the blocks on iPad/iPhone (moving, editing, connecting and deleting) was easy?	
Very easy	47%
Easy	44%
Neutral	6%
Difficult	3%
Very difficult	0%
Setting up the simulation on the iPad/iPhone for the exercise was pretty easy?	
Very easy	58%
Easy	33%
Neutral	6%
Difficult	3%
Very difficult	0%
You feel more comfortable checking the relationship of the location of poles and zeros with the frequency response of a system?	
Strongly agree	42%
Agree	50%
Neutral	8%
Disagree	0%
Strongly disagree	0%
When doing the exercise, you found out something you didn't know before?	
Yes	36%
No	64%

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, I present my research on the design and implementation of i-JDSP – an interactive laboratory for digital signal processing in the iOS devices. An overview of the current trend in mobile computing and the advantages of using Apple iOS devices as tools in the field of digital signal processing are discussed. A description of the user interface design is given along with a more in depth explanation of the application's internal architecture. A set of functions are developed and discussed in detail to perform fundamental DSP laboratory exercises. A number of example simulations that can be used as part of a DSP course are demonstrated. Finally, a summary of assessment results of using i-JDSP among undergraduate students from EEE 407 is provided.

Being a mobile application, i-JDSP provides more accessibility with a very convenient all-graphical interface without the need of desktops or Internet connectivity. It can be ideal for university programs applying new trends in education and distance learning. The overall feedback from the students is positive and very encouraging. Students with previous J-DSP experience can easily adapt to the new platform and some of them think i-JDSP exceeds J-DSP in terms of accessibility and overall user experience.

6.2 Future Work

New function blocks can be easily added and integrated to the existing framework to cover more topics in DSP. Possible improvements can be made such that i-JDSP allows saving figures and to provide unlimited drawing area. Another possibility may be to use i-JDSP as a control hub that communicates with other hardware such as sensors and chips for real time signal collecting and processing. It seems that with the ongoing advancement in mobile technology, the further development of i-JDSP may produce a revolutionized portable tool, that can be used in education as well as industry.

REFERENCES

[1] H. Benko, A. D. Wilson, and P. Baudisch, "Precise selection techniques for multi-touch screens," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 1263–1272.

[2] D. Siewiorek, A. Smailagic, J. Furukawa, A. Krause, N. Moraveji, K. Reiger, J. Shaffer, and F. L. Wong, "Sensay: A context-aware mobile phone," in *Proceedings of the 7th IEEE International Symposium on Wearable Computers*, ser. ISWC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 248–.

[3] Nielsen Company, "U.S. smartphone battle heats up: Which is the "most desired" operating system?" http://blog.nielsen.com/nielsenwire/online_mobile/us-smartphone-battle-heats-up, December 2010, [Online; accessed 15-March-2011].

[4] Apple Inc., "iOS4: The world's most advanced mobile operating system," http://www.apple.com/iphone/ios4/, 2011, [Online; accessed 15-March-2011].

[5] ——, "Develop for iOS," http://developer.apple.com/technologies/ios/, 2011, [Online; accessed 15-March-2011].

[6] ——, "App Store," http://www.apple.com/iphone/features/app-store.html, 2011, [Online; accessed 15-March-2011].

[7] ——, "Apple special event 2010," http://www.apple.com/apple-events/september-2010/, September 2010, [Online; accessed 15-March-2011].

[8] P. Anderson and A. Blackwood, "Mobile and PDA technologies and their future use in education," JISC Technology and Standards Watch, Tech. Rep., November 2004.

[9] A. Holzinger, A. Nischelwitzer, and M. Meisenberger, "Mobile phones as a challenge for M-Learning: Examples for mobile interactive learning objects (MILOs)," *Pervasive Computing and Communications Workshops, IEEE International Conference on*, vol. 0, pp. 307–311, 2005.

[10] G. De Leo and G. Leroy, "Smartphones to facilitate communication and improve social skills of children with severe autism spectrum disorder: special education teachers as proxies," in *Proceedings of the 7th international conference on Interaction design and children*, ser. IDC '08. New York, NY, USA: ACM, 2008, pp. 45–48.

[11] Pearson Education Inc., "Algebraprep," http://www.pearsonhighered.com/AlgebraPrep/, 2010, [Online; accessed 15-March-2011].

[12] Element Collection Inc., "The elements," http://periodictable.com/ipad/, 2010, [Online; accessed 15-March-2011].

[13] MathWorks Inc., "Matlab mobile," http://www.mathworks.com/mobile/, 2010, [Online; accessed 15-March-2011].

[14] A. Spanias, *Digital Signal Processing: An Interactive Approach*. Lulu Publishers, 2007.

[15] A. Spanias and V. Atti, "Interactive online undergraduate laboratories using J-DSP," *Education, IEEE Transactions on*, vol. 48, no. 4, pp. 735 – 749, 2005.

[16] A. Clausen, A. Spanias, A. Xavier, and MayaTampi, "A Java signal analysis tool for signal processing experiments," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, May 1998, pp. 1849 –1852 vol.3.

[17] A. Spanias, S. Urban, A. Constantinou, M. Tampi, A. Clausen, X. Zhang, J. Foutz, and G. Stylianou, "Development and evaluation of a web-based signal and speech processing laboratory for distance learning," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, vol. 6, 2000, pp. 3534 –3537.

[18] V. Atti and A. Spanias, "On-line simulation modules for teaching speech and audio compression techniques," *Frontiers in Education, Annual*, vol. 1, pp. T4E–22, 2003.

[19] M. Yasin and L. Spanias, "On-line laboratories for image and two-dimensional signal processing using 2D J-DSP," in *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, vol. 3, 2003, pp. III – 785–8 vol.3.

[20] T. Thrasyvoulou, K. Tsakalis, and A. Spanias, "J-DSP-C, a control systems simulation environement: labs and assessment," in *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, vol. 1, 2003, p. T4E11T4E16.

[21] A. Spanias, "Teaching genomics and bioinformatics to undergraduates using J-DSP," in *ASEE Annual Conference and Exposition*, June 2004.

[22] M. Zaman, A. Suppappola, and A. Spanias, "Advanced concepts in time-frequency signal processing made simple," in *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, vol. 1, 2003, pp. T2E – 10–15 Vol.1.

[23] Y. Ko, T. Duman, and A. Spanias, "On-line laboratory for communication systems using J-DSP," in *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, vol. 1, 2003, pp. T3E–13 – T3E–18 Vol.1.

[24] A. Spanias, K. Ramamurthy, and L. Hinnov, "Workshop - designing multidisciplinary signal and data analysis laboratories using Java-DSP," in *Frontiers in Education Conference, 2009. FIE '09. 39th IEEE*, 2009, pp. 1 –2.

[25] R. Santucci, T. Gupta, M. Shah, and A. Spanias, "Advanced functions of Java-DSP for use in electrical and computer engineering courses," in *ASEE Annual Conference and Exposition*, June 2010.

[26] A. Spanias, V. Berisha, H. M. Kwon, C.-W. Huang, A. Natarajan, and R. Ferzli, "Using the Java-DSP real-time hardware interface in undergraduate classes," in *Frontiers in Education Conference, 36th Annual*, 2006, pp. 12 –17.

[27] H. Kwon, V. Berisha, V. Atti, and A. Spanias, "Experiments with sensor motes and Java-DSP," *Education, IEEE Transactions on*, vol. 52, no. 2, pp. 257 –262, May 2009.

[28] A. Spanias, "Java-DSP interface with MATLAB and its use in engineering education," in *ASEE Annual Conference and Exposition*, June 2004.

[29] ——, "Using JDSP and LabView to perform undergraduate labs," in *ASEE Annual Conference and Exposition*, June 2007.

[30] Oracle Corporation, "Java Applets," http://java.sun.com/applets/, July 2010, [Online; accessed 15-March-2011].

[31] ——, "The Java Virtual Machine specification," http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, December 2010, [Online; accessed 15-March-2011].

[32] D. Norman, J. Miller, and A. Henderson, "What you see, some of what's in the future, and how we go about doing it: Hi at apple computer," in *Conference companion on Human factors in computing systems*, ser. CHI '95. New York, NY, USA: ACM, 1995, pp. 155–.

[33] M. Hassenzahl and N. Tractinsky, "User experience - a research agenda," *Behaviour & Information Technology*, vol. 25, no. 2, pp. 91–97, Jan. 2006.

[34] D. Barnard, "The magic trackpad is just a better mouse," http://davidbarnard.com/post/875415140/the-magic-trackpad-a-better-mouse, December 2010, [Online; accessed 15-March-2011].

[35] Wikipedia, "Defensive programming," http://en.wikipedia.org/wiki/Defensive_programming, January 2011, [Online; accessed 15-March-2011].

[36] Apple Inc., "Xcode 4," http://developer.apple.com/xcode/, September 2010, [Online; accessed 15-March-2011].

[37] ——, "iOS Cocoa Touch," http://developer.apple.com/technologies/ios/cocoa-touch.html, 2011, [Online; accessed 22-March-2011].

[38] J. A. Jacko, *The human-computer interaction handbook: fundamentals, evolving technologies, and emerging applications*. Lawrence Erlbaum Associates, 2008, pp. 64–67.

[39] A. Hillegass, *Cocoa Programming For Mac OS X*. Addison Wesley, 2008, p. 4.

[40] Apple Inc., "Delegation and the Cocoa frameworks," http://developer.apple.com/library/ios/#documentation/general/conceptual/DevPedia-CocoaCore/Delegation.html, 2011, [Online; accessed 22-March-2011].

[41] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, May 1998, pp. 1381 –1384 vol.3.

[42] Core Plot, "Cocoa plotting framework for Mac OS X and iOS," http://code.google.com/p/core-plot/, 2011, [Online; accessed 22-March-2011].

[43] A. Spanias, *Digital Signal Processing: An Interactive Approach*. Lulu Publishers, 2007, p. 74.

APPENDIX A

ADDING NEW FUNCTION BLOCKS TO I-JDSP

i-JDSP is designed in the way that multiple developers can work together at the same time with no intervention in i-JDSP core structure. Adding new function blocks to the existing function set in i-JDSP is simple. It is simply to implement the function algorithm and tell i-JDSP how it should look and behave. Thus, developers can focus on their own related function blocks while not knowing how i-JDSP works internally. Basic steps to be followed to add a new function block are:

- Create the function class as a model
- Generate the associated view controller (if needed)
- Tell i-JDSP how the function block should be drawn
- Register the function block in i-JDSP

A.1 Creating the function class as a model

All function blocks to be added should be implemented as a model class inherits from the Part class. The class interface in the header file (NewPart.h) should have a similar outline as below:

```
1 @interface NewPart : Part {
2         iVars;  // declare NewPart's instance variables
3 }
4 @property iVars;  // property for iVars
5 - (void)publicMethods;  // declare public methods
6 @end
7 @interface NewPart ()
8 - (void)privateMethods;  // declare private methods
9 @end
```

After that, implement the algorithm of the new block in the implementation file (NewPart.m). If the algorithm to be written is too complicated or lengthy, it can be implemented as a separate class. If certain functions required may need to be used by more than one blocks then is better to include them in a separate class. There are no

specific rules to dictate how to implement each algorithm or function. But basic outline of the implementation should look like something as follows:

```
1  @implementation NewPart
2  @synthesize iVars;
3
4  #pragma mark -
5  #pragma mark init methods
6  -(id)init{
7          self = [super init];
8          if (self) {
9                  // init iVars if needed
10                 }
11         }
12         return self;
13 }
14 #pragma mark -
15 #pragma mark public methods
16 - (void)publicMethods{
17 }
18 - (void)executeBlock
19 {
20         [self someMethod]; // call the function/algorithm the block
               performs
21         [self setSignalType:sigtype withIndex:index]; // set the
               signal type of the output
22         [self setSignalSize:sigsize withIndex:index]; // set the
               signal size of the output
23         [self setData1:data withIndex:index]; // set the data
               buffers of the output
24         [self updateBlock]; // pass data specified above to its
               child blocks
25 }
26 - (void)dealloc
```

```
27 {
28         [iVars release];  // release iVars if needed
29         [super dealloc];
30 }
31
32 #pragma mark -
33 #pragma mark private methods
34 - (void)privateMethods{
35 }
36 @end
```

A.2 Generating the associated view controller (if needed)

When a new function block needs an interface for changing the parameters or viewing results graphically, a view controller of that function should be created. There are no specific rules for creating the view controller. However, if the view controller is used to change or update the parameters of a function block, it must extend the PartDetailViewController and conform to the AttributeSavedDelegate protocol. An example of the FFTDetailViewController is listed below:

```
1 // FFTDetailViewController.h
2 #import <UIKit/UIKit.h>
3 #import "AttributeEditor.h"
4 #import "PartDetailViewController.h"
5 @class FFT;
6 @class PickerAttributeEditor;
7 @interface FFTDetailViewController : PartDetailViewController <
      AttributeSavedDelegate> {
8         NSArray* rowLabels;
9         NSArray* rowKeys;
10        NSMutableDictionary* params;  // stores parameters
11
12        FFT* fft;  // FFT class instance
```

```
13          PickerAttributeEditor* pickerAttributeEditor;  // view
                controller for changing the value
14 }
15 @property (nonatomic, retain) NSArray* rowLabels;
16 @property (nonatomic, retain) NSArray* rowKeys;
17 @property (nonatomic, retain) NSMutableDictionary* params;
18 @property (nonatomic, retain) PickerAttributeEditor*
       pickerAttributeEditor;
19 @property (nonatomic, retain) FFT* fft;
20 - (void)passInParams:(FFT*)fft;
21 @end
```

```
1 // FFTDetailViewController.m
2 #import "FFTDetailViewController.h"
3 #import "FFT.h"
4 #import "PickerAttributeEditor.h"
5 #import "PartsController.h"
6 @implementation FFTDetailViewController
7 @synthesize rowLabels, rowKeys, params;
8 @synthesize fft, pickerAttributeEditor;
9
10 // AttributeSavedDelegate Method
11 - (void)saved:(NSString*)value forKey:(NSString*)key
12 {
13          [params setValue:value forKey:key];
14          [self.tableView reloadData];
15 }
16 - (void)passInParams:(FFT *)fft
17 {
18          NSArray* keys = [NSArray arrayWithObjects:@"fft_size", nil];
19          NSArray* objects = [NSArray arrayWithObjects:[NSString
                stringWithFormat:@"%d", fft.fft_size], nil];
```

```
20          self.params = [NSMutableDictionary dictionaryWithObjects:
                objects forKeys:keys];
21 }
22
23 #pragma mark -
24 #pragma mark View lifecycle
25 - (void)viewDidLoad {
26                  self.title = @"FFT";
27          rowLabels = [[NSArray alloc] initWithObjects:@"FFT Size",nil
                ];
28          rowKeys = [[NSArray alloc] initWithObjects:@"fft_size",nil];
29      [super viewDidLoad];
30 }
31
32 // add part to rootviewcontroller
33 - (void)addPart
34 {
35          FFT* _fft = [[FFT alloc] init];
36          _fft.fft_size = [[params objectForKey:@"fft_size"] intValue];
37          int index = [[PartsController sharedInstance].partsDic count
                ];
38          NSString* partNameWithIndex = [NSString stringWithFormat:@"
                FFT %d", index];
39          _fft.partName = partNameWithIndex;
40          [[PartsController sharedInstance].partsDic setObject:_fft
                forKey:_fft.partName];
41
42          [delegate partAdded:_fft.partName]; // notify delegate
43          [_fft release];
44 }
45
46 - (void)updatePart
47 {
```

```
48          FFT* fft = [[PartsController sharedInstance].partsDic
                objectForKey:partNameToUpdate];
49          fft.fft_size = [[params objectForKey:@"fft_size"] intValue];
50          [fft executeBlock];
51
52          [self.navigationController popToRootViewControllerAnimated:
                YES];
53  }
54
55  #pragma mark -
56  #pragma mark Table view data source
57  - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
58      // Return the number of sections.
59      return 1;
60  }
61
62  - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection
        :(NSInteger)section {
63      // Return the number of rows in the section.
64      return 1;
65  }
66
67  // Customize the appearance of table view cells.
68  - (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
69
70      static NSString *CellIdentifier = @"Cell";
71
72      UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:CellIdentifier];
73      if (cell == nil) {
74          cell = [[[UITableViewCell alloc] initWithStyle:
                UITableViewCellStyleValue2 reuseIdentifier:CellIdentifier
```

```
            ] autorelease];
75      }
76      // Configure the cell...
77          NSUInteger row = [indexPath row];
78          NSString* rowKey = [rowKeys objectAtIndex:row];
79          NSString* rowLabel = [rowLabels objectAtIndex:row];
80          cell.textLabel.text = rowLabel;
81          cell.detailTextLabel.text = [[params valueForKey:rowKey]
                description];
82
83      return cell;
84  }
85
86  #pragma mark -
87  #pragma mark Table view delegate
88  - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(
        NSIndexPath *)indexPath {
89      // Navigation logic may go here. Create and push another view
            controller.
90          NSUInteger row = [indexPath row];
91          NSString* rowLabel = [rowLabels objectAtIndex:row];
92          NSString* rowKey = [rowKeys objectAtIndex:row];
93
94          self.pickerAttributeEditor = [[PickerAttributeEditor alloc]
                initWithStyle:UITableViewStyleGrouped];
95          self.pickerAttributeEditor.delegate = self;
96          self.pickerAttributeEditor.keyPath = rowKey;
97          self.pickerAttributeEditor.keyValue = [[params valueForKey:
                rowKey] description];
98          NSArray* possibleData = [[NSArray alloc] initWithObjects:@"8"
                , @"16", @"32", @"64", @"128", @"256", nil];
99          self.pickerAttributeEditor.pickerData = possibleData;
100         [possibleData release];
```

```
101         self.pickerAttributeEditor.title =rowLabel;
102         [self.navigationController pushViewController:self.
                pickerAttributeEditor animated:YES];
103 }
104
105 #pragma mark -
106 #pragma mark Memory management
107 - (void)dealloc {
108         [rowKeys release];
109         [rowLabels release];
110         [fft release];
111         [pickerAttributeEditor release];
112         [params release];
113         [super dealloc];
114 }
115 @end
```

A.3 Telling i-JDSP how the function block should be drawn

For each newly added function block, i-JDSP needs to know how to draw the view representing the block such as how many pins are needed. First, tell i-JDSP how many pins are associated with the new block. In PartView.m, add the following to the existing code:

```
1 - (id)initWithFrame:(CGRect)frame andTitle:(NSString*)title {
2         ...
3         // Add pins
4         ...
5         else if([partClassName isEqualToString:@"NewPart"])
6         {
7                 pinNums = [NSArray arrayWithObjects:@"0", @"2", @"3",
                        nil];  // added necessary pins
8         }
9         ...
```

```
10 }
```

Second, find the code that handles the user touch input and add the necessary code as follows so that i-JDSP knows how to perform when the user double taps the block. Still in PartView.m, add the following code:

```
1 - (void)handleEventsFromTouches:(NSSet*)touchs{
2         // ...
3         else if([className isEqualToString:@"NewPart"])
4                 {
5                 // perform the necessary action e.g. bring up the
                      view controller etc.
6                 }
7         // ...
8 }
```

A.4 Registering the function block in i-JDSP

Finally, the developer needs to register the new function block in i-JDSP. First of all, import the header file of the new block in PartList.h, which maintains the whole listing of the available function blocks. Secondly, add the following codes to PartsTableView-Controller.m so that the new block will appear when the user taps the "+" button on the navigation bar.

```
1 - (void)viewDidLoad {
2         // ...
3         NSArray* arr = [[NSArray alloc]
4                                 initWithObjects: ...,
5                                 @"NewPart",
6                                 nil];
7         // ...
8 }
9 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(
      NSIndexPath *)indexPath {
```

```
10          // ...
11          if (part == @"NewPart") {
12                  // perform the necessary action when the user selects
                        the         NewPart
13          }
14          // ...
15  }
```

APPENDIX B

I-JDSP LABORATORY ON DIGITAL FILTER BASICS

In this section, we introduce our first i-JDSP laboratory exercise. The purpose of this exercise is to familiarize the readers with i-JDSP and to demonstrate how this iOS-based simulation environment can be used to obtain hands-on experiences with discrete-time signals and digital filters. It is important for the students to read this section carefully and to go through the initial steps in order to become familiar with i-JDSP.

B.1 General information on i-JDSP

This book provides a series of laboratory exercises on several DSP topics. The laboratories are based on an object-oriented iOS application called i-JDSP. i-JDSP was developed at Arizona State University (ASU) and is written as a native Cocoa application that can be installed on any iOS device with iOS version 3.2 and above. i-JDSP has a rich suite of signal processing functions that include signal generators, digital filters, pole-zero root computation, filter design algorithms, an FFT, plot functions, a sound player, and many others. The software provides a user-friendly programming environment that embodies iOS devices' graphical capabilities. Its highly intuitive graphical user interface (GUI) is easy to learn. Selecting and establishing individual blocks can be done by a drag-and-drop process. Each block is linked to software that performs a specific signal processing task. Block parameters can be edited through dialog windows and simulation results can be plotted. Figure B.1 shows the i-JDSP environment. By connecting blocks, signal flow is established and an algorithm can be simulated. Signals at any point of a simulation can be visualized through the appropriate plot blocks. Blocks can be manipulated (i.e. edit, move, delete and connect) using multi-touch technologies. System execution is dynamic, i.e., any change at any point within a system will automatically take effect in all subsequent blocks.

B.2 Getting started - generating and analyzing signals

The easiest way to explain some of the functions of i-JDSP is to work through a simple example. Open the i-JDSP application. When the welcome page appears press

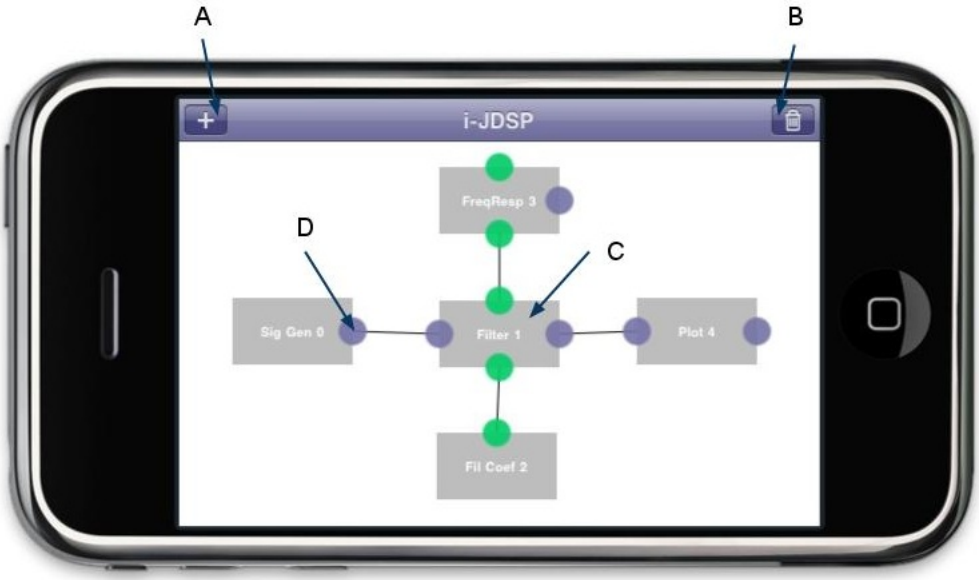


Figure B.1: i-JDSP simulation environment. In the figure, 'A': Add button (to add function blocks); 'B': Trash button (to delete all the blocks); 'C': Simulation diagram; 'D': Connection pin.

"Start. The i-JDSP environment appears and is ready for programming a DSP simulation. Please read the instructions on the screen.

To program a simulation that involves a signal, a filter and a plot block, proceed with the following steps. Press the "+" button on the left corner of the navigation bar. When the function list appears, select the *SignalGenerator*. A new parameter list of the signal generator block should appear. Tap the "Add" button on the right corner of the navigation bar to use the default settings for the block. Note that the signal generator block is established. Similarly, create a *Filter* and a *Plot* block. Note that each block has signal input(s) denoted by the small blue dot(s) on its left side and signal output(s) on its right side. Some blocks carry parameter inputs and parameter outputs denoted by green dot(s) at their bottom and top respectively. For example, the Filter block has a coefficient input at the bottom and a coefficient output at the top of the block. Parameter inputs are used to interface and run functions such as filter design, frequency response, pole-zero plot, etc.

You can then move the established block by placing your finger over it and drag it to a new location. To delete a block, place your finger over the block and hold for 3 seconds. When the pop-up menu appears, select “Delete”. To link blocks, tap once inside the blue dot on the right side of the signal generator box, then tap once the blue dot one the left side of the filter box. A line will be drawn when the connection is successful. Always make the connections in the direction of the signal flow. The $FilterCoeff.$ block is used to specify filter coefficients. The block is connected to the *Filter* block parameter input as shown in Figure B.2. Now, connect the *Filter* block to the *Plot* block so that your screen looks like the block diagram shown in Figure B.2. Note that you can view the dialog box (if available) of each block by double tapping on the block.

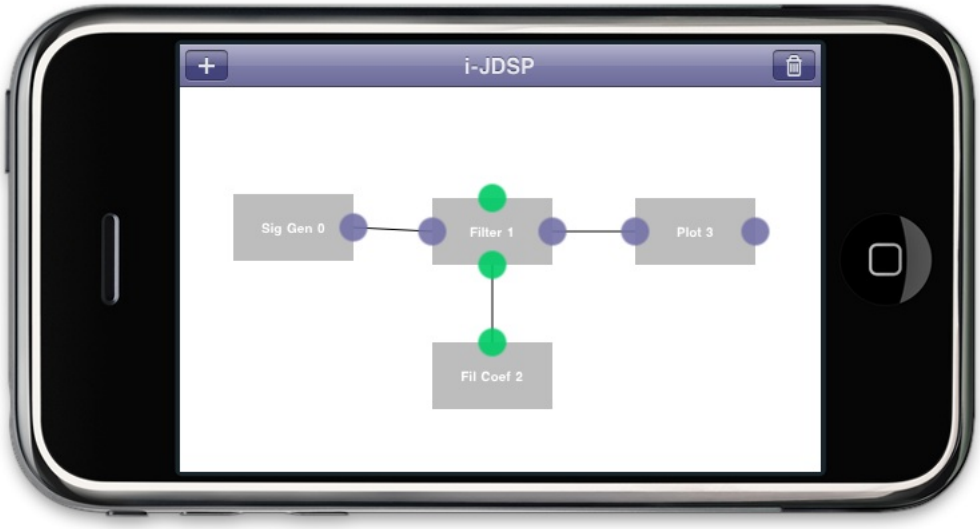


Figure B.2: Lab-1 working with i-JDSP.

B.3 Generating signals

Let us now form a signal using the signal generator. Double click inside the *Sig.Gen.* box and a dialog window will appear.

You may change the “Gain, the “Pulsewidth, the “Period and the “Time Shift by selecting the entry and typing the desired value in the following dialog (Figure



Figure B.3: The signal generator (*Sig.Gen.*) dialog window in i-JDSP.



Figure B.4: The editing dialog window in i-JDSP.

B.4). The signal type can be changed by selecting on the picker menu and selecting a signal (Figure B.5). It is important to note that when finishing editing the new value, remember to tap the ”Save” button on the right corner of the navigation bar. That will brings you back to the view in Figure B.4, after you are done with editing all the parameters of that particular box, tap the ”Update” button on the right corner of the

navigation bar, otherwise, the new parameter sets will not be saved.



Figure B.5: The picker menu for selecting different signal types in i-JDSP.

For all the signals, except audio, i-JDSP assumes a normalized sampling frequency of 1Hz. Hence, the sampling frequency in terms of radians is 2π. All frequencies are entered as a function of π, e.g., 0.1π, 0.356π, etc. Note that any sinusoidal frequency at or above π will result in aliasing. The following steps will guide you through simulations with sinusoids.

Step 1.1: Create a sinusoid with Frequency 0.1π, "Gain 3.75, and "Pulsewidth 40. When all of the parameters have been entered, press the "Update" button to update the signal. Remember that whenever changes are made to this box, the "Update" button must be pressed in order for the changes to take effect. Now back to the main i-JDSP environment, double tap the *Plot* box to view the signal. Count the number of samples within a period. How many samples do you have within a period? (ans: 20 samples).

Step 1.2: Create a sinusoid with "Frequency π, Gain 3.75, "Pulsewidth 40 (remember to press "Update" for changes to take place). Double tap the *Plot* box. What happens? (ans: no signal due to aliasing, i.e., the signal has been sampled precisely at the zero crossings).

Step 1.3: Create a sinusoid with "Frequency 1.3π, "Gain 3.75, "Pulsewidth 40. What happens? Double tap the *Plot* box. Count the number of samples within a period. (ans: we have aliasing again; sinusoidal frequencies must be chosen to be smaller than to avoid aliasing).

B.4 Performing simulations

Next, we want to perform a digital filter simulation. Establish the $FilterCoeff.$ block and connect it to the bottom parameter input of the Filter block. Double click on the $FilterCoeff.$ block and note that it has a default coefficient of $b_0 = 1$; the rest of the coefficients are zero. Hence the default gain of the filter is unity. Navigate back to the main environment. Open again *Sig.Gen.*, and set the signal values as per Step 1.1. Double-click the *Plot* block and a new dialog window will appear. You should again see the same input signal as in Step 1.1 because the filter is letting the signal pass through unaffected. This is because all the coefficients are zero except $b_0 = 1$. Note that if you press the "View Values" button on the right, a table with the values of the signal appears. Navigate back and continue.

Step 2.1: Let us now see the filter in action. Double click the $FilterCoeff.$ block. You should see the coefficient panel as shown in Figure B.6. Note that the filter coefficients (b_0-b_{10} and a_0-a_{10}) correspond to the following difference equation:

$$y(n) = \sum_{l=0}^{L} b_l x(n-1) - \sum_{m=1}^{M} a_m y(n-m) \tag{B.1}$$

Step 2.2: Keep the values in *Sig.Gen.* as per Step 1.1. Change the filter coefficient to $b_0 = 4$, and press "Update". Observe the Plot block. You should see that the amplitude of the sinusoid has changed (ans: peak amplitude 4x3.75= 15).

Step 2.3: Implement a pure delay by setting b_5=1 and the rest of the coefficients (including b_0) to zero and press "Update". What happens to the sinusoid? (ans: the sinusoid is delayed).



Figure B.6: The coefficient panel.

Step 2.4: Implement a simple lowpass filter, set $b_0 = 0.2$ and $a_1 = -0.8$ (set the rest of the coefficients to zero and $a_0 = 1$) and press "Update". Use *Sig.Gen.* and generate a sinusoid with "Gain 1, "Frequency 0.1π, and "Pulsewidth 256. What do you observe? (ans: amplitude change and phase shift).

Step 2.5: Select the *FreqResp* block, and place it above the *Filter* block. Connect the parameter output to the *FreqResp* block. Double click the Freq-Resp block. You can choose to see the magnitude and phase responses of the filter. Change the coefficient to $a_1 = 0.8$ instead of $a_1 = -0.8$. What do you see in the frequency response and output panels? (ans: highpass filter frequency response and decrease in the time-domain amplitude).

B.5 Visualizing results graphically

To view the signal in the frequency domain, insert an *FFT* box between the *Filter* and the *Plot* box (you may need to delete the *Plot*, add *FFT* and another *Plot*) as shown in Figure B.7.

Step 3.1: Set the *Filter* parameters and input as per Step 2.4. Double click on the *FFT* block and change the "FFT size to 256 points and press "Update". Now

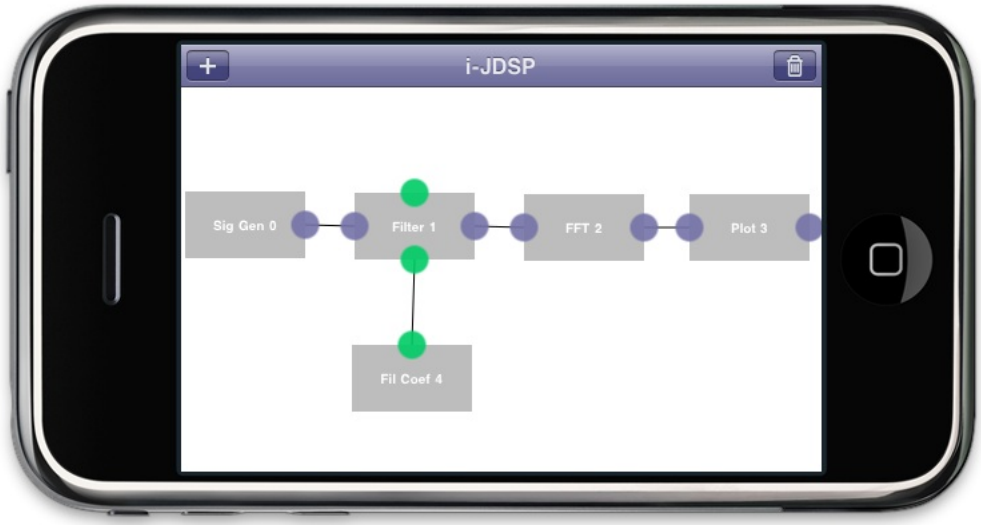


Figure B.7: A i-JDSP simulation diagram.

you can see the magnitude and the phase of the signal in the frequency domain. The magnitude has a sharp peak approximately at 0.31, i.e., the frequency of our sinusoidal signal is 0.1x3.1459 rads.

Step 3.2: Change the sinusoidal frequencies as per Step 1.2 and Step 1.3 but with "Pulsewidth 256, and observe the changes in the plot panel.

Step 3.3: Delete the filter. Set the sinusoidal Frequency in *Sig.Gen.* as per Step 1.1 but with "Pulsewidth 256. Now create a second *Sig.Gen.* block and an *Adder* (mixer) block. Your editor window should then look as in Figure B.8. Next, edit the second *Sig.Gen.* block. Open the dialog window and change the the signal type to "Random. Choose a "Variance of 4 and extend the "Pulsewidth to 256 samples in order to have noise over the full length of the signal. Now take a look at the output signal. In the time domain it is very hard to see that a sinusoid is present. However, if you view the signal in the frequency domain with an FFT of size 256, then you still find a peak at approximately 0.31.

Step 3.4: Change the "Gain of the sinusoid up or down and observe the spectra (FFT plot). Try different values in order to mask the sinusoid or to mask the noise.

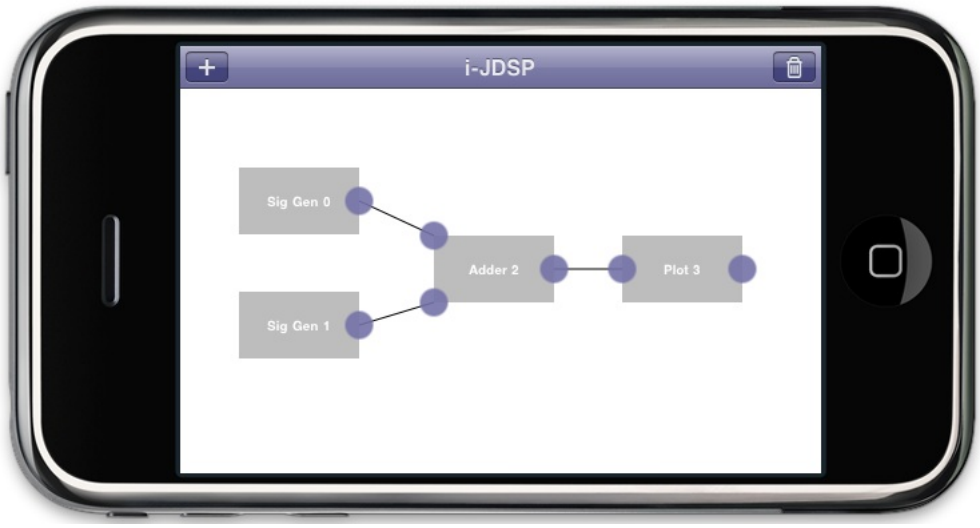


Figure B.8: Sinusoid plus noise simulation.

B.6 Spectral estimation of MIDI and DTMF signals

In this part of the experiment, we will use the FFT to measure the frequencies of music tones and telephone dual dialing tones from the MIDI and DTMF application standards respectively.

B.6.1 Working with the MIDI signal generator

MIDI, which stands for Musical Instrument Digital Interface, is a standard for communicating music information to computers, music synthesizers, video games, etc. i-JDSP has a simple MIDI tone generator with a compact keyboard GUI that generates a sinusoid of an appropriate frequency upon pressing a key. The frequencies of the sinusoids are consistent with standard music tones. We follow a step-by-step procedure to make frequency and period measurements for simple MIDI tones.

Step 4.1: Establish the *MIDI* block (Figure B.9). Double click the *MIDI* block to see the graphical interface that consists of a small piano-like keyboard. Connect the *MIDI* to a *Plot* block and then to an FFT block. Use a second *Plot* block to produce a frequency-domain graph. The i-JDSP block diagram is shown in Figure B.9. Double click on the FFT and set its length to 256 for all experiments. For the piano keys C, D,

E, F, G, A, B, observe the sinusoidal signal in the time domain and the corresponding spectral lines in the frequency domain. Note that the C key is the white key before the two black keys in the middle (i.e., the 7th white key). The music notes D through B correspond to the 6 white keys that follow. Press each of the keys, hear the sound, and observe the plots before and after the FFT. The sampling frequency is 8 kHz and hence one sample period is at 1/8000Hz = 0.125 mS. Note also that FFT lines are located at integer multiples of the frequency 8000/256 Hz. Use the "View Values" button in the *Plot* to find and measure the periods from the time domain plot (before FFT) and the music note frequencies from the peaks in the graph that follows the FFT. Record in a table, such as the one shown below (Table B.1), the measured periods and frequencies. Note that the period in seconds should theoretically correspond to the reciprocal of the frequency in Hz. If that is not the case with the measurements try to provide an explanation.

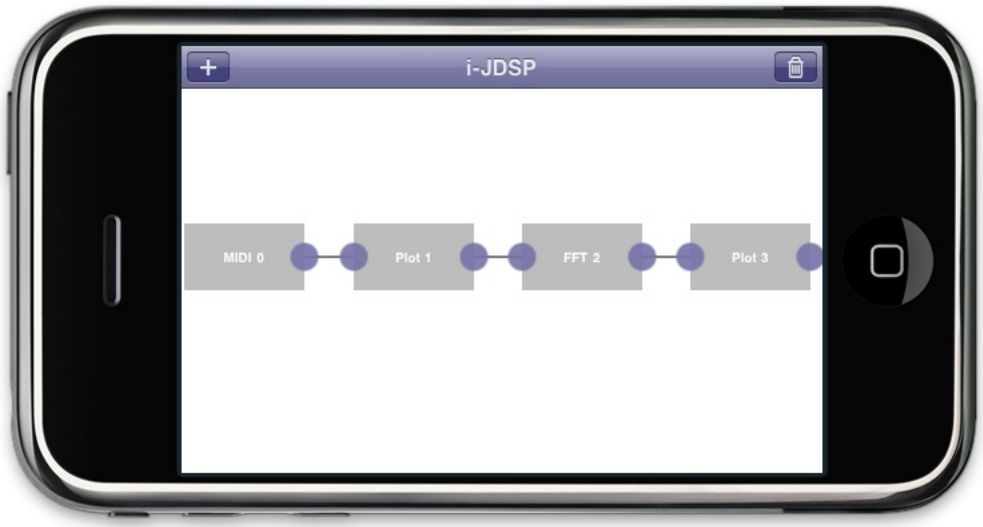


Figure B.9: The MIDI simulation diagram.

Find from the Internet the theoretical frequencies for the piano keys C, D, E, F, G, A, B, and compare. Hint: Use the "View Values" button to see precisely the indexes of the frequencies.

Table B.1: Table for recording measured MIDI tone periods and frequencies.

Note	Frequency (rad) 1	Frequency (Hz)	Period (samples)	Period (seconds)
C				
D				
...				

B.6.2 DTMF generator

Dual tone multi-frequency (DTMF) generation is part of every cellular and wireline telephone. The idea of DTMF is to encode telephone numbers in terms of dual tones that are transmitted to the telephone switch which detects them using an appropriate detection system. A dual tone consists of two frequency components that are generated upon dialing a number on the numeric keypad of any telephone. The two tone frequencies correspond to a row and column matrix arrangement. Since the DTMF standard has provisions for acoustic coupling, each combination of frequencies has been designed specifically to avoid confusion of dual tones with human voice. i-JDSP has a simple DTMF generator programmed as a block. The following task takes us through frequency measurements of DTMF waveforms.

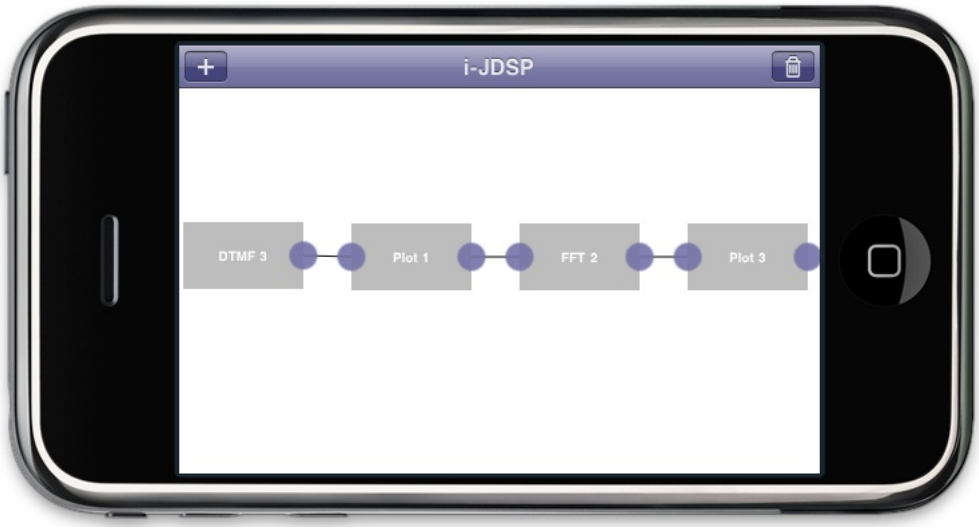


Figure B.10: The DTMF simulation diagram.

Step 4.2: Establish the *DTMF* block (Figure B.10). Use the same block diagram as in the previous MIDI simulation but with the *DTMF* block in place of the *MIDI*. By double clicking on the *DTMF* block you can see the GUI with a numeric keypad. Again use a 256-point FFT for all the experiments. For the DTMF keys 0, 1, 2, 3, , 9, *, #, measure the frequencies of the sinusoids (two tones per key) from the FFT plots. Press each of the keys, hear the sound, and observe the plot after the FFT. Measure the frequencies corresponding to each number. Construct a table to show these frequencies (fill in f1 and f2 for each digit). The sampling frequency is again 8 kHz. Give all the frequencies in Hz.

Table B.2: Table for recording measured DTMF waveform frequencies.

	Column 1	Column 2	Column 3
Row 1	1 (f1, f2)	2	3
Row 2	4	5	6
Row 3	7	8	9
Row 4	*	0	#

APPENDIX C

LABORATORY EXERCISES USING I-JDSP

C.1 Digital filter simulations

The rest of this section will demonstrate the use of the i-JDSP application when performing the laboratory exercises described in Chapter 2 from the *Digital Signal Processing An Interactive Approach* book written by Dr. Andreas Spanias. Screenshots of the simulations are provided for the exercises. Please refer to the book for detailed information on the exercises.

Exercise 1: Use i-JDSP to simulate a digital filter with the following coefficients $a_0 = 1$, $a_1 = 0.9$, and $b_0 = 1$.

Figure C.1 shows the digital filter simulation diagram for this exercise in i-JDSP.

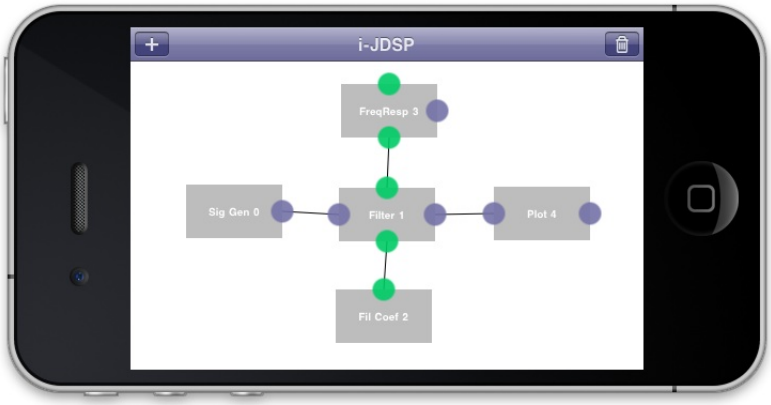


Figure C.1: Digital filter simulation diagram.

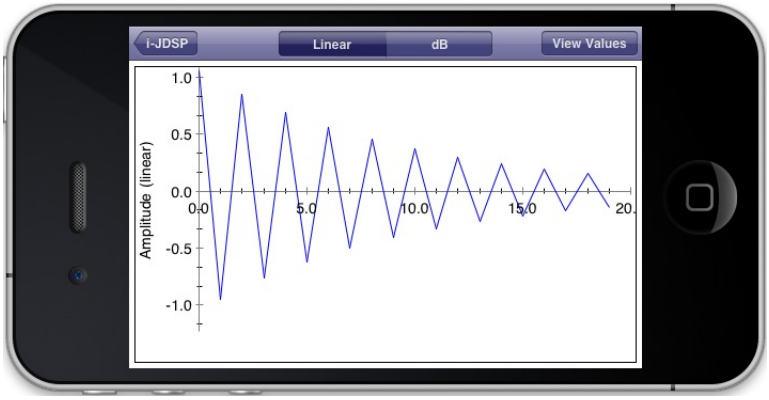


Figure C.2: Exercise 1 part a impulse response.

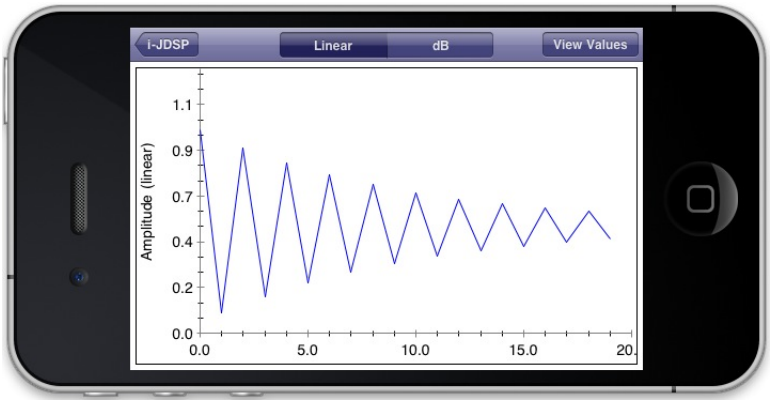


Figure C.3: Exercise 1 part b unit step response.

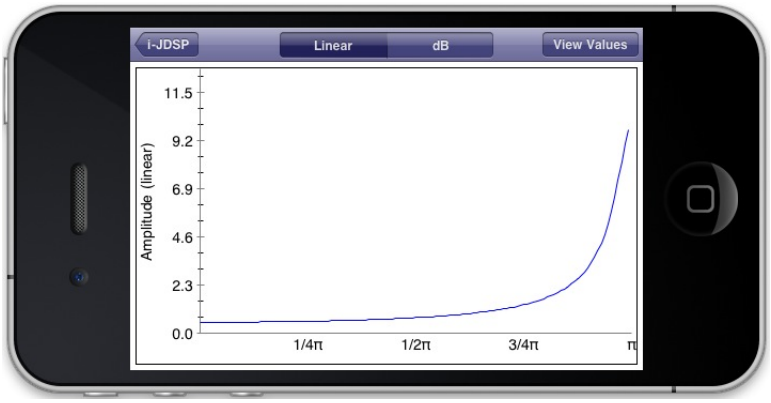


Figure C.4: Exercise 1 part c(1) frequency response (magnitude).

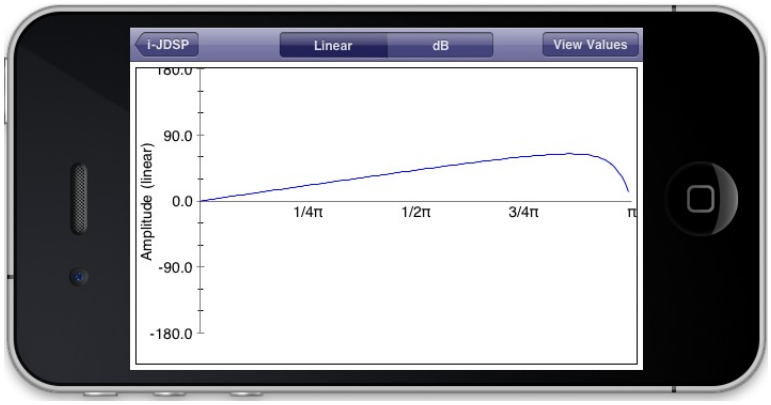


Figure C.5: Exercise 1 part c(2) frequency response (phase).

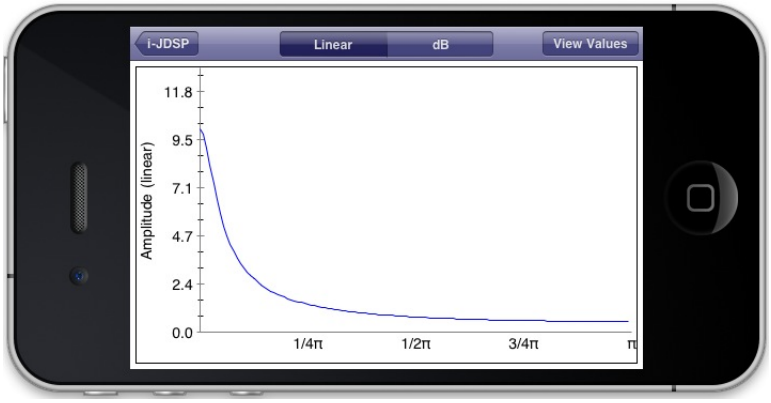


Figure C.6: Exercise 1 part d(1) frequency response (magnitude).

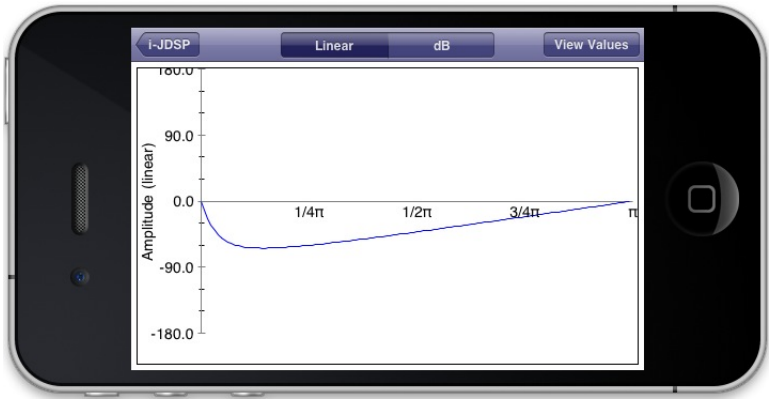


Figure C.7: Exercise 1 part d(2) frequency response (phase).

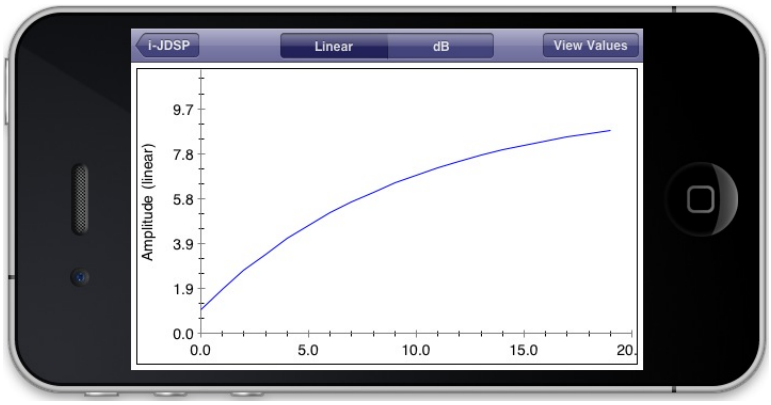


Figure C.8: Exercise 1 part e unit step response.

Exercise 2: Use i-JDSP to simulate a digital filter that has the following coefficients $a_0 = 1$, $a_1 = 0$, $a_2 = 0.8$, and $b_0 = 1$. Use the similar setting in Figure C.1 to establish the simulation.

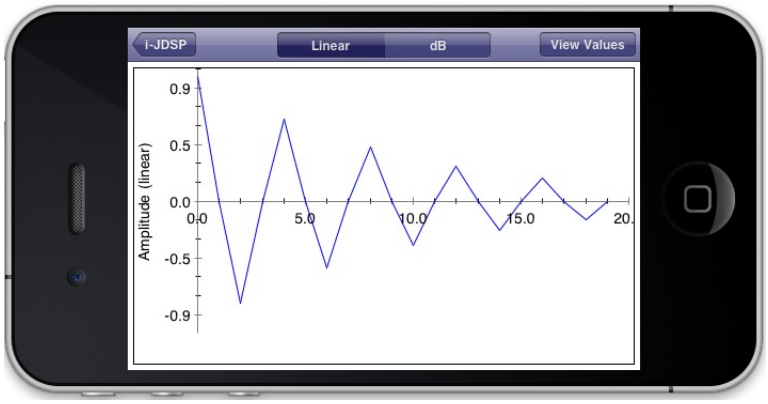


Figure C.9: Exercise 2 part a impulse response.

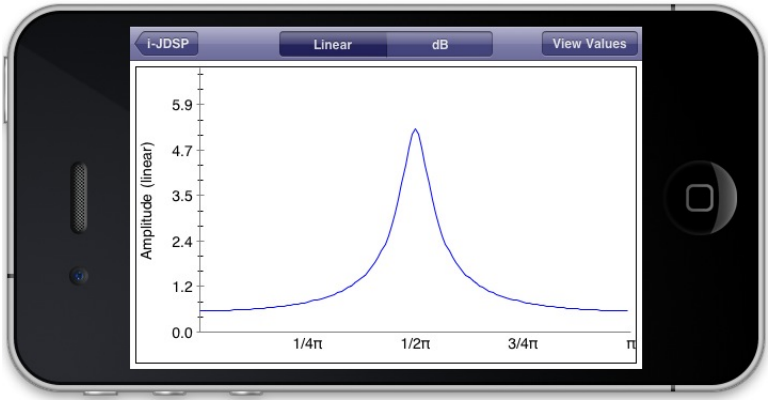


Figure C.10: Exercise 2 part b(1) frequency response (magnitude).

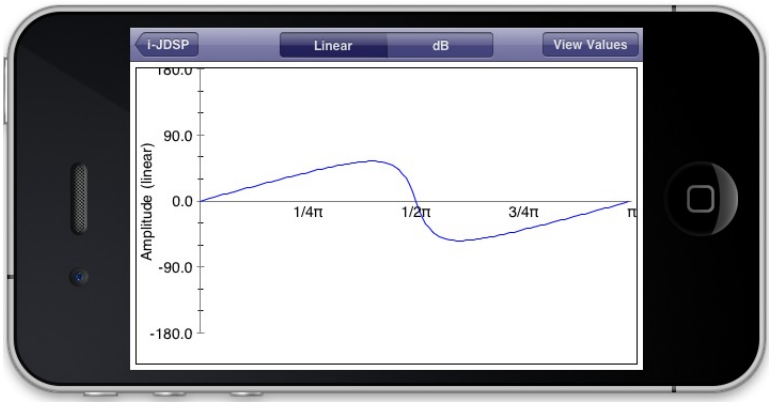


Figure C.11: Exercise 2 part b(2) frequency response (phase).

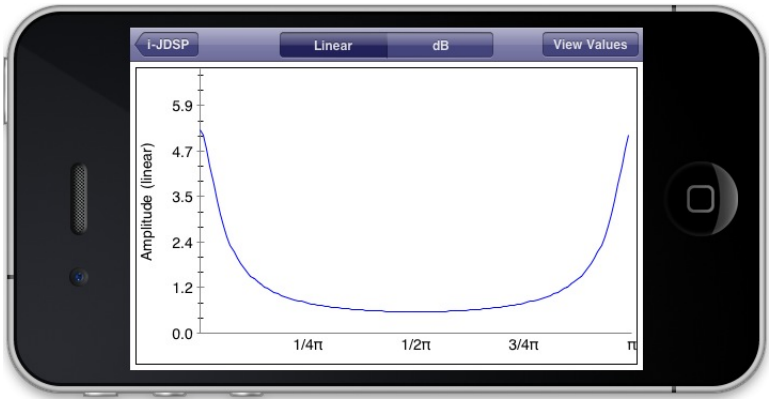


Figure C.12: Exercise 2 part c(1) frequency response (magnitude).

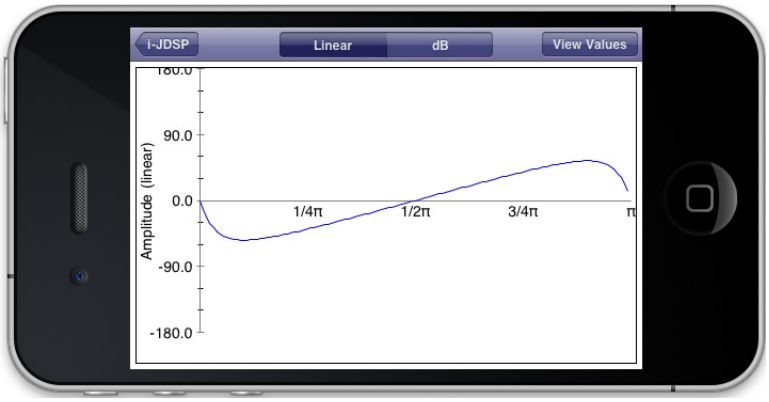


Figure C.13: Exercise 2 part c(2) frequency response (phase).

Exercise 3: Use i-JDSP to simulate a digital filter with the following coefficients $a_0 = 1$, $b_0 = 1$, and $b_1 = 1$.

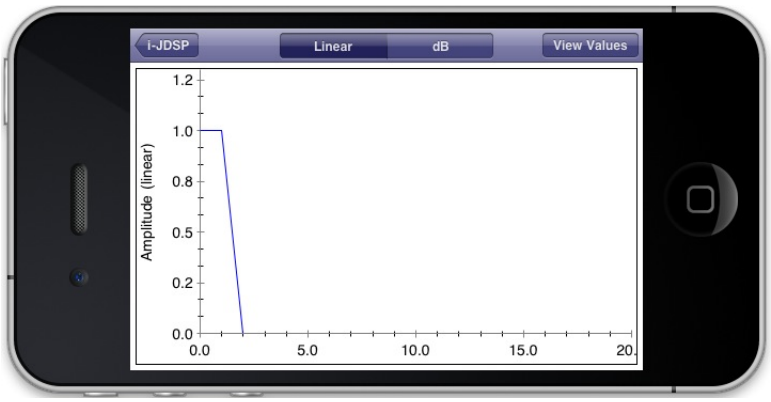


Figure C.14: Exercise 3 part a impulse response.

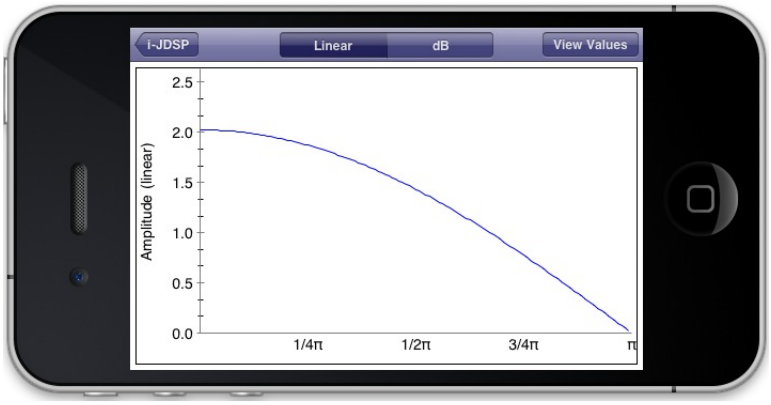


Figure C.15: Exercise 3 part b frequency response (magnitude).

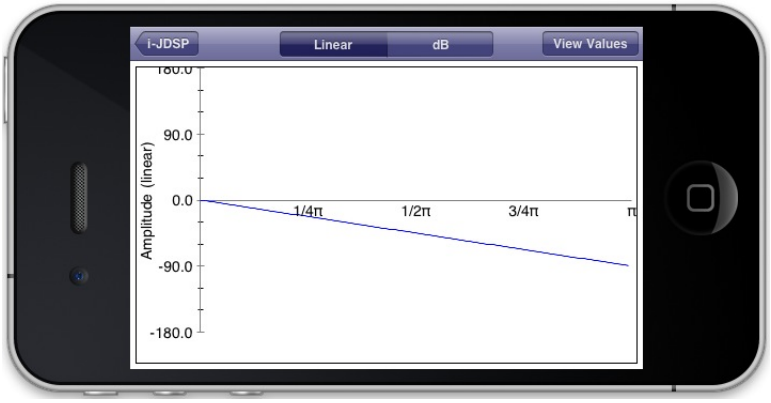


Figure C.16: Exercise 3 part b frequency response (phase).

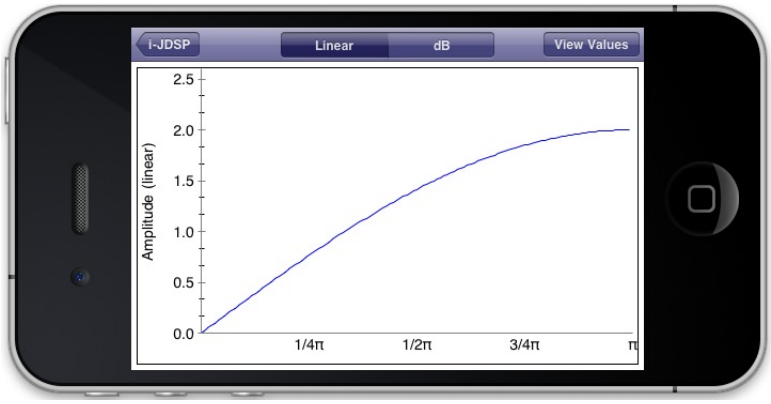


Figure C.17: Exercise 3 part c frequency response (magnitude).

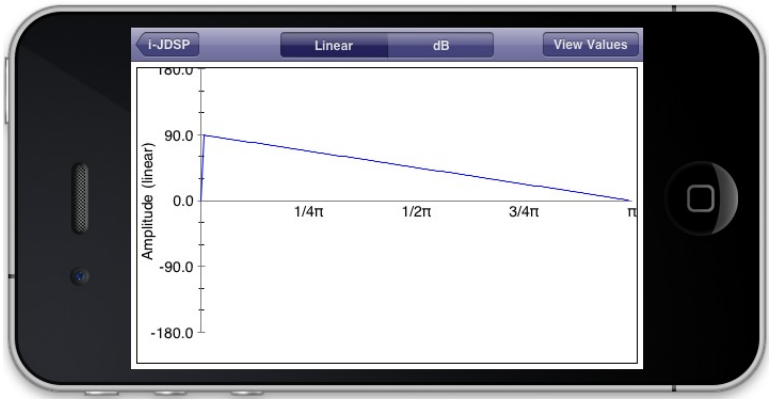


Figure C.18: Exercise 3 part c frequency response (phase).

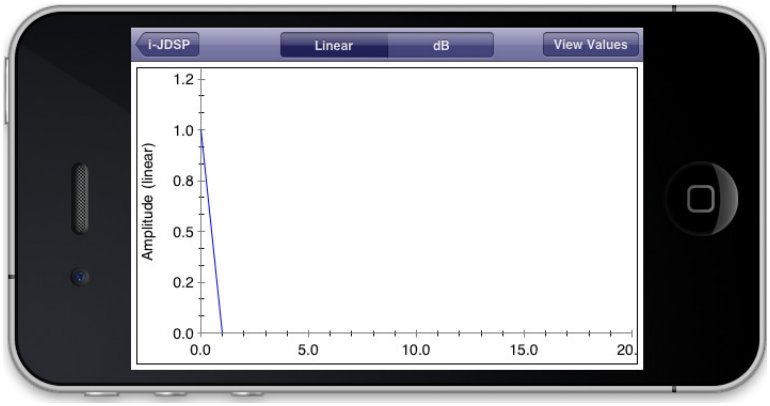


Figure C.19: Exercise 3 part d impulse response.

Exercise 4: Use i-JDSP to simulate a digital filter with the following coefficients $a_0 = 1$, $a_1 = 0.5$, and $b_0 = 1$.

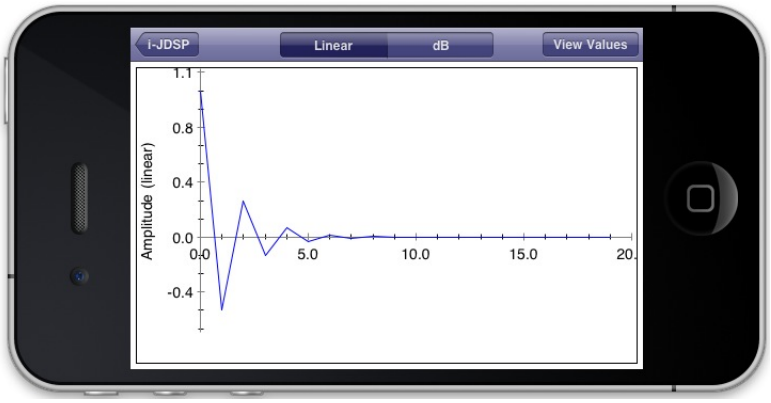


Figure C.20: Exercise 4 part a impulse response.

Exercise 5: Use i-JDSP to simulate the transient and steady-state response of a digital filter that has the following coefficients $a_0 = 1$, $a_1 = 0$, $a_2 = 0.9$ and $b_0 = 1$.

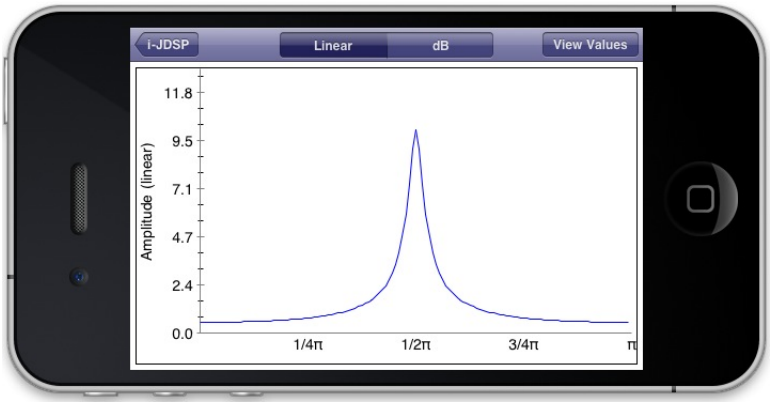


Figure C.21: Exercise 5 part a frequency response (magnitude).

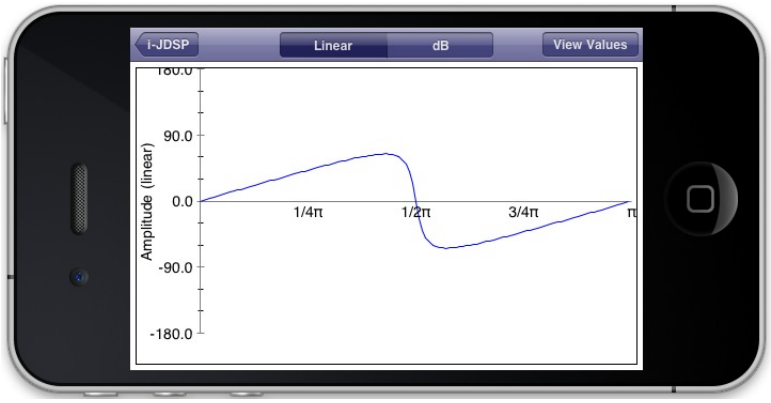


Figure C.22: Exercise 5 part a frequency response (phase).

Refer to Figure C.23 to establish the simulation for part b.

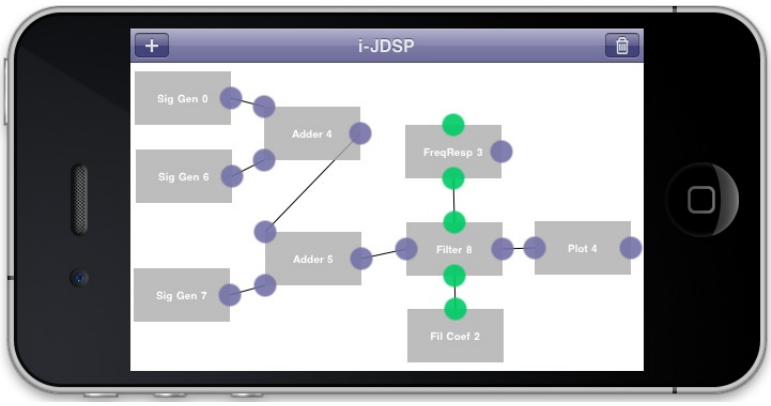


Figure C.23: Exercise 5 part b simulation diagram.

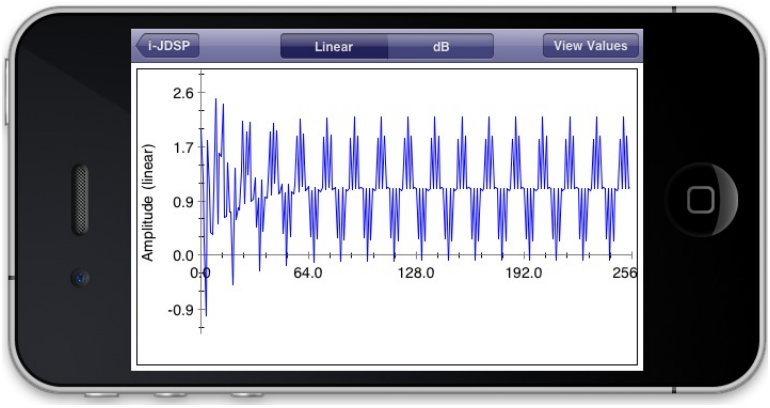


Figure C.24: Exercise 5 part b output of the filter.

Exercise 6: Use i-JDSP to simulate the averaging filter for $L = 5$ and $L = 50$.

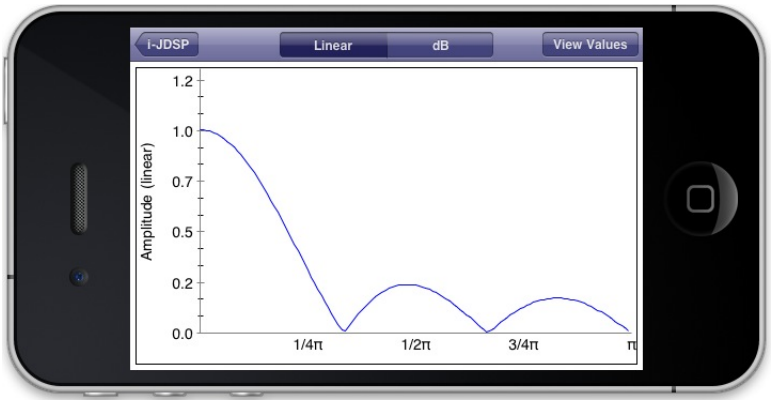


Figure C.25: Exercise 6 part a frequency response (magnitude).

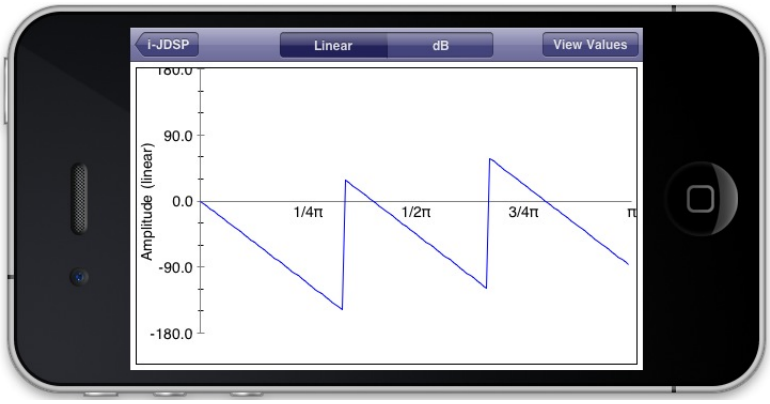


Figure C.26: Exercise 6 part a frequency response (phase).

Use the similar setting illustrated in Figure C.27.

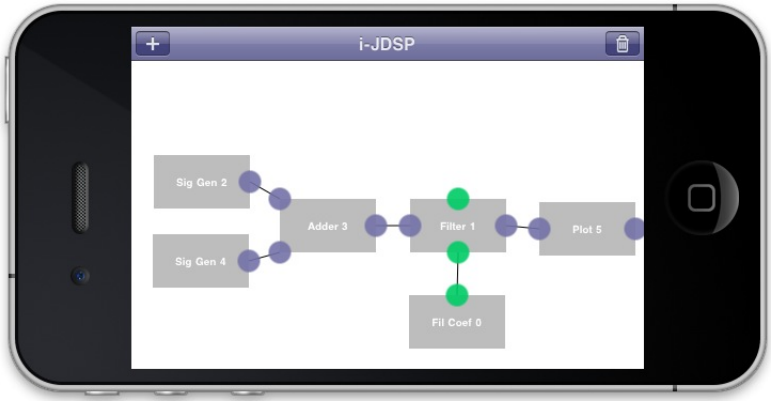


Figure C.27: Exercise 6 part b simulation diagram.

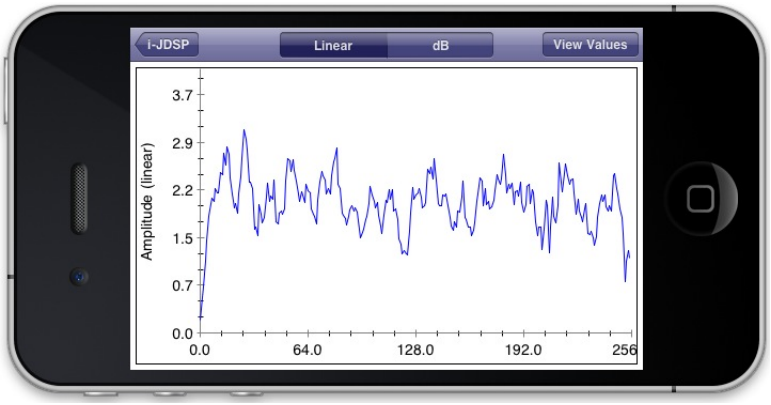


Figure C.28: Exercise 6 part b output of the filter.

C.2 The z-transform and frequency response

In this section, we will use i-JDSP to simulate the computer exercises discussed in Chapter 3 from the DSP book. Specifically, we use the Filter block of i-JDSP to invert the z-transform of various signals. Freq-Resp block and PZ Plot block are used for examine the frequency response and the poles and zeros location respectively.

Exercise 1: Exponential Sequences Design and simulate with i-JDSP a digital filter that has the following impulse response $h(n) = 0.9^n u(n)$.

Use the following diagram to establish the connections of the blocks.

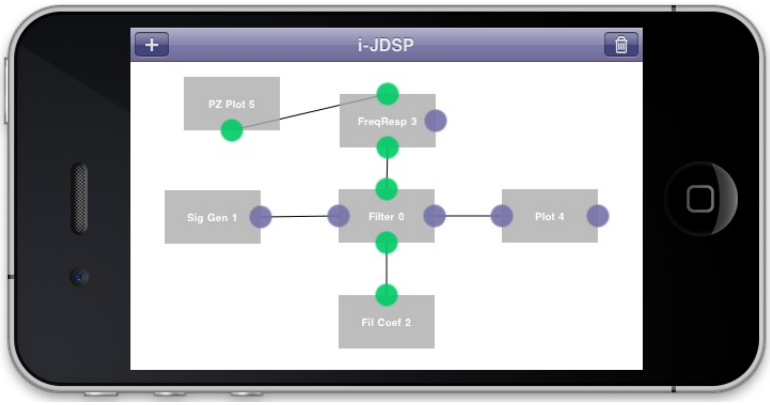


Figure C.29: Exercise 1 simulation diagram.

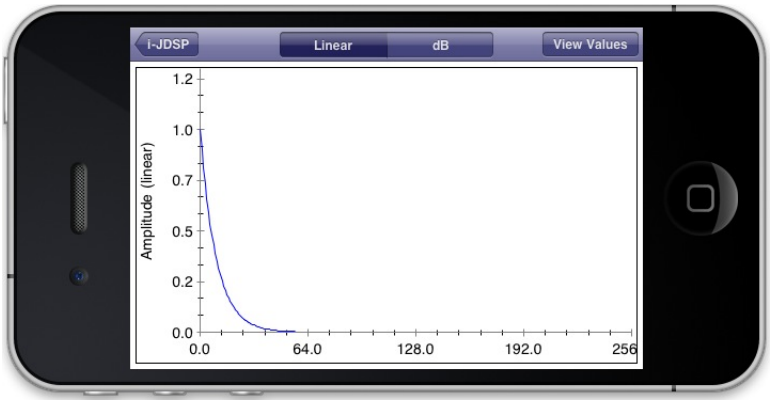


Figure C.30: Exercise 1 part b impulse response.

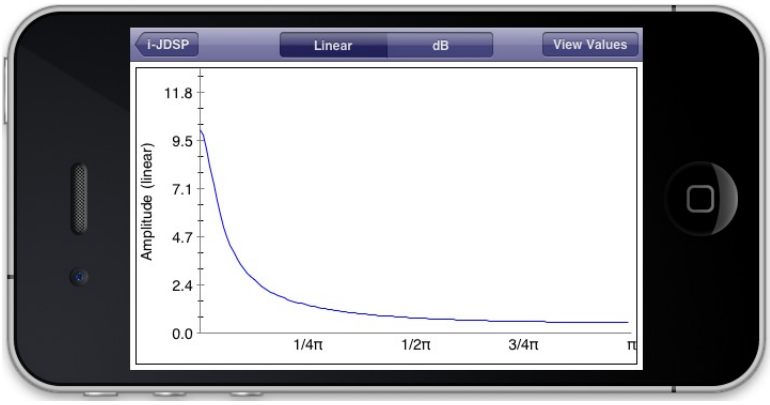


Figure C.31: Exercise 1 part c frequency response (magnitude).

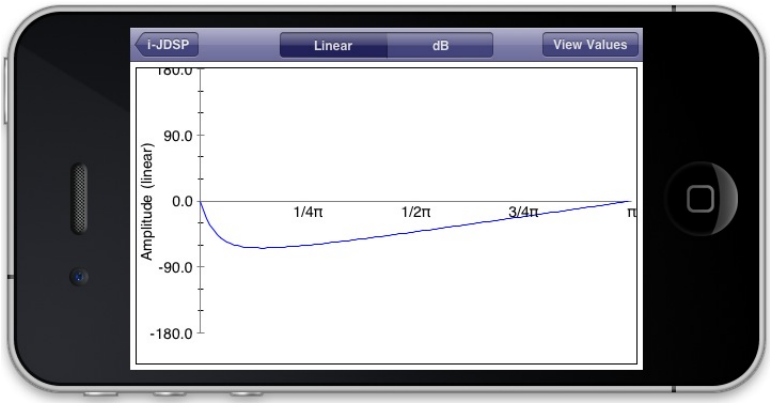


Figure C.32: Exercise 1 part c frequency response (phase).

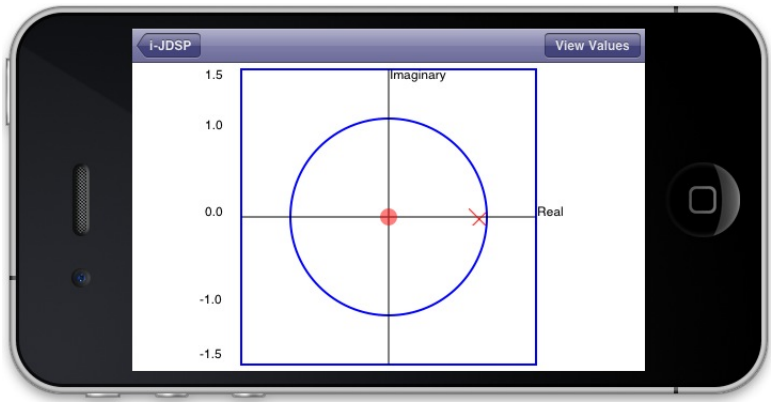


Figure C.33: Exercise 1 part c poles and zeros.

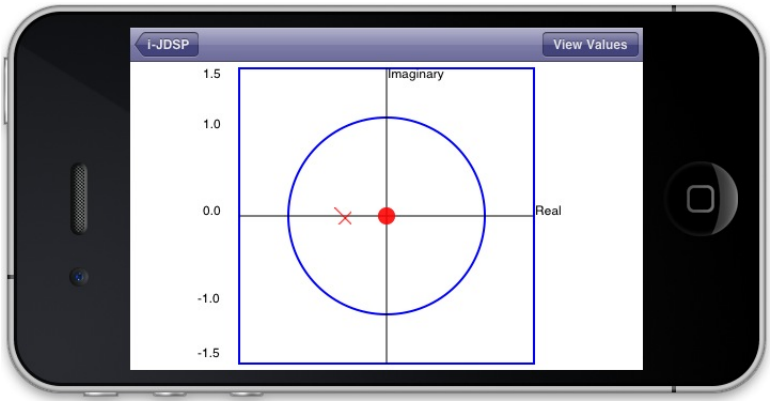


Figure C.34: Exercise 1 part e poles and zeros.

Exercise 2: Digital Oscillator Digital oscillators can be designed by selecting by coefficients of the digital filter so that we obtain oscillatory behavior when the excitation is a unit impulse.

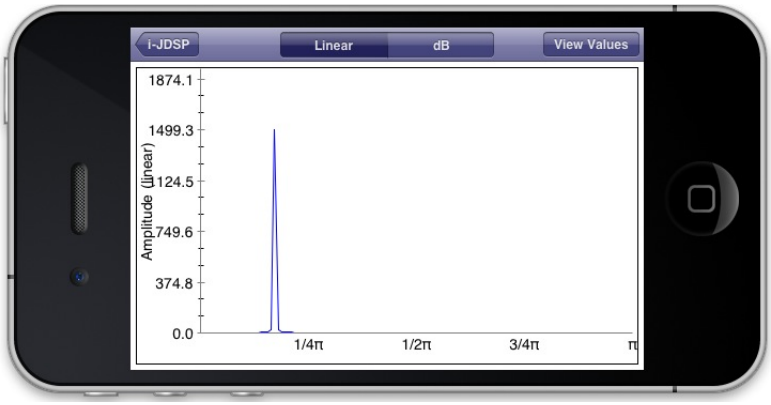


Figure C.35: Exercise 2 part b frequency response (magnitude).

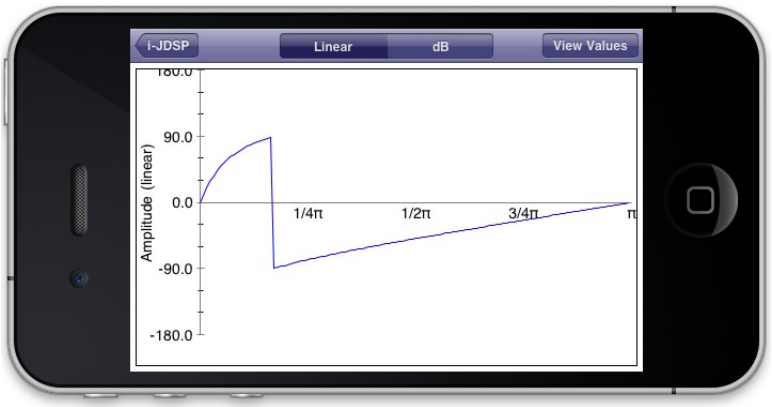


Figure C.36: Exercise 2 part b frequency response (phase).

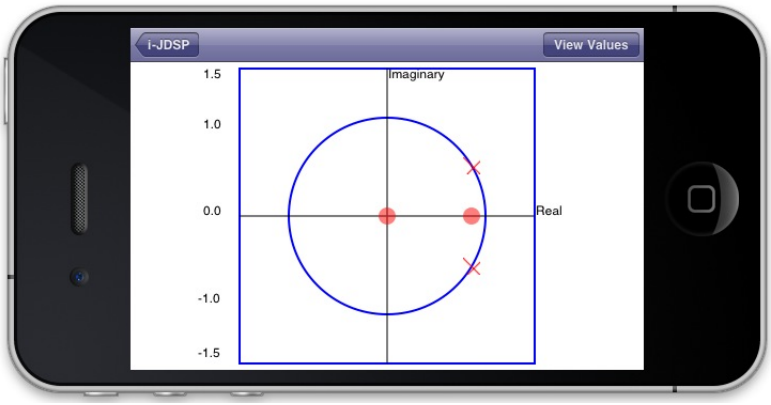


Figure C.37: Exercise 2 part b poles and zeros.

Exercise 3: Canceling Sinusoidal Components

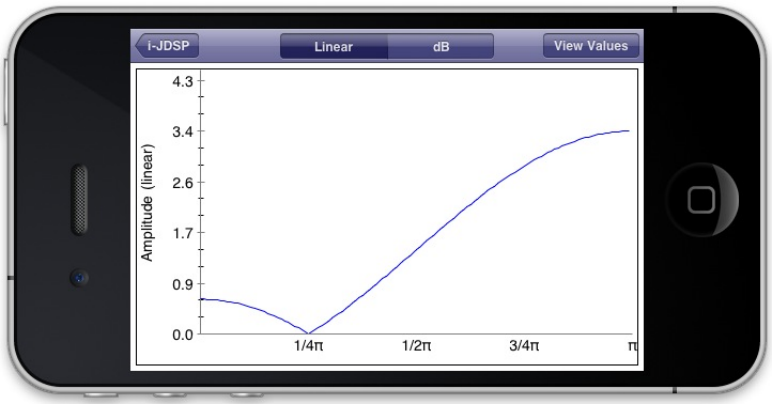


Figure C.38: Exercise 3 part a frequency response (magnitude).

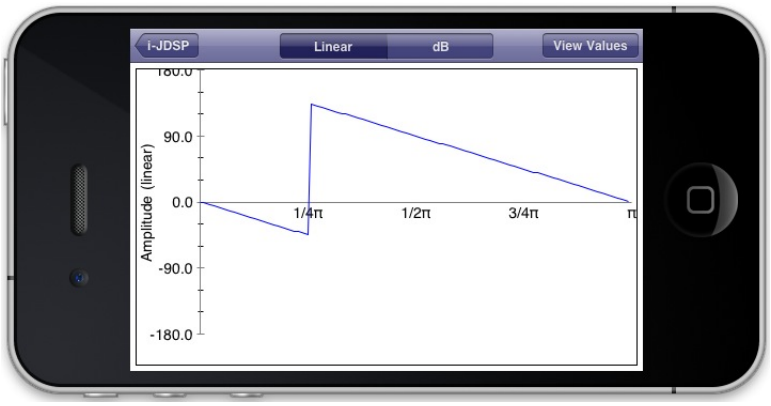


Figure C.39: Exercise 3 part a frequency response (phase).

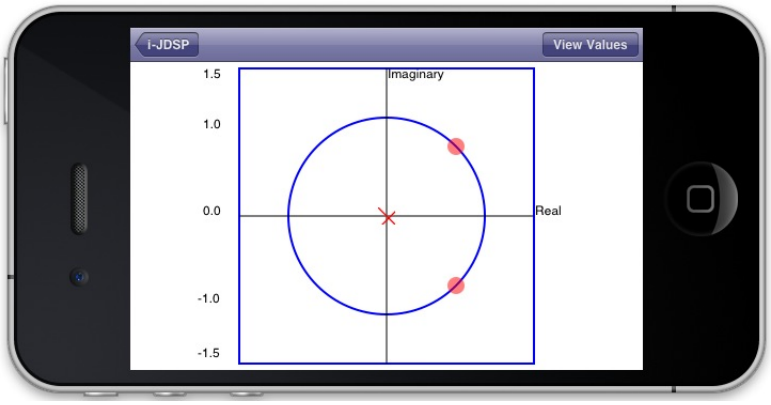


Figure C.40: Exercise 3 part a poles and zeros.

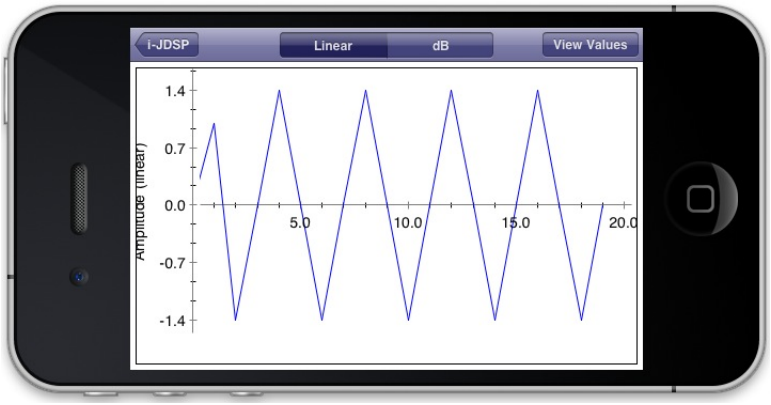


Figure C.41: Exercise 3 part b output of the system when $\Omega_1 = \pi/2$.

Exercise 4: Symmetric Impulse Response and Linear Phase

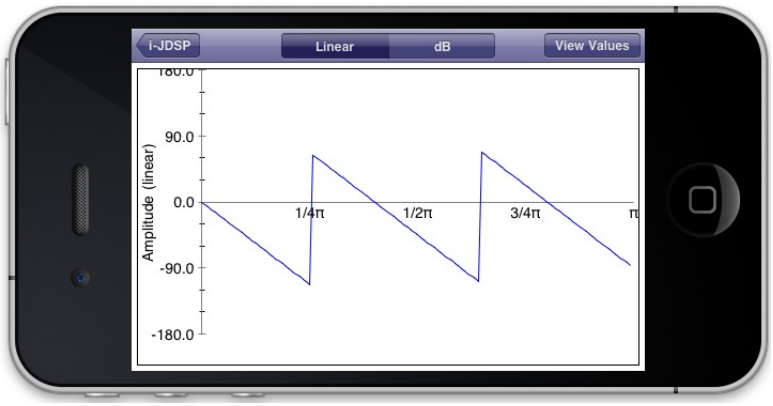


Figure C.42: Exercise 4 part a phase response.

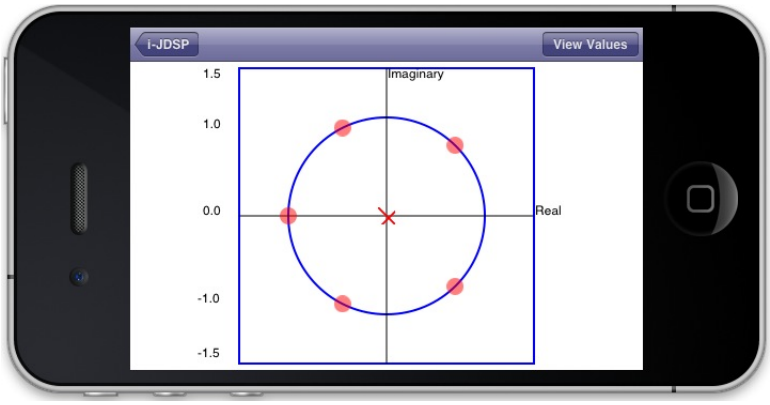


Figure C.43: Exercise 4 part b poles and zeros.

Exercise 5: Pole-Zero Plots
Refer to Figure C.29 for the simulation diagram.

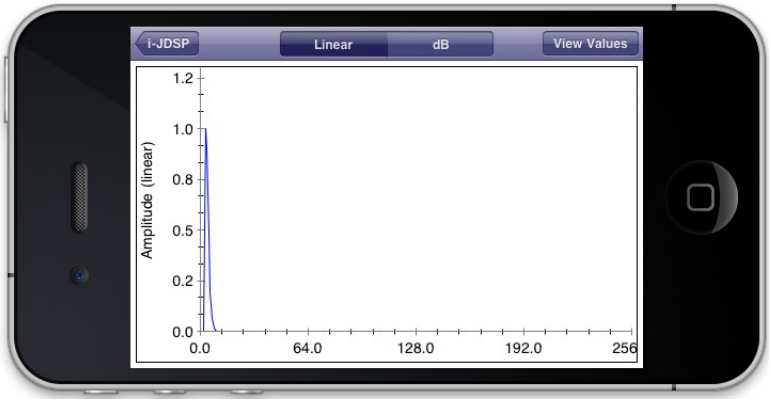


Figure C.44: Exercise 5 part b impulse response of $H_1(z)$.

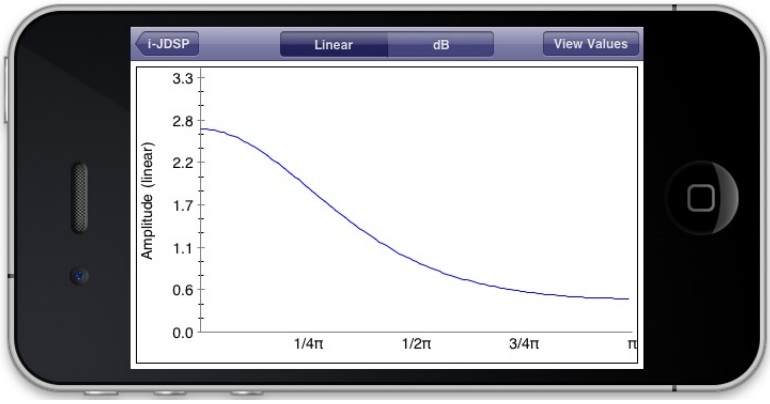


Figure C.45: Exercise 5 part b frequency response (magnitude) of $H_1(z)$.

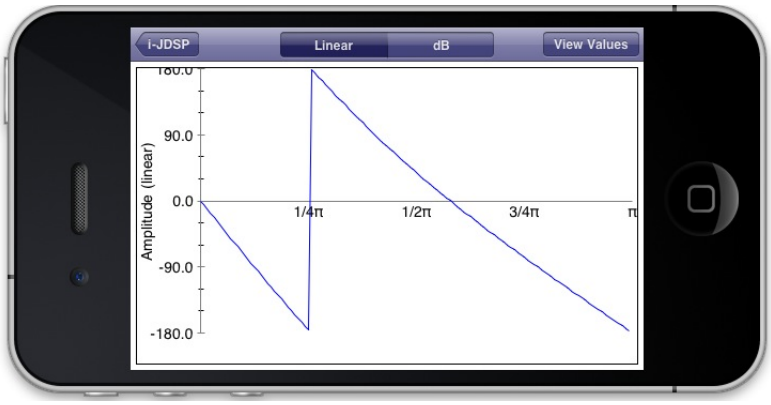


Figure C.46: Exercise 5 part b frequency response (phase) of $H_1(z)$.

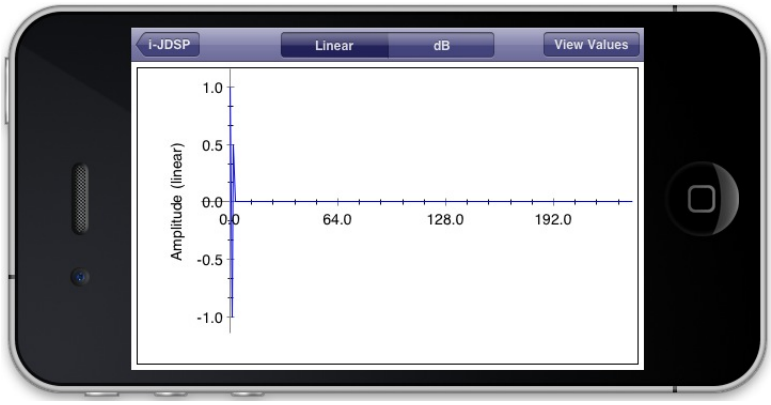


Figure C.47: Exercise 5 part b impulse response of $H_2(z)$.

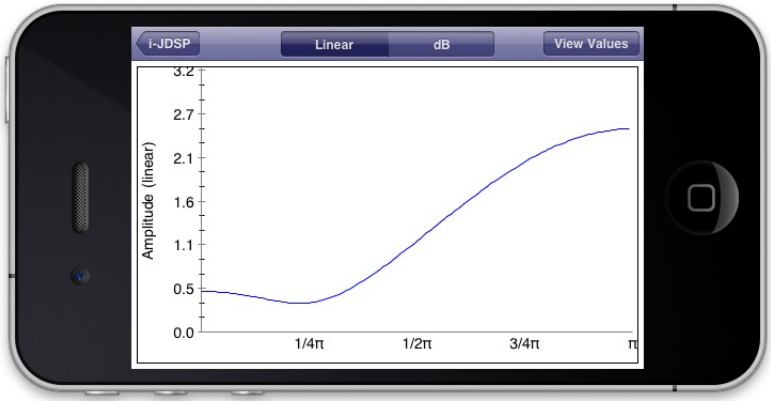


Figure C.48: Exercise 5 part b frequency response (magnitude) of $H_2(z)$.

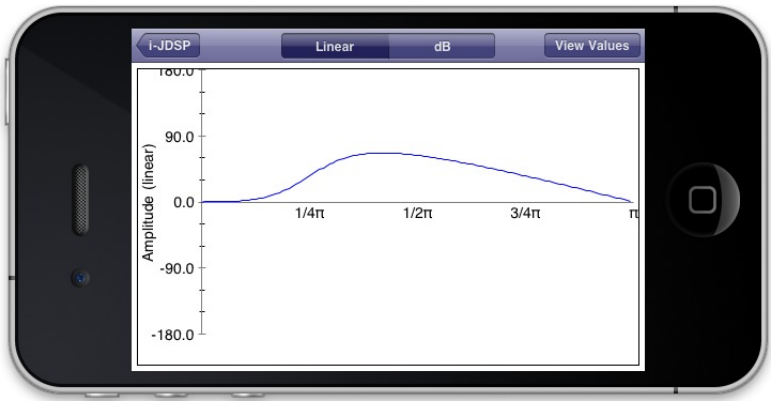


Figure C.49: Exercise 5 part b frequency response (phase) of $H_2(z)$.

Exercise 6: Cascade and Parallel System Configurations
Use the following diagram to establish the simulation.

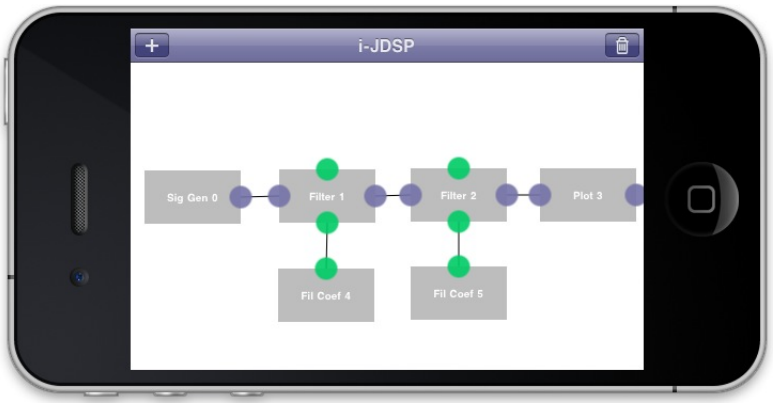


Figure C.50: Exercise 6 part a simulation diagram.

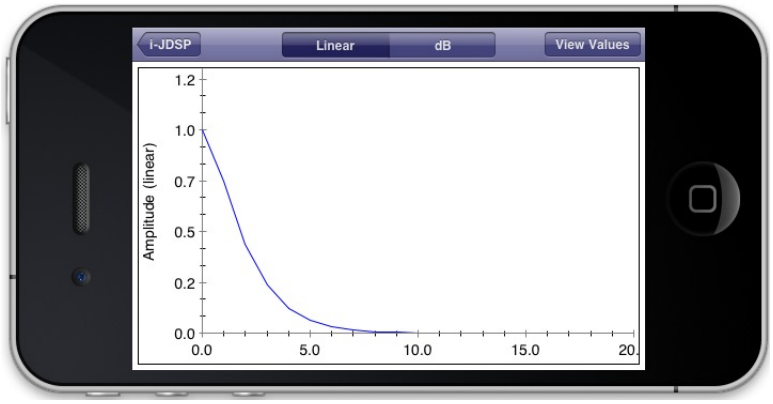


Figure C.51: Exercise 6 part a output of the system.

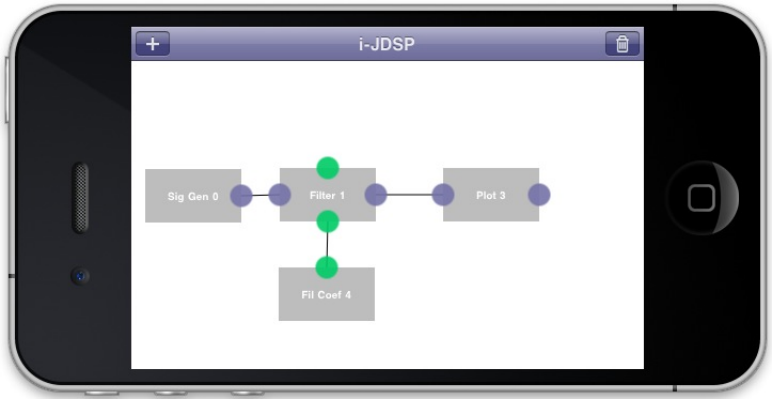


Figure C.52: Exercise 6 part b simulation diagram.

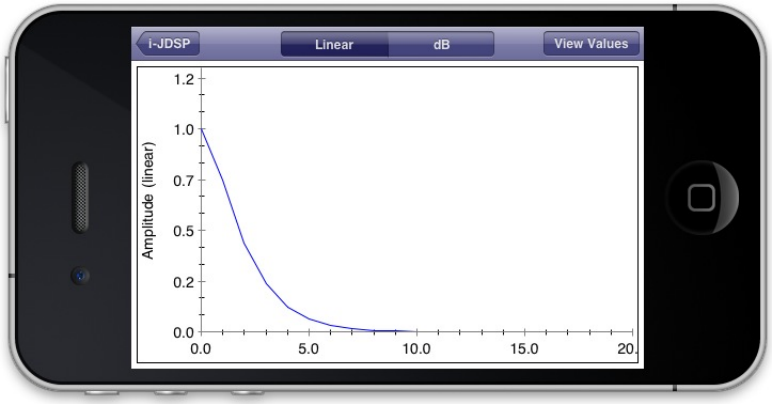


Figure C.53: Exercise 6 part b output of the system.

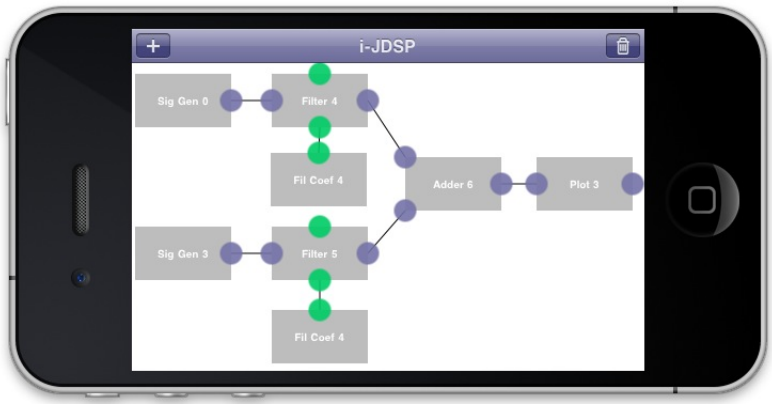


Figure C.54: Exercise 6 part c simulation diagram.

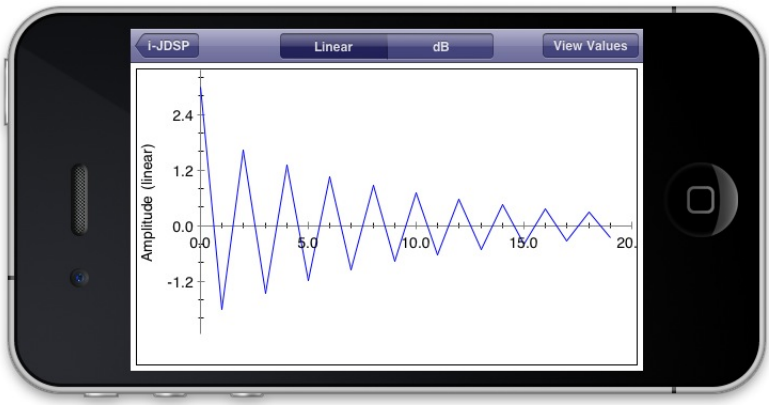


Figure C.55: Exercise 6 part c output of the system.

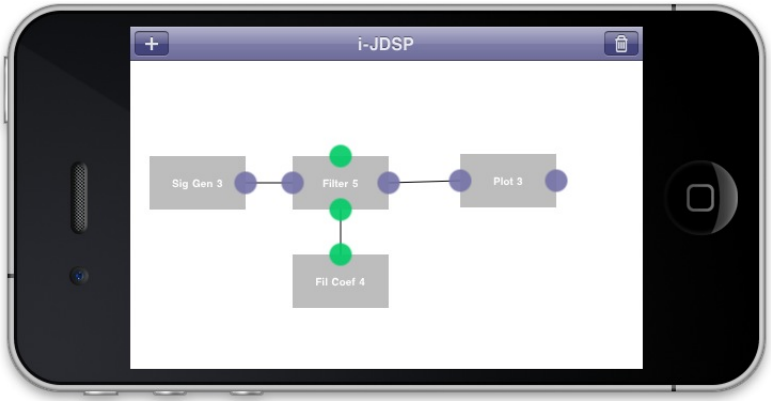


Figure C.56: Exercise 6 part d simulation diagram.

Exercise 7: Pole-Zero Plots

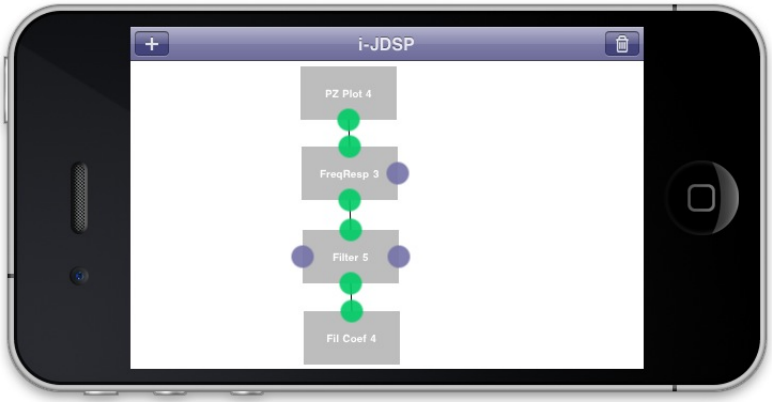


Figure C.57: Exercise 7 simulation diagram.

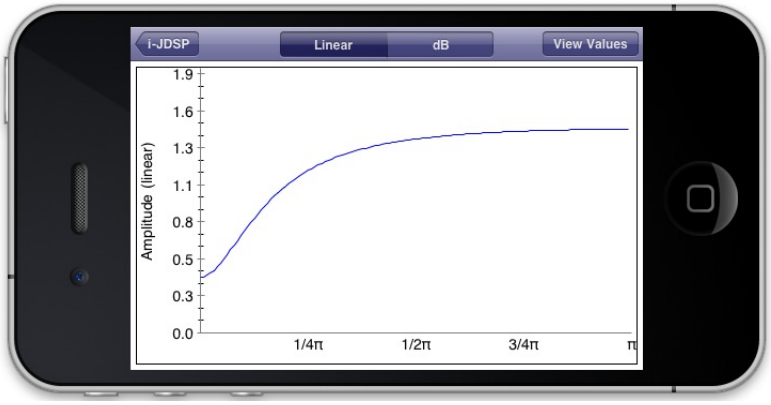


Figure C.58: Exercise 7 part a frequency response (magnitude) of $H_1(z)$.

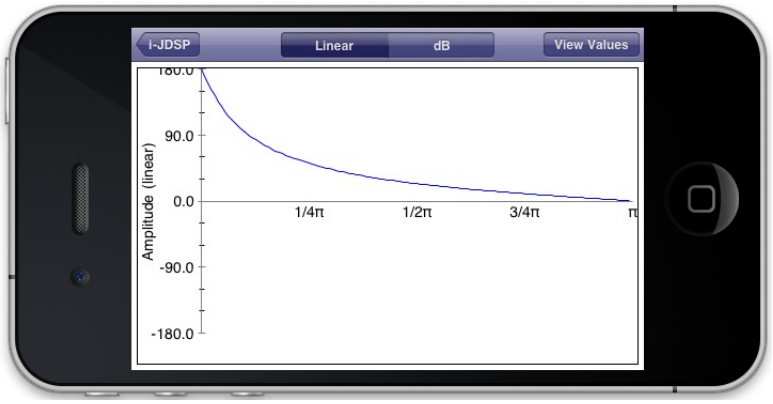


Figure C.59: Exercise 7 part a frequency response (phase) of $H_1(z)$.

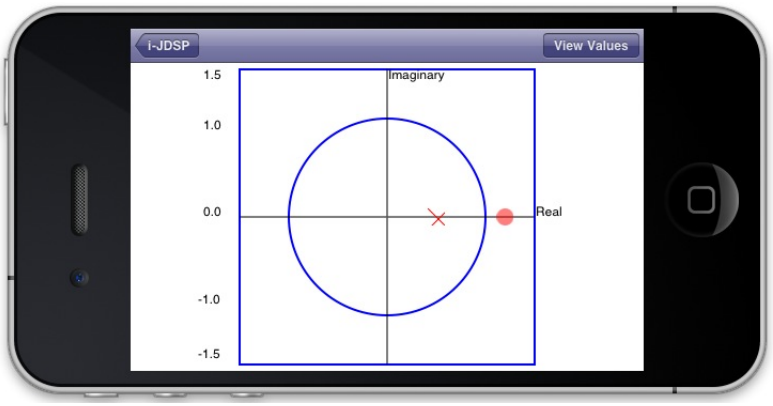


Figure C.60: Exercise 7 part a poles and zeros of $H_1(z)$.

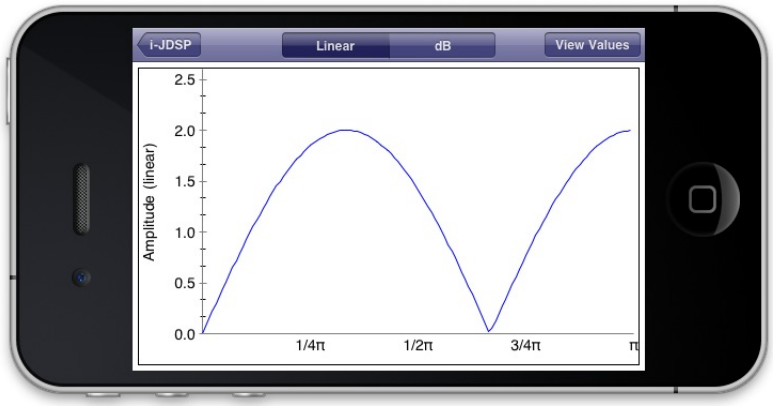


Figure C.61: Exercise 7 part b frequency response (magnitude) of $H_2(z)$.

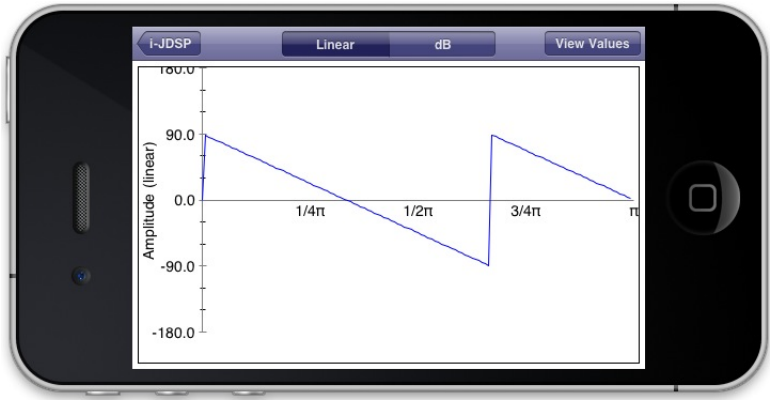


Figure C.62: Exercise 7 part b frequency response (phase) of $H_2(z)$.

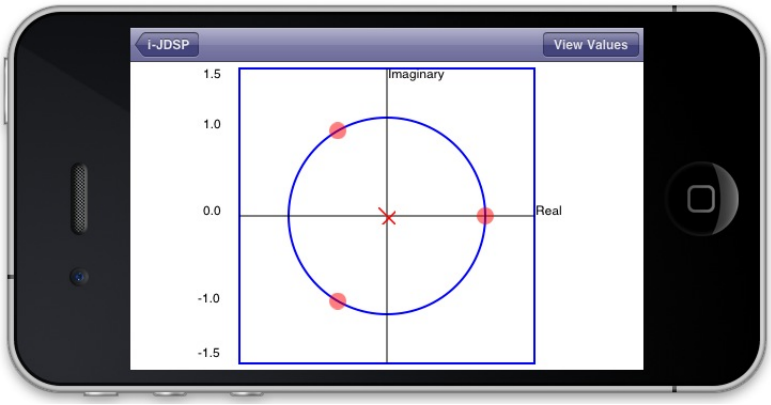


Figure C.63: Exercise 7 part b poles and zeros of $H_2(z)$.

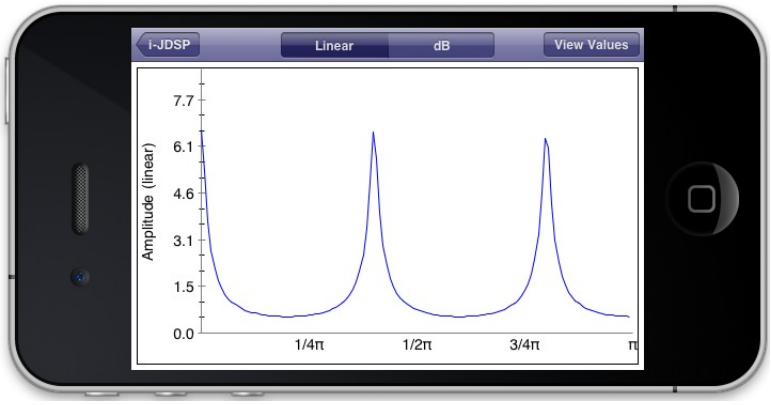


Figure C.64: Exercise 7 part c frequency response (magnitude) of $H_3(z)$.

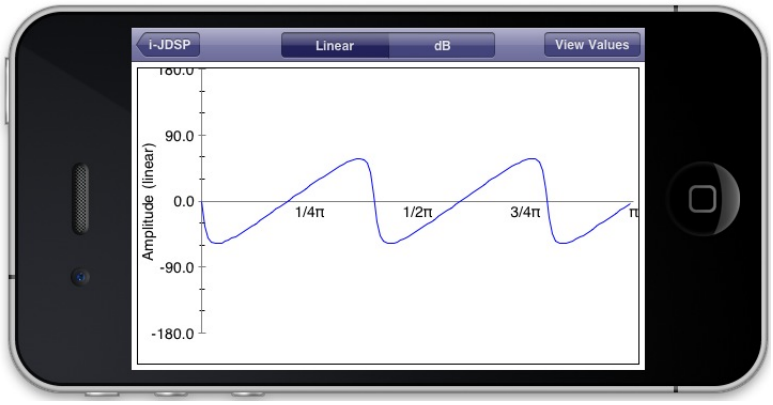


Figure C.65: Exercise 7 part c frequency response (phase) of $H_3(z)$.

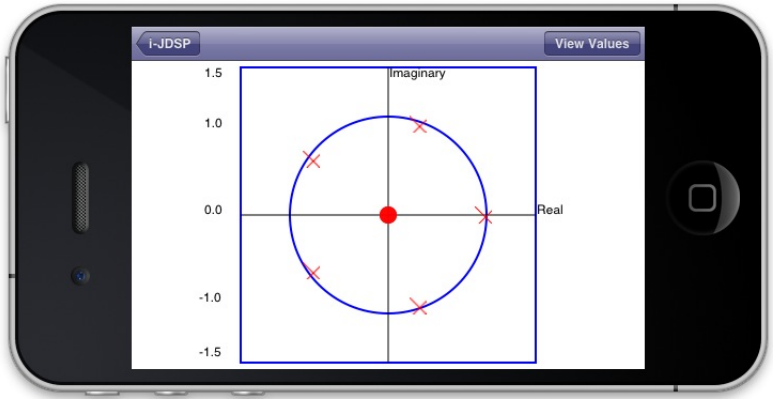


Figure C.66: Exercise 7 part c poles and zeros of $H_3(z)$.

Exercise 8: Varying the Magnitudes of Poles and Zeros

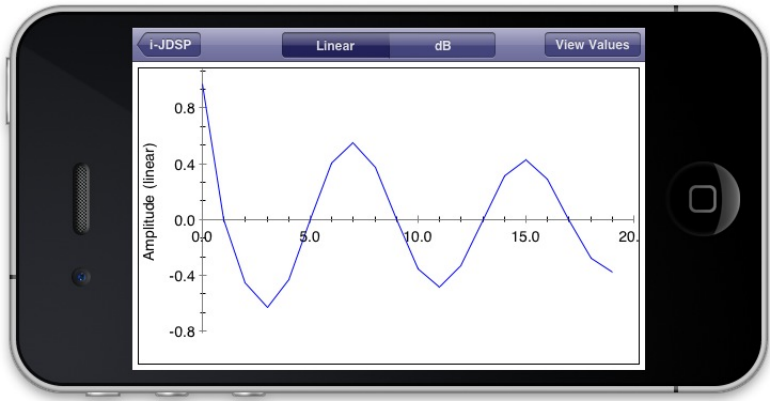


Figure C.67: Exercise 8 part b impulse response when $r = 0.96$.

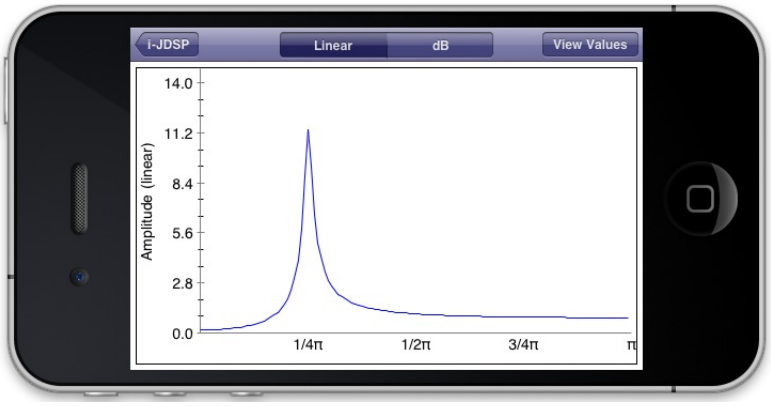


Figure C.68: Exercise 8 part c frequency response (magnitude) when $r = 0.96$.

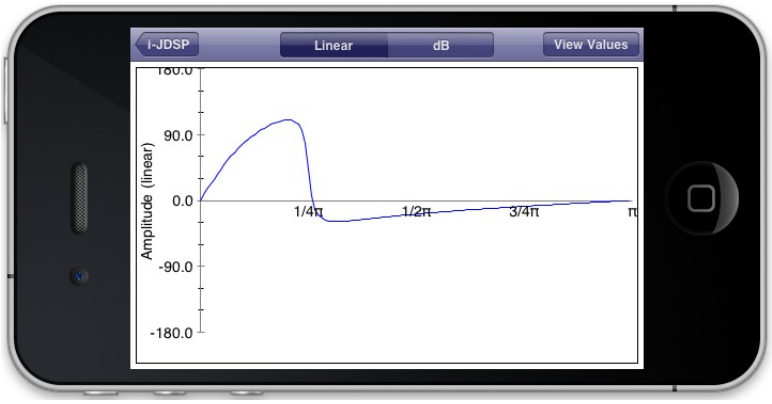


Figure C.69: Exercise 8 part c frequency response (phase) when $r = 0.96$.

Exercise 9: Lowpass/highpass Filter Design by Pole-Zero Placement
Use the PZ Placement block for this exercise.

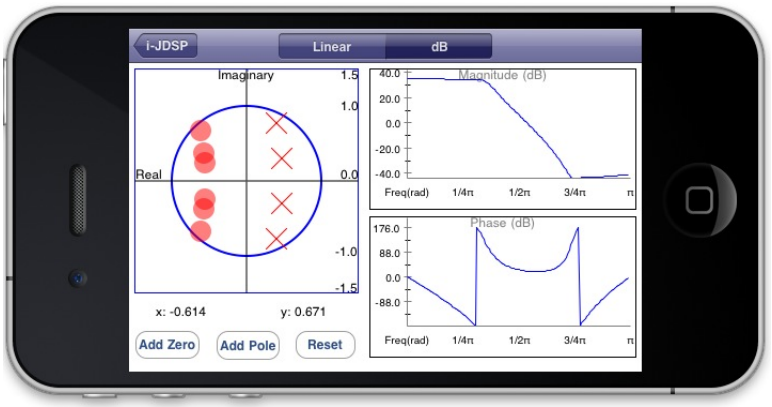


Figure C.70: Exercise 9 part a when $\Omega_c = \pi/3$.

Exercise 10: An Interesting Frequency Response

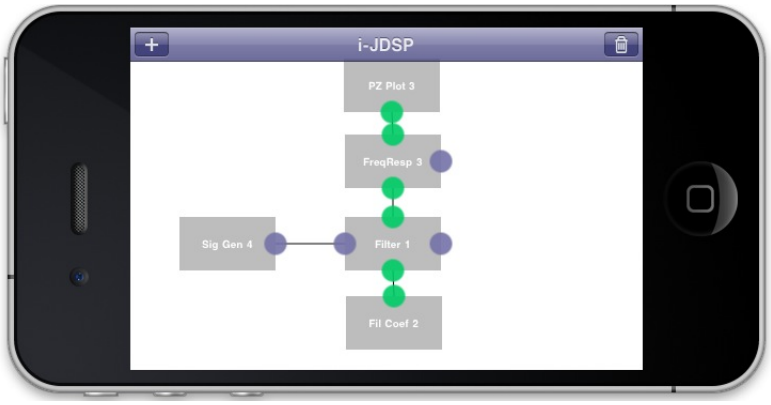


Figure C.71: Exercise 10 simulation diagram.

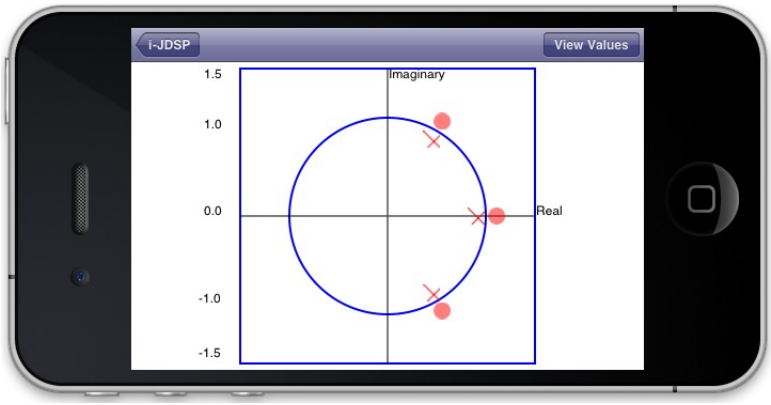


Figure C.72: Exercise 10 part a poles and zeros.

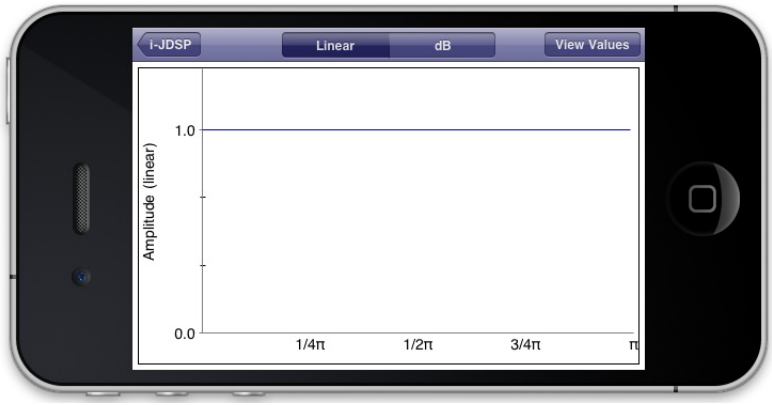


Figure C.73: Exercise 10 part b frequency response (magnitude).

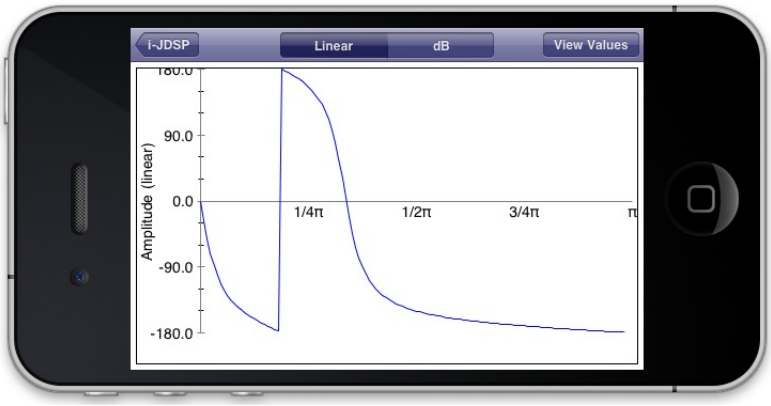


Figure C.74: Exercise 10 part b frequency response (phase).

Exercise 1: Symmetries in the time domain and frequency domain

Use i-JDSP and describe how the symmetries affect the FFT spectra. Simulation can be established using the diagram below.

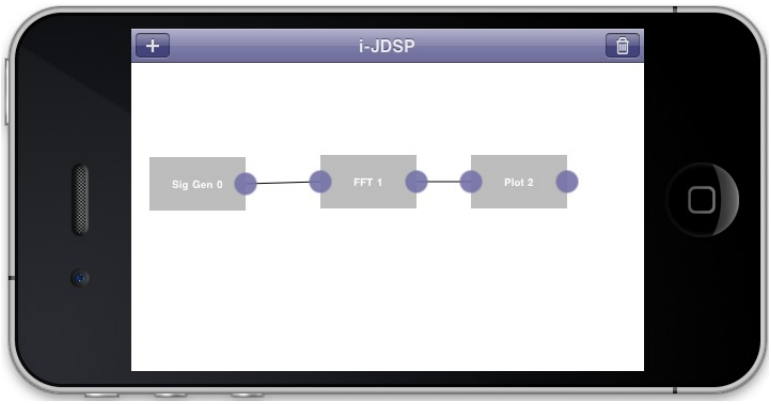


Figure C.75: Exercise 1 simulation diagram.

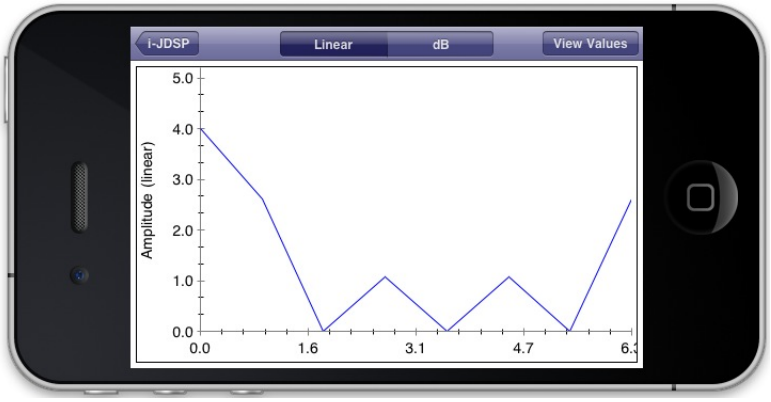


Figure C.76: Exercise 1 part a FFT output of $x_1(n)$.

Exercise 2: The Rectangular Window Use the following diagram to establish the simulation.

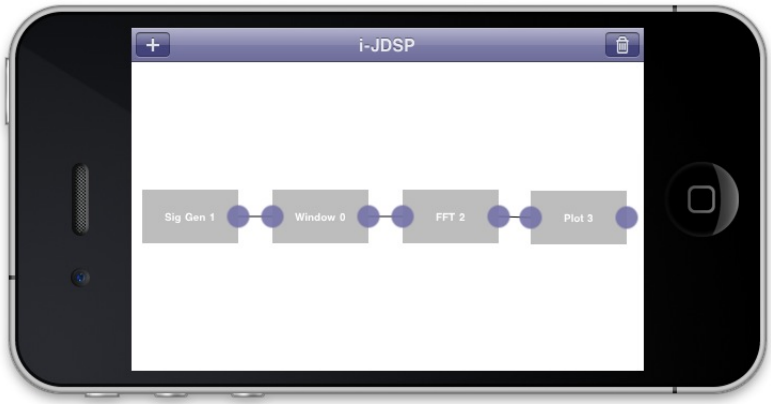


Figure C.77: Exercise 2 simulation diagram.

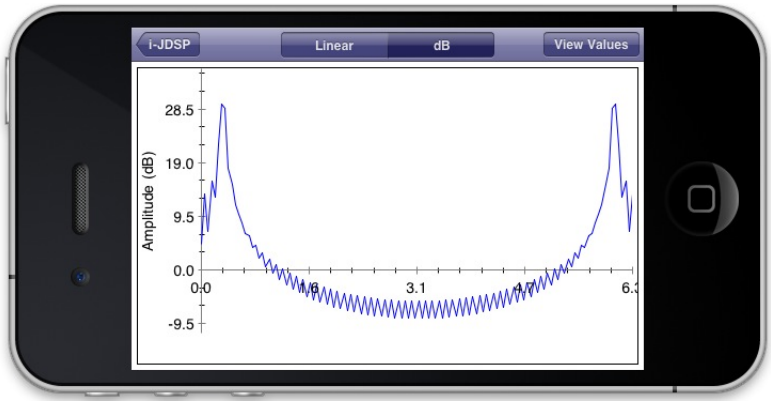


Figure C.78: Exercise 2 part a FFT output.

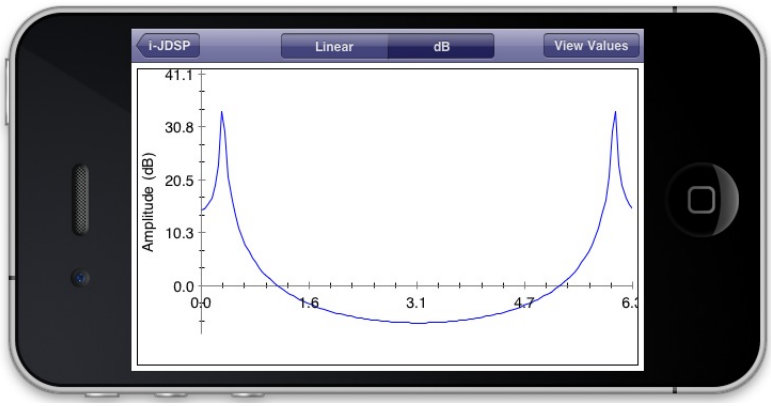


Figure C.79: Exercise 2 part b FFT output.

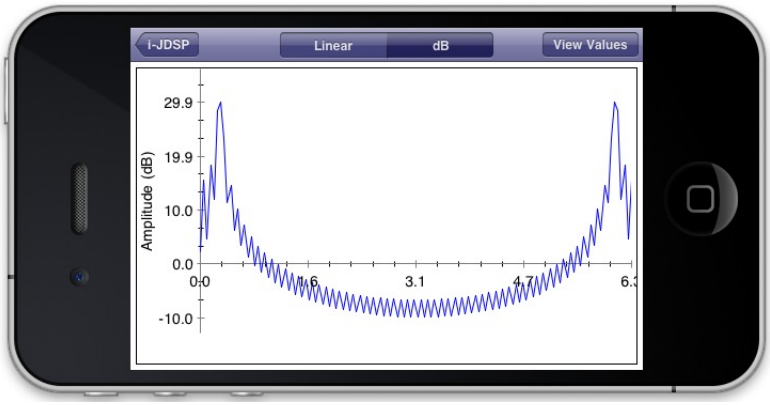


Figure C.80: Exercise 2 part c FFT output when $freq = \pi/11$.

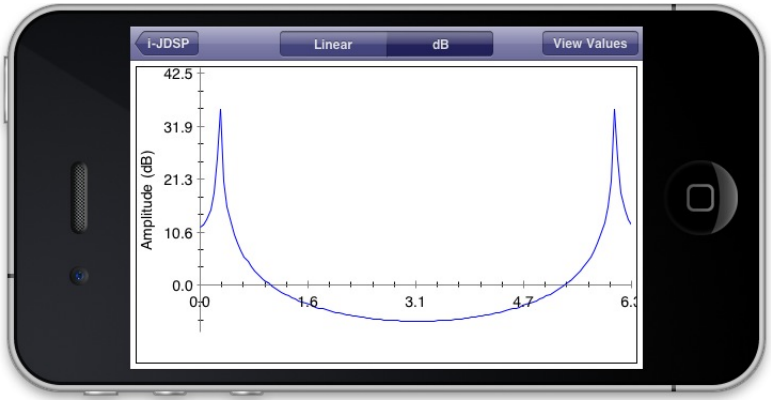


Figure C.81: Exercise 2 part c FFT output when $freq = \pi/11$.

Exercise 3: Window Tradeoffs Refer to Figure C.82 for the simulation setup.

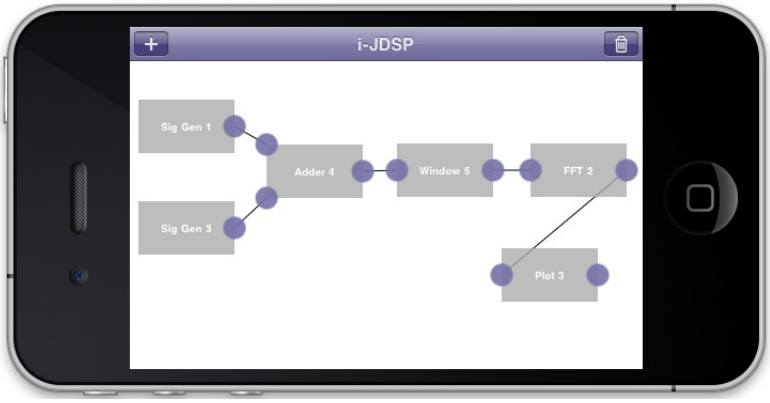


Figure C.82: Exercise 3 simulation diagram.

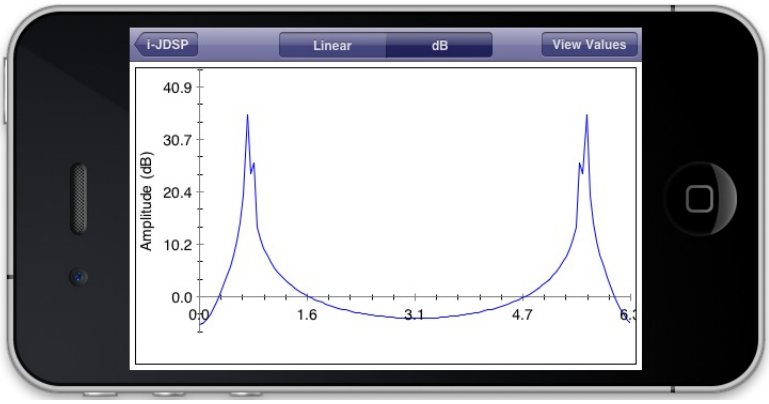


Figure C.83: Exercise 3 part a FFT output using a rectangular window.

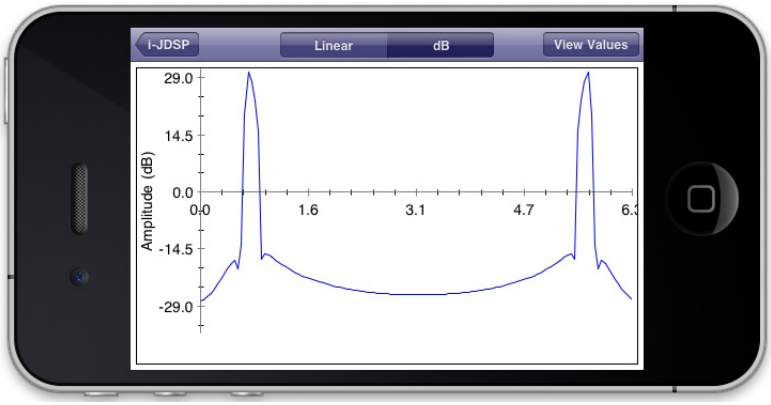


Figure C.84: Exercise 3 part b FFT output using a Hamming window.