Fall 2012

3D spatio-temporal analysis for compressive sensing in magnetic resonance imaging of the murine cardiac cycle

Brice Aaron Hirst

Department:

3D SPATIO-TEMPORAL ANALYSIS FOR COMPRESSIVE SENSING IN MAGNETIC RESONANCE IMAGING OF THE MURINE CARDIAC CYCLE

by

BRICE AARON HIRST

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

2012

Approved by

Yahong Rosa Zheng, Advisor
Kurt Kosbar
Randy H. Moss

PUBLICATION THESIS OPTION

The papers presented in this thesis have been prepared in the styles utilized by the SPIE 2013 Medical Imaging Conference and the ASEE 2013 Annual Conference. Pages 3-22 will be submitted for publication in the former conference; pages 23-44 will be submitted for publication in the latter. Appendices A and B have been added for purposes normal to thesis/dissertation writing, and contain material prepared in the original style of the instructional purpose for which it was meant.

ABSTRACT

This thesis consists of two major contributions, each of which has been prepared in a conference paper. These papers will be submitted for publication in the SPIE 2013 Medical Imaging Conference and the ASEE 2013 Annual Conference.

The first paper explores a three-dimensional compressive sensing (CS) technique for reducing measurement time in MR imaging of the murine (mouse) cardiac cycle. By randomly undersampling a single 2D slice of a mouse heart at regular time intervals as it expands and contracts through the stages of a heartbeat, a CS reconstruction algorithm can be made to exploit transform sparsity in time as well as space. For the purposes of measuring the left ventricular volume in the mouse heart, this 3D approach offers significant advantages against classical 2D spatial compressive sensing

The second paper describes the modification and testing of a set of laboratory exercises for developing an undergraduate level understanding of Simulink. An existing partial set of lab exercises for Simulink was obtained and improved considerably in pedagogical utility, and then the completed set of pilot exercises was taught as a part of a communications course at the Missouri University of Science and Technology in order to gauge student responses and learning experiences. In this paper, the content of the laboratory exercises with corresponding educational approaches are discussed, along with student feedback and future improvements.

ACKNOWLEDGMENTS

Special thanks are due to Dr. Yahong Rosa Zheng, who has guided me through the entire process of conducting research and writing papers, and has also served as the chair of my thesis defense committee. Also due thanks are Dr. Kurt Kosbar and Dr. Randy Moss, who have not only taught me in several courses at this university, but have also served as members of my thesis defense committee.

Acknowledgements for the MRI compressive sensing research go out to Dr. Lixin Ma and Mr. Ming Yang, who provided all of the raw MRI test data used in the study as well as general guidance on MRI physics and terminology. Information about compressive sensing algorithms and concepts was provided by Mr. Zengli Yang and Mr. Hamed Kajbaf, and Mr. Saurav Subedi was very helpful in facilitating the Spring 2012 test of the Simulink labs on a class of undergraduate students.

The research on compressive sensing for MRI was supported by the University of Missouri Research Board fund, and the Simulink project design was supported by Dr. Zheng's NSF Career award #ECCS-0846486.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

Paper I

Paper II

LIST OF TABLES

Paper I

Paper II

ABBREVIATIONS

Abbreviation	Description
ADM	Alternating Direction Method
AM	Amplitude Modulation
AMI	Alternating Mark Inversion
ASEE	American Society for Engineering Education
AWGN	Additive White Gaussian Noise
CCD	Charge-Coupled Device
CS	Compressive Sensing
CT	Continuous Time
DC	Direct Current (also an alias for the first output term in a DFT)
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DSB-AM	Double Sideband Amplitude Modulation
DSP	Digital Signal Processing
DT	Discrete Time
DWT	Discrete Wavelet Transform
ECG	Electrocardiogram
FD	Finite Differences
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FM	Frequency Modulation
IEEE	Institute of Electrical and Electronics Engineers

ISI	Intersymbol Interference
MEX	MATLAB Executable
MRI	Magnetic Resonance Imaging
NRZ	Non-Return to Zero
NRZI	Non-Return to Zero Inverted
ODE	Ordinary Differential Equation
PDF	Probability Density Function
PLL	Phase Locked Loop
PM	Phase Modulation
PSD	Power Spectral Density
PSF	Point Spread Function
PSNR	Peak Signal-to-Noise Ratio
RF	Radio Frequency
RZ	Return to Zero
SNR	Signal-to-Noise Ratio
SPIE	Society of Photo-Optical Instrumentation Engineers
TV	Total Variation
VCO	Voltage-Controlled Oscillator
ZF	Zero-Forcing
ZP	Zero Padding

1. INTRODUCTION

1.1. OVERVIEW OF RESEARCH WORK

The research work conducted for this thesis consists principally of two parts: the formation of a 3D compressive sensing method specifically designed for MR imaging of beating mouse hearts (Paper I), and the design and implementation of a set of laboratory exercises for teaching Simulink to undergraduates (Paper II). These topics are only marginally related; the only real connection between them is that MATLAB was used for both. Nonetheless, both topics are important in their own rights, and have important implications in their relevant fields.

1.2. 3D COMPRESSIVE SENSING FOR MR IMAGING OF MOUSE HEARTS

Compressive sensing, in general, refers to the exploitation of randomness and transform sparsity to construct an approximation to an image using sub-Nyquist random sampling. The basic concept was described by Donoho in 2006 [6]; since that time, it has been seen as a promising method to reduce the burden of measurement in imaging methods that are slow and/or expensive in their acquisition of data. The field of MRI is one of the imaging methods where CS shows the most potential, mainly because the measuring process is slow, and because all conventional methods for speeding up data acquisition have serious negative consequences for either the physics of the machine or the health of the patient inside.

The main breakthrough paper for CS as applied to MRI was published by Lustig *et. al.* in 2007 [4]. In that paper, a CS method for undersampling k-space (which is essentially the frequency domain, and the native measurement domain of MRI) and

reconstructing accurate approximations to many static images was described. Since then, others have applied CS to different subsets of MR imaging. The application of CS to mouse heart imaging in particular was explored by Wech *et al.* in 2011 [2], but only in the 2D sense; transform sparsity in the time domain was not considered.

In collaboration with the University of Missouri VA Hospital, a method was developed in Paper I of this thesis to expand the compressive sensing reconstruction for time-varying MR imaging to all 3 dimensions: 2 spatial and 1 temporal. While it was mainly applied to mouse cardiac imaging, there is no reason why this method would not see success with any other time-varying image sequences acquired by MRI. An overview of the MATLAB software used in this research is presented in Appendix A.

1.3. SIMULINK LAB DESIGN

In our university as well as many others, MATLAB is the standard simulation tool that is taught in laboratory courses alongside lecture topics such as discrete linear systems and communications. In 2011, Chaitri Aroskar created a set of Simulink exercises with the aim to replace MATLAB as the simulation exercises in communications [12]. As an undergraduate, the author of this thesis attempted these labs only to run into several difficulties. Paper II describes the process of improving the educational value of these labs, as well as the results of testing them on a class of undergraduates taking communications. These exercises were generally received well by the students, and in the future they may be used by other universities attempting to teach Simulink. The full set of lab manuals used in this experiment is presented in Appendix B.

PAPER

I. 3D spatio-temporal analysis for compressive sensing in magnetic resonance imaging of the murine cardiac cycle

Brice Hirst[a], Yahong Rosa Zheng[a], Ming Yang[b], and Lixin Ma[b]

[a]Missouri University of Science and Technology, Department of Electrical and Computer Engineering;

[b]University of Missouri, Department of Radiology, Nuclear Science and Engineering Institute, and Harry S. Truman Memorial Veteran's Hospital.

ABSTRACT

This paper explores a three-dimensional compressive sensing (CS) technique for reducing measurement time in magnetic resonance imaging (MRI) of the murine (mouse) cardiac cycle. By randomly undersampling a single 2D slice of a mouse heart at regular time intervals as it expands and contracts through the stages of a heartbeat, a CS reconstruction algorithm can be made to exploit transform sparsity in time as well as space. For the purposes of measuring the left ventricular volume in the mouse heart, this 3D approach offers significant advantages against classical 2D spatial compressive sensing.

Keywords: Compressive sensing, MRI, three-dimensional, random sampling, mouse heart

Further author information

Brice Hirst - E-mail: bahrkb@mst.edu, Telephone: 1-573-368-9504

Dr. Zheng - E-mail: zhengyr@mst.edu, Telephone: 1-573-341-6632

Ming Yang - E-mail: my5f2@mail.missouri.edu, Telephone: 1-573-814-6000

Dr. Ma - E-mail: mal@health.missouri.edu, Telephone: 1-573-814-6000

1. INTRODUCTION

For many imaging methods (such as video recording), data acquisition is fast enough and cheap enough that acquiring fully (i.e. Nyquist rate or greater) sampled sets of data is not a problem. However, taking full sets of measurements with other imaging methods can be costly in several ways. For example, a camera CCD sensor array may be very cheap to produce, but microwave imaging requires expensive sensors which generally take a long period of time to acquire data.[1] In magnetic resonance imaging, an image is created by using radio-frequency (RF) pulses to spin the protons in living tissue at different frequencies and phases, then reading the total RF emissions from all of them combined, creating a frequency-based measurement domain known as "k-space". A full set of data can take a long time to acquire, which can cause unacceptable motion artifacts in cases such as real-time imaging of the mouse cardiac cycle. By reducing the number of measurements, compressive sensing can reduce either the amount of imaging time or the number of sensors, or both, depending on the imaging method. For preclinical MRI, the prospect of reduced measurement time is of great interest to the field of medicine because:

1. An MRI procedure is expensive. Less measurement time = less money spent.
2. Fast-moving targets (such as mouse hearts) exhibit less severe motion artifacts when measurement times are shorter.
3. A shorter measurement time means less exposure to magnetic & RF energy for the animal under test.
4. Preclinical research can be completed faster when more scans can be made in the same period of time.

An example of a complete cardiac cycle for a mouse is shown in Figure 1, with the important stages labeled. The primary diagnostic goal of these images is to estimate the volume of the left ventricular cavity (the prominent white region in each frame of Figure 1) as the heart beats.[2] A notable feature of this sort of measurement is that the areas of

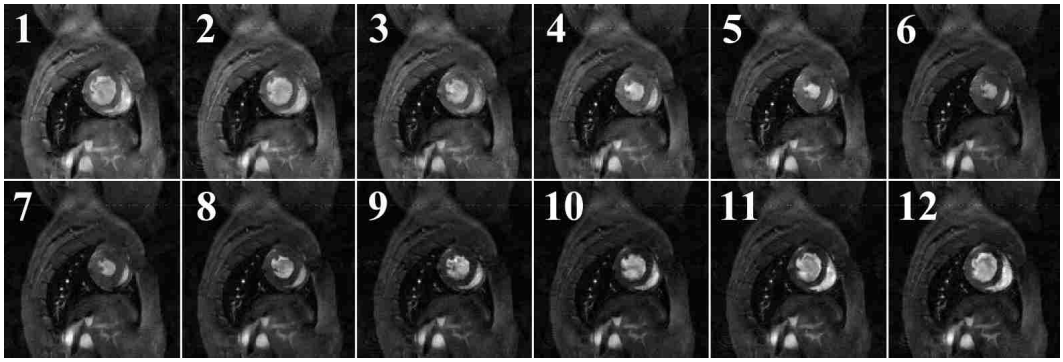


Figure 1. Sample cardiac cycle for a mouse heart. (Frames 1-6) The heart contracts in the systolic phase until it reaches end-of-systolic (minimum) volume. (Frames 7-12) The heart expands in the diastolic phase until the end-of-diastolic (maximum) volume is reached again. The horizontal line of interference near the top of each image was caused by RF noise entering the MRI machine.

the heart surrounding this cavity do not change much with time; hence, they will be sparse in any time-based transform domain that can exploit this, such as the discrete Fourier transform (DFT) domain. By applying a classical 2D sparsifying transform such as the discrete cosine transform (DCT) or discrete wavelet transform (DWT) in the x-y domain, and subsequently applying a DFT in the time domain, the performance over the entire "video" will be improved after a 3D CS reconstruction is applied as compared to the case when a 2D CS reconstruction is applied to each frame individually.

2. THE COMPRESSIVE SENSING METHOD

The concept of compressive sensing is heavily rooted in information theory. The important "information" in an image can be condensed into fewer samples after a transformation is applied; CS is a method that exploits this "transform sparsity" to capture most of this condensed information directly by sub-Nyquist sampling. Randomness and sparsity play heavy roles in the quality of image reconstruction from this undersampled data in both 2D and 3D methods.

2.1 Compressive Sensing Background

The premise of compressive sensing centers highly upon the concept of *sparsity*. Consider a signal $\boldsymbol{x} \in \mathbb{C}^n$, where n is the full number of samples in the signal. A signal is considered to be S-sparse if only S of the samples are significant (i.e. above the noise floor of the image).

In general, the lower the ratio S/n, the more compressible a signal is and hence the better quality the reconstruction will be when CS is applied. Now, many signals are not very sparse in their original form, so image reconstruction in CS can make use of a *sparsifying transform*. A sparsifying transform can be defined as an orthogonal basis $\boldsymbol{\Psi}^{n*n}$ such that:

$$\boldsymbol{s} = \boldsymbol{\Psi}\boldsymbol{x}\,, \tag{1}$$

where $\boldsymbol{s} \in \mathbb{C}^n$ is the transformed signal. If the basis $\boldsymbol{\Psi}$ is chosen correctly, the resulting signal $\boldsymbol{s}$ will be much sparser than the original signal $\boldsymbol{x}$. The best sparsifying transform $\boldsymbol{\Psi}$ for these purposes is highly dependent on the data; for instance, the JPEG compression scheme uses a block-based discrete cosine transform to sparsify data because its target, photographic images, is well suited to compressibility with this transform.

In standard notation, $\boldsymbol{\Psi}$ is defined as a matrix which, if all of the pixels in the 2D image are arranged into a single dimension to form $\boldsymbol{x}$, can be multiplied by $\boldsymbol{x}$ to find the sparse result $\boldsymbol{s}$. Indeed, all sparsifying transforms can be expressed this way in matrix form if desired, but this is not advised; the main reason is because most common sparsifying transforms are computationally reducible (i.e. they can be computed a special way that takes much less time than a straight matrix multiplication). In this case, matrix multiplications for large values of n are unwieldy and unnecessary; for example, if the DFT is the sparsifying basis, a straight DFT matrix multiplication has a computational complexity of $\mathrm{O}(n^2)$, whereas the fast Fourier transform (FFT) method can do it with a complexity of $\mathrm{O}(n \log n)$.[3] Computational advantages apply to all of the transforms used in this paper; henceforth, $\boldsymbol{\Psi}$ *will instead be defined as a function of* $\boldsymbol{x}$ for all subsequent purposes.

Using only a small subset of the k-space data points, it is our goal to approximate the fully sampled MRI image as accurately as possible. Let $\boldsymbol{x} \in \mathbb{C}^n$ represent the full reconstructed image (in the pixel domain), where the visual representation of the image is contained in the magnitude data $|\boldsymbol{x}|$. Let $\boldsymbol{y} \in \mathbb{C}^m$ represent the randomly undersampled k-space points. In standard MRI, $m = n$, and so the process of moving from k-space to

the image domain is a simple matter of a 2D Fourier Transform on each individual video frame in question. However, when $m \ll n$, the reconstruction necessitates the solution of an underdetermined system for $\boldsymbol{x}$. A naive approach would involve simply filling in all of the unsampled k-space points with zeroes to create a full n-dimensional k-space, then performing the Fourier Transform as before. The problem with this is that zero-filling in this manner tends to create coherent aliasing, or noise that looks like multiple periodic superimposed copies of the image.[4] Using the concept of transform sparsity, we can create a much more accurate image reconstruction using the same points in k-space, provided we know of a transform domain where the image would be relatively sparse. In order to take advantage of this sparsity, we would like to minimize the following equation:

$$\min \quad f(\boldsymbol{\Psi x}) \quad \text{subject to} \quad \boldsymbol{Ax} = \boldsymbol{y}, \tag{2}$$

where $\boldsymbol{A}^{m*n}$ is an undersampled inverse Fourier Transform matrix, and $f(\boldsymbol{\Psi x})$ is a function that calculates a specific sparsity measure from the sparse representation of $\boldsymbol{x}$. The standard sparsity measure used in compressive sensing is the L_1 norm, defined as:

$$||\boldsymbol{s}||_1 = \sum_{i=1}^{n} |s_i| \quad \text{for all} \quad \boldsymbol{s} \in \mathbb{C}^n \tag{3}$$

This sparsity measure, while not as ideal other measures such as the Gini index,[5] is a good candidate for CS because its minimization is well-defined as a convex optimization problem.[6] The final equation, then, is:

$$\min \quad ||\boldsymbol{\Psi x}||_1 \quad \text{subject to} \quad \boldsymbol{Ax} = \boldsymbol{y} \tag{4}$$

This is a standard basis pursuit problem which is solvable with minimal computational complexity.[7] However, it does not always result in an optimal reconstruction when noise is involved. Another, more flexible minimization is the constrained L_1/L_2 problem, which is:

$$\min \quad ||\boldsymbol{\Psi x}||_1 \quad \text{subject to} \quad ||\boldsymbol{Ax} - \boldsymbol{y}||_2 \leq \delta \tag{5}$$

Here, the parameter δ can be adjusted to change the trade-off between transform sparsity and the noise floor of the reconstruction. Our experiments indicate that this problem takes longer to solve, but produces more accurate results. With computation power being so cheap (compared to MR imaging time), it is arguably the best choice for this application. It is also possible to include a TV (total variation) penalty if desired, but this will increase computation time. In the experiments performed in this paper, the YALL1 v1.4 ADM minimization algorithm[8] was used with $\delta = 5 \times 10^{-4}$ and a stopping tolerance of 1×10^{-4}.

2.2 2D Application of CS for MRI

Since only a small subset of the measurements are to be taken, proper selection of the samples to measure is of paramount importance. In general, the more "random" the sampling pattern, the more incoherent the undersampling noise will appear, and hence the better the resulting reconstruction will be.[4] However, when the sampling is done in the frequency domain (as MRI is), there is an additional consideration. It is well known that most natural 2D images have most of the "information" concentrated near the origin in the frequency domain, and MRI images are no exception.[9] Thus, a balanced approach to undersampling would use a frequency distribution that is more likely to pick points closer to the origin than further away; the Gaussian distribution is often used to this effect.[2] Also, it is important to note that, due to the nature of MRI data acquisition patterns, only sampling in complete lines can provide practical savings in scan time. Though research on new MRI pulse sequences is in progress, the available pulse sequences at the time of this writing cannot sample single points individually. Thus, this paper will only concern itself with line-based k-space sampling.

The process of image formation for ordinary time-varying applications of MRI is shown in Figure 2. A fully-sampled set of k-space data points is acquired at each of several time instances, much like a video camera captures full "frames" of image data at evenly spaced

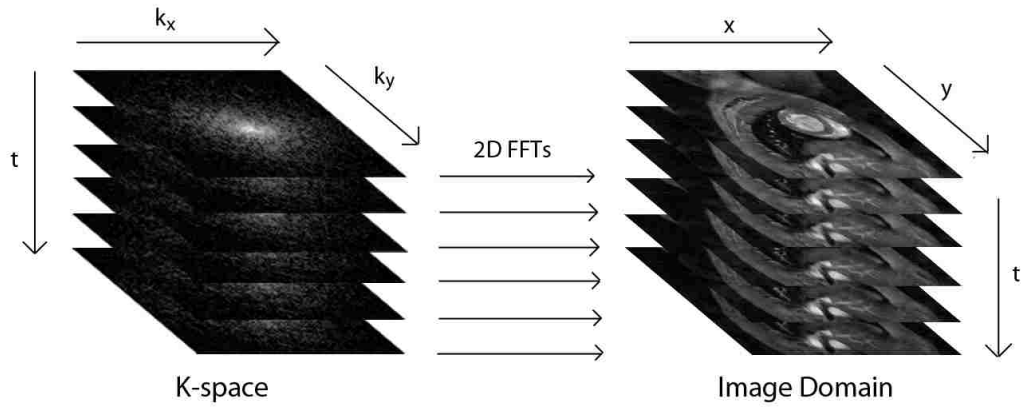


Figure 2. Example of image formation for non-CS mouse heart imaging. An entire set of k-space points is measured for each time period of interest, then a 2D FFT is applied to each frame to recover the image sequence.

points in time. However, since k-space cannot be sampled fast enough to keep up with a mouse's rapid heartbeat (300-500 beats per minute), an assumption of periodicity in the cardiac cycle is made (more details in Section 3.1). After all k-space points are measured, a 2D FFT is applied to each frame individually to create the final image sequence.

In classical 2D compressive sensing, each k-space image is treated as a separate reconstruction. As shown in Figure 3, a single set of k-space points is randomly undersampled, then an L_1 minimization is applied using an appropriate 2D sparsifying transform (such as the 2D DCT). This approach is optimal for static images, but for time-varying sequences of images such as those obtained during mouse cardiac imaging, time-domain sparsity can be considered as well, resulting in better overall reconstruction quality for the entire image sequence.

2.3 3D Extension to CS for MRI

In contrast to the 2D approach, our 3D approach treats an entire sequence of k-space images as a single problem, using a 3D sparsifying transform that can take advantage of sparsity in time. By treating the entire "video" as a single optimization problem, the computation time will be increased in comparison to a 2D approach, but the reconstructions will be of far better quality. Figure 3 shows the contrast between 2D and 3D methods; the 3D case

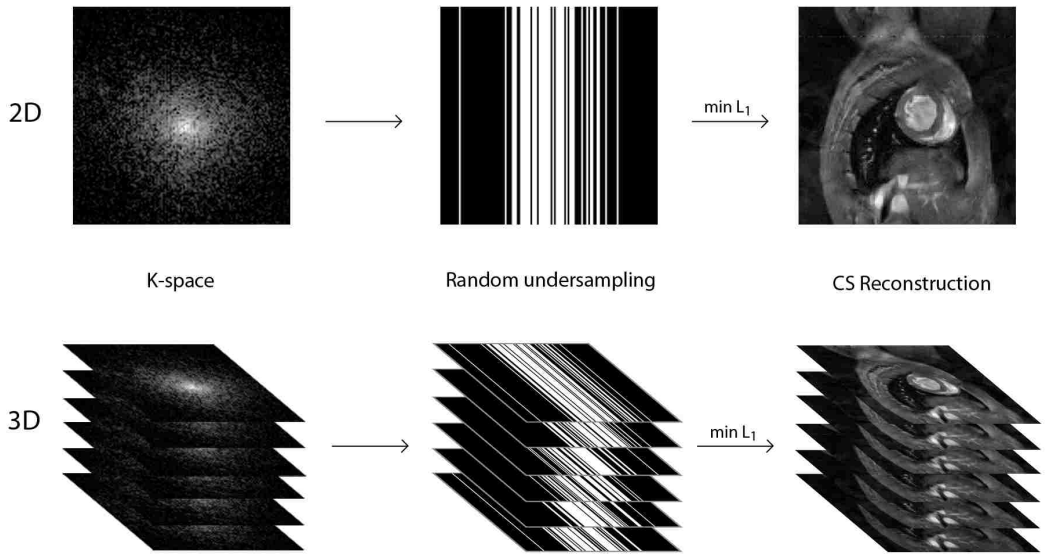


Figure 3. Contrasting 2D and 3D approaches to compressive sensing. The 2D method only performs minimizations on a single frame at a time; hence, it only exploits transform sparsity in space. The 3D method performs a minimization over the entire image sequence at once, so sparsity in time can be exploited as well. In both cases, the random undersampling patterns are shown using black-and-white masks where white pixels indicate k-space points that are sampled and black pixels indicate ones that are not.

considers the entire image sequence as a single L_1 minimization. Treating the entire set of images as a single minimization problem will increase total reconstruction time due to the fact that computation time increases more than linearly with problem size (especially when 3D sparsifying transforms are considered), but this is an acceptable trade-off for achieving better image quality.

In our experiment, sets of k-space data consisting of 12 frames of 128x128 pixels were used. Traditional 2D CS would dictate 12 separate reconstructions of size $n = 16834$. We can find the required number of samples m to acquire under line-based random sampling using:

$$m = \text{ceil}(\frac{rn}{128}) \,\text{x}\, 128 \tag{6}$$

where r is the approximate undersampling ratio (i.e. $r = 0.3$ for 30% undersampling) and ceil() is the operation of rounding up to the nearest integer. In 3D CS, both n and m are

simply multiplied by 12, resulting in a problem size of $n = 196608$ and $59904 \leq m \leq 79872$ for sampling rates between 30% and 40%. Using MATLAB with YALL1 v1.4 on a computer with an Intel Core2 Quad Q9550 CPU and 8GB of RAM, 3D CS reconstructions for L1/L2 constrained problems of this size using the parameters listed in Section 2.1 take between 11 and 120 seconds each depending on the sparsifying function $\mathbf{\Psi}$, the sampled MRI data, and the undersampling percentage. Further improvement on these times is possible via code optimization of the various 3D sparsifying functions, and also possibly with a massively-parallel GPU implementation.[10]

3. THE EXPERIMENT

The main goal of the proposed experiment is to compare the image quality obtained by 3-dimensional time-space CS reconstructions with that of the classical 2-dimensional approach in the mouse cardiac cycle. To do this, a set of seven image sequences was obtained from the University of Missouri VA Hospital, each of which contains 12 frames that form a single complete murine cardiac cycle (example in Figure 1).

3.1 Equipment and Imaging Procedure

During the MRI scans, the animals were anesthetized with around 2% isoflurane in oxygen and their heart rates were maintained around 360 beats per minute. The animals were positioned supine in a cradle, where ECG electrodes were inserted into the right forepaw and left rear paw and a respiration sensor was taped on the chest as shown in Figure 4. The cradle was inserted into a 300 MHz quadrature driven birdcage RF coil with an inner diameter of 38 mm (shown in Figure 5), which was then placed into the center of a Varian Unity Inova 7 Tesla 210 mm horizontal bore MRI system equipped with a 400 mT/m gradient system (example shown in Figure 6). The ECG was used to trigger the MR scan at a particular point in the cardiac cycle (for example. immediately after the R-wave). The animals' body temperature was supported by warm air circulated through the MRI bore, and the ECG and respiratory monitoring/gating as well as the body temperature maintenance were done using a small animal physiological monitoring system.

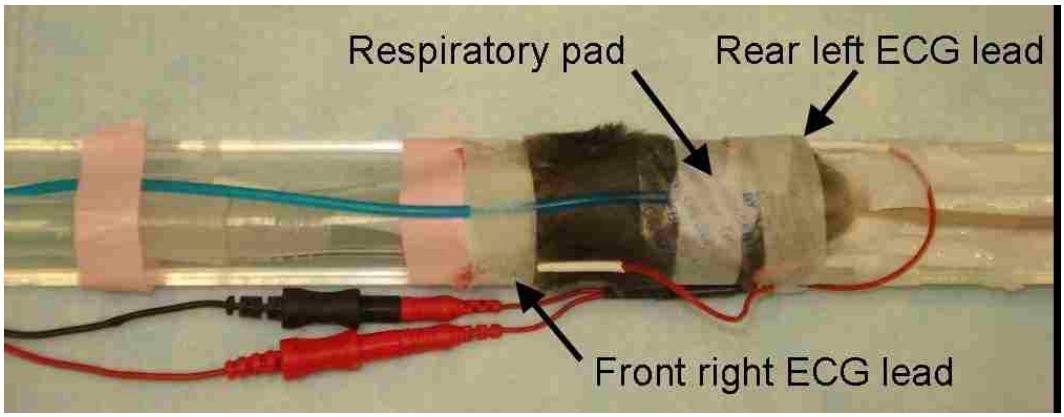


Figure 4. Picture of anesthetized mouse taped in cradle with sensors attached.

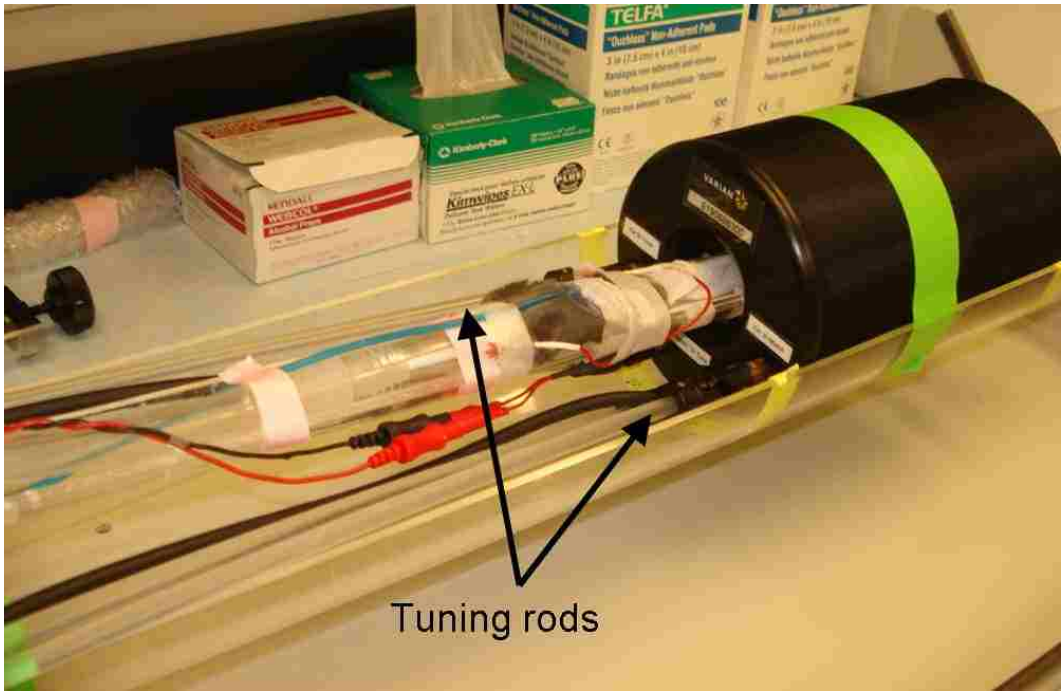


Figure 5. Picture of cradle being inserted into RF coil.

Figure 6. Example of full MRI system used in experiments.

Scout images at axial, coronal and sagittal planes were obtained at the very beginning of the procedure to localize the animal. Three images were acquired at each plane to confirm the correct positioning of the animal. After that, we applied a multi-slice gradient echo pulse sequence to obtain three sagittal images across the heart location based on the scout images. These sagittal images were used for determining the long-axis (coronal) plane of the left ventricle by positioning the slice across from the aortic valve to the apex. The short-axis plane was located by positioning the slice perpendicular to the long-axis plane. Finally, a typical steady state free precession (SSFP) sequence was applied to collect the first read-out line in k-space from each of the 12 frames in one cardiac cycle; this was then repeated for each subsequent line until the full set of measurements was acquired. The TR (repetition time) was calculated in real-time as the RR-delay (the time between R waves in the ECG) divided by 12, and the TE (echo time) was 1.57ms.

Since even a sedated mouse has a heart rate of over 300 beats per minute, there is

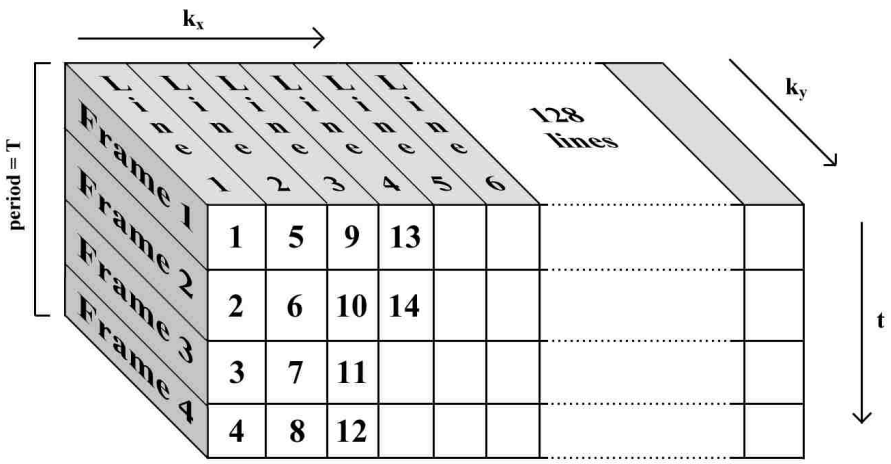


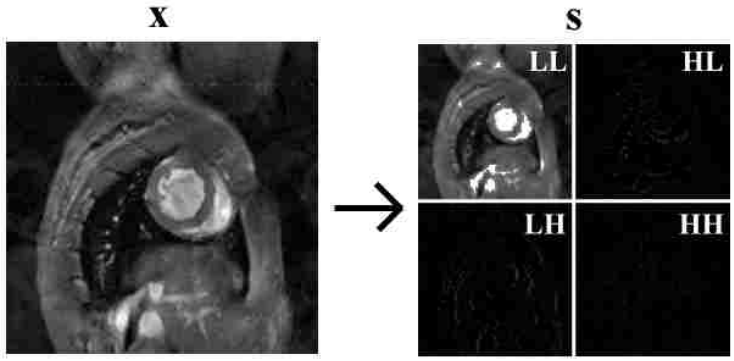
Figure 7. Example sampling method for acquiring a full video of a mouse heartbeat consisting of 4 frames. Starting with the leftmost line of k-space, the same line is measured 4 times in succession as the heart cycle progresses in time. After one full period T (i.e. one full heartbeat), the machine begins measuring the second k-space line as the heart goes through another full cycle. At the end, 128 lines will have been measured each of 4 times, taking a total of 128 full heartbeats to complete. Note that our experimental procedure divides the heart cycle into 12 frames instead of the 4 shown here.

obviously not enough time to sample anywhere close to a full set of k-space data before the heart moves to the next stage in the cycle. Because of this, the sampling scheme shown in Figure 7 was used to collect fully-sampled sets of k-space data for each of 12 stages in the cardiac cycle that are equally-spaced in time. The reason this works is because of the periodicity of the cycle. Each full heartbeat can be assumed to be identical (or mostly identical) to every other one; in this way, we can collect enough data for a full video in roughly 128 sequential heartbeats from the mouse. True undersampling performed in the pulse sequence can cut this down to 40 or less, resulting in fewer image artifacts due to the fact that not all heartbeats will look exactly identical.

3.2 2D and 3D Sparsifying Transforms

In CS, common 2D sparsifying transforms include the identity transform, the DFT, the DCT, and the DWT. The definitions of these are detailed in Table 1. For simplicity of analysis, the wavelet transform variants use periodic extension (i.e. the filters use circular convolution) to keep the size of the image unchanged upon transformation. The width and height of the image are specified by m and n, respectively. Also worthy of mention is that,

Table 1. Common 2D sparsifying transforms. In all cases, the transformed image $\boldsymbol{s} \in \mathbb{R}^{m \mathrm{X} n}$ is a function $\boldsymbol{\Psi}$ of the input image $\boldsymbol{x} \in \mathbb{R}^{m \mathrm{X} n}$, and j is the imaginary unit. The primary goal of these transforms is to increase the sparsity of $\boldsymbol{s}$ relative to $\boldsymbol{x}$.

Transform	**Description**
Identity	The identity transform $\boldsymbol{s} = \boldsymbol{x}$
DFT2	2D Discrete Fourier Transform $s_{u,v} = \sum_{i=0}^{m-1} \sum_{k=0}^{n-1} x_{i,k}\, e^{-j2\pi(\frac{iu}{m}+\frac{kv}{n})}$ for $0 \le u \le m-1, \quad 0 \le v \le n-1$
DCT2	2D Discrete Cosine Transform $s_{u,v} = \sum_{i=0}^{m-1} \sum_{k=0}^{n-1} x_{i,k} \cos[\frac{\pi}{m}(i+0.5)u] \cos[\frac{\pi}{n}(k+0.5)v]$ same u,v as above
DWT2	2D Discrete Wavelet Transform Let g_L be a wavelet low-pass filter and g_H be a wavelet high-pass filter Filter $\boldsymbol{x}$ by g_L in both the x-direction and y-direction, then downsample by a factor of 2 in both dimensions to get section LL Use g_L in the x-direction and g_H in the y-direction to get LH Use g_H in the x-direction and g_L in the y-direction to get HL Use g_H in both the x-direction and y-direction to get HH 

for the DCT, it is common to break the image up into a number of identically sized blocks before applying a 2D DCT to each block individually. For the DWT, it is common to repeatedly take the upper-left section and decompose it further and further. In this paper, an 8x8 pixel block DCT is used, and each DWT decomposition is performed to as many levels as MATLAB will allow.

In order to extend our analysis to three dimensions, it is necessary to apply a sparsifying transform that also extends to the time domain. The FFT was chosen as the most suitable

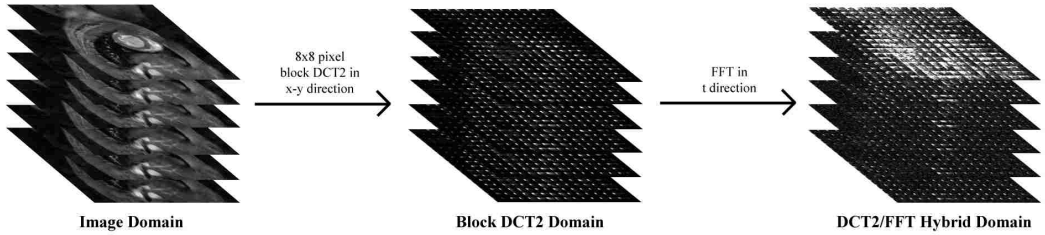


Figure 8. Example of a composite 3D sparsifying transform. First, a 2D DCT is applied on every individual 8x8 block of pixels in each frame of the video. Second, an FFT is applied along the time dimension to condense information into the first frame, which now contains all of the time-based DC terms.

transform for this purpose because of its computational simplicity in MATLAB as well as its propensity to condense the information from unchanging pixels into just one significant coefficient (the DC term). The full composite transform, then, consists of applying one of the 2D transforms from Table 1 on each frame of the video individually, then applying a 1D FFT along every pixel in the video as it changes with time. This process is shown in Figure 8 using the 8x8 pixel block DCT as an example of a 2D transform.

3.3 Analysis Method

The analysis method can be represented in pseudo-code as follows:

```
For undersampling percentages of x = 30%, 35%, and 40%, do these steps:
    For each of the 7 videos of interest, do these steps:
        Find visual reconstructions for the fully sampled k-space data
        For each of 3 random trials, do these steps:
            Use Gaussian random undersampling to pick x% of the k-space lines
            For each of 10 sparsifying transforms, do these steps:
                Perform constrained L1/L2 minimization with YALL1
                Calculate PSNR error metric of undersampled reconstruction

The 10 sparsifying transforms applied in the x-y dimensions include:

Identity transform
2D DFT
Full 2D DCT
8x8 pixel block 2D DCT
2D DWT with Haar wavelet
2D DWT with Daubechies wavelet of order 10
2D DWT with Symlet wavelet of order 2
2D DWT with Symlet wavelet of order 5
2D DWT with Coiflet wavelet of order 2
2D DWT with Coiflet wavelet of order 5.

The DFT is always the transform applied in the time dimension
when 3D reconstruction is attempted.
```

Because of variance in the random sampling patterns, it is necessary to repeat each reconstruction for a total of 3 random trials, each time picking a different sampling mask.

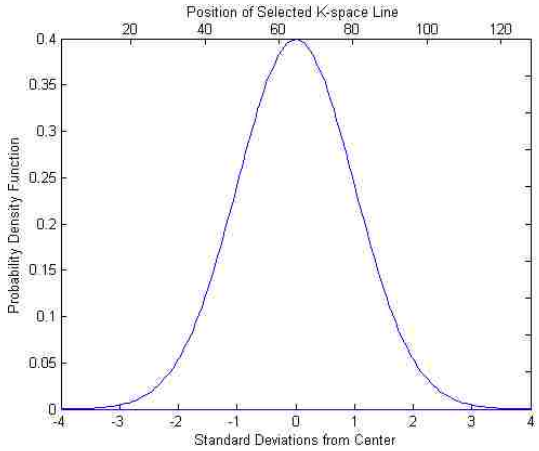


Figure 9. Gaussian PDF with $\sigma = 16$ pixels for random undersampling of a 128 line k-space grid. The mean of the distribution corresponds to line 65, and this line is always sampled even if the random sampling process does not select it (which is rare). Lines will be sampled from this curve until the required percentage of coefficients is reached. This process is repeated independently for every frame of 2D k-space data to form the final 3D sampling pattern.

The line-based sampling mask was drawn from a Gaussian probability distribution with a standard deviation of 16 pixels. The center line in k-space is the mean of the distribution, and it is always chosen for sampling no matter what. If a random result ends up choosing a line that either has already been chosen or is located past the image boundary, that result is discarded and a new line is chosen to replace it. Figure 9 demonstrates the probability of any given line being chosen at a point during random sampling.

One more consideration to note is that randomly chosen sampling masks may still not perform well, because it is very much possible to "randomly" generate a mask that happens to have a measure of coherence by chance. In order to insure the most incoherent sampling mask possible, we can generate several masks and choose the one with the least interference in the point spread function (PSF) as proposed by Lustig et al.[4] In our experiment, 100 different masks were generated each time a new sampling pattern was called for, and the one with the lowest peak PSF out of those was chosen for use.

3.4 Performance Metrics and Image Examples

In order to judge the success of our proposed CS method, and also to discern the best 2D sparsifying transform, it is necessary to use a performance metric. In theory, the transform

that results in the sparsest representation of the signal will create the highest quality image upon reconstruction, so a sparsity measure such as the Gini Index could be used. However, since all of the fully-sampled images are at our disposal, a more practical approach is to simply take the error of the undersampled reconstructions when they are compared to the fully sampled ones. The peak signal-to-noise ratio (PSNR) is a fairly standard measure of image quality that takes into account the average error, or "noise", of every pixel in the image; it is defined as:

$$PSNR = 10\log_{10}\left(\frac{n\max(\boldsymbol{x})^2}{\sum_{i=0}^{n-1}(x_i - \hat{x}_i)^2}\right) \quad \text{for all} \quad \boldsymbol{x}, \hat{\boldsymbol{x}} \in \mathbb{R}^n, \tag{7}$$

where $\max(\boldsymbol{x})$ is the maximum possible intensity of any single point in the image, $\boldsymbol{x}$ is the original image, and $\hat{\boldsymbol{x}}$ is the "estimate" of that image; in our case, it is the reconstruction. This metric can be easily applied in our case, and since it is so widely used as a measure of image quality, it also provides a way to compare our results to those of other MRI CS studies. For this reason, and others discussed by Huynh-Thu and Ghanbari,[11] the PSNR was chosen as the main performance metric for this study.

Figure 10 provides a demonstration of the various PSNR levels for CS reconstructions with different transforms. These images were generated by performing 2D reconstructions and adjusting the undersampling rate for each transform until the desired PSNR level was reached. The left side of the figure contains the fully sampled image for reference, and the right side shows the artifacts of CS undersampling on the left ventricle (of which estimating the volume is the main objective for this type of imaging) as the PSNR drops. The identity transform tends to cause moderate-to-severe aliasing artifacts as the sampling rate goes down, whereas the 8x8 pixel block DCT presents the classic "block artifact" seen in heavy JPEG compression. On the other hand, when certain wavelet transforms are used, they tend to exhibit noise that is less disruptive in nature, and as a result tend to require fewer samples to achieve the same subjective image quality as other transforms.

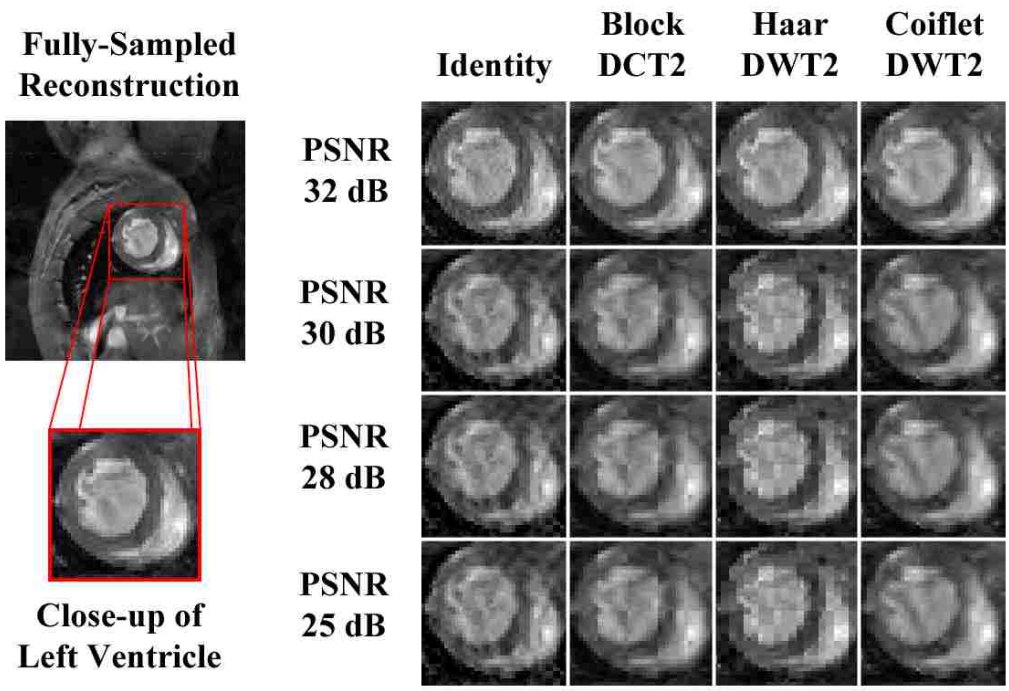


Figure 10. Examples of PSNR ratings for various 2D CS reconstructions. Sparsifying transforms include the identity transform, the 2D DCT performed upon each 8x8 pixel block, the 2D DWT with the Haar wavelet, and the 2D DWT with the third-order coiflet wavelet. The full k-space data set from which samples are drawn is the same for all reconstructions. Images with PSNR below 25 dB are mostly too distorted to be useful, so they are not shown.

3.5 PSNR Results for 2D and 3D Compressive Sensing on Mouse Hearts

To obtain a good idea of the significance of 3D reconstruction, it is necessary to compare it to the 2D case. Upon running the analysis detailed in Section 3.3 and obtaining the PSNR metrics, we can deduce the general performance for a particular undersampling rate by averaging the obtained PSNR over all images for each transform in turn. This analysis was performed for the following two cases:

1. All 84 frames (12 frames in each of 7 videos) were considered as independent images, and classical 2D CS reconstruction was performed on all of them using each of 10 different 2D sparsifying transforms.

2. Each of the 7 videos was considered as a single 3D CS reconstruction. The same 2D transforms as the first step were applied to each image frame, but an FFT was

Table 2. PSNR results for each sparsifying transform over all images for 30%, 35%, and 40% undersampling. The best-performing transform for each case is highlighted in red. The 3D method significantly increases image quality compared to the 2D method even at lower sampling rates.

2D Sparsifying Transform	PSNR (dB) for 2D CS			PSNR (dB) for 3D CS		
	30%	35%	40%	30%	35%	40%
Identity	24.78	25.97	27.08	31.61	32.45	33.28
DFT2	24.26	25.70	27.32	29.81	31.26	32.58
DCT2, 8x8 pixel blocks	26.12	27.41	28.67	31.78	32.93	33.68
DCT2, full image	23.63	25.15	26.71	30.76	32.00	33.11
DWT2, Haar wavelet	25.68	26.81	28.02	31.27	32.22	33.20
DWT2, Daubechies order 10	26.09	27.48	28.76	32.24	33.14	34.07
DWT2, Symlet order 2	26.20	27.39	28.63	31.64	32.46	33.50
DWT2, Symlet order 5	26.33	27.64	28.92	32.25	33.25	34.24
DWT2, Coiflet order 2	26.34	27.68	29.00	32.29	33.30	34.19
DWT2, Coiflet order 5	26.15	27.64	29.04	32.36	33.26	34.14

performed in the time direction as well before performing an L_1 minimization on the whole video at once.

The final results for 2D and 3D CS at undersampling rates of 30%, 35%, and 40% are shown in Table 2. Under 2D reconstruction, the images exhibit a best-case PSNR of between 24 and 29 dB depending on the chosen transform and the number of k-space lines sampled. Referring back to Figure 10, the area of the image corresponding to the left ventricle is still somewhat distorted in this range, though it is still possible to roughly estimate the volume. In order to obtain a more acceptable PSNR, higher undersampling percentages would be required under 2D compressive sensing. However, the 3D reconstructions offer an average PSNR of over 32 dB when the best transform is used, even with only 30% undersampling. The PSNR for 30% undersampling (which is a standard target undersampling rate in MRI CS) experiences an average increase of around 6 dB, which is a major improvement.

In terms of PSNR, the best performing 2D transforms were the higher-order wavelets, but wavelet transforms also offer another advantage over the DCT block transforms that is not reflected in the PSNR. As Figure 10 shows, the block artifacts in the DCT distort the shape and size of the left ventricular cavity more than the time-frequency artifacts in the

wavelet transforms, and so the volume measurement will be impacted negatively. Clearly, the high-order symlet and coiflet wavelets should be used whenever it is computationally feasible in order to obtain the best image quality.

4. CONCLUSIONS

The analyses performed in this paper demonstrate that exploiting time sparsity in murine cardiac MRI with compressive sensing offers a significant improvement to image quality. Furthermore, since the reduced measurement time will reduce motion artifacts from the heartbeat, the results in practice should be even more impressive than those predicted by the experiment performed here, given that we simply discarded points from sets of fully (and thus slowly) sampled data.

Warranting future exploration is the use of sparsifying transforms other than the FFT in the time domain for better sparsification. Expanding the dictionary of 2D transforms (adding more high order wavelets in particular) would also allow the algorithm to explore more possibilities for 2D transform sparsity, further improving performance. Finally, as proposed by Zonoobi et al.,[5] using a stochastic algorithm to minimize the Gini index rather than the L_1 norm could provide even greater reconstruction accuracy, as the Gini index is usually a much better measure of sparsity than the L_1 norm.

ACKNOWLEDGMENTS

Special thanks are due to Ming Yang and Dr. Lixin Ma at the Veteran Affairs Hospital and the University of Missouri - Columbia, who provided all of the raw MRI data for this study. Also due thanks are Hamed Kajbaf and Zengli Yang, who contributed knowledge of CS methods as well as pieces of MATLAB code, and Ken Hanson, for providing the LaTeX template for SPIE manuscripts. This work was financially supported by the University of Missouri Research Board Fund.

REFERENCES

[1] Kajbaf, H., T.Case, J., Zheng, Y. R., Kharkovsky, S., and Zoughi, R., "Quantitative and Qualitative Comparison of SAR Images from Incomplete Measurements Using Compressed Sensing and Nonuniform FFT," *RADAR Conference IEEE 2011*, 592–596 (2011).

[2] Wech, T., Lemke, A., and Medway, D., "Accelerating Cine-MR Imaging in Mouse Hearts Using Compressed Sensing," *Journal of Magnetic Resonance Imaging* **34**(5), 1072–1079 (2011).

[3] Cooley, J. W. and Tukey, J. W., "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation* **19**(90), 297–301 (1965).

[4] Lustig, M., Donoho, D. L., and Pauly, J. M., "Sparse MRI: The Application of Compressed Sensing for Rapid MR Imaging," *Magnetic Resonance in Medicine* **58**, 1182–1195 (2007).

[5] Zonoobi, D., Kassim, A. A., and Venkatesh, Y. V., "Gini Index as Sparsity Measure for Signal Reconstruction from Compressive Samples," *IEEE Journal of Selected Topics in Signal Processing* **5**(5), 927–932 (2011).

[6] Donoho, D. L., "Compressed Sensing," *IEEE Transactions on Information Theory* **52**(4), 1289–1306 (2006).

[7] Chen, S. S., Donoho, D. L., and Saunders, M. A., "Atomic Decomposition by Basis Pursuit," *SIAM Review* **43**(1), 129–159 (2001).

[8] Zhang, Y., Yang, J., and Yin, W., *Users Guide for YALL1: Your ALgorithms for L1 Optimization*. http://www.caam.rice.edu/~optimization/L1/YALL1/User_Guide/YALL1v1.0_User_Guide.pdf.

[9] Fuderer, M., "The Information Content of MR Images," *IEEE Transactions on Medical Imaging* **7**(4), 368–380 (1988).

[10] Andrecut, M., "Fast GPU Implementation of Sparse Signal Recovery from Random Projections," *Engineering Letters* **17**(3) (2009).

[11] Huynh-Thu, Q. and Ghanbari, M., "Scope of validity of PSNR in image/video quality assessment," *Electronics Letters* **44**(13) (2008).

II. Utilization of MATLAB Simulink Exercises for an Undergraduate Communications Course

Brice Hirst, Yahong Rosa Zheng

Abstract

A set of six MATLAB Simulink laboratory exercises was previously designed in 2011 for an undergraduate entry-level communication course. This paper presents our experience of a pilot test on these exercises, followed by their modification and enhancement, and concluding with an application of the modified exercises in the Spring 2012 offering of the course. The pilot test by a student in the Spring 2011 class identified several areas of improvement for the previous design of the lab exercises, including details of lab instructions, high level of difficulty in the first two labs, and partial completeness of the last two labs. Significant effort was then put into the modification and re-design of these labs in Fall 2011, and the enhanced labs were applied in the Spring 2012 semester in a class with an enrollment of nine students. Feedback from students was solicited after each lab exercise. The results show that the Simulink labs were well received by students, in comparison with a traditional lecture-only approach or a MATLAB script programming approach. Some slight changes were also made to address the minor flaws in these labs. Currently, the Simulink labs are ready to be disseminated to the public and are available for free download by other universities offering similar courses.

Background

The first communications course at our undergraduate level covers three main topics:

1) Review of linear systems topics such as the Fourier series, the Fourier transform, power spectral analysis, and impulse/frequency responses.
2) Basic analog modulation and demodulation techniques such as Amplitude Modulation (AM), Frequency Modulation (FM), and Phase Modulation (PM).
3) Digital baseband transmission concepts such as line coding, pulse shaping, Inter-Symbol Interference (ISI), and Zero Forcing (ZF) Equalization.

This course is open to students who have taken continuous-time linear systems but not necessarily discrete-time linear systems. It is meant to provide students with a theoretical foundation for advanced courses such as Communication Systems II, Communication Circuits, and Wireless Communications. The course has been traditionally offered by the lecture-only approach; however, in a couple of semesters, lab projects using MATLAB scripts offered in the textbook were incorporated into the course in addition to the lectures. Although the MATLAB projects provide a good aid for learning the heavy communications theory and are welcome by many of the students, more than 50% of the enrolled students were not ready for the extensive MATLAB programming. This is mainly due to their limited training in MATLAB scripting and lack of the foundation of discrete-time linear systems. Therefore, a set of Simulink projects was designed in Spring 2011 for this course[12] because Simulink offers easy-to-use block diagram models that inherently take care of the sampling issues.

MATLAB Simulink has been used as an educational tool for teaching block-diagram based simulations in other institutes and for many courses[13,14,15]. It is also widely used in industry and research institutes for real-life applications in areas such as control system design and signal processing.[16,17] Advantageous for the beginner is that Simulink has an intuitive point-and-click interface, where circuit blocks can be drag-and-dropped from a library, then connected with wires. Figure 1 shows a basic illustration of the Simulink design layout, in this case used to create a graphical representation of a third-order FIR (finite impulse response) filter.

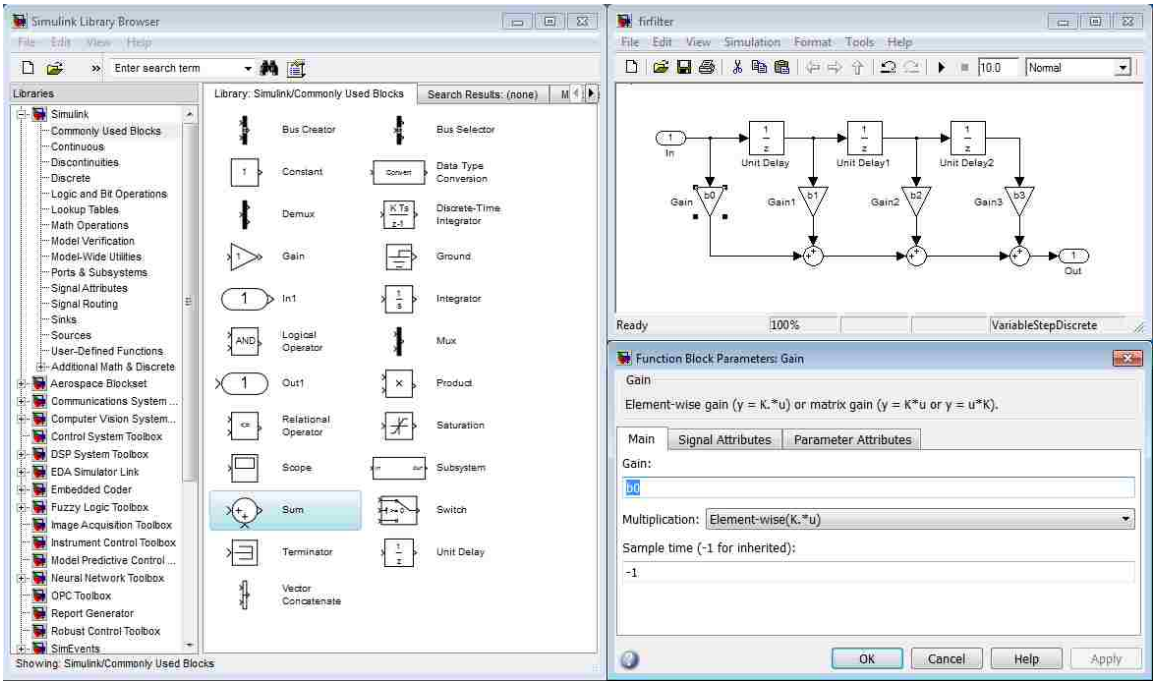


Figure 1 – Example of Simulink program interface.

Despite its ease-of-use in creating the most basic of block diagrams for simulation, the Simulink software harbors many unique nuances that must be understood in order to make good use of the software. The original design of the Simulink lab

exercises aimed at teaching the most important skills relevant to Simulink in combination with teaching concepts from communications theory. The layout of the six lab projects was well thought out, as shown in Table 1. The original lab manuals were created by following Gagne's pedagogical model of nine levels of instruction.[18]

Table 1 - Simulink Laboratory Projects covering theory in combination with Simulink skills

Lab	**Topics covered**	**Simulink skill**
I	Frequency Domain Analysis	Building a Model
II	Linear Systems	Subsystems & Masks
III	Amplitude Modulation	Library Building
IV	Frequency & Phase Modulation	Model Referencing
V	Pulse Code Modulation & Line Codes	Using Stateflow
VI	Zero Forcing Equalizer	Interacting with MATLAB

Pilot Test of the Original Labs

A pilot test of these labs was performed by the first author of this paper, who took the Communications course in Spring 2011 that used the MATLAB scripting approach. That author then became a master's student in Fall 2011 and performed the pilot test. Despite being an outstanding student in the Spring 2011 class and having just learned the theory thoroughly, the first author had to make a great effort to follow the original lab instructions and had to spend a large amount of time to perform the tasks required in the lab manuals. The pilot test identified several areas of improvement for the original design of the lab exercises. These include:

1) Unrealistic assumptions:

The original design required that the students work through a Simulink demo before performing the lab exercises so that they gain the basic skills of making a model, using the Simulink help files, and navigating through libraries. The lab exercises were then designed with expectations and requirements that were far too high for the level of the course. In practice, it is rare that an undergraduate student in our university would spend much time on a demo if it is not an assignment that would be graded. It is also difficult for an average undergraduate to grasp all of the required skills just by working through a single demo.

2) Lack of details in lab instructions:

Based on the unrealistic assumption that a student would know where to find help instruction in Simulink, the original lab manuals contain limited instructions on how to perform the lab exercises. The first author attempted to perform these labs as an undergraduate, having no prior Simulink experience, and was repeatedly confused by what was being asked and how to accomplish it. It was simply too difficult to search through the help manuals trying to figure out how to perform the tasks required by the lab manual. The frustration was worse during the first two lab exercises because the student just started to learn the basic of Simulink skills and was overwhelmed by the large size of the Simulink help system, most of which is designed as a reference rather than a tutorial.

3) Inappropriate level of difficulty and unrealistic requirements:

The level of difficulty for all six of the original labs was considered too high for average undergraduate students at our university, especially for this first communications course. Some models required advanced knowledge in Digital Signal Processing (DSP) or Digital Communications that are not prerequisites of this course. The amount of work required by the original lab manuals was also enormous and unrealistic for the given length of lab sessions. For example, multiple types of modulators and demodulators were required in Lab 3 and Lab 4 that could not be done in a two-hour session.

4) Incomplete lab designs:

The original designs of Lab 5 and Lab 6 were partially functional except for a few example modules. Lab manuals for these two labs were completely missing.

5) Selection of Simulink solvers:

The original design of the labs used a large amount of library blocks that utilize continuous-time (CT) solvers such as ODE45. The primary reason for selecting the CT solvers was that the theory in the course is based mainly on CT systems and some students may not have a discrete-time linear system background yet. However, the available CT blocks in Simulink are very limited, thus making the tasks difficult to accomplish. On the other hand, Simulink has a much larger selection of discrete-time (DT) blocks that not only run much faster than CT blocks, but also provide more convenient solutions to the tasks required in the labs.

In addition, the pilot test found out that the Simulink help manual fails to explicitly teach/emphasize the difference between the DT and CT solvers. This caused a significant amount of confusion and grief to both the original and new designers of the labs. After balancing the pros and cons of the DT and CT solvers, the decision was made to switch all labs but one to the DT solver. The ODE45 CT solver was kept for Lab 2 to explicitly teach the difference between the two types of solvers in Simulink. All of the other five labs now use DT solvers, with the lab manuals instructing the students to set specified sampling frequencies for signal sources and having all subsequent blocks inherent this sampling frequency. This approach allows the students to use the DT models in Simulink without requiring them to understand the Nyquist sampling theorem.

Redesign of the Laboratory Exercises

After the pilot test, significant effort was put into the modification and re-design of these labs to address the issues that were found. The main considerations in modifying the labs were reorganization of objectives, addition of informational content, and adjustment of difficulty. The coverage and layout of the six lab projects were kept the same as shown in Table 1. Since the labs are meant to be supplemental to a lecture course, the number of required tasks in each lab was trimmed down to fit into a 2-hour lab session per project; while the excluded tasks were listed as extra credit for more capable students. Those extra tasks may be used for a full laboratory course if needed. The manual instructions were completely rewritten to provide much more detail for each lab. In particular, the instructions for the first two labs provide the student with step-by-step procedures to perform each task in order to learn the basics of Simulink without

searching the help manual. The instructions to the later labs gradually reduce the amount of explicit details and instead provide general guidance. All labs except Lab 2 were redesigned to use Simulink DT blocks rather than CT blocks.

Although the difficulty and scope of the labs were reduced overall to create an experience that is not overly demanding yet still educationally significant, the amount of information in the lab manuals pertaining to the remaining material was increased. Also, explanations for the functioning of certain Simulink features (such as CT vs. DT solvers and normal vs. accelerated simulation modes) were added; it is always a good idea for students to understand the "why" for the use of a software feature rather than to be told to "just do it this way because the lab manual says so."

Overview of the Newly Designed Lab Exercises

The complete set of exercises presented in this paper consists of six labs designed to build understanding in the fundamental operations of Simulink as well as provide experience in applying concepts learned in the communications systems lecture. It is important to note that, since these exercises were taught as an additional requirement of a 3-credit hour lecture course (class meets 3 hours per week), less material is included than would be expected for a full laboratory course. However, some extra credit material that was cut from the original set of exercises could most certainly be added back in to create a new laboratory course.

The six lab exercises under test were as follows:

Lab 1 - Intro to Simulink and Frequency Analysis *(shown in Figure 2, left side)*

The first exercise covers the most basic of operations in Simulink, which includes placing blocks from libraries, connecting them together, and using basic signal sources and scope outputs. Basic FFT operations and Fourier series signal representations are also covered here. Since the students are assumed to have no prior experience with Simulink at this point, the lab manual consists of very explicit instructions right down to every mouse click and keypress. This way, the students are eased into the Simulink program and can come away from the first lab feeling good about it.

Lab 2 - Hierarchical Design and Linear Systems *(shown in Figure 2, right side)*

The second exercise steps up the complexity by introducing subsystems and masks. Students are required to use these tools to implement a switchable source similar to the built-in Signal Generator block, which is then used in another model. A demonstration of system linearity is also performed here using transfer function blocks. This is the only lab out of the six that uses a continuous-time Ordinary Differential Equation (ODE) solver in Simulink; all other exercises use discrete-time solvers, which are much more practical and flexible in most cases.

Lab 3 - Simulink Libraries and Amplitude Modulation *(shown in Figure 3, left side)*

The third exercise requires students to create their own Simulink library with custom-made amplitude modulator and demodulator blocks. Unlike the previous two labs, the students are not explicitly told what blocks to use in the AM modulator; they are required to call upon previously learned knowledge to construct a model that simulates

the basic AM equation. Also, the demodulator was designed as a product detector, which must be implemented by finding and using a low-pass digital filter. Both of these tasks serve to reinforce previously learned material while expanding upon the knowledge of how discrete-time sampling works in Simulink.

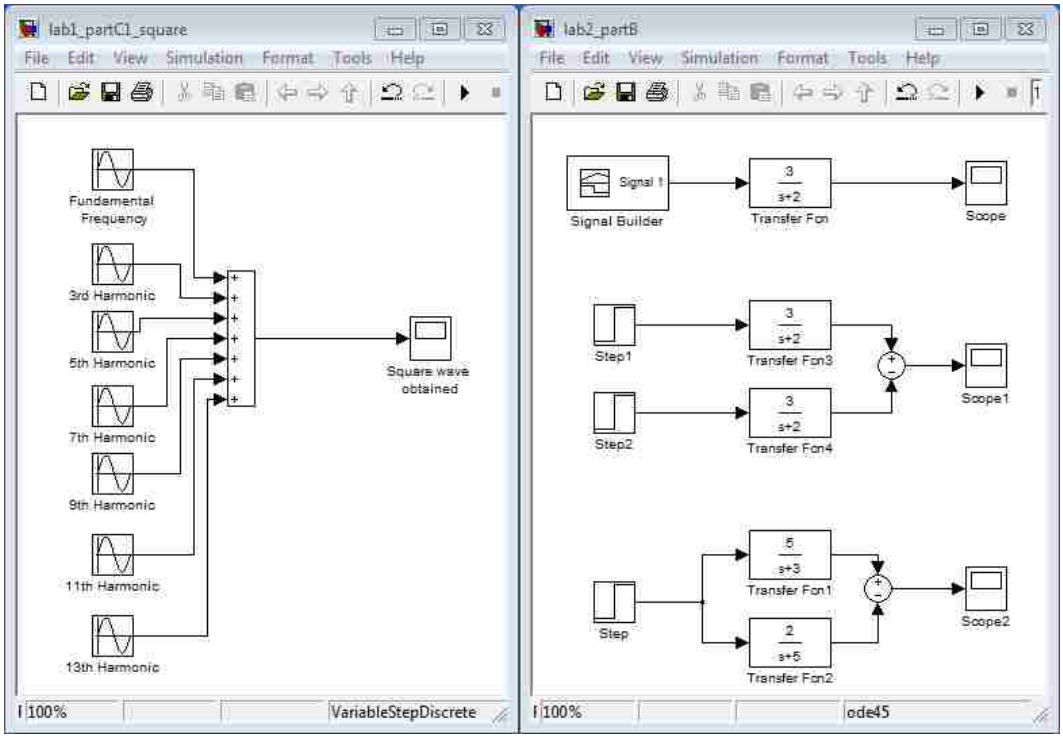


Figure 2 – Example screenshots from Simulink projects.
Left: In Lab 1, students construct a square wave from the sum of its Fourier series harmonics.
Right: In Lab 2, students visually observe the property of system linearity.

Lab 4 - Model Referencing and Angle Modulation *(shown in Figure 3, right side)*

The fourth exercise centers around the creation of PM and FM modulators, this time using model referencing rather than simple subsystems or libraries. When model files are referenced in this manner, they can be accelerated by pre-compiling code, which is one of the important features of Simulink that is worth knowing about. Basic information about PM and FM is given, but students are still required to seek out and

learn about the VCO block as well as external triggering for a sinusoidal source. This lab is not quite as long as the previous one, so it gives students a slight reprieve before the fifth one. Also, students are allowed extra credit by building an FM demodulator from scratch, which is quite difficult for a beginner at Simulink but allows students who might have missed a lab to catch up.

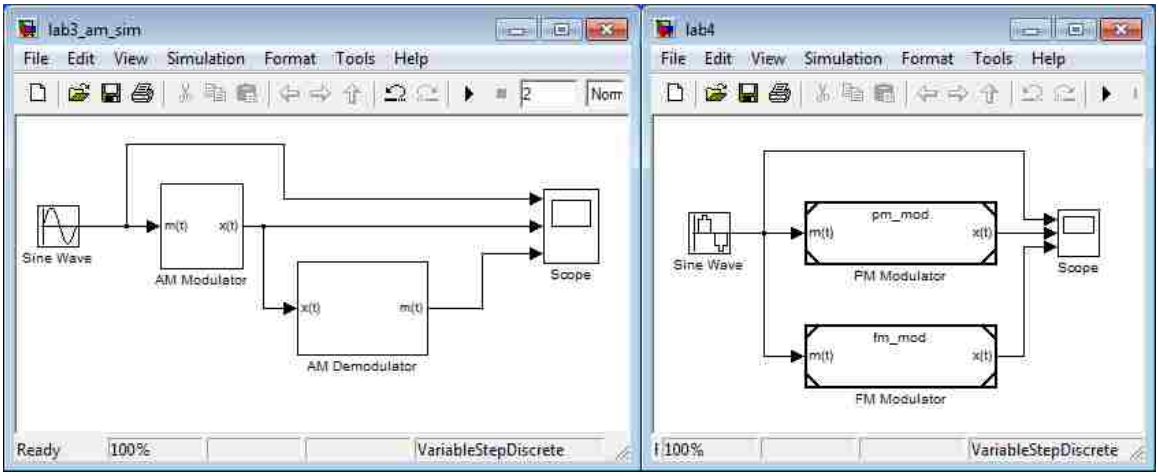


Figure 3 – Example screenshots from Simulink projects.
Left: In Lab 3, students construct AM library blocks which are used to modulate/demodulate an input.
Right: In Lab 4, students use external model file referencing to create PM and FM modulators.

Lab 5 – Line Coding and Decoding *(shown in Figure 4, left side)*

Unlike the previous four lab exercises, no new Simulink features are presented in the fifth one. Instead, students are asked to research various line coding schemes before coming into class, then implement them in Simulink from scratch using blocks of their choice. This lab is the turning point where students are freed from detailed instructions and are expected to experiment and look up blocks that will allow them to accomplish the task at hand. By requiring students to invent their own solutions, they learn how to

complete tasks on their own rather than being led by the hand as was done in previous labs.

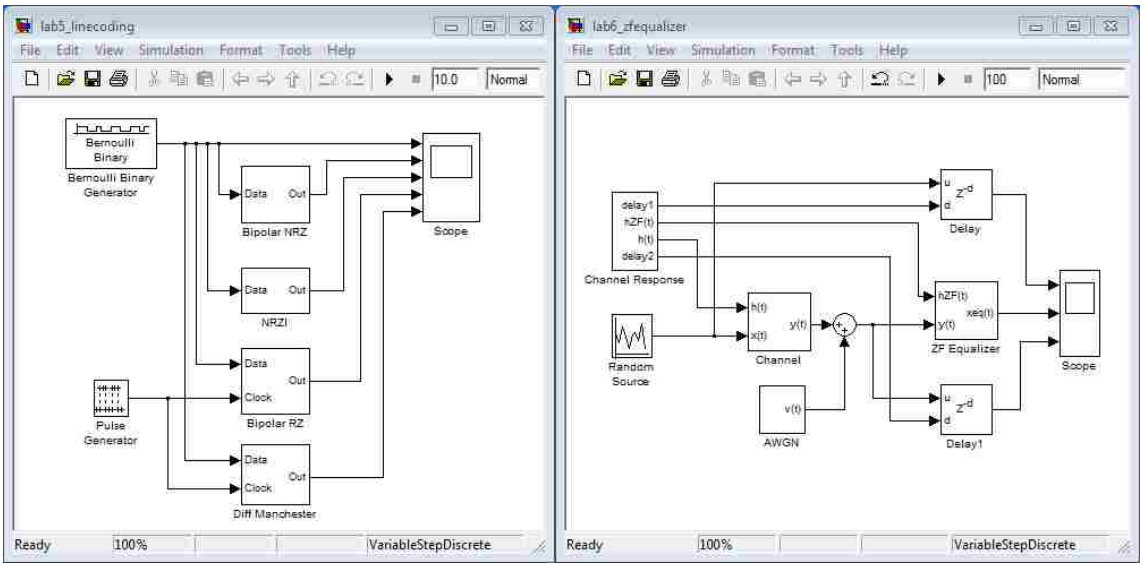


Figure 4 – Example screenshots from Simulink projects.
Left: In Lab 5, students construct subsystems that implement common line coding techniques.
Right: In Lab 6, students observe the effects of a ZF equalizer on a static channel with additive noise.

<u>Lab 6 – Zero-Forcing Equalization</u> *(shown in Figure 4, right side)*

The final exercise is mainly an observation task where students connect pre-constructed blocks to create a zero-forcing equalizer that removes ISI from a signal sent through a static channel with additive white noise. Since most of the model construction is already provided for the students, the emphasis of this lab is on noting the effects of the equalizer as various parameters (such as input signal type, static channel response, noise power, and equalizer length) are adjusted. In research, careful observation and interpretation of simulation results is just as important as constructing the simulations in the first place, so this final exercise rounds off the educational experience quite well.

Each of the six lab exercises consists of three major tasks. These are:

1. Preliminary – There is a short period of time between the posting of each lab manual for download and the start of the corresponding in-class lab session, and there is often some preliminary work that is to be done during this time so that the student can come to the lab session with all of the knowledge necessary to perform it. The preliminary work usually consists of some simple calculations or research; examples include calculating the Fourier series for some simple waveforms before the first exercise, and researching line coding schemes before the fifth one.

2. Procedure – The actual construction and simulation of models in Simulink is performed during in-class laboratory sessions where teacher's assistants are available in person to ask for help. The simplest lab exercises may only require a single 1-hour session to complete, whereas more complicated ones (such as the line coding exercise) may require two or more.

3. Report – After completing a lab exercise, a detailed report on what was done and what results occurred is due before the next lab session. The content requirements for the lab report are documented in the syllabus, so there should be little ambiguity over what material is expected. In each report, a series of post-lab questions pertaining to theory and/or lab results must also be answered.

This three-step approach is designed to obtain retention of material, as well as reinforcement of previously learned concepts from linear systems and communications. Since this set of labs is designed to be taught alongside a course in electrical engineering communications, the material in the labs should be sequenced so that the corresponding lectures have covered it a week or so in advance. For example, not much will have been learned during the first few weeks of lecture, so the first two labs only present material from linear systems, which is a standard prerequisite for communications. The remaining four labs present material from communications in the order they are usually taught, starting with AM, then progressing to PM, FM, line coding, and finally equalization. By lining up the material in this manner, concepts from communications will be subject to a greater degree of retention.

Utilization of the Newly Designed Labs in Classroom Teaching

The enhanced labs were applied in a communications course in Spring 2012 with an enrollment of nine students. The lecture portion of the course was conducted in a normal classroom with three one-hour sessions per week. The lab projects were added in weekly one-hour sessions as the lecture progressed. The first author served as Teaching Assistant (TA) for the lab sessions and graded the students' lab report work. A preliminary exercise was required before most labs, and the students were required to conduct the lab exercises in the designated computer learning lab during the allocated lab sessions. Each student worked individually through the lab exercises. Only five students conducted the sixth lab because only the best 5 out of 6 lab reports were counted in the final grade. The full set of labs was worth 20% of the final grade in the course.

The final grading scheme for the laboratory reports is as follows. These descriptions for each required section of the report are taken directly from the syllabus presented to the students:

Objectives (5%) – Basic objectives of the lab, in your own words.

Procedure (10%) – Brief description of how you did the lab exercise, step-by-step. This section does not have to be long; it is not necessary to copy the steps from the lab manual.

Results (30%) – The final results of the lab, including any screenshots and figures requested in the lab manual. Include any important observations or difficulties you had in this section.

Conclusions (25%) – Describe any important conclusions you have taken away from the lab here. Important things to consider for this section are: *What have you learned from this lab? How is the material here important in the real world? Overall, how valuable was this lab to your learning of the EE243 material in your opinion?*

Answers to Post-Lab Questions (10%) – Simply provide the answers to the post-lab questions here. Nothing fancy needed here; short-answers will be fine.

Model Files (20%) – Submit all model files, M-files, and other related files resulting from your completion of the lab as email attachments with your report.

This format is fairly standard in undergraduate lab reports at our university; since students were likely already familiar with this arrangement, it was a sensible choice.

Results and Student Feedback from Laboratory Sessions

Feedback from students was solicited after each lab exercise in order to gauge teaching effectiveness. The results show that the Simulink labs were well received by students, in comparison with the traditional lecture-only approach and the MATLAB script programming approach. Ordinarily, the communications course includes MATLAB-based exercises instead; the students seemed pleased that this was being replaced with a chance to learn the more visually-oriented Simulink.

Though the labs themselves were received fairly well, there was still a bit of displeasure among the students due to the fact that they felt this was too much extra work for a 3 credit-hour lecture course. Nonetheless, when it became well-known that this set of exercises was the alternative to doing the MATLAB exercises and the total amount of work would remain the same for the whole semester, the students reverted to a somewhat neutral stance.

A chronological breakdown of the student experiences for each lab exercise is as follows:

Lab 1 - Intro to Simulink and Frequency Analysis

Since the tasks necessary to complete the first lab were outlined very clearly step-by-step in the lab manual, students had little trouble with it. The main purpose of this lab was to familiarize students with the most basic operations in Simulink, such as connecting blocks and running simulations, and in that capacity it accomplished its purpose. The only difficulty experienced by students was in the preliminary for the lab, where students were expected to calculate Fourier series coefficients, which would then be used to construct basic waveforms from a sum of sinusoids. However, many of them had trouble with this part, even when given the coefficients directly, because of some oddities in the way Simulink handles sine wave sources. Overall, the smooth conduct of the first lab raised the students' interest in Simulink.

Lab 2 - Hierarchical Design and Linear Systems

The second lab was more in-depth than the first, and some students were confused by the instructions for masking subsystems. Since the author of the lab manual was present to clarify what needed to be done, the lab was completed by the students successfully, but since the lab exercises may not always be taught directly by their author, several clarifications were made in the lab manual itself for use in future classes.

Lab 3 - Simulink Libraries and Amplitude Modulation

The third lab included the first instance where students were told to create a model to accomplish a task without explicitly being told which blocks to use. Most students were able to create the AM modulator successfully without hints, but a few did

need to be led in the right direction first (i.e. they were told how many gain and sum blocks they needed, but not the order in which they needed to be arranged). The demodulator was trickier because it involved a digital filter that is generally learned in a higher level DSP course, but it was nevertheless essential to create a working product demodulator. Therefore, the lab manual went into detail on how to make the model without explaining how the filter works; as a result, most of the students performed this part very well.

Lab 4 - Model Referencing and Angle Modulation

The fourth lab was initially designed to showcase the performance difference between normal and accelerated models. However, the lab computers did not have the necessary MEX compilers to use the accelerator mode, so this part had to be excluded. The angle modulation part of the lab was completed without much difficulty. The extra credit demodulation part was not attempted by any of the students despite being worth a large amount of bonus points; the most likely reason for this is that the students who might have done this part didn't need the extra points, and the ones who did need them didn't care about their grades enough to do it.

Lab 5 - Line Coding and Decoding

Despite being the first lab where almost no explicit instruction was given, students performed reasonably well on the fifth lab. This was taken as evidence that the students were indeed learning the intricacies of Simulink well. A couple students even found an easier approach to the problem than the instructor's solution using special

blocks. Hints were eventually given to stragglers, but these hints were more along the lines of a “push” to get them going in the right direction rather than an outright spoiler to the solution.

Lab 6 - Zero-Forcing Equalization

Because only the best 5 out of 6 labs counted for points, many students who had already gotten sufficient scores on the previous labs did not attempt the sixth. Roughly half of the class participated in this lab, but a problem was revealed as the lab session started: the pre-built models made use of new blocks that were not available on the old version of Simulink loaded on the computers. To fix this, students were forced to wait 5 minutes while the instructor quickly rebuilt the model using older blocks and re-uploaded it. To avoid problems like this in the future, all blocks used in example models should be version-checked before being posted for download by the students. Once the correct model was uploaded, students had an easy time performing this lab.

The educational impact of these labs was best seen in the performance of the students on the line coding lab. In that lab, they were required to research four line coding schemes such as bipolar NRZ and differential Manchester, then implement each one as a masked subsystem in Simulink using any method of their choice. This served as both a learning experience and an evaluation of their accumulated Simulink skills. The results of this evaluation are as follows: around half of the students completed the full exercise with only a slight amount of guidance (mostly on the transition-based line codes, where a memory element was required), whereas the other half needed a bit more help to get the

lab done. Since student abilities vary, an outcome like this is satisfactory. On average, the students learned most of the basics of Simulink quite well and the interest on learning the theory was also increased. With a few simple improvements and the possible addition of one or two more lab exercises to the six presented in this paper, they could be used to teach a full undergraduate laboratory course on Simulink.

As this was the first teaching experience for the first author, he learned a great deal about teaching and working with students. The experience gained through this teaching may also be helpful to other instructors using this set of lab exercises.

1) It is important to test the lab exercises on the computers that students use; this is true for any laboratory-based course. Compatibility of software versions, missing software or hardware components, access rights to drivers, etc. would cause many problems, so these components must be tested prior to the lab sessions.

2) It is important to clearly state deadlines and consequences of late submission. In our testing of the lab exercises, a lack of hard deadlines and late-submission consequences was assumed by many students. Despite repeated reminders, a lot of students forgot to submit the model files they used in each lab. The eventual solution was to grade late submissions much more harshly; it is fine if a student needs more time to complete a report, but the quality of the submission must reflect this extra time spent.

3) It is important to clearly specify expectations in a grading rubric. Since the syllabus did not clearly specify a grading scheme for writing quality in particular, the average quality of the lab reports was below acceptable standards. As a trade-off, the decision was made to remove a great deal of points from unacceptable work, with an offer to restore them if corrections were made by the student. In this manner, the student is not penalized for initially expecting low standards, but they must still improve the quality of their work to acceptable levels if they want to obtain full credit.

The authors also realized the importance of the assessment scheme on the quality of student learning, and it was noticed that the students' training on technical writing was not emphasized enough in the practice of previous lab courses. Many problems existed in the reports submitted by the students, including deficiencies in basic structure and flow of English, lack of proper grammar and spelling, etc. This issue was also reported to the undergraduate curriculum committee of the department.

Conclusions

A set of six MATLAB Simulink laboratory exercises that was previously designed for an undergraduate communications course has been thoroughly tested by an undergraduate student in a pilot test. The initial lab designs were then greatly improved as a result of the knowledge gained during the pilot test. These newly modified labs have since been utilized in classroom teaching, and some additional minor modifications were made to the lab manuals based on the observed student feedback and performance. As a result, the current-form laboratory exercises are ready to be disseminated to the public.

Acknowledgements

This work was supported by Dr. Zheng's NSF Career award #ECCS-0846486.

References

[12] C. Aroskar and Y. R. Zheng, "Design of Simulink Projects for an Undergraduate Communications Course," *Annual Conference of American Society of Engineering Education (ASEE)*, pp. 1-9, June 26-29, 2011.

[13] K. Modi, E. Hong and B. Bhattacharya, "Interactive models for teaching digital signal processing," *DSP/SPE 2009*, pp. 238-243, January 4-7, 2009.

[14] J. Turner and J.P. Hoffbeck, "Putting Theory into Practice with Simulink," *Proceedings 2005 ASEE Annual Conference & Exposition*, June 2005.

[15] M. Rice, "*Teaching Digital Communication Theory with Simulink at Brigham Young University*," MATLAB Digest: Academic Edition, vol. 3, no. 2, April 2009.

[16] J. Eker and A. Cervin, "A MATLAB toolbox for real-time and control systems co-design," *Real-Time Computing Systems and Applications*, pp. 320-327, 1999.

[17] M. Karimi-Ghartemani, H. Mokhtari, M.R. Iravani and M. Sedighy, "A signal Processing system for extraction of harmonics and reactive current of single-phase systems," *IEEE Transactions on Power Delivery*, vol. 19, pp. 979-986, July 2004.

[18] R.M. Gagne, *The Conditions of Learning and Theory of Instruction*, Holt, Rinehart and Winston Inc., New York, 1985.

SECTION
2. CONCLUSIONS

2.1. CONCLUSIONS FOR 3D COMPRESSIVE SENSING IN MRI

The methods for improving image quality in the MR imaging of mouse hearts using 3D sparsification that were presented in Paper I are very promising, especially for the current research being conducted at the University of Missouri VA Hospital. Just as standard video compression schemes exploit the fact that pixels do not change much from frame to frame, the DFT method presented here for compressing unchanging pixels into a single DC coefficient causes great improvements over 2D methods which do not exploit time sparsity. Future research can improve upon this method by finding a transform other than the DFT which might increase sparsity in the time domain.

2.2. CONCLUSIONS FOR SIMULINK LABORATORY EXERCISES

The new Simulink exercises presented in Paper II are a great improvement over the originals tested in the previous year. Overall, the students that attempted these exercises learned the operation of Simulink quite well, with only minimal struggles. The main place for improvement is in the explicit declarations of expectations with regard to technical writing standards, as well as adjustment of the grading rubric to match. It is also advisable to ensure that the lab computers are equipped with all of the necessary compilers and updated software to make full use of the features described in the exercises.

APPENDIX A.

SOFTWARE FOR MRI COMPRESSIVE SENSING

All of the CS results for Paper I were obtained using a special MATLAB software program that I explicitly designed for those applications. I designed the following two versions of the software program to use for my research:

1) MRI CS Toolbox V1.0
2) 3D MRI CS Toolbox V0.1

These two software programs are nearly identical in user interface and internal functionality; the 3D toolbox was essentially a fork from the latest 2D toolbox. Both programs utilize YALL1 v1.4 to perform L1 minimization, and accept k-space data stored in MATLAB's native .MAT format. The user interfaces for these programs are shown in Figures A.1 and A.2. In truth, the only differences between the two are that the 3D program loads in full sequences of images instead of single images, and also provides options to change the 3D sparsifying basis.

Both of these programs were developed during the process of my research, and are released under the GNU GPL v3. The help file for the 2D version of the software is included in this appendix after the figures in order to convey the extent of its functionality.

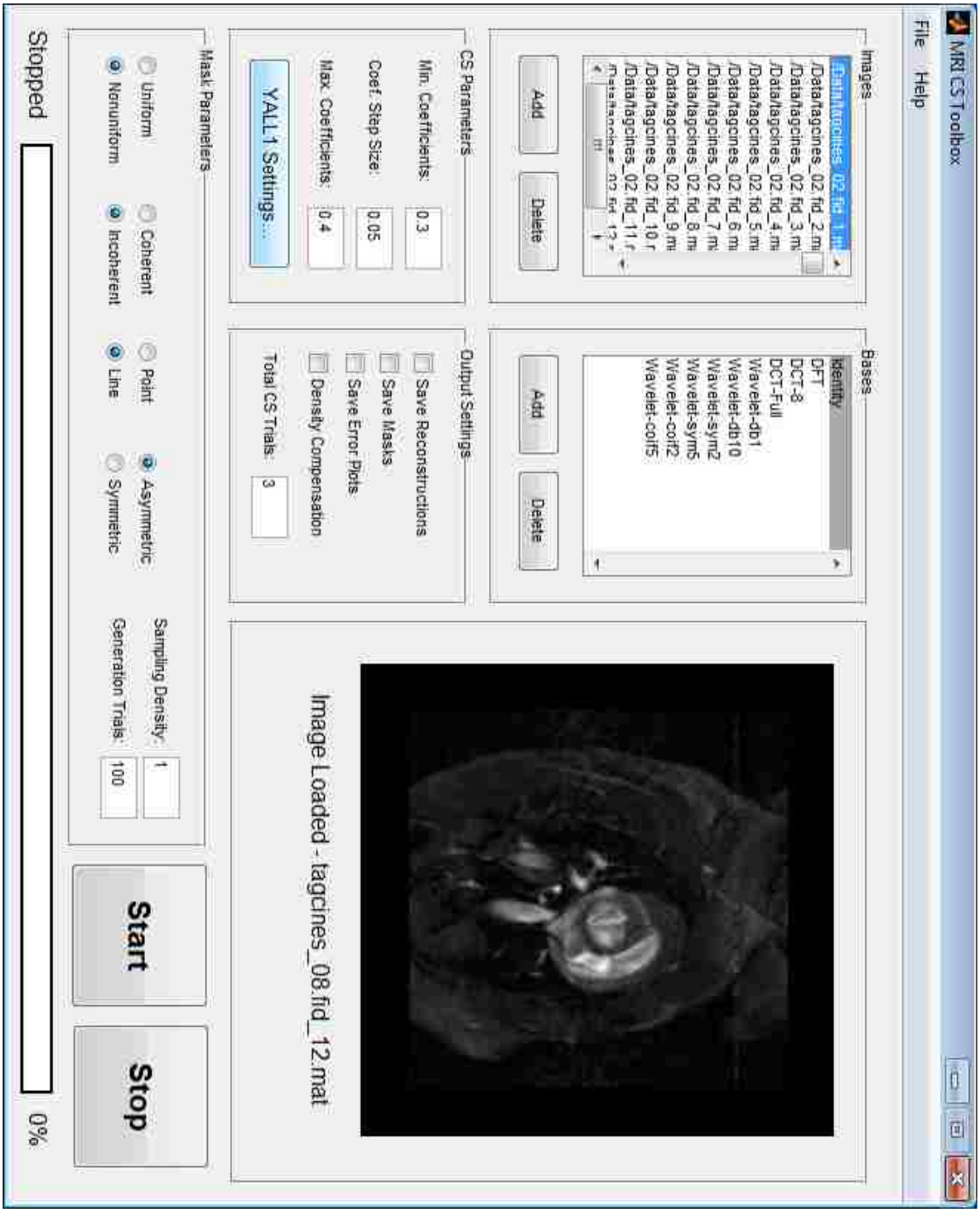


Figure A.1 – Example of 2D CS toolbox user interface

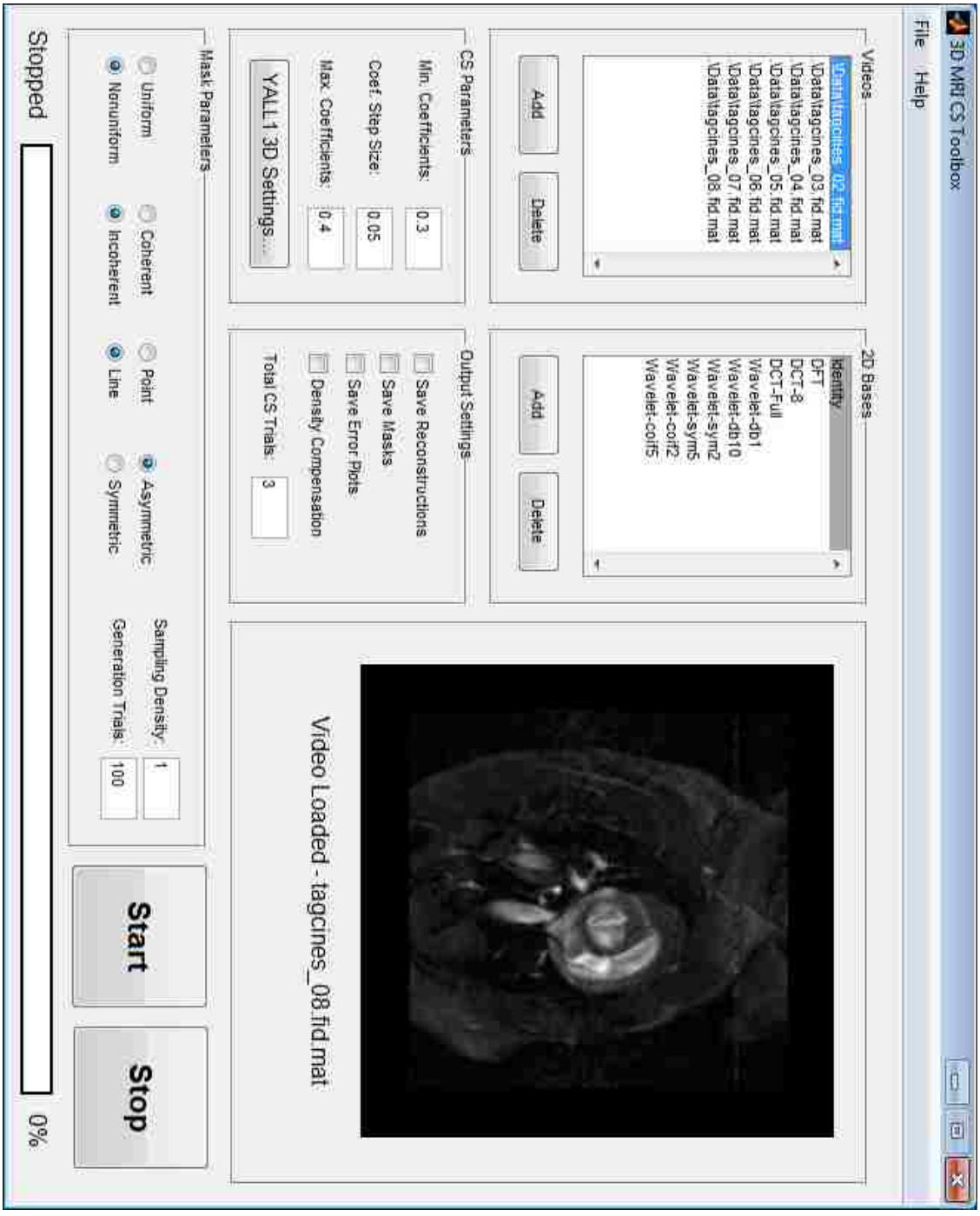


Figure A.2 – Example of 3D CS toolbox user interface

MRI CS Toolbox Overview

The MRI CS Toolbox is a convenient program for experimenting with CS reconstruction by undersampling full sets of k-space data. It has support for many bases and many sampling methods, and is designed to run unattended with batch processing in order to analyze a large amount of data and sampling techniques at once.

Opening an Image

The "Add" and "Delete" buttons on the image list will allow you to import one or more k-space images into the program. Any images imported into the program should be in the .MAT format, with the variable "d" containing a 2D complex double array that consists of a full set of k-space data points. If a single image is imported (as opposed to multiple at once), the full image will be displayed in the output window.

Adding Reconstruction Bases

The "Add" and "Delete" buttons on the basis list will allow you to select which sparsifying transform bases to use in your experiment. The following bases are supported:

ZP	Zero-padding (all missing k-space points are filled with zeros, no CS is performed)
Identity	Identity basis (the image domain itself)
DFT	Fourier basis (the k-space domain itself)
DCT	The block DCT bases (i.e. DCT-8 operates on 8x8 pixel blocks, DCT-Full treats the whole image as a block)
Wavelet	Various wavelet transforms (see MATLAB "wfilters" documentation for more info)
FD	Finite difference transforms in X or Y direction (number specifies order)

Each wavelet decomposition is performed to the maximum level allowed by MATLAB. To add all of the bases at once, choose "Select All" at the bottom of the list.

Adjusting CS Parameters

The CS parameters section allows you to adjust which undersampling percentages to try. For example, if the minimum value is 0.3, the step size is 0.05, and the maximum value is 0.5, the program will attempt sequential reconstructions using 30%, 35%, 40%, 45%, and 50% of the k-space coefficients. This is a good way to find the best trade-off between undersampling rate and image quality.

The L1 minimization program used by this toolbox is YALL1. There are four major minimization models that can be used in this toolbox; these models along with

their relevant parameters are available for adjustment by clicking the "YALL1 Settings" button. See the YALL1 reference manual for more details on what these settings mean.

Program Output Settings

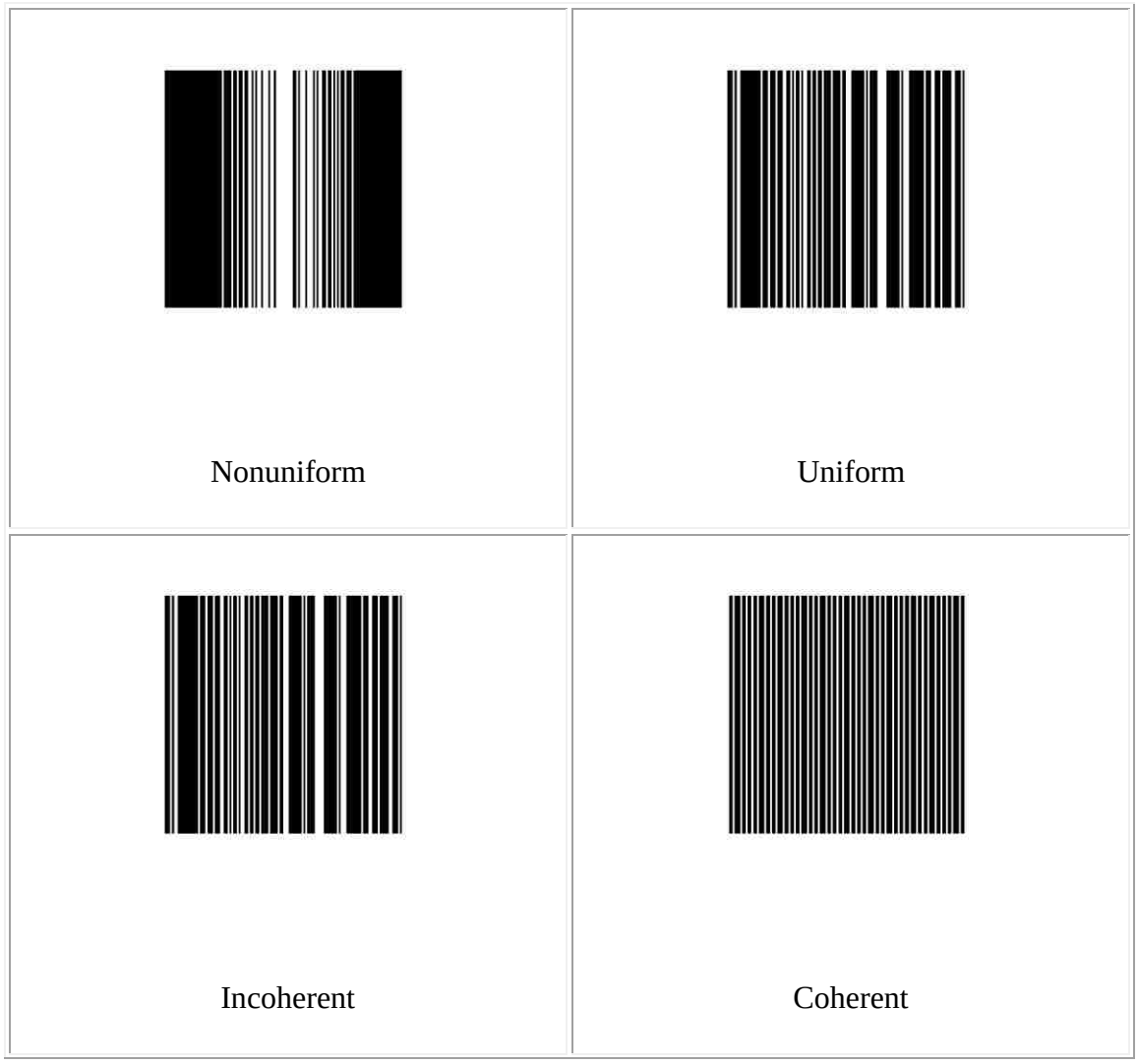
By default, the program will send the fully sampled reconstructions as well as a .MAT file containing all of the final output values to the "Output" directory located in the main program folder. However, it is also possible to save each sampling mask, reconstructed image, and PSNR error graph using the checkboxes in the output section. Be careful when using these settings, as a massive number of images can be generated in some cases.

The Density Compensation option adjusts the reconstruction such that the less likely samples will carry more weight. For example, if a k-space line only has a 30% chance of being picked, and it gets picked, the pixel values will be divided by 0.3 before reconstruction. This can help reconstruction in some cases, but is not usually recommended.

The total trials box indicates the number of times to run the simulation on each image. This is important because of the random variance in the reconstruction patterns. If more than 1 trial is performed, the error metric will be calculated based on the average error over all the trials. Also, the total simulation time will depend on the product of all the test parameters. For example, if there are 3 images, 4 bases, 5 sampling percentages, and 5 trials, the simulation will be performing 3*4*5*5=300 reconstructions.

Choosing Sampling Patterns

There are several different ways to construct a sampling pattern for the reconstruction. Which points are picked and which ones are not is dependent on the different sampling types. The examples below illustrate the different available choices of sampling pattern. Black pixels represent k-space points that are not used, and white pixels represent those that are. Each example below uses 30% undersampling.

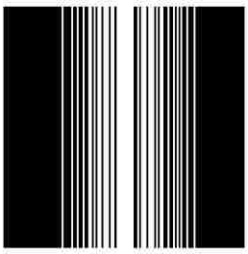


Nonuniform

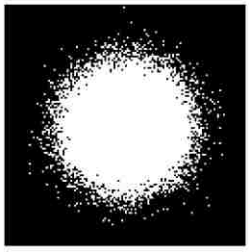
Uniform

Incoherent

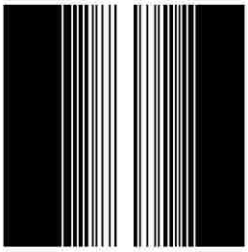
Coherent



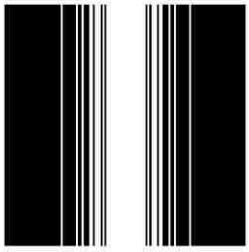
Line-based



Point-based



Asymmetric



Symmetric

For most cases, nonuniform, incoherent, asymmetric line-based sampling is recommended for MRI. The relative sampling density allows you to adjust how dense the random distribution is around the center of k-space in the case of nonuniform sampling. A value of 1 means that the edges of the image correspond to 4 standard deviations of the Gaussian distribution. If it is changed to 1.5, the edges become 6 standard deviations and the distribution gets narrow. Setting this value too low will just result in near-uniform

sampling, while setting it too high will result in a non-random block of the center coefficients being selected.

The mask trials setting allows you to adjust how many different masks will be generated before the best one is picked. The "best sampling mask" is considered to be the one with the minimum peak interference in the point spread function.

Running the Simulation

When you hit "Start", the simulation commences and runs all the way through until it is finished. Hitting "Stop" will abort the simulation after it finishes the current reconstruction. Upon completion, the program will print the results for the best basis (based on the mean PSNR value) in the command window. It will also save the individual PSNR values as well as some sparsity measures in a .MAT file in the Output directory.

APPENDIX B.

SIMULINK LAB MANUALS

The following pages are near-verbatim copies of the lab manuals used in the teaching of Simulink described in Paper II. They have been reformatted to fit the thesis margin requirements for printing, and the figures have been renumbered so that they can be listed in the illustrations section.

Lab 1: Intro to Simulink and Frequency Analysis

Objectives:

1) To learn how to build a basic model in Simulink.
2) To simulate signals and view their frequency domain representation.
3) To synthesize signals from their Fourier series coefficients.

Introduction:

A signal is ordinarily described as a function of time. This is what we visualize when we view the waveform on an oscilloscope. However, in communication systems it is also important that we know the frequency content of a signal. The spectral density of a signal characterizes the distribution of the signal's energy or power in the frequency domain. This concept is important when considering filtering in communication systems.

The mathematical tool which relates the frequency domain description of the signal to its time domain description is called the Fourier Transform. The Fourier Transform of a signal specifies the amplitudes and phases of the frequency content of the signal given its behavior in the time domain. The inverse Fourier transform can then recover the original time domain signal given its frequency domain description.

Simulink is a tool that allows us to visually create signal processing systems by connecting individual "blocks" that describe the actions of the system. After specifying the blocks that make the system do what we want, we can then pass any signal we want through it and view the output on a virtual "scope" in either the time or the frequency domain.

Preliminary:

The complex Fourier series coefficients are given by:

$$c_n = \frac{1}{T}\int_0^T f(t)e^{-j2\pi n\frac{t}{T}}\,dt$$

1) Find the Fourier series coefficients of a square wave of odd symmetry of amplitude *A* and frequency *f* Hz (assume 50% duty cycle and zero DC).
2) Find the Fourier series coefficients of a sawtooth wave of odd symmetry of amplitude *A* and frequency *f* Hz (assume zero DC).
3) Sketch the amplitude and phase of the Fourier series coefficients obtained in 2 & 3.

Procedure:

Part A – Getting Familiar with Simulink

To open Simulink, type '*simulink*' at the MATLAB command prompt. The Simulink library browser will open. It consists of various blocksets specific to different applications such as communications, signal processing, control systems, power systems etc. It is advisable to explore the various block libraries on your own to get a good feel for the capabilities of Simulink.

In order to familiarize yourself with Simulink, you will first build a simple system by following the steps given in this document. To open a new model, go to the *File* menu and select *New -> Model*. A blank model will open. This will serve as your canvas to build your system model.

In the Simulink library browser, go to the *Sources* option and select a *'Sine Wave'* source block. In order to include this block in your model, click and drag the block onto your model window. Similarly search for a *'Gain'* block (to amplify the signal) and a *'Scope'* block (to visualize the signal) in the various Simulink libraries. You should find the gain block under the *Commonly Used Blocks* or the *Math Operations* libraries. You should find the scope block in the *Commonly Used Blocks* or *Sinks* libraries. Include both in your model window.

For this model, we need two gain blocks. In order to duplicate a block, simply right click on the block and drag and place this duplicate block in your model window too. We would also like to have a *'Switch'* block to be able to select from multiple inputs and a multiplexer block to view multiple signals on the same scope. The multiplexer is found in the libraries as the *'Mux'* block. Search for these blocks in the basic libraries and include them in your model.

We also require a *'Clock'* block to use as a decision maker for switching between two signals using the switch. This block simply outputs the current simulation time. Look for it in the *Sources* library and include it in your model window.

(*Note:* This is not the clock used for synchronization of digital logic systems.)

Your model window should now have all of the blocks shown in Figure B.1.1.

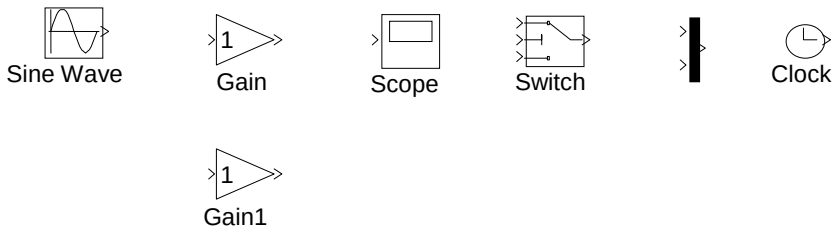


Figure B.1.1 - Model window after creating blocks

In order to begin connecting the blocks, click on the sine wave block, then hold down the CTRL key and click on the first gain block. A connecting line will form between the two blocks. An alternate way to do this is to place your cursor near the output port of a block until the cursor changes into a cross-hair, then click and drag a line to the input port of the block to be connected. The same sine wave input is to be connected to the second gain block too. In order to draw a second branch, right click on the original branch and drag a line to the input port of the second gain block. Connect the chosen blocks in the following manner.

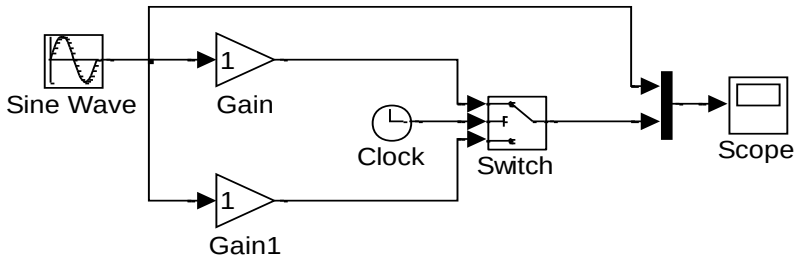


Figure B.1.2 - Model window after connecting blocks

Next, we need to set the parameters for the individual blocks. Double clicking on the blocks opens the block parameters pane. First, double click on the sine wave source

block. Set the block parameters as given in Figure B.1.3, and click OK to apply the settings.

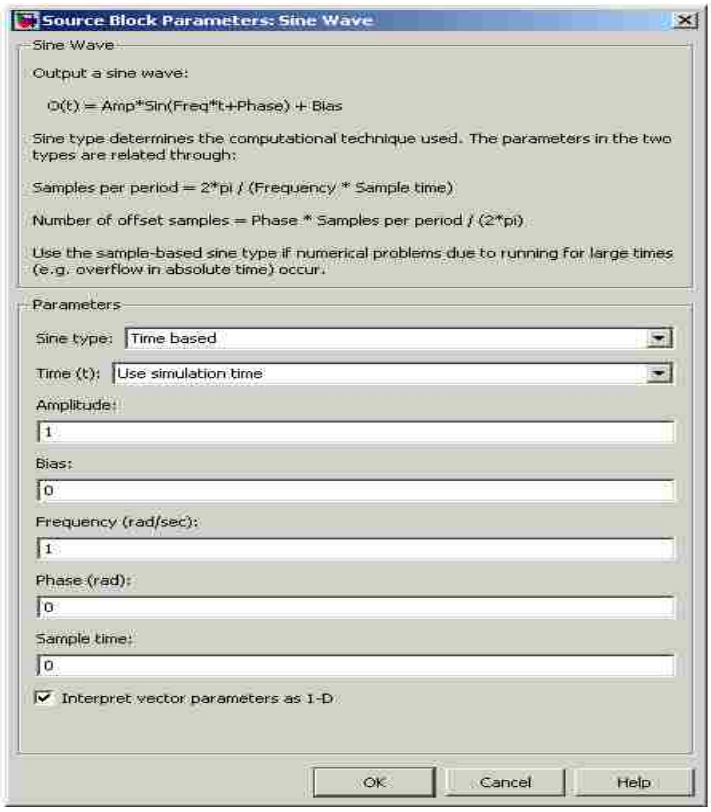


Figure B.1.3 - Sine wave source parameters

Open the two gain block settings and set them up for gains of 0.5 and 3 respectively. The switch should have the parameters shown in Figure B.1.4 in order to change from the second gain block output to the first after a simulation time of 5 seconds. The mux block will have two inputs by default. The clock and scope block settings need not be altered.

To add a title to the model, double-click at the location you want to type, then type “Building a Basic Model in Simulink”. You may use the *Format* menu to change the

font and font size. You may also right-click on the title box and select *'Show Drop Shadow'* to emphasize the title. The completed model is seen in Figure B.1.5. Include a screenshot of the model in your lab report.

Set the simulation parameters by selecting *Configuration Parameters* from the *Simulation* menu. Under the Solver tab, set the solver to *'discrete'*, the stop time to 10.0, and the max step size to 0.03. Then click OK. Now the model is ready for simulation.

Go to Simulation > Start to begin simulation. In order to visualize the results, double click on the scope block. The scope output should be as shown in Figure B.1.6. Include a screenshot of this result in your lab report.

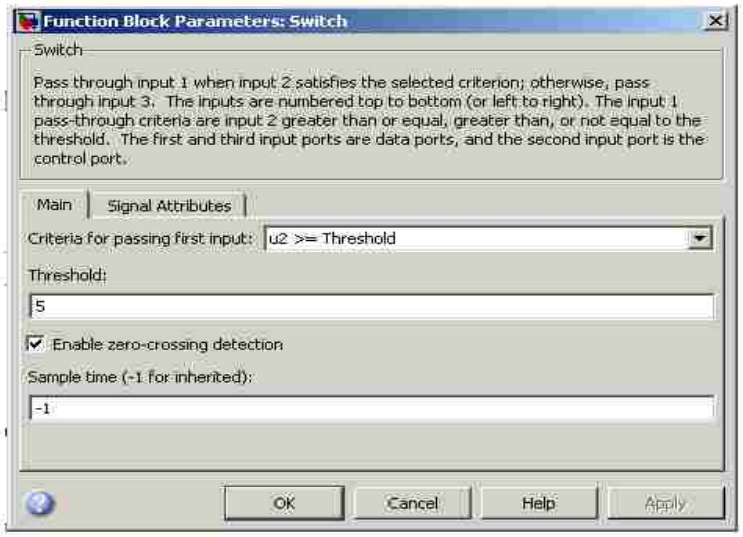


Figure B.1.4 - Switch block parameters

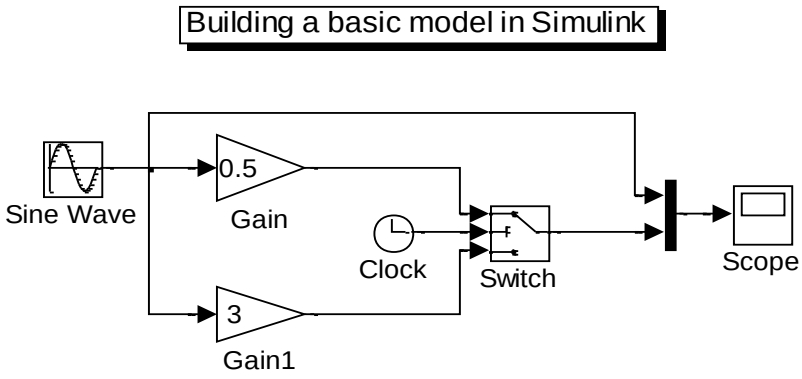


Figure B.1.5 - Final model

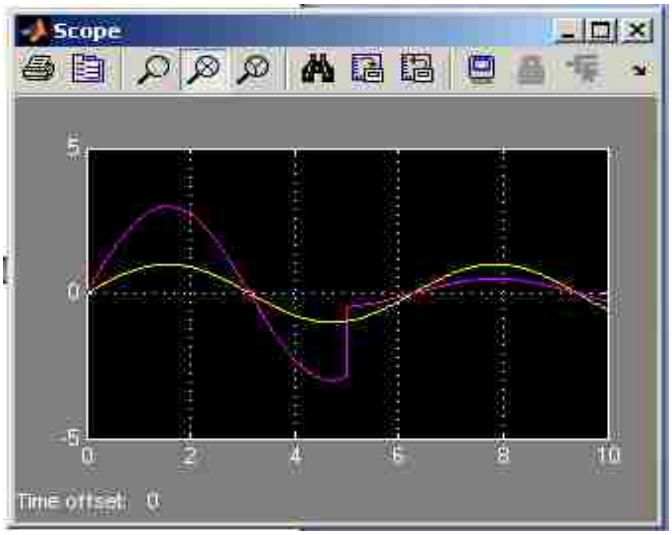


Figure B.1.6 - Scope display

Change the frequency of the sine wave generator to 5*2*pi rad/sec (i.e. 5 Hz) and change the phase to pi/2 rad to turn it into a cosine wave. Simulate the model and observe the scope, then include a screenshot of the new scope output in your report. Similarly, observe the scope for the following variations of the original model and submit the outputs in your report:

1) Change the switch block to a *Product* block (remove the clock).
2) Change the clock to a cosine of frequency 4 rad/sec, amplitude 10, and bias 5.
3) Change two or more parameters of your choice in the original model.
4) Replace one of the gain blocks with any block of your choice.

Part B - Frequency Domain Analysis

In order to view the properties of a signal in the frequency domain, some different steps must be taken. Ordinarily, a signal source will generate a single value at each point in time, which then propagates through the entire system as Simulink processes the simulation. However, a proper analysis in the frequency domain requires several time-domain points to be stored up and processed at the same time, generally with an FFT block.

A handy block used to compute and display the frequency components of a signal is the *Spectrum Scope*, found in the Signal Processing Toolbox libraries. To start using it, create a new model and add a title of "Frequency Domain Analysis of a Signal". Then find the spectrum scope block in the library and add it to your new model.

Upon opening the block settings, a number of options will need to be set. To display a raw magnitude spectrum (as opposed to the PSD), change the units to "dBW", and set the display options to one-sided. In order to use the standard signal sources from the normal toolbox, it is necessary to check the *Buffer Input* box. When this is selected, the scope will take time-domain samples from a source and accumulate them in a buffer until it has enough to calculate the required FFT. Set the buffer size to 512, buffer overlap

to 0, and spectral averages to 1. Do not change any other settings from their default values.

A convenient signal source from the continuous-domain toolbox is the *Signal Generator* block. This block can generate a number of different periodic signals given only an amplitude and a frequency. Find this block and add it to your project, then open the settings. Leave the signal type at ‘sine’ and amplitude at 1, but change the frequency to 5 Hz.

It turns out that these two blocks cannot be directly connected together due to the way Simulink processes signals. The signal generator produces *continuous-time signals*, which are numerically computed using differential equations. However, the FFT scope requires a *discrete-time signal* to function, which is handled in an entirely different way by Simulink’s model solvers. In order to convert the continuous signal into a discrete one, we require the *Zero-Order Hold* block. Find this block in the library, then place it between the signal generator and the spectrum scope and connect them together in a line. Your final model should resemble Figure B.1.7.

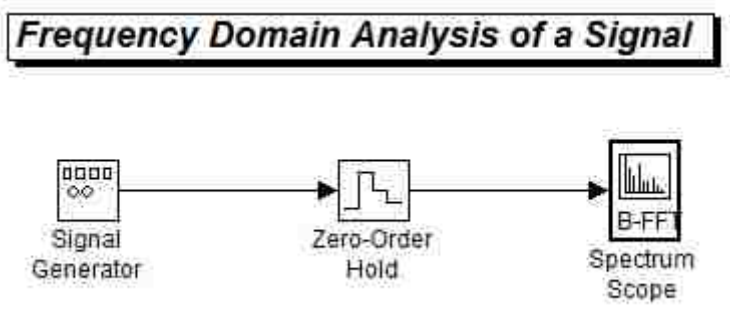


Figure B.1.7 – Final model

There are still a few more things to do. By opening the zero-order hold settings, we can choose the sampling period for the discrete signal. Set it to 0.01 (f_s = 100 Hz). Finally, open the simulation configuration settings and use the same settings as the previous example, but make the max step size 0.001. We are now ready to run the simulation.

Upon finishing the simulation, the spectrum scope should pop up with the results of the FFT. For the sine wave, there should be a single peak at 5 Hz and nothing else. It may also be a good idea to connect a normal scope to the zero-order hold output to view the signal in the time-domain.

Now we are ready to look at a few more types of signals. Change the signal type in the generator to *'square'* and rerun the simulation without changing any other settings. There should be harmonics present this time; if they are not visible, go into the scope settings and lower the minimum Y-value. Compare the magnitudes of these harmonics to the results from the preliminary and note your observations in your lab report. Also, include a screenshot of your magnitude spectrum.

Next, change the generated signal to a sawtooth wave and repeat the above process, noting your observations on the harmonics and taking a screenshot of the spectrum. Finally, change the generator to output a random signal and record your observations on the spectrum produced.

Part C - Signal Synthesis from Fourier Series Coefficients

Given the Fourier coefficients calculated in the preliminary exercise, it is possible to reconstruct the square wave and sawtooth wave signals using only sine wave sources.

To do this, open a new model, then create a sine wave source for the fundamental frequency of the square wave and one for each harmonic (use as many harmonics as you feel are necessary to reconstruct the signal accurately). Set the amplitudes, frequencies, and phases of each according to the Fourier coefficients you calculated previously.

To add all these sine waves together, the *Sum* block is required. Find this block in the libraries and add it to your project. Open the settings, and under the list of signs, add as many plus (or minus) signs as there are signals to add. This will expand the sum block to accept several inputs. Attach a scope to the input and observe how close the sum of sinusoids comes to representing the shape of the original signal, then include a screenshot of the scope output in your lab report. Repeat the above steps to recreate a sawtooth wave and report your observations.

Post-Lab Questions:

Q1. In the demo model built in this lab, we used a mux block to visualize two signals on a single scope. Is it possible to see two signals on the scope without using a mux block? If yes, explain how.

Q2. Does changing the max step size in the configuration parameters have any effect on the way the model runs?

Q3. In part A, does changing the simulation time have any effect on the model output?

Q4. In part B, what is the relation of the sample time to the signal frequency to get a reasonably good representation of the frequency domain components?

Q5. In part B, what effect does changing the FFT buffer size have on the output spectrum?

Q6. In part C, change the phase and magnitude of each harmonic. Which is more important: the phase of a harmonic or the magnitude?

Lab 2: Hierarchical Design and Linear Systems

Objectives:

1) To learn how to build and mask subsystems in Simulink.
2) To simulate linear systems and observe their properties.

Introduction:

In circuit modeling, a network of elements such as resistors, capacitors, and inductors compose a system with the properties of linearity and time-invariance. This lab is concerned with the property of linearity, which states that a weighted sum of any number of inputs, when sent through the system, will produce an output exactly the same as if each individual input was sent through the system one-at-a-time and the outputs added together with the same weights. In mathematical terms, if input $x_1(t)$ results in output $y_1(t)$, and input $x_2(t)$ results in output $y_2(t)$, the output due to $x(t) = a_1x_1(t)+ a_2x_2(t)$ (where a_1 and a_2 are scalar constants) will be:

$$y(t) = a_1y_1(t)+ a_2y_2(t)$$

A key feature of Simulink is that it allows us to make hierarchical designs. This means that we can compose a complicated system that consists of many individual "subsystems", The main reason for doing this is that it encapsulates the underlying details of each subsystem so that only the top level of the design is visible to the user. In doing

this, the model window becomes less cluttered, and the end-user is protected against accidentally making unwanted changes to the system inside.

Preliminary:

1) Consider the input $x_1(t) = 2[u(t) - u(t-3)]$

   Find $y_1(t)$ for $h_1(t) = 3\,e^{-2t}u(t)$

2) Consider the input $x_2(t) = 4\,u(t)$

   Find $y_2(t)$ for $h_2(t) = 5\,e^{-3t}u(t) - 2\,e^{-5t}u(t)$

(Hint: Use Laplace transforms to convert x(t) and h(t) to the s-domain, then multiply.)

Procedure:

Part A – Designing a Hierarchical System

To begin, download the file "*lab2.mdl"* from Blackboard and open it. The model shown in Figure B.2.1 will open:

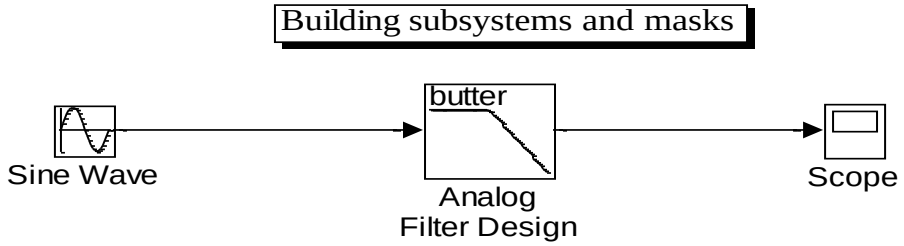


Figure B.2.1 - Initial model

Simulate the given model and observe the output on the scope. It is important to note that, unlike the previous lab, this lab uses *continuous-time* solvers. In this mode, Simulink solves the system by solving a set of differential equations symbolically, as opposed to approximating the system using discrete methods. This allows us to simulate analog systems to a high degree of accuracy. Sample times do not exist in this mode, and any block that requires discrete-time operation will not work properly.

(If you try to connect continuous blocks directly to discrete ones, Simulink will automatically change the solver and the whole system will become discrete.)

To start, we want to add some more signal sources to the model. Open the Simulink libraries and search for a *Chirp Signal*, a *Step* source, and a *Random Number* source, then add all of these to your project. Next, find and insert a *Multiport Switch* block and a *Constant* block into your model. The subsystem will consist of these various sources as inputs, with the switch and constant for selecting between them. Connect all inputs to the multiport switch and the constant to the selector input of the switch as shown in Figure B.2.2. In order to make a subsystem, select all of these blocks (all blocks except the filter and the scope), then right click and choose *Create Subsystem.* All selected blocks will be replaced with a single block containing all the functionality of the original blocks.

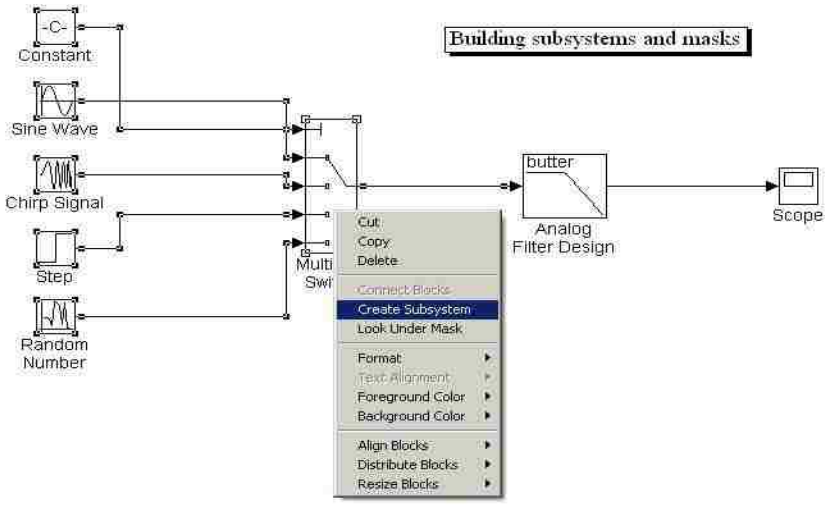


Figure B.2.2 - Select the create subsystem option

Rename the resulting subsystem as *"Input"* by clicking the text box beneath the block. You can reveal and change the original components of this new block by double-clicking it. However, it is much more elegant to create a user interface to control the various settings than to open the block and change the parameters inside every time. For example, one way to change the selected source is to go into the block and change the constant C manually, but this can be time-consuming for frequently-used blocks, and it also brings the hazard of accidentally changing something you didn't want to change. The solution to this problem is called *masking* the subsystem, which creates a handy user interface much like the ones you see when you open the pre-made blocks to change the parameters.

In order to create a mask for the subsystem, right-click on the subsystem block and select *Mask Subsystem*. A window will open that will allow you to specify all of the details necessary to fully define a custom user interface for your new subsystem.

1) The first tab in this window defines the behavior of the block icons and the input-output ports. Under *Icon Drawing Commands*, we can represent the block graphically using common MATLAB functions such as *disp*. Use a simple command such as:

```
disp('Input Signal');
```

to draw the name of the subsystem on the body of the block.

2) The next tab, *Parameters*, is where we define the parameters of the system that can be changed dynamically by the user. Obviously, we require an option for the user to select which of the input sources to use, but we also require many other parameters related to the source selected. For example, the sine wave requires an amplitude, frequency, and phase to be defined, whereas the random source requires a mean and variance. We shall make the mask dynamic such that, for each source selected, only parameters related to that source will appear to the user for modification. To accomplish this, first we edit the window to look as given in Figure B.2.3.

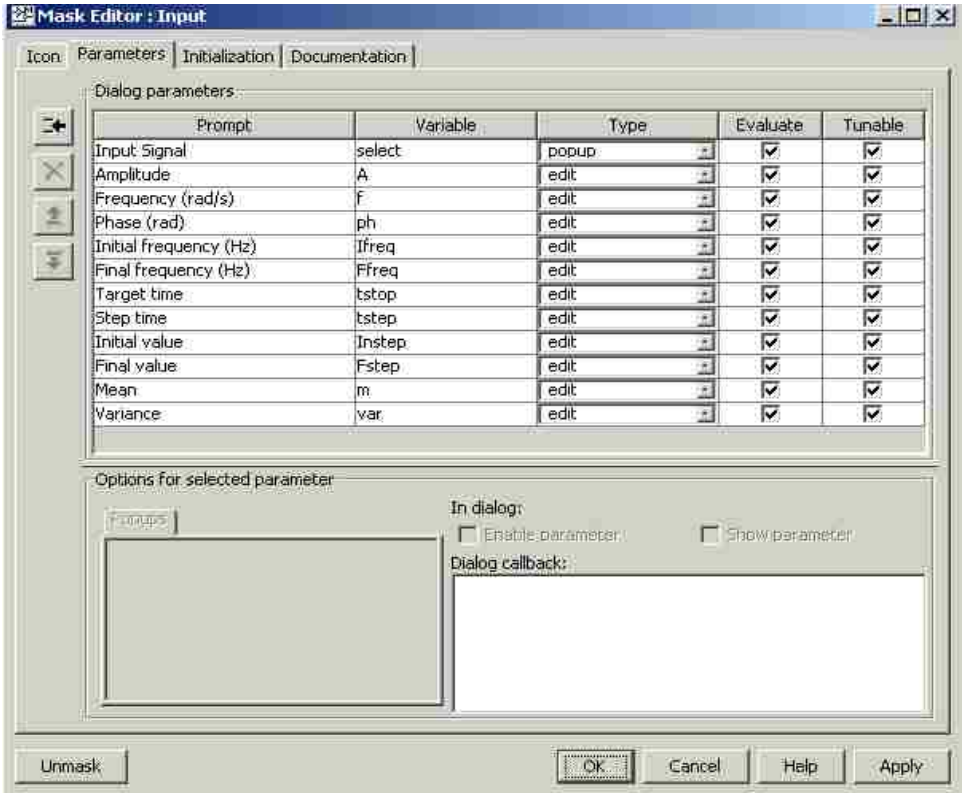


Figure B.2.3 - Parameter pane in the mask window.

All parameters except for the input signal are considered “edit boxes”, which mean that the user simply enters whatever value they want for these variables when they open the settings panel. For the choice of input source, however, we want a popup menu that lists the various sources. To enter the possible choices, type the following lines into the *Popups* tab under the Input Source parameter:

Sine Wave

Chirp Signal

Step Input

Random Number

3) Besides the input signal, each of these parameters relates to only one source. We do not want parameters to appear that are unrelated to the source that is selected. For example, if the chirp signal is selected, the user should not be able to see and modify the mean or variance variables. In order to control which variables are visible for each source, we require a bit of MATLAB code that will execute whenever the input source is changed. Highlight the Input Signal parameter, then under its *Dialog callback* box, enter the following code:

```
c=get_param(gcb, 'MaskValues'); % Get mask parameter values

% Check the value selected in the pop-up menu and change visible
parameters
if strcmp(c{1},'Sine Wave')
      set_param(gcb,'MaskVisibilities',{'on';'on';'on';'on';'off'
      ;'off';'off';'off';'off';'off';'off';'off'})
end
if strcmp(c{1},'Chirp Signal')
      set_param(gcb,'MaskVisibilities',{'on';'off';'off';'off';'o
      n';'on';'on';'off';'off';'off';'off';'off'})
end
if strcmp(c{1},'Step Input')
      set_param(gcb,'MaskVisibilities',{'on';'off';'off';'off';'o
      ff';'off';'off';'on';'on';'on';'off';'off'})
end
if strcmp(c{1},'Random Number')
```

```
        set_param(gcb,'MaskVisibilities',{'on';'off';'off';'off';'o
        ff';'off';'off';'off';'off';'off';'on';'on'})
    end
```

(Note: The names in the code must match the names in the popups tab EXACTLY)

4) The third tab, *Initialization*, is where we can define the initial values for any the parameters in the mask. However, the catch is that this code executes every time you run the simulation, which means that any values you chose for the parameters in the dialog box prior to running the model will be overwritten by this code. For that reason, leave this blank for now.

5) The last tab, *Documentation*, contains areas to enter helpful information about your mask for any user who might use it in the future. Since this is a tutorial lab, you may leave this blank for now. Click OK to save the changes to the mask.

With the mask completely defined, our last step is to make it so that the variables actually correspond to parameters in the constituent blocks. Right-click on the subsystem and select *Look Under Mask* to see the original blocks making it up. Inside each source, we want to make the value of each parameter dependent on the variables we defined. To do this, simply replace the value in each parameter with the corresponding variable. For example, in case of the sine wave you would have replaced the amplitude value with ‘A’,

the frequency with ‘f’ and the phase with ‘ph’. The value of the constant that switches between sources will become ‘select’.

To test your subsystem, double click the block and look at the parameters that are visible to change. Depending on which source you select, the available parameters should change. Experiment with different sources and different values for the parameters, then take a screenshot of the scope output for each source (pick any parameters you want) and include them in your report. Comment on how the low-pass filter affects each type of source.

Part B – Simulating Linear Systems

Next, we are going to simulate the linear systems given in the preliminary. Start a new, blank model and find the *Signal Builder* block. When you add the block and open the settings panel, a screen will open which will allow you to create a time-domain signal from points. Use this tool to create the input signal $x_1(t)$ from the preliminary. Next, find the *Transfer Fcn* block in the libraries and place it. This block is used to filter a system given its transfer function in the s-domain. During the preliminary, you should have found the Laplace transforms of h_1 and h_2. Enter the expression for $H_1(s)$ in this block, and connect it to the input source. Finally, attach a scope to the output. Include a printout of the scope output in your report, and observe how close the output comes to your calculated value of $y_1(t)$ in the preliminary exercise.

To demonstrate the properties of linearity, we are now going to modify this model. Replace the signal builder source with the two separate step sources that comprise it. Duplicate the transfer function block, then connect each step function to a separate

instance of $H_1(s)$. Subtract the two outputs using a *Sum* block, then connect the final result to the scope. If the system is linear, the final result should be exactly the same as before. Include a printout of the scope output in your report, then observe whether or not it is identical to the output in the previous step.

Finally, use the knowledge from the previous steps to construct $x_2(t)$ and run it through the system $H_2(s)$. Include the final output in your lab report, and observe how closely it matches the theoretical output for $y_2(t)$ you calculated in the preliminary.

For extra credit, you may make a masked subsystem containing x_1, x_2, h_1, and h_2, with the entire block connected to the scope. In the settings for the block, allow the user to choose between either input, and between either system. The block should output the result of the chosen input filtered by the chosen system. If you choose to do this, include printouts of the final model, the contents of the subsystem, and the mask parameters window in your lab report.

Post-Lab Questions:

Q1. What happens if the given code for the popup dialog is not entered at all?

Q2. What problems will occur if the signal names in the popup dialog do not match those defined in the callback code?

Q3. What happens if the order of the parameters is changed to be different from Figure 3? (You may test this by using the “move up” and “move down” options).

Q4. Define a possible function for h(t) that would NOT be linear.

Lab 3: Simulink Libraries and Amplitude Modulation

Objectives:

1) To learn to build libraries in Simulink.
2) To simulate Double Sideband Amplitude Modulation (DSB-AM).
3) To simulate demodulation of the signals that were modulated as above.

Introduction:

A useful feature in Simulink is the ability to create user-made libraries. For example, if you implement a certain modulation scheme using various blocks and wish to use this scheme in several different models down the road, you can create a library with all of the blocks you need, then open it later and pull out the blocks you need just as you do with the built-in Simulink libraries.

Modulation, by definition, is a process by which a certain characteristic of a fixed carrier wave is varied in accordance with an information-bearing signal. The primary motivation for modulation is to facilitate transmission of the information-bearing signal over a communication channel with a specific frequency range. For example, a human voice may have frequencies between 100 and 3000 Hz, but if this signal were to be transmitted via RF directly at those frequencies, only one voice could be transmitted at a time without interference (and it would require an antenna the length of the United States). By modulating and transmitting the signal at a much higher frequency, we can communicate over a much wider variety of channels, and also share these channels with other signals.

Amplitude modulation refers to the manipulation of the amplitude of a carrier wave in some manner to transmit the required information. In the simplest case, the information signal is multiplied by the carrier wave, creating a new signal that has frequencies above and below the carrier wave, but none at the original frequencies of the message signal. This is very useful for a wide array of applications, since we can effectively choose which frequency range our message gets transmitted over.

Preliminary: None

Procedure:

Part A – Designing a Simulink Library

To begin, open a new library by choosing *New -> Library* from the *File* menu. This top-level window is the area where our final user-made blocks will reside, ready to be taken and used in another program. The basic principle is that, just as we did in Lab 2, we will build a system out of the default blocks and then enclose it all in a subsystem to create a user-made block. The main difference is that, in a library, it is only a collection of these blocks. You cannot simulate anything within a library file; you must take the blocks out and place them into a model file if you wish to simulate anything.

For this lab, we will be constructing a block that performs DSB-AM modulation on a given signal. The defining equation for AM modulation in this manner is:

$$x(t) = A(1 + \alpha m(t))cos(2\pi f_c t)$$

where A is the amplitude of the carrier, f_c is the frequency of the carrier, α is the modulation index, and m(t) is the *normalized* message signal (i.e. it is modified to occupy the range between -1 and +1). This subsystem will consist of one input, one output, and three user-changeable parameters. In order to create this input and output, search for the blocks *In1* and *Out1* in the Library browser and add them to your library. You may rename these components m(t) and x(t), respectively, for clarity.

The first part to build in this model is the normalizer, which takes the input message signal m(t) and constrains it to fall between -1 and 1. Ordinarily, normalization would entail dividing every sample of the signal by the absolute maximum value the signal would take. However, this is somewhat tricky to implement in Simulink, and it runs into trouble where noisy signals are concerned. For the purposes of this lab, we will instead *clip* the signal so that, if it falls outside the range of -1 to +1, it will be forced to one of those values. This can be accomplished with the *Saturation* block in the library. Add this block to your model and set its limits accordingly.

After clipping, you must implement the above equation using blocks from Simulink. You should have the necessary knowledge to build a system that computes this equation. When defining the parameters for the blocks, there are three of them that we may wish the user to change: the amplitude *A*, the carrier frequency *fc*, and the modulation index *alpha*. Make sure to define these parameters as variable names rather than constants. When you are finished, create a subsystem out of these blocks and define a mask that will allow the user to change the three parameters that were defined by variables. Your finished block should appear like Figure B.3.1.

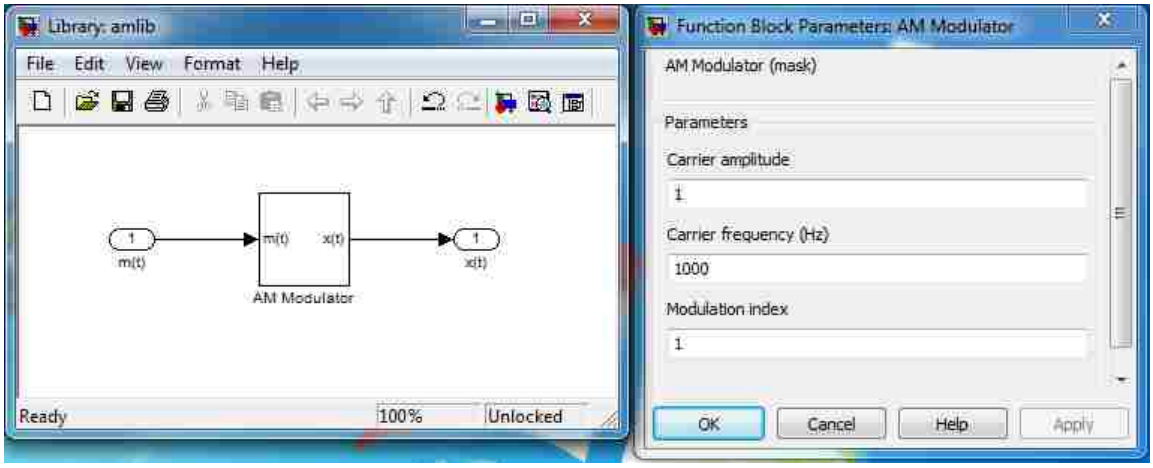


Figure B.3.1 – Finished AM modulator block

Part B – Using a Simulink Library

After creating the above block, save the library as "am_lib.mdl". Now, open a new model file, set the solver to discrete time, and save it as "am_sim.mdl". This model file will be where we test the modulator built in the previous step. Drag the modulator block you just designed into the new model. In this way, user-made libraries are much like the ones that are built into Simulink; you just drag them into model files as needed.

To test this model, create a sine wave block and connect it to the input of the modulator. Set its amplitude to 1 and its frequency to 1 Hz. Now, in the modulator parameters, set the carrier amplitude to 1 and the modulation index to 0.5. Set the carrier's frequency to around 20 times the frequency you set for the source sine wave. Connect a scope to the output, then set it to accept two inputs so that you can view the original source sine wave on it at the same time. Set the max step size to 0.001 and the simulation run time to 2 seconds, then run it. After autoscaling the axes, you should get an output that looks like Figure B.3.2.

Save a screenshot of this scope output, then change the modulation index to 0.8 and run it again. Also save screenshots of the scope outputs for modulation indices of 1 and 1.5. Comment on the effects the modulation index has on the modulated signal. Finally, set the modulation index back to 1, but set your sine wave source amplitude to 2. You should notice the clipping effect on the final output.

For extra credit, you may connect an FFT block to the output of the modulator and observe the resulting spectrum. There should be a large impulse at the carrier frequency, along with two smaller ones on either side which correspond to the message signal.

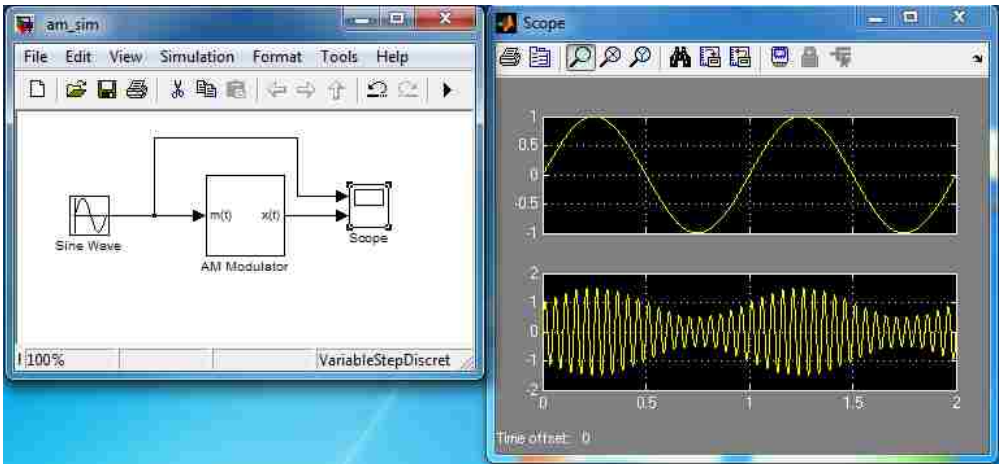


Figure B.3.2 – Output of AM modulator with α=0.5

Part C – AM Demodulation

Return to the library file you created in part A. We are now going to create a demodulator block and keep it in the same library file. It is important to note that, if you closed the library and then reopened it, it is now *locked*, meaning it won't let you make

changes to it. In order to change it again, you must choose *Edit –> Unlock Library* from the top menu.

An AM demodulator can be constructed by multiplying the output of the modulator by another cosine wave equal to the frequency of the original carrier, then low-pass filtering the result. This is known as a *product detector*. To begin, place the blocks seen in Figure B.3.3 in your library window, then configure the parameters as shown and create a subsystem out of them.

After doing this, create a mask for the subsystem. There will be three user-modifiable parameters: the carrier frequency *fc*, the filter sampling frequency *Fs*, and the filter order *n*. Set these parameters as shown in Figure B.3.4, then save the mask. You now have a fully-functional AM demodulator block. Save the library, and then reopen your simulation model file.

(Note: the filter coefficients are designed at runtime. Make sure this line of code is entered properly, or the filter will not work.)

```
firpm(n,[0 (1/2*fc)/(Fs/2) fc/(Fs/2) 1],[1 1 0 0])
```

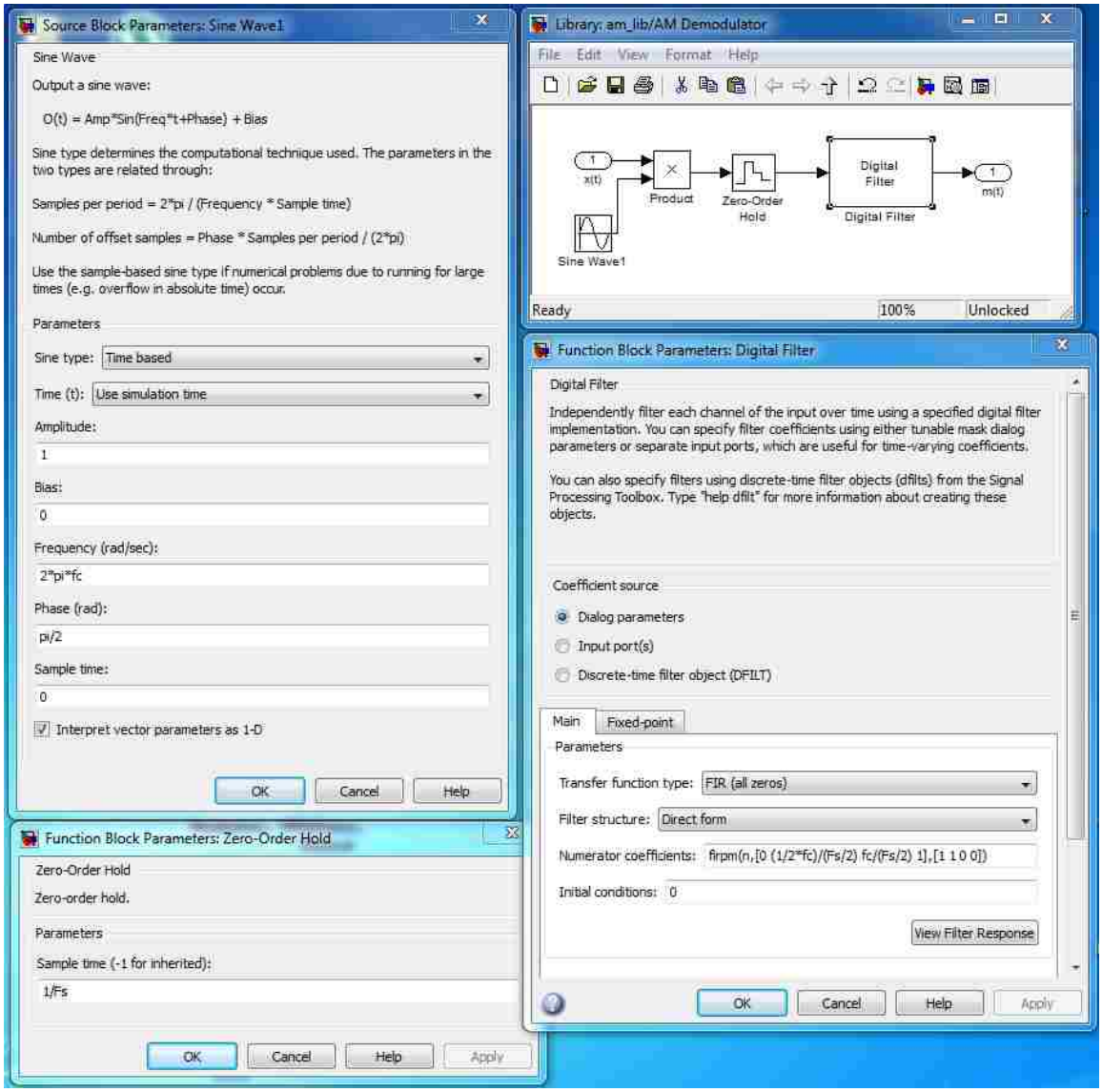


Figure B.3.3 – AM demodulator model with block parameters

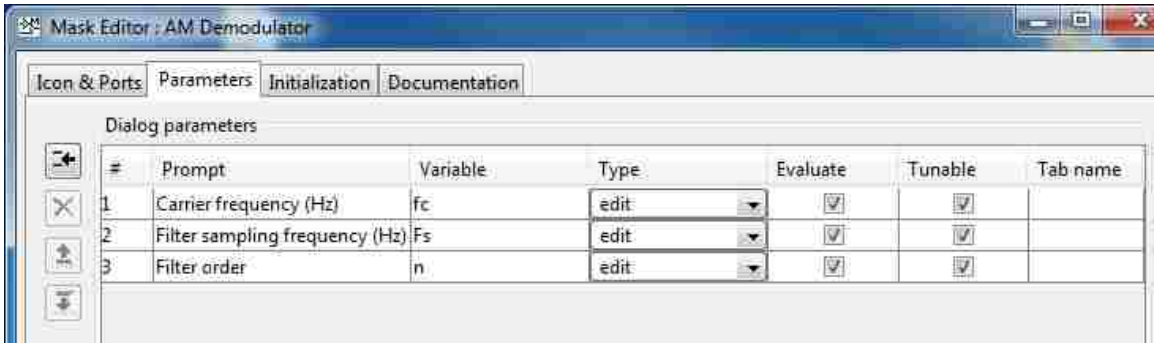


Figure B.3.4 – Mask parameters for AM demodulator

Essentially, this block multiplies the modulated signal by a cosine wave at the carrier frequency, then prepares to filter the result by sampling at regular intervals specified by Fs by using a zero-order hold block. A digital low-pass filter is created using the MATLAB function “firpm”, which takes a given set of specifications (in our case, the cutoff frequencies) and attempts to create an optimal FIR filter of the given order. Just as Simulink can evaluate block parameters that include variables (such as 1/Fs), it can also evaluate functions whenever the model is simulated for the first time. In our case, it was not possible to use the analog Butterworth design block because our model is being executed in discrete time.

After designing the block, drag it into your main model file and connect it to the output of your modulator. Add another input to the scope, then connect it and run the simulation. Using a sampling frequency of 500 Hz and a filter order of 100, you should get results that look similar to Figure B.3.5.

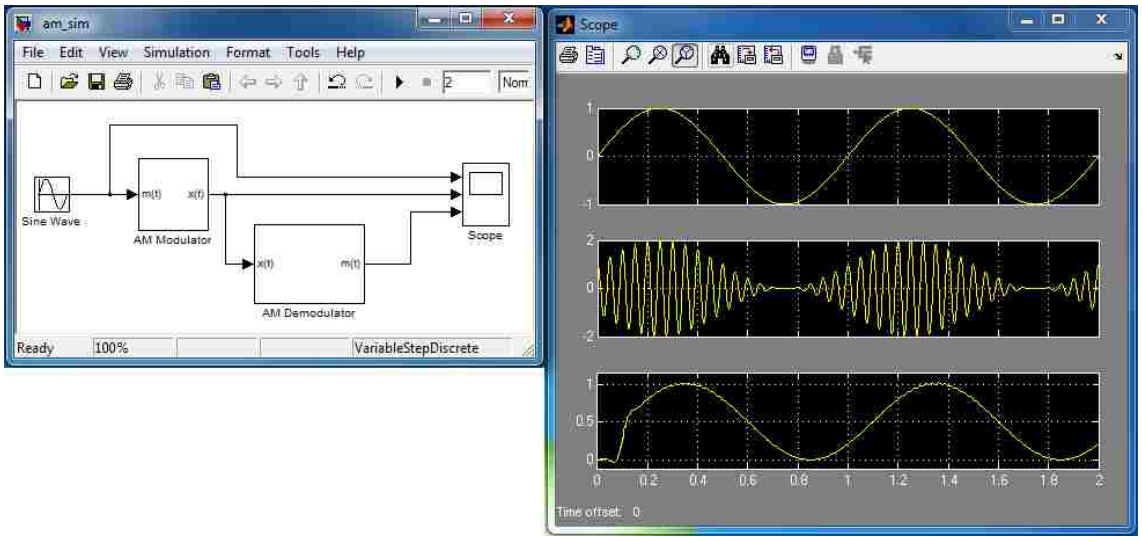


Figure B.3.5 – Final results for AM modulation

The output from the demodulator is going to be distorted at the beginning and slightly delayed when compared to the original input. The reason for this is that an FIR filter consists of a series of shift registers (in our case, 100 of them) that all begin at 0. It takes some time for these registers to be filled, so the filter output will not be correct until a certain amount of time has passed. The higher the filter order, the longer this will take.

After completing the above steps, save your model files and scope outputs and submit them in your report. For extra credit, use the blocks *From Wave File* and *To Wave File* to run a sound file (use "nineoneone.wav" from Blackboard) through your system and record the output. The final output should sound similar to the original input.

Post-Lab Questions:

Q1. In part B, change the carrier signal to a square wave. How does the modulated output change?

Q2. In part B, what are the consequences of setting the modulation index above 1? Will this signal be demodulated properly, or will there be problems?

Q3. What will happen if the modulation index is set too *low*? How will this affect the transmission efficiency in a real system?

Q4. In this lab, we defined the carrier frequency to be around 20 times the message frequency. What will happen if the carrier frequency is not that much higher than the message? Will this cause problems in demodulation?

Q5. Adjust the FIR filter order in part C to different values. What happens to the output if the filter order is set very low? What if it is set very high?

Lab 4: Model Referencing and Angle Modulation

Objectives:

1) To learn to use model referencing in Simulink.
2) To simulate Frequency Modulation (FM).
3) To simulate Phase Modulation (PM).

Introduction:

Along with subsystems and libraries, model referencing is another way to implement hierarchical design. Compared to libraries, the main advantage in using model references is that they can be *accelerated,* meaning that the referenced model itself can be pre-compiled in order to create a performance increase. In this lab, model referencing will be used in the implementation of FM and PM modulation.

Angle modulation is a process in which the angle of the carrier wave is varied according to the message waveform. In this modulation technique, amplitude of the carrier wave is maintained constant, which makes the transmission power constant. However, this benefit is achieved at the cost of increased transmission bandwidth.

The two primary methods of achieving angle modulation are phase modulation (PM), in which the instantaneous angle is varied linearly with the message signal, and frequency modulation (FM), in which the instantaneous frequency is varied linearly with the message signal. Both techniques will be explored in this lab.

Preliminary: None

Procedure:

Part A – Designing a Simulink Submodel

The model reference feature is provided by the *Model* block in the Ports & Subsystems library in the Simulink library browser. The instance of a Model block which represents another model is called a *referenced model* or *submodel*. The model that contains a referenced model is called its parent model. In order to demonstrate these features, we will create a model file that performs PM modulation, then use that model as a reference in another model.

To start, create a new blank model and save it as "**pm_mod.mdl**". Next, go into the configuration options and change the solver to discrete time. In order for this model to function as a reference within another model, there is one more configuration option that must be set. On the side bar of the configuration panel, go to *Optimization -> Signals and Parameters* and check the box labeled "*Inline parameters*". If this box is not checked, the model will generate errors when you try to run it as a reference within a parent model.

Next, we will create the model itself. In PM modulation, the phase of the carrier is varied according to the amplitude of the message signal. There are a number of different ways to do this in Simulink, but perhaps the easiest is to manipulate the time variable of the carrier wave directly. Create a sine wave generator in the new model, then open the parameters and set *Use External Signal* for the time variable. This will essentially create a block that will output **sin(ωt)**, where **t** is a signal that you feed into the block.

In order to make the output of this block vary according to the current simulation time, we need to create a *Digital Clock* block which will supply the base value for t. By

connecting one of these blocks directly into the sine wave generator, it will generate an ordinary sine wave which only varies with the simulation time, which works exactly like sine wave blocks that don't use an external signal for time. However, we also want this **t** variable to be influenced by the input signal, in order that we might change the current phase of the carrier. To do this, we add a *Sum* and *Gain* block, which has the net effect of adding or subtracting from the time that the sine wave generator sees, thus altering its phase. The complete system, along with all relevant parameters, is shown in Figure B.4.1. Note that the gain block, as configured below, will vary the phase of the carrier between $-\pi$ and π so long as the message signal goes between -1 and 1. If the message exceeds these limits, the modulation will make some message signal levels indistinguishable from others as the phase wraps around. Make sure that any input to this model is limited between -1 and 1 for this reason.

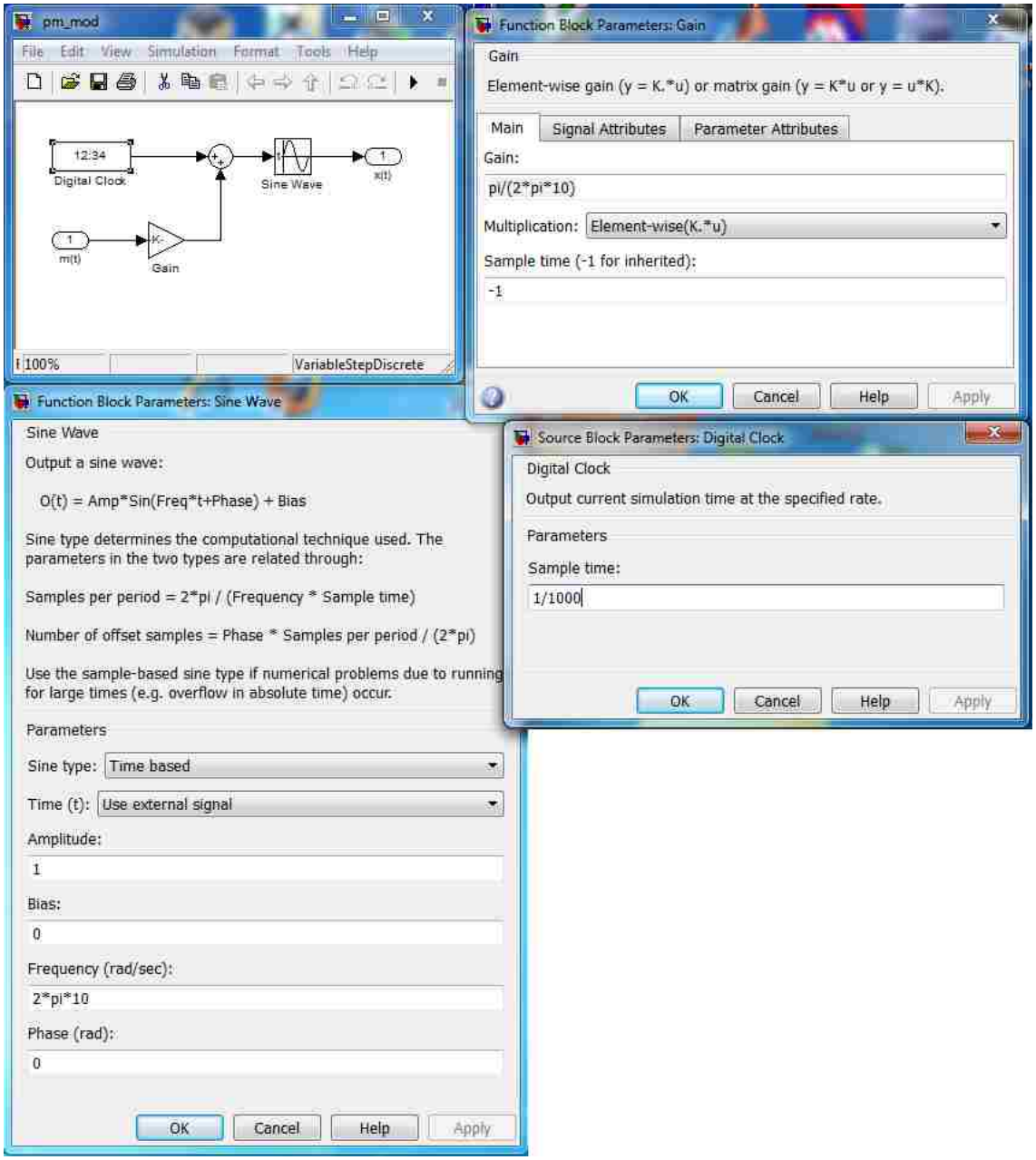


Figure B.4.1 – Finished PM modulator model

Using these parameters, the sampling frequency is 1000 Hz and the carrier frequency is 10 Hz. Make sure that any input signal you use in the parent model is much less than 10 Hz, or else the modulator will distort the output. Increasing the sampling

frequency and carrier frequency is also a solution, if you wish to modulate a signal with higher frequencies.

Part B – Designing a Simulink Parent Model

Now that we have a working modulator, it is time to put it to use. Create a new model file and save it as "**lab4.mdl**". Make sure that the same configuration parameters are set for this new model as were set for the previous one; the solvers must be consistent across all models when referencing is used. However, it is not necessary to set *Inline parameters* for the top-level model.

In order to use the modulator we just built in this model, we must first create an empty submodel block. Find the *Model* block in the Simulink library and place it. Initially, it starts out red and empty, because no model has yet been placed in it. Now, open the parameters window and browse for the model file you created the PM modulator in. Make sure that the simulation mode is set to *Normal.* The Accelerated mode is useful for speeding up models, but it requires a MATLAB compiler to create machine-level code to do this. Many machines do not have the appropriate compilers installed, so we will not experiment with this feature in this lab.

We need to create an FM modulator next. FM modulation is remarkably easy in Simulink, as the only block it requires is a VCO, or voltage-controlled oscillator. Open a new model file and add the *Discrete-Time VCO* block, along with one input and one output port. Set the quiescent (or center) frequency to the carrier frequency, and the input sensitivity to approximately half the carrier frequency. Perform the same steps as in Part

A to create a valid submodel, then save it as "**fm_mod.mdl".** Then add this model to your parent model as a reference as before.

Now that we have referenced these models, we will create a message signal, send it through the modulators, then look at them on the scope. When this is done, you should get a result that looks like that of Figure B.4.2. In this example, the carrier frequency for both modulators was 10 Hz, and the message signal was a sine wave of 1 Hz. Save a screenshot of the scope result, then comment on why the output waveforms look like they do. Next, change the message signal from a sine wave to a square wave and repeat this simulation.

For extra credit, you may implement a demodulator for the FM modulation scheme above. Do not use the built-in demodulator blocks from the Communications Toolbox; you must create your own model, block-by-block. Common FM demodulators include the quadrature detector and the PLL. A properly working demodulator may be worth up to 50% extra credit. Make sure to submit both the demodulator model files and the scope output for the demodulated signal.

(Note: In order for FM demodulation to be accurate, the frequency deviation should be orders of magnitude lower than the carrier frequency. Leaving the carrier at 10 Hz and the deviation at 5 Hz will make any attempt at demodulation very problematic.)

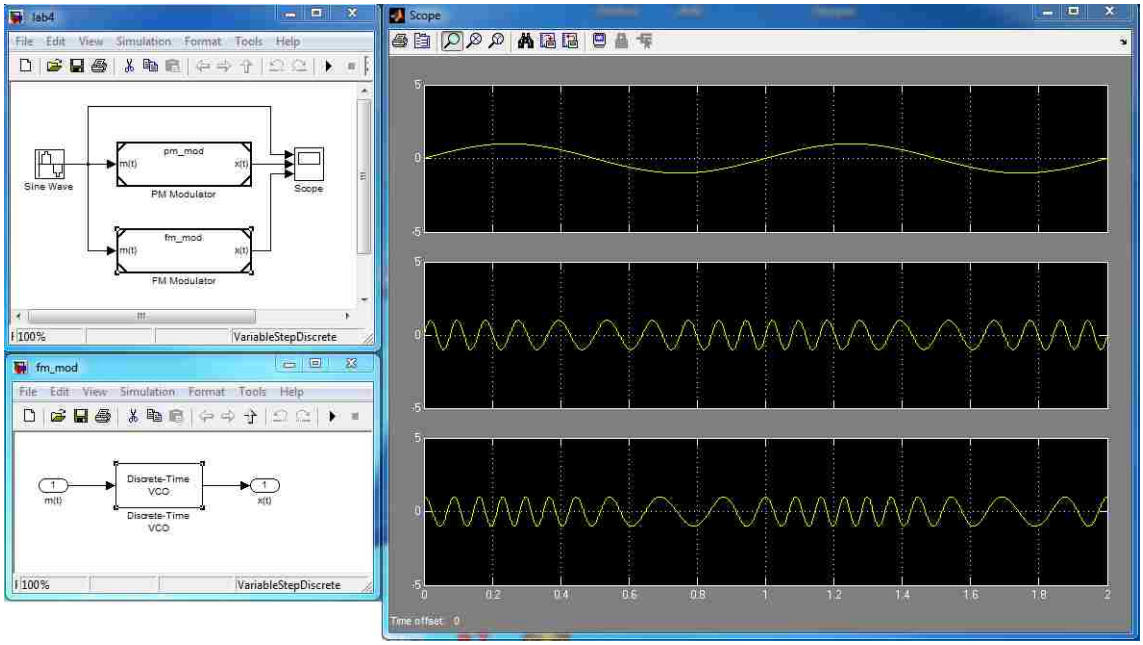


Figure B.4.2 – Finished parent model and scope results

Post-Lab Questions:

Q1. In what ways is a referenced model block different from a subsystem block?

Q2. In accelerated mode, the runtime speed of a referenced model is greatly increased, with the tradeoff of spending a large amount of time initially to compile it. In what situations would it NOT be a good idea to use this mode?

Q3. In Part B, look at the modulated signal for both PM and FM. If you know the input signal is a sine wave, and you are given the modulated output without any other information, is it possible to tell for certain whether the modulation was FM or PM? Why or why not?

Q4. Is it possible to tell the modulation type for certain if you know that the input signal is a square wave instead? Assume that the fundamental frequency of the square wave is many times lower than the carrier frequency.

Q5. In both types of angle modulation, the transmission power is kept constant no matter what the message signal is. Why is this advantageous over AM, especially for transmission of music?

Lab 5: Line Coding and Decoding

Objectives:

1) To implement 4 different types of line coding in Simulink and apply them to random data
2) To achieve more autonomy in creating Simulink models

Introduction:

In order to transmit data across any sort of channel, it is necessary to encode the information bits into some electrical waveform. The relation of this waveform to the input bits is called *line coding*, and the choice of this waveform can have a substantial impact on the performance of the system. For example, the simplest line coding is called *unipolar NRZ*, which creates a waveform that goes to 0V when the input is 0, and some positive voltage when the input is 1.

Simple forms of line coding such as NRZ are sufficient for some applications, but they can cause problems in others. For instance, suppose your data includes a very long run of 0's or 1's. With no transitions of the voltage level, the receiver may lose sync with the data when it attempts to sample it, and this will cause unnecessary errors. More advanced line codes can embed the clock signal within the data, ensuring that enough transitions occur to maintain sync. Differential line codes will encode the data in transitions rather than levels, which helps in some applications involving phase modulation.

Unlike the previous labs, this lab will expect you to figure out how to construct most of the models on your own. Most of the line codes can be implemented using basic blocks such as gains, sums, and constants, but a couple of them require delay blocks, which will be explained.

Preliminary:

Research the following types of line coding, and think about how they might be implemented in Simulink using the blocks you already know about:

Bipolar NRZ	*NRZI*
Bipolar RZ (without AMI)	*Differential Manchester*

Procedure:

Part A – Implementing Bipolar NRZ Coding

In order to begin constructing line coding models, we must have a source of data. An ideal source would be one that returns 0's and 1's with equal probability one sample at a time. The best choice of block to do this is the *Bernoulli Binary Generator*. Add this block to your model and set it up to spit out a new bit every second (sample time = 1). Make sure to remember to set the simulation to use the discrete solver. Also, add a scope to visualize the results.

Between the generator and the scope, we need to create a system that takes in a single bit, either 0 or 1, and outputs a waveform that goes between -1 and 1 for bipolar

NRZ. This can be done solely by using basic math blocks such as sums, gains, and constants. Once this system is finished, package it up into a submodel so that the top-level model looks clean. The data source and the scope will be shared by all line encoders, so do not include those in the submodel.

Part B – Implementing NRZI Coding

NRZI line coding is a bit trickier, as the current output will depend on what the output was *in the previous sample*. Basically, if the current data bit is 0, the output voltage level remains constant, but if it is 1, the voltage level transitions to its opposite state. Of course, in order to keep the voltage level the same (or change it), the model must know what value it had to begin with. The block that will accomplish this is called the *unit delay*. The output of this block is simply whatever the input to it was in the previous sample. This creates a sort of memory in the system, which is necessary because it must "remember" what the last voltage level was in order to be able to keep it the same or change it.

To use this block effectively, its input must be *the final output of the line encoder*. That way, when the simulation advances by a sample, the previous output level of the line encoder will have shifted to the output of the unit delay block. By using this value along with the value of the current bit, math functions can be used to create the appropriate waveform.

An example of this sort of "feedback loop" is shown in Figure B.5.1. Note that this is *not* the solution to the line coder; it is merely an example of how the blocks should be arranged. The example arrangement as it is shown is simply an integrator; the real

solution will involve a *subtraction* rather than an addition, as well as other math blocks. When you finish this system, package it into a submodel as well.

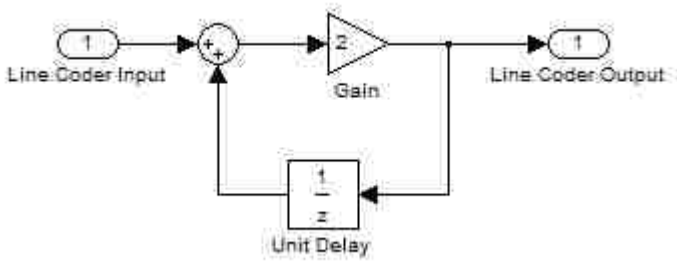


Figure B.5.1 – Example arrangement for use of unit delay block

Part C – Implementing Bipolar RZ Coding

For bipolar RZ line coding, a clock is required in addition to the data source. This clock will have the same period as the data rate, and its transitions are added to the line encoder output. To create this clock, add a *Pulse Generator* block. Set it to be sample based with sample time 0.5, period 2, and pulse width 1. This will make a clock that will alternate between 0 and 1 one time each for every data bit.

Using this clock as well as the data source, standard bipolar RZ line coding (*not* AMI) can be implemented using simple math blocks (no delays are required). You may implement the AMI version if you wish, but this will require a delay block to track the previous output polarity.

Part D – Implementing Differential Manchester Coding

The most difficult line code to implement is the Differential Manchester code. In ordinary Manchester code, for every data bit, the line encoder will use the clock to output

a high-to-low transition on an input of 1 and a low-to-high transition on an input of 0. When this is made differential, the type of transition stays the same on a 0 and switches on a 1. You may refer to Figure B.5.3 for a pictorial example of this.

When all encoders are completed and simulated, you should get a top-level model similar to Figure B.5.2 and scope results similar to Figure B.5.3. After confirming that the line codes are correct for different sequences of input bits, take a screenshot of your top-level model as well as the inside of every submodel and include them in your report. Include the scope output as well, and discuss in your report how these results compare to the theoretical results.

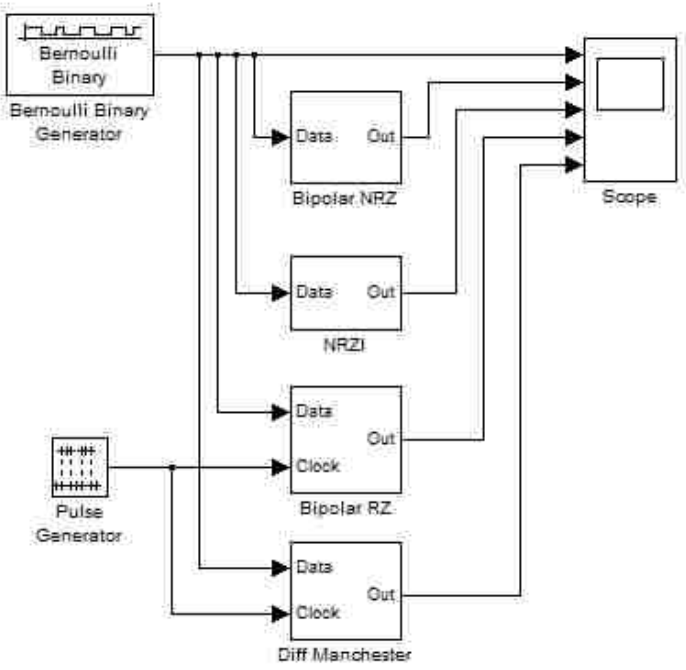


Figure B.5.2 – Example of final top-level model

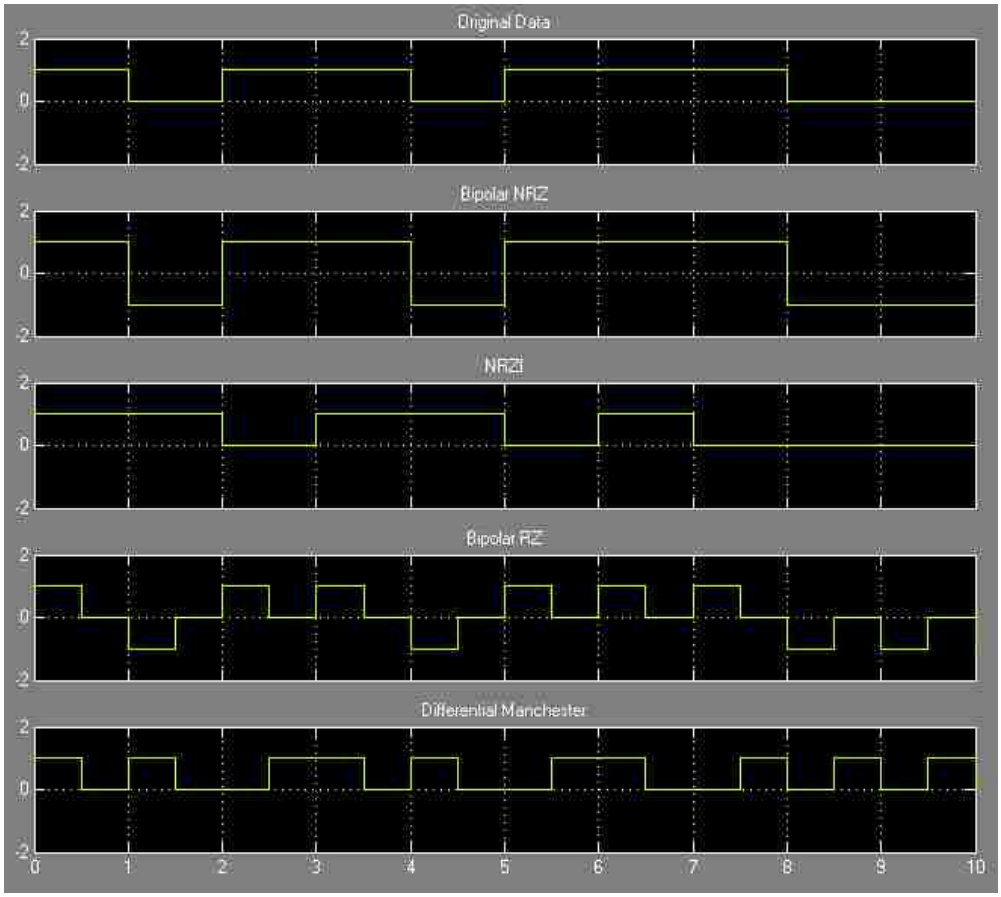


Figure B.5.3 – Example of scope results for all line codes

For extra credit (10 points each), you may design decoders for any of the line encoders created above. The output of any line decoder should give back the original bit sequence. Include scope results for these if you do them, and remember to submit all model files with the report.

Post-Lab Questions:

Q1. For the bipolar NRZ line code, what is an advantage of using bipolarity as opposed to unipolarity? Why could it be better to have a negative voltage level as opposed to 0 volts?

Q2. NRZI uses differential coding to encode the data in transitions. When the receiver is first turned on, having no previous information, can it immediately decipher the first bit received?

Q3. Bipolar RZ can be combined with AMI (alternate mark inversion), which leaves the voltage level at zero on a “0” bit and alternates between positive and negative voltages for successive “1” bits. What is a possible advantage or disadvantage of doing this?

Q4. Differential Manchester combines most of the advantages of the above line codes at the cost of high complexity and low bit rate. To what applications might this line code be best suited?

Lab 6: Zero-Forcing Equalization

Objectives:

1) To implement a simple zero-forcing equalization system in MATLAB
2) To observe the conditions under which zero-forcing equalization is effective

Introduction:

In most communications systems, the channel over which signals are sent is not ideal in any sense. Echoes, reflections, and time delays can cause a signal to be split into multiple copies which arrive at the receiver at different times. In many wired systems, these imperfections can be modeled by a static impulse response h(t). When this impulse response is convolved with an incoming signal, it basically has the effect of blurring adjacent symbols together, which is known as inter-symbol interference, or ISI. We wish to construct a matching filter on the receiving system that can undo the effects of ISI given the approximate channel response. These filters are known as *equalizers*, of which many types exist for different types of channels.

The zero-forcing equalizer is arguably the simplest of these. The basic idea of the ZF equalizer uses the fact that the channel impulse response h(t) can be transformed into a frequency response H(f). By constructing another filter on the receiver that has the inverse response $H^{-1}(f)$, we can undo the effects of the original channel, since $H(f)H^{-1}(f) = 1$. The main problem with this is that we are limited to a finite number of equalizer coefficients in a real system, and to get this inverse perfectly, we would need an infinite number of coefficients. Fortunately, we can still get an approximation to this inverse by

truncating the response to a finite number of terms. The other major problem, most prevalent in wireless communications, is that the ZF equalizer amplifies noise in any case where H(f) has a significant drop-off. If H(f) is close to zero at some frequency, the inverse will have a very large gain at that frequency, and any noise in this region will be amplified excessively. For this reason, it is mostly useful in systems that have a very high SNR.

Preliminary: Read the zero-forcing equalization section in your textbook.

Procedure:

Part A – Creating the Basic Model

First, download the library file for this lab from Blackboard. There should be four blocks inside:

1. Channel Response (generates the channel response h(t) and the equalizer response $h_{ZF}(t)$)
2. Channel (filters the input signal according to the coefficients set in #1)
3. ZF Equalizer (filters the received signal according to the equalizer coefficients set in #1)
4. AWGN (generates additive white Gaussian noise with a certain power in dB)

The first block is the heart of the operation: it takes in a given impulse response and equalizer length and uses the standard ZF matrix equation to generate the appropriate coefficients for the equalizer. It also provides two delays as outputs that will line up the input and channel output with the equalizer output on the scope. The second and third blocks are functionally identical; they are FIR filters that use the coefficients provided from the first blocks. The fourth block is a simple noise generator that is used to simulate the effects of random Gaussian noise on the equalizer's output.

For the first part of this experiment, construct the model shown in Figure B.6.1. Set the solver to discrete-time and the total simulation time to 100. For the random source block, use a Gaussian distribution with the mean at 0, variance at 1, and sample time at 1. When finished, run the model to ensure that everything is working properly.

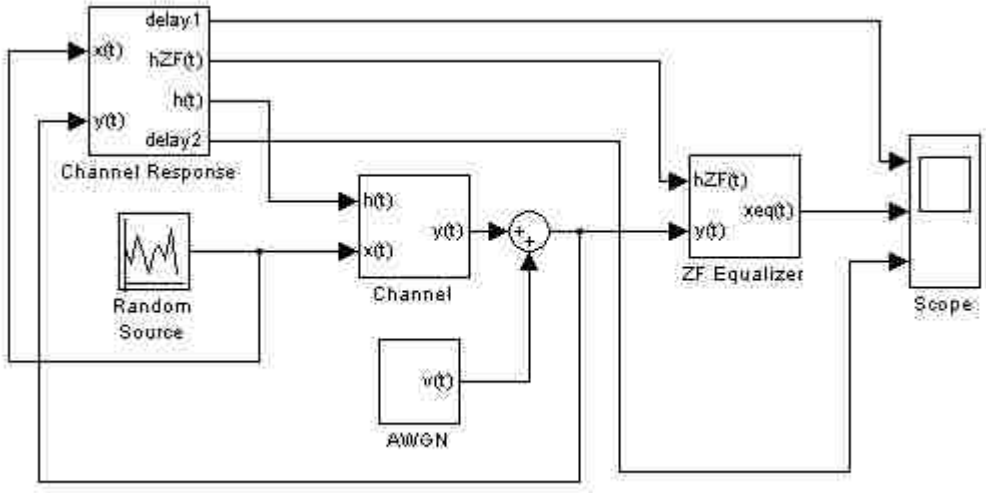


Figure B.6.1 – System setup for ZF equalizer simulation

Part B – Equalizer Simulation

For the first test of the channel equalizer, we would like to simulate a static channel with:

$$h(t) = [10\ 2\ 0\ 4\ 1]$$

This is a simple example of a channel with a decaying response and a single echo. When the input signal is run through this channel, the echoes of past symbols will interfere with the present ones. The first simulation will be noise-free, so temporarily disconnect the AWGN source. In order to separate these out, we would like to run it through a ZF equalizer with L = 31 taps. Simulate this for 100 symbols and compare the equalized signal to the original source signal. Also, look at the signal before equalization to see how the previous and current symbols interact. You should get a scope output that resembles Figure B.6.2. Save this scope output and discuss your observations in your report.

For the second test, we would like to reduce the length of the equalizer and see how that affects the final results. Set the equalizer length to L = 9 and run the simulation again, then save the scope output and note any differences in the equalized signal in your report. Repeat for L = 101 taps. Is there any noticeable difference between L = 31 and L = 101 taps?

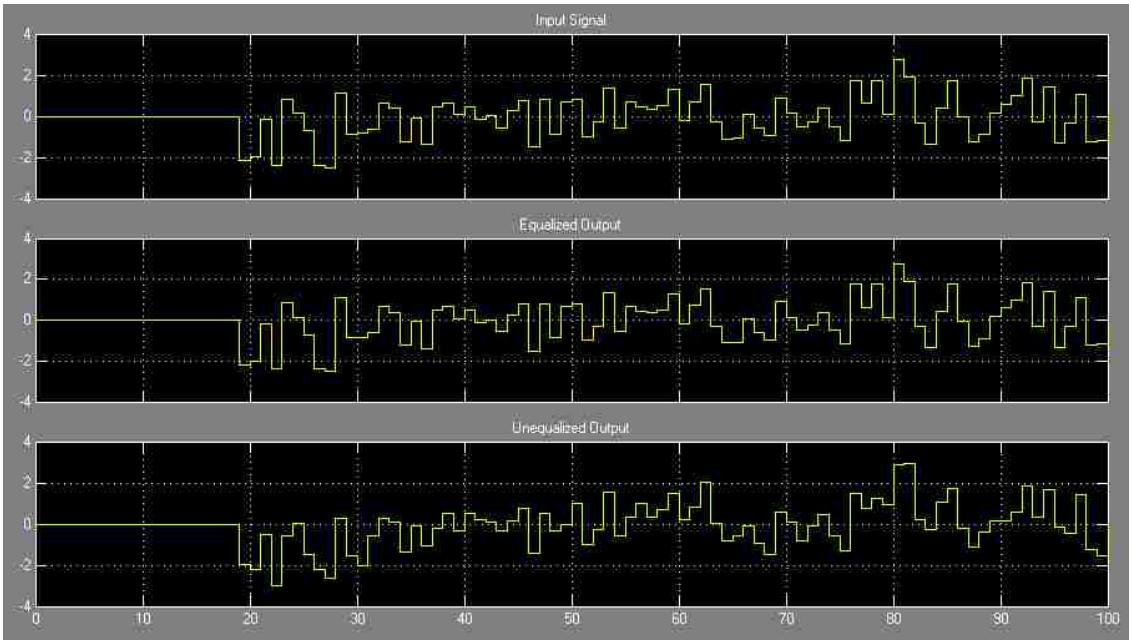


Figure B.6.2 – Example scope output

Part C – Equalizer Simulation with Noise

The main fault in the ZF equalizer lies in its poor handling of noise under certain channel responses. For this simulation, we would like the channel to be:

h(t) = [10 6 3 1 0 10 3 1]

This particular channel has a frequency response with large drop-outs, meaning that the equalizer response will have large peaks. Reconnect the AWGN block and set the noise power to -80 dB, and set the equalizer length to L = 101, then simulate the channel. Since this is a very high SNR, the equalizer output should still be fairly accurate. Save your scope output and note any differences between the equalized output and the original signal.

To see the equalizer's downfalls, we must simulate the model with much larger quantities of noise, similar to those found in wireless communications. Repeat the above simulation for noise powers of -40 dB, -30 dB, and finally -20 dB, which will make the noise just about 1/100 as powerful as the signal itself. At what point does the equalizer begin to choke and amplify the noise so much that the original signal becomes lost? Make sure to save the scope outputs and note in your report the noise level at which the equalizer becomes unreliable.

Post-Lab Questions:

Q1. In part A, what is the relation of the time delay on the output to the length of the channel impulse response and/or equalizer response?

Q2. An ideal ZF equalizer would require an infinite number of taps to perfectly cancel all ISI in a channel. Why would it be computationally infeasible to create an equalizer with an incredibly large number of taps to approximate this?

Q3. Assuming it *were* possible to create an equalizer with thousands of taps that is computable in a reasonable amount of time, why might it still not be a good idea to do this in a real system?

Q4. In wireless communications, the channels are no longer completely static, which means that h(t) can change with time as the transmitter, receiver, or other objects move around. What are some possible difficulties with designing an equalizer under these conditions?

VITA

Brice Aaron Hirst was born in the town of St. Louis, Missouri on the date of June 16, 1988. He graduated from Sacred Heart High School in Sedalia, Missouri in the year 2006, and received an Associate of Arts with Highest Honors from State Fair Community College in Sedalia in 2008. He graduated Summa Cum Laude with a bachelor's degree in Electrical Engineering from the Missouri University of Science & Technology in the year 2011, and was a student member of IEEE as well as a 3-year member of the Phi Kappa Phi Honor Society. He received his Master's degree in Electrical Engineering from the Missouri University of Science and Technology in December of 2012.