

Enhancing the DSP Toolkit of LabVIEW

Murat Tanyel
Dordt College

Abstract

Most Digital Signal Processing (DSP) courses rely heavily on MATLAB and/or C, representing the state of the art in textual programming, for their standard computer tools. Many textbooks are published containing examples, if not sections, devoted to these textual languages. We have argued, in a previous paper, that whereas this environment may be efficient in manipulating equations, textual implementation of processes best described by block diagrams loses its intuitive substance and gave examples in LabVIEW of implementations that are better left graphical. However, the standard DSP toolkit of LabVIEW is aimed at the practicing engineer/scientist who needs to process acquired data to reach other ends in contrast to a student whose aim is to learn about signal processing. LabVIEW's DSP toolkit is rich with high level algorithms but needs to be enhanced in order to serve the pedagogical needs of students of DSP. Having decided to teach *DSP* with LabVIEW a year ago, I have found myself writing many routines to complement the standard DSP toolkit as I have tried to demonstrate basic concepts. This paper will describe this additional toolkit that has been growing to make LabVIEW a better teaching tool in a DSP class.

I. Introduction

When I started teaching *DSP* last year, I chose LabVIEW as the computer tool for hands-on experiences and demos. The decision was against the common trend, for MATLAB has become the de facto standard for numerical computation in signal processing¹. This uncommon decision was taken because of two major reasons: i) My previous experience with LabVIEW²⁻⁶ has been very rewarding and I would like my students to have LabVIEW programming skills in addition to MATLAB programming skills they acquire in their *Introduction to Engineering* class at the freshman level and in *Linear Systems* at the sophomore level. ii) It was my contention that "processes that [are best] describe[d in terms of] what happens to various inputs to achieve an output, so easily depicted by block diagrams in control systems, communication systems and DSP, are better candidates for simulation and/or realization in a graphical programming environment than in a textual environment."⁷ My personal preference, as given in (i) above was not adequate to justify this choice, while reason (ii) could be addressed by other choices, such as MATLAB's SIMULINK and HP's VEE. SIMULINK has provided an effective graphical environment and thus 'valuable additional simulation capability' for control systems⁸. HP's VEE is another contender in the graphical programming realm which is also aimed at data acquisition and instrument control⁹. In the end, the ready availability of LabVIEW's full development system in

our electronics lab, mainly for the purpose of instrument control and data acquisition through IEEE 488 and RS 232, as opposed to the additional investment required for the alternatives won the day for my preference.

National Instruments' LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is based on the concept of data flow programming and is particularly suited to test and measurement applications¹⁰. The three important components of such applications are data acquisition, data analysis and data visualization. LabVIEW offers an environment which covers these vital components. Therefore, the full development version offers a number of *Signal Processing* functions, grouped under the headings of **Signal Generation**, **Time Domain**, **Frequency Domain**, **Measurement**, **Windows** and **Filters**, as displayed in Fig. 1, for the analysis of data acquired through its *Data Acquisition* or *Instrument Drivers* functions.

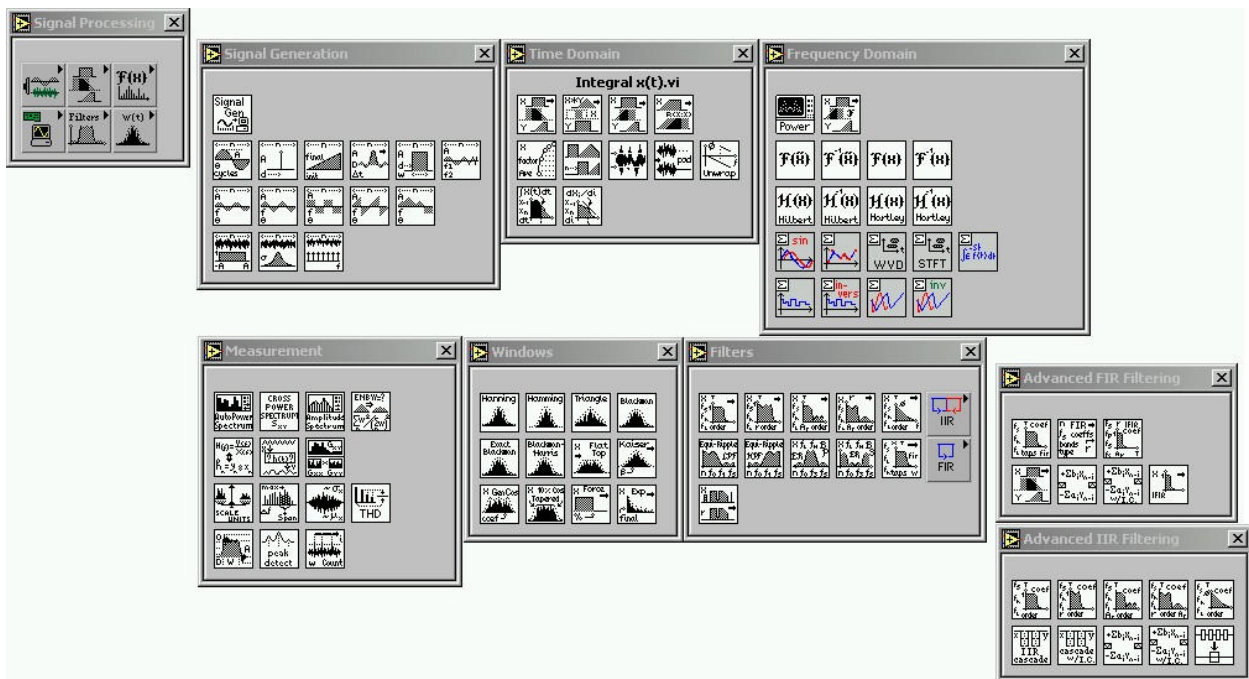


Figure 1: Different Signal Processing tools provided by the full development version of LabVIEW.

The functions depicted in Fig. 1 are meant to be used by engineers or scientists for whom the analysis of data per se, rather than a study of how data analysis is accomplished, is the main focus. For example, the Butterworth Filter.vi will filter an array of numbers according to the user-provided criteria (the cut-off frequency[ies], the sampling frequency, order, filter type), and the output will provide the filtered signal, but other features such as the coefficients and/or the poles/zeroes of this filter that a student of DSP might be interested to find out are all transparent to the user. Therefore, a typical DSP class, such as the one described in ⁷, requires many more 'functions' than the ones that are provided by National Instruments programmers. In the two

years that I have taught DSP, I have generated many additional ‘functions’ or virtual instruments (VIs) that constitute an additional toolkit for my DSP course. This paper will introduce this additional toolkit. Section II will give a list of all the VIs in the toolkit with brief descriptions. Section III will provide examples using some of these VIs. Section IV will conclude with my comments on how LabVIEW has served in DSP and on future work that can be undertaken.

II. VIs Developed and Used in Class

The VIs developed and used in class are tabulated in this section (see Table 1). The first column contains the names under which these VIs were originally saved. The second column contains the inputs to these VIs that the user must provide. The third column lists the outputs from the VI. The last column describes the VI briefly. The types of the inputs and outputs are also supplied. It should be noted that DBL stands for a double precision floating point number, CDB stands for complex number with double precision floating point components. Since LabVIEW employs a data driven language, called G, it does not have variable names. Instead, the data inputs and outputs can be labeled. It is these labels that are listed here and to those accustomed to traditional programming languages, some inputs and outputs may appear to have strange names. Almost all the VIs developed here perform functions that are beyond the scope of the signal processing functions of the full development system. The few that duplicate the original set have been indicated by italicized names. These few have been duplicated for pedagogical reasons and may have different nuances from those in LabVIEW’s set.

| Name | Inputs (type) | Outputs (type) | Function |
|---------------------------------------|---|---|--|
| <code>adc.vi</code> | x (DBL) R (DBL) B (integer) | b (Boolean array) | Converts the decimal number x to binary (2’s complement) with the given range R and the number of bits B . |
| <code>Arb0padder.vi</code> | Input Array (DBL array) n (integer) | Output Array (DBL array) | Pads the input array by n 0’s. |
| <code>Averager.vi</code> | N (integer) | Coeffs (DBL array) | Generates FIR coefficients for an N -length running averager. |
| <code>Canonical.vi</code> | x[n] (DBL array) bk (DBL array) ak (DBL array) | y[n] (DBL array) | 2 nd order canonical filter |
| <i><code>Cascade.vi</code></i> | x (DBL array) A matrix (DBL 2D array) B matrix (DBL 2D array) | y (DBL array) | Implements cascade filtering of arbitrary order with 2 nd order canonical segments. |
| <code>Coeffs to Pole Zeroes.vi</code> | b coeffs (DBL array) a coeffs (DBL array) | poles (CDB array) zeroes (CDB array) | Computes poles & zeroes from the numerator and denominator coefficients of a discrete system transfer function. |
| <i><code>Cosforex913.vi</code></i> | L (integer) fo (DBL) fs (DBL) | cos(2*pi*fo*t) (DBL array) | Generates a cosine sequence with frequency fo at the sampling frequency fs of length L . |

| | | | |
|----------------------|--|---|---|
| dac.vi | b (Boolean array) R (DBL) | dac (DBL) | Converts a binary (2's complement) number to decimal in the given range. |
| Direct form.vi | x[n] (DBL array) bk (DBL array) ak (DBL array) | y[n] (DBL array) | 2 nd order direct form filter |
| DTFT.vi | x (DBL array) f (DBL) | X(jf) (CDB) | Computes the DTFT of the sequence x at the frequency f . |
| DTFT_Horner.vi | x (DBL array) f (DBL) | X(jf) (CDB) | Computes the DTFT of the sequence x at the frequency f using Hörner's rule. |
| EvnOd.vi | N (integer) | Even/Odd (Boolean) | Determines whether the input N is even or not. |
| FIRL-HPKaiser.vi | fpass (DBL) fstop (DBL) Apass (DBL) Astop (DBL) fs (DBL) HP/LP (Boolean) | Coeffs (DBL array) | Generates coefficients of a LP or a HP Kaiser windowed FIR filter with the given frequency specs. |
| FIRWindowedFilter.vi | M (integer) ω_c or ω_b , $\times\pi$ (DBL) ω_a , $\times\pi$ (DBL) LP/BP-HP/BS (Boolean) | Coefficients (DBL array) | Generates coefficients of a LP, HP, BP or BS rectangular windowed FIR filter of length $2M+1$ with frequency specs (obtained from the appropriate combination of inputs). |
| GenDelay.vi | x[n] (DBL) n (integer) D (integer) w-in (DBL array) | w-out (DBL array) | Implements a delay of D for the incoming signal point $x[n]$ at index n . |
| Generate Pattern.vi | Array to be manipulated (DBL array) DONE (Boolean) | Array out (DBL array) | Allows the user to generate an arbitrary sequence and edit it until DONE button is pressed after which the last version of the sequence is kept. |
| LPFIRrect.vi | M (integer) w_c/π (DBL) | Coeffs (DBL array) | Generates coefficients of a lowpass rectangular windowed FIR filter of length $2M+1$ with cutoff frequency of w_c . |
| MagSpect.vi | x (DBL array) dB-Linear (Boolean) Refinement (integer) fs (DBL) | X (DBL array) xo (DBL) df (DBL) Magnitude (graph) | Making use of LabVIEW's DFT/FFT block, computes the magnitude spectrum of the input sequence x , outputs it in graph form and in numeric form as an array ($ X $), xo (beginning point for the horizontal axis) and df (interval between horizontal data points). |
| ModuloNReduce.vi | Input Sequence (DBL array) N (integer) | Modulo-N Reduced Sequence (DBL array) | Modulo-N reduces the input sequence. |
| Norm0Padder.vi | Input Array (DBL array) n (integer) | Output -n 0s padded (DBL array) | Pads the input array by n 0's and scales the array such that the power spectrum is preserved. |
| Overlap-Add.vi | Filter Coeffs (DBL Array) Data Chunk (DBL array) Trailing edge of prev. (DBL array) | Output-steady (DBL array) Trailer (DBL array) | Performs each chunk of convolution for the overlap-add method of convolution. |

| | | | |
|---------------------|--|---|---|
| ParEq.vi | fs (DBL) f0 (DBL) R (DBL) r (DBL) | ak (DBL array) bk (DBL array) | Generates forward and reverse coefficients of a 2 nd order parametric equalizer from the input specs. |
| ParPes.vi | fs (DBL) fo (DBL) Δf (DBL) | ak (DBL array) bk (DBL array) | Generates forward and reverse coefficients of a 2 nd order parametric resonator from the input specs. |
| PoleZeroDiag.vi | Poles (CDB array) Zeroes (CDB array) | Pole/Zero Diagram (graph) | Plots the pole/zero diagram. |
| PoleZeroDiagDisp.vi | Poles (CDB array) Zeroes (CDB array) | <NONE> | Displays the plot of the pole/zero diagram in a separate window. |
| SimpleWavetable.vi | Wavetable (DBL array) Delay (integer) STOP (Boolean) | Periodic Waveform (graph) | Generates a periodic waveform from a wavetable. |
| Wavetable3.vi | Open from file (Boolean) Delay (integer) # of periods (integer) fs, kHz (DBL) f, kHz (DBL) Method (integer) | Periodic Waveform (graph) | Generates a periodic waveform from a wavetable (from an existing file or using Generate Pattern.vi) at the frequency specified. |
| xtoXthruDTFT.vi | x (DBL array) fo (DBL) fend (DBL) N (integer) fs (DBL) | X(jf) (CDB array) xo (DBL) Δf (DBL) | Computes the spectrum of the input sequence x using the DTFT. The frequency range is defined by the inputs fo and fend with N increments. |

Table 1: DSP tools developed in house.

III. Examples

I will provide three examples of how I use LabVIEW in class. The first example, IDFTdemo.vi, uses only LabVIEW's signal processing toolkit. The second and third examples make use of the additional toolkit developed in house and described in section II.

IDFTdemo.vi: The inverse discrete-time Fourier transform (IDFT) may be obtained from the discrete-time Fourier (DFT) transform by the relation¹¹

$$\text{IDFT}(\mathbf{X}) = \frac{1}{N} [\text{DFT}(\mathbf{X}^*)]^* \quad (1)$$

where \mathbf{X} is the discrete-time Fourier transform of the sequence \mathbf{x} , N is the number of elements in either sequence and $*$ stands for complex conjugation.

Fig. 2 is the front panel of the VI that demonstrates Eq. 1. The user may choose from a sinusoidal, a rectangular pulse or a ramp generator to produce the time sequence x . In this example, a sinusoid is chosen. The parameters of the generator may be specified by the knobs “#

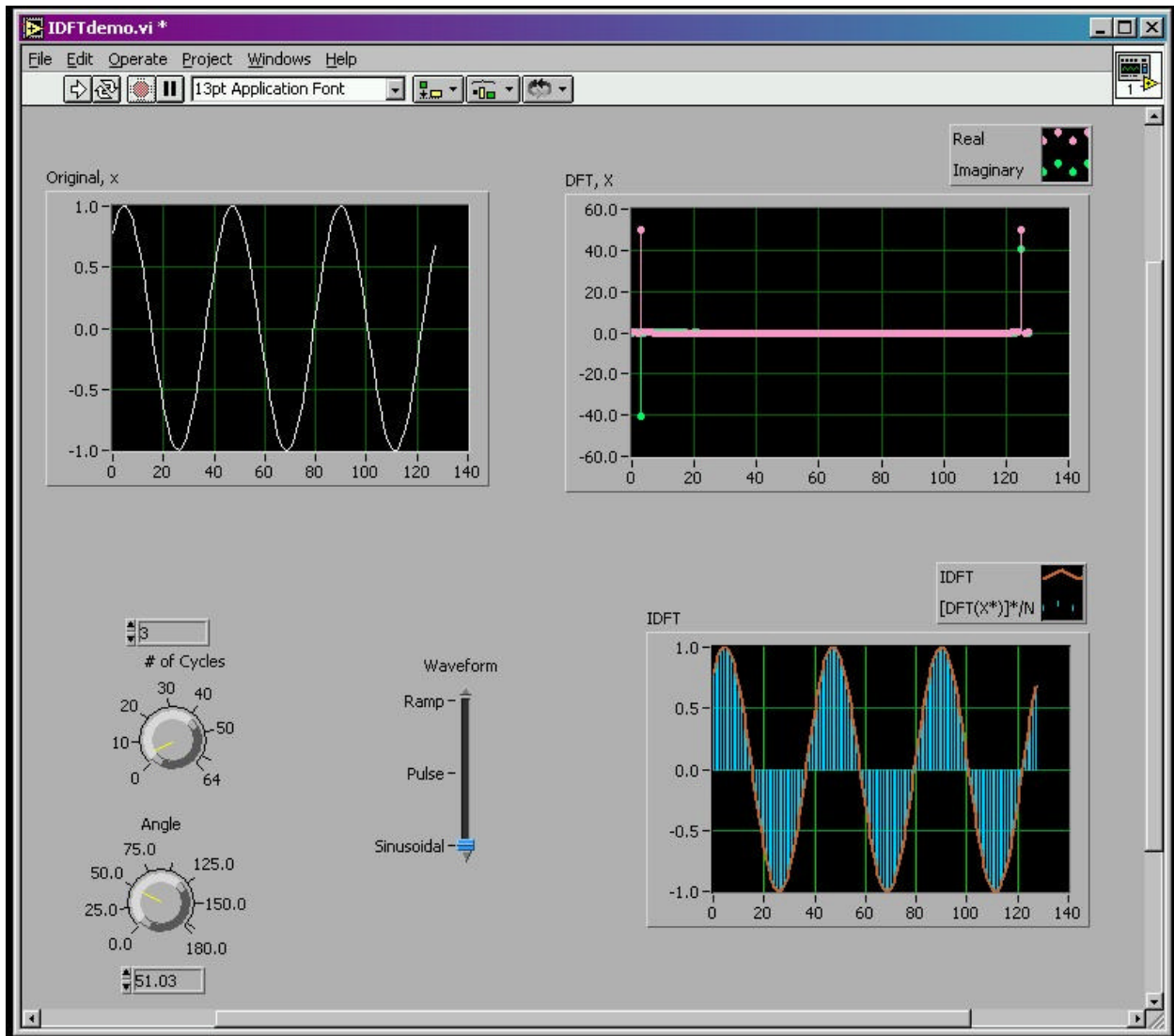


Figure 2: The front panel of *IDFTdemo.vi* that demonstrates how the IDFT may be obtained by using the DFT.

of Cycles” and “Angle”. The captions for these knobs are dynamic, if *Pulse* is selected, the captions become “Pulse Width” and “Delay” and if *Ramp* is selected, they are “Starting Value” and “Ending Value”. The first graph labeled “Original, x ” displays the sequence generated. The second graph, labeled “DFT, X ” displays the real and imaginary parts of the discrete Fourier transform of the original sequence. The final graph, labeled “IDFT” displays the inverse discrete

Fourier transform of the sequence X obtained in two ways: the solid red line is the plot of the sequence obtained by LabVIEW's IDFT function and the discrete blue plot is the IDFT obtained by employing LabVIEW's DFT function as specified in Eq. 1. In class, I make a note of pointing out that this demonstration in no way proves Eq. 1, but should be considered alongside the analytical development in the textbook¹¹ spelling out the derivation of Eq. 1.

Fig. 3 depicts the implementation of Eq. 1 in this demo. All the blocks in this diagram are available in the full development version of LabVIEW 5.1. The "Waveform" switch chooses

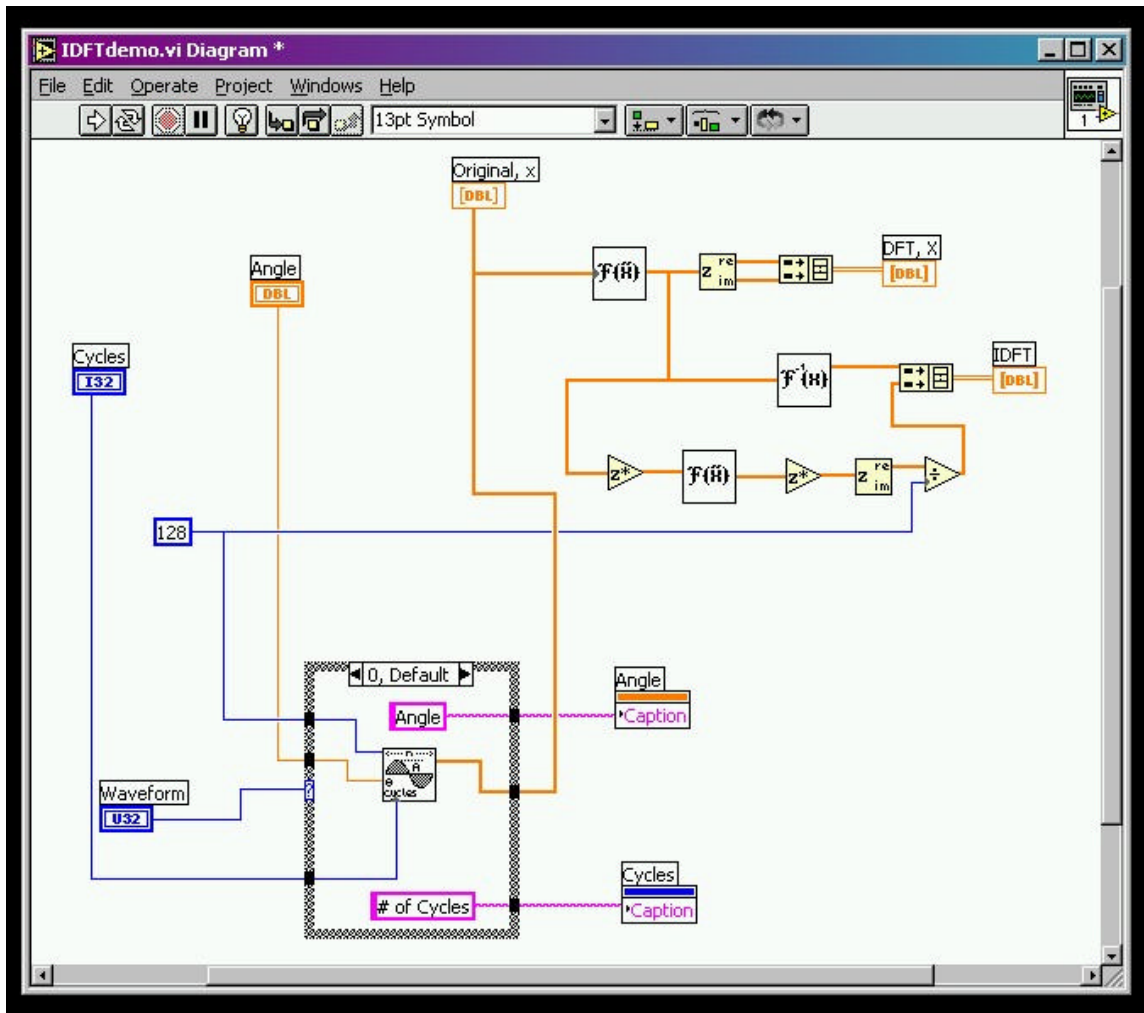


Figure 3: The block diagram of *IDFTdemo.vi* utilizing Eq. 1.

between the three cases of the case structure (the rectangular area in the lower left). This exposure has captured case '0' which generates the sine wave. The string constants "Angle" and "# of Cycles" (in magenta borders) determine the captions of the related knobs and change with each case. We note that the labels for these knobs are "Angle" and "Cycles", chosen before the

versatility was programmed in and hence the display of labels in the front panel is disabled to avoid confusion. The array coming out of the case structure is connected to the graph “Original, x ” and proceeds to the DFT block of LabVIEW, whose icon spells $\mathbf{F}\{\tilde{\mathbf{x}}\}$. The real and imaginary parts of its output (\mathbf{X}) are separated and graphed in “DFT, \mathbf{X} ”. To get back to the time domain, the output of the DFT block (\mathbf{X}) is connected in one path to LabVIEW’s IDFT block (whose icon spells $\mathbf{F}^{-1}\{\mathbf{x}\}$), which in turn is connected to one of the plots of the graph “IDFT”. The other path takes \mathbf{X} to the complex conjugate block (with the icon \mathbf{z}^*) to obtain \mathbf{X}^* . \mathbf{X}^* is connected to another DFT block and its output is again connected to the complex conjugate block, making up $[\mathbf{DFT}(\mathbf{X}^*)]^*$. The real part of this array is divided by 128 ($N = 128$ in this example) and graphed together with the array from the alternate path. We note that if Eq. 1 is applied correctly, the resulting array should have all real values and the last conversion is for the sake of avoiding a type conflict in the graph.

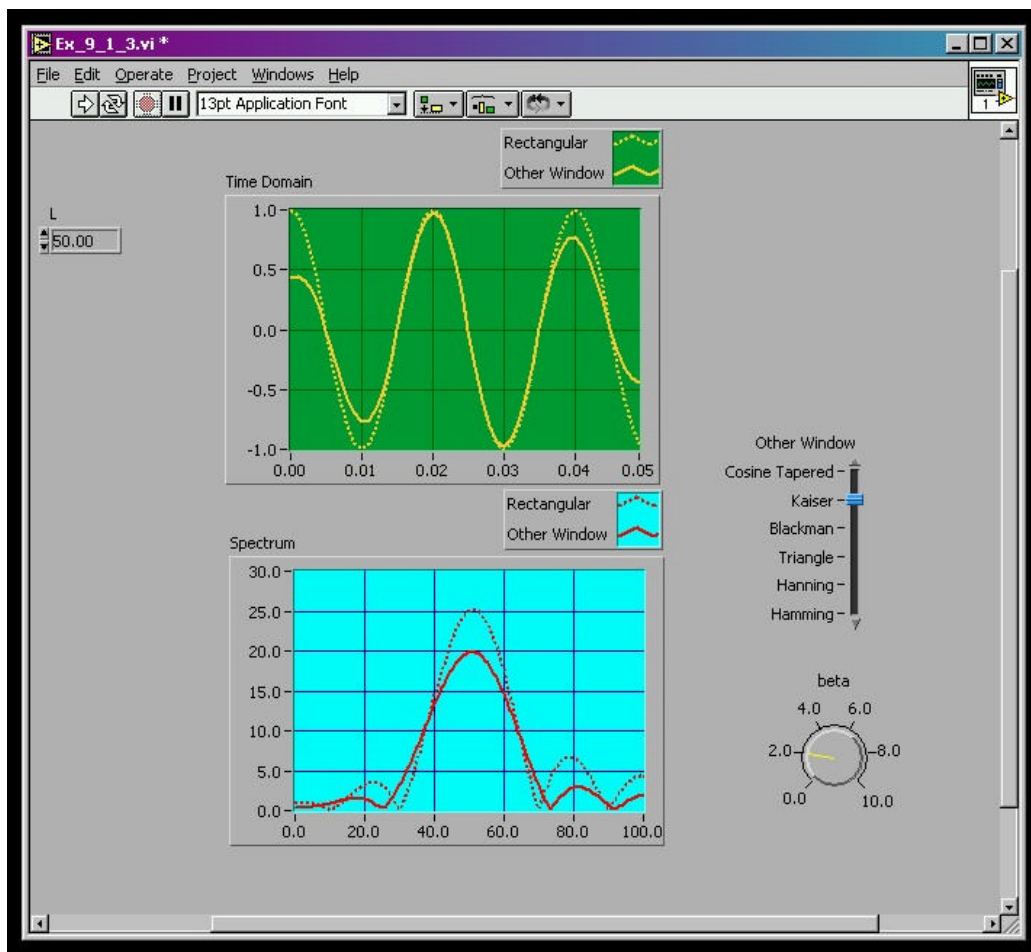


Figure 4: The front panel of *Ex_9_1_3.vi* which compares the effect of various windowing techniques on the spectrum of a 50 Hz sinusoid.

Ex_9_1_3.vi: Fig. 4 shows the front panel of a VI that demonstrates the effect of various windows as well as the record length (L) on the estimation of the spectrum of a 50 Hz sinusoid.

This front panel depicts the result of the run with $L = 50$ and Kaiser window with $\beta = 2$ selected. The front panel is otherwise self-explanatory. This VI is best run in continuous mode, selecting different windows and changing the relevant parameters while the VI is running.

A look at the block diagram of Ex_9_1_3.vi reveals that this demo utilizes `xtoXthruDTFT.vi` from our toolkit (Fig. 5). This VI (whose block has the icon with the characters spelling “x ? X thru DTFT”) estimates the spectrum of the input array in a range of frequencies making use of the discrete time Fourier transform.

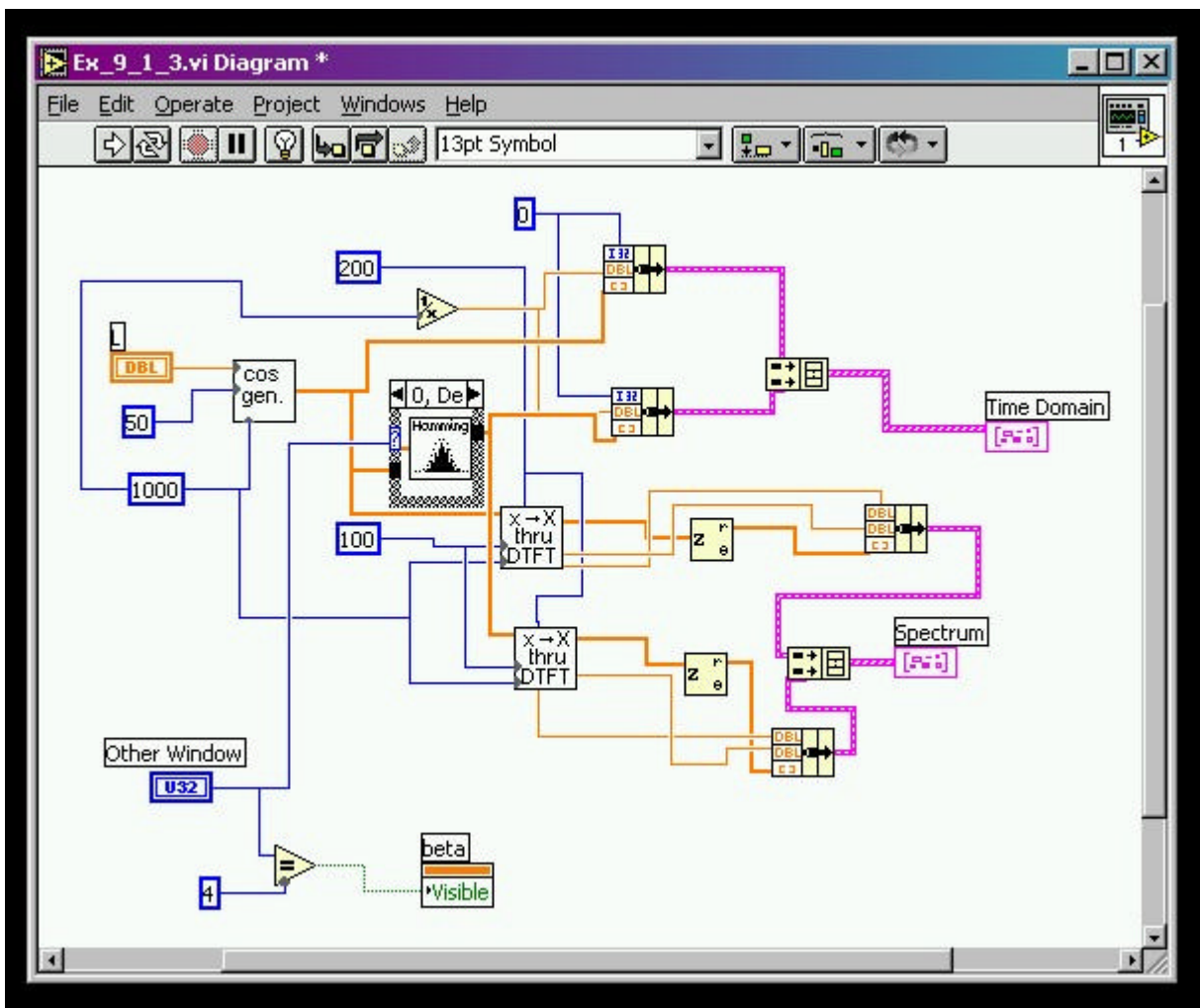


Figure 5: The block diagram of Ex_9_1_3.vi revealing the use of `xtoXthruDTFT.vi`.

The discrete time Fourier transform at a single frequency ω is given by¹¹:

$$X(\omega) = \sum_{n=0}^{L-1} x(n)e^{-j\omega n} \quad (2).$$

XtoXthruDTFT.vi makes use of DTFT.vi, another item in the home-brewed toolkit (Fig. 6), which implements Eq. 2. A study of Fig. 6 will reveal that XtoXthruDTFT.vi has inputs “x” (the time sequence), “fo” (beginning frequency), “fend” (final frequency), “fs” (sampling frequency), “N” (number of frequency points over which the spectrum is to be estimated). From the frequency range specification and N the appropriate inputs for DTFT.vi are calculated and this subVI is called in a for loop. The output of DTFT.vi at each frequency is accumulated in an array making up $X(jf)$. We note that LabVIEW’s help window is invoked, giving information on DTFT.vi: indicating its inputs and outputs on the icon and displaying explanatory text as entered when the VI was written.

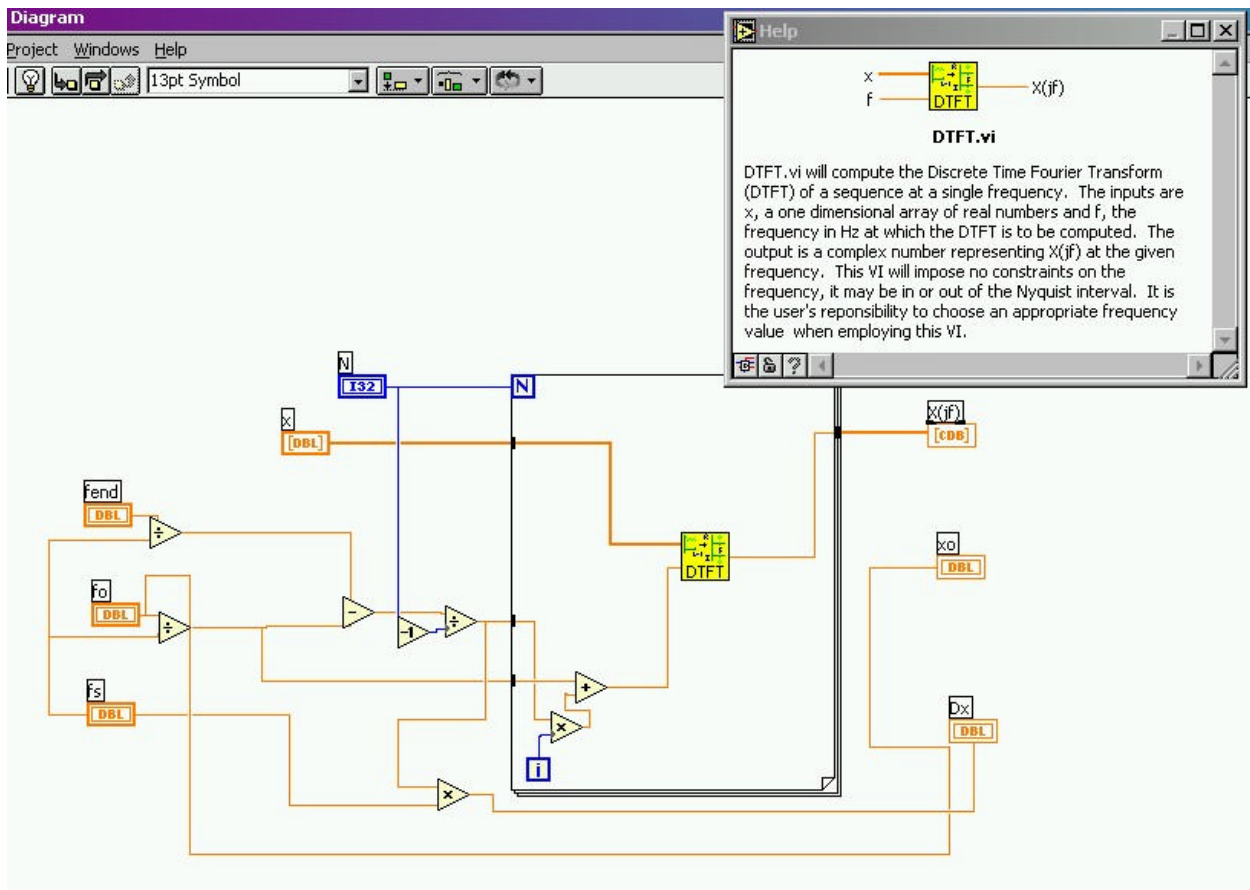


Figure 6: The block diagram of xtoXthruDTFT.vi.

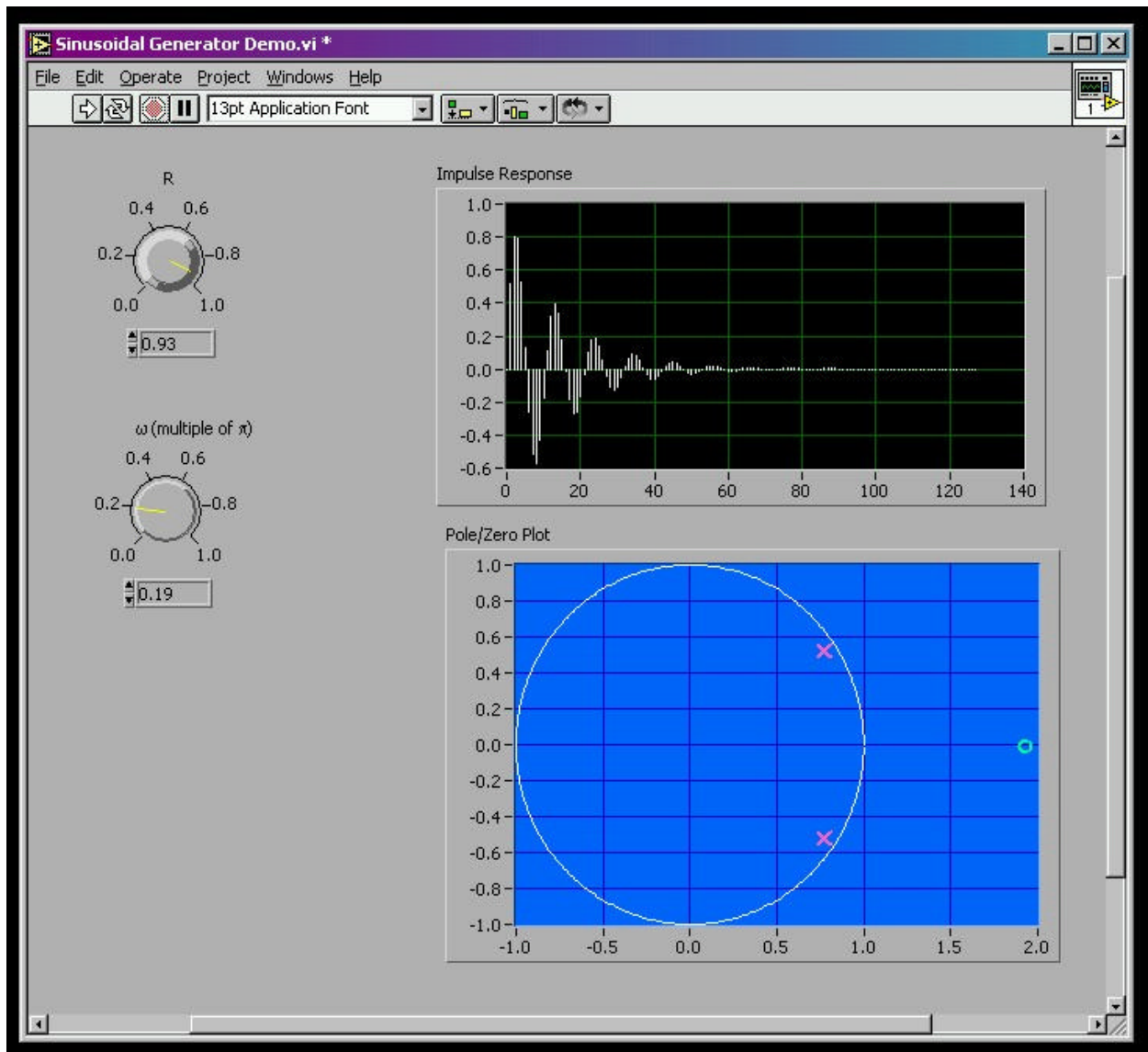


Figure 7: The front panel of Sinusoidal Generator Demo.vi.

Sinusoidal Generator Demo.vi: A filtering approach to generating a sinusoid would be to design a filter whose impulse response, $h(n)$, is the desired waveform. The filter whose transfer function is given by

$$H(z) = \frac{R \sin \omega_0 z^{-1}}{1 - 2R \cos \omega_0 z^{-1} + R^2 z^{-2}} \quad (3)$$

generates an exponentially decaying sinusoid of frequency ω_0 for $0 < R < 1$ as its impulse response¹¹. For $R = 1$, it generates a pure sinusoid. Fig. 7 depicts the front panel of *Sinusoidal Generator.vi* which demonstrates such a filter. Its inputs are R and ω_0 in terms of π (in other words, in normalized frequency units). The outputs are the impulse response of the filter and the pole-zero plot of its transfer function. Like the previous demo, this VI is best run in continuous mode. By changing the value of R , we can observe its effect on the impulse response and the pole/zero pattern of the filter and watch the impulse response become a pure sinusoid when the poles are exactly on the unit circle ($R = 1$).

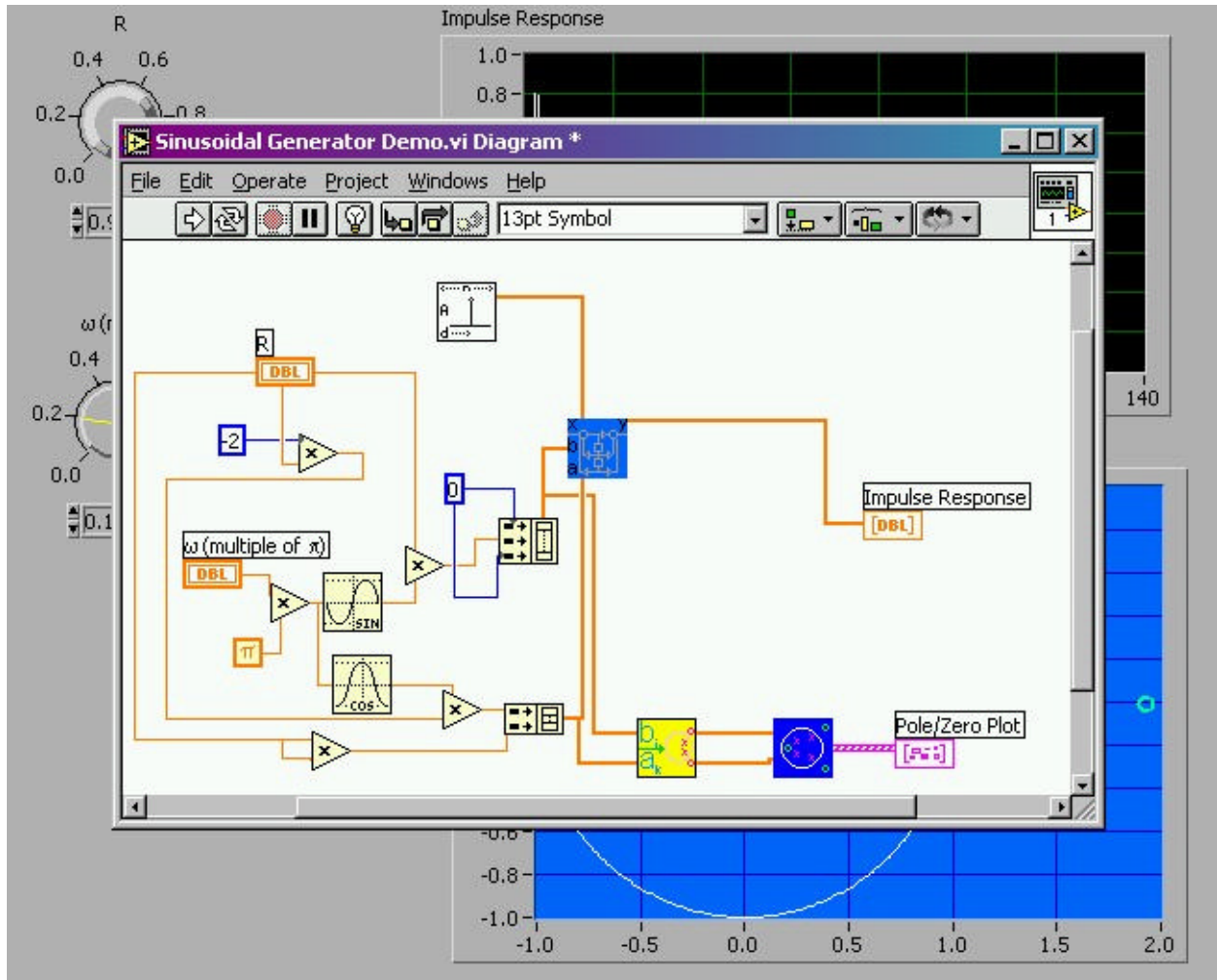


Figure 8: The block diagram of *Sinusoidal Generator Demo.vi*.

Fig. 8 depicts the block diagram that describes the computations for this simulation. We note that this VI employs *Canonical.vi*, listed in section II of this paper and described in⁷. Its b coefficients (or forward coefficients) are supplied by an *Build Array* block whose individual inputs

are 0 (a constant), $R\sin\omega_0z^{-1}$ (the reader may be convinced by following the blocks from the input labeled “ ω (multiple of π)” to the second input of *Build Array*) and another 0 (the same constant connected to the third input). Likewise, its a coefficients (or reverse coefficients) are supplied by another *Build Array* block with two inputs. It should be noted that *Cascade.vi* assumes $a_0=1$ and therefore only two a coefficients are supplied. The other VIs from this toolkit that this demo employs are *CoefFs to Pole Zeroes.vi* (the predominantly yellow block) that computes poles and zeroes from the coefficients of a transfer function and *PoleZeroDiag.vi* (the predominantly blue block to the left of “Pole/Zero Plot”) which prepares a plot of the poles and zeroes with the unit circle.

V. Conclusions

At the end of last year’s offering of DSP we reported on the suitability of LabVIEW as a computer tool for DSP and concluded⁷: “... mathematical formulae are better dealt with by textual languages and there are some algorithms, such as the implementation of circular buffers, that are very simple in the C language but present great difficulty in LabVIEW. Therefore, we should not abandon the use of such languages, but use every tool in applications where [their] strengths excel.” We noted that the instrument-like user interface of LabVIEW, its abundance of analysis VIs make it very attractive and inviting to implement most DSP routines and recommended that in applications where it is cumbersome to program in LabVIEW, we could have an experienced programmer write the painstakingly difficult algorithm and make it available as a subVI to the rest of the class so that students do not get bogged down by a few non-intuitive programs. This paper is a report on the state of the progress that has been made to achieve this end.

The works that remains is twofold. I plan to expand the toolkit in future offerings of the course. An equally important task is to “polish” all the VIs in the toolkit so that they all have their unique icons and their documentation so that LabVIEW’s help facility may display adequate information on each of the VIs. Right now, only a limited number of these VIs benefit from such luxuries (see Fig. 6 for an example).

The student response has been enthusiastic again this year. Part of the enthusiasm is due to the aesthetics of LabVIEW virtual instruments. I have seen many a student revisiting and perfecting the way the front panel looks long after his/her VI has achieved its computational goals and long after the class period has ended. If it is going to help students enjoy DSP, I am happy to go the extra mile and develop the additional DSP toolkit for this not-quite-standard-for-DSP programming environment.

ACKNOWLEDGEMENTS: The LabVIEW software used in this course was purchased through a grant to upgrade our electronics lab from Johnson Controls in Holland, MI. I would like to thank

Johnson Controls for helping make the teaching of not only *electronics* but also *DSP* a pleasant experience.

Bibliography

1. Ingle, V. K., Proakis, J. G. *Digital Signal Processing using MATLAB*, Pacific Grove, CA: Brooks/Cole (2000).
2. Tanyel, M., Quinn, R., Barge, E., "An Engineering Laboratory for Freshmen - Computer Utilization", in *1990 ASEE Annual Conference Proceedings*, Toronto, June 26-29 1990.
3. Scoles, K., Tanyel, M., Onaral, B., "Computing in Electrical Engineering Education at Drexel University", in *IEEE Transactions on Education*, vol. 36, no. 1, pp. 198-203, Feb. 1993.
4. Tanyel, M., *Engineering Explorations with LabVIEW*, Philadelphia, PA: Harcourt Brace Custom Publishers (1994).
5. Abu Zeid, O. A., Tanyel, M., "Innovation in Teaching Mechanical Engineering Applications", in *Proceedings of 1994 Frontiers in Education Conference*, pp. 82-86, Nov. 1994.
6. Tanyel, M., "Virtual Experimentation in Freshman and Sophomore Years," in *Proceedings of 58th Annual ASEE North Midwest Section Meeting*, Oct. 1996.
7. Viss, M. and Tanyel, M. "From Block Diagrams to Graphical Programs in DSP," in *Proceedings of the 2001 American Society for Engineering Education Annual Conference & Exposition*, Albuquerque, NM, June 24-27 2001.
8. Bishop, R. H., *Modern Control Systems Analysis & Design Using MATLAB[®] & SIMULINK[®]*, Menlo Park, CA: Addison-Wesley (1997).
9. Helsel, R., *Cutting Your Test Development Time with HP VEE*, Englewood Cliffs: Prentice Hall (1994).
10. Chugani, M. L., Samant, A. R., Cerna, M., *LabVIEW Signal Processing*, Upper Saddle River, NJ: Prentice Hall (1998).
11. Orfanidis, S. J., *Introduction to Signal Processing*, Upper Saddle River, NJ: Prentice Hall (1996).

MURAT TANYEL

Murat Tanyel is a professor of engineering at Dordt College. He teaches upper level electrical engineering courses. Prior to teaching at Dordt College, Dr. Tanyel taught at Drexel University where he worked for the *Enhanced Educational Experience for Engineering Students (E⁴)* project, setting up and teaching laboratory and hands-on computer experiments for engineering freshmen and sophomores. For one semester, he was also a visiting professor at the United Arab Emirates University in Al-Ain, UAE where he helped set up an innovative introductory engineering curriculum. Dr. Tanyel received his B. S. degree in electrical engineering from Bogaziçi University, Istanbul, Turkey in 1981, his M. S. degree in electrical engineering from Bucknell University,

Lewisburg, PA in 1985 and his Ph. D. in biomedical engineering from Drexel University, Philadelphia, PA in 1990.