Lab 0 – Introduction

The DSP lab consists of four of hardware experiments illustrating the programming of real-time processing algorithms on the Texas Instruments TMS320C6713 floating-point DSP. Programming of the DSP chip is done in C (and some assembly) using the Code Composer Studio (CCS) integrated development environment. In addition, the lab includes two MATLAB-based software experiments on digital filtering.

Familiarity with C programming is necessary in order to successfully complete this lab course. All of the C filtering functions in the textbook [1] translate with minor changes to the CCS environment. MATLAB is also necessary and will be used in the software experiments and to generate input signals to the DSP and to design the filters used in the various hardware examples.

The hardware experiments are real-time sample-by-sample processing examples and include aliasing and quantization effects; the circular buffer implementation of delays, FIR, and IIR filters; voice scramblers; wavetable generators; and several digital audio effects, such as comb filters, plain, allpass, and lowpass reverberators, Schroeder's reverberator, and several multi-tap, multi-delay, and stereo-delay type effects, tremolo, vibrato, flangers, wah-wah filters and phasers, as well as the Karplus-Strong string algorithm; various guitar distortion effects, such as fuzz and overdrive.

The lab assignments contain a short introduction to the required theory. More details, as well as several concrete C and MATLAB implementations, may be found in the book [1], which may be freely downloaded from the web page:

```
http://www.ece.rutgers.edu/~orfanidi/intro2sp/
```

0.1. Lab Guidelines

Attendance is *required* in all lab sessions (see the lab schedule at the beginning of this manual.) It is not possible to receive a grade of "A" if one of these sessions is missed. Due to the limited number of workstations and tight space, missed hardware labs cannot be made up. In addition, a 1–2 page lab report on each hardware lab must be submitted at the next lab session. A full multi-page reports is required for the software experiments (Lab-4 and Lab-6).

Students work in pairs on each workstation. Each lab section section has been split into two groups, A & B, that meet on alternate weeks (see lab schedule on the lab web page). Please make sure that you attend the right group (if in doubt please contact your TA).

0.2. Running C Programs

Most of the C programs will be written and run under the CCS IDE. However, practicing with and learning C can be done on any departmental computer in ELE-103. Computer accounts on `ece.rutgers.edu` may be obtained by contacting the system administrator of the ECE department, Mr. John Scafidi.

C programs may be compiled using the standard Unix C compiler `cc` or the GNU C compiler `gcc`. Both have the same syntax. It is recommended that C programs be structured in a modular fashion, linking the separate modules together at compilation time. Various versions of GCC, including a Windows version, and an online introduction may be found in the web sites:

```
http://gcc.gnu.org/
http://www.delorie.com/djgpp/
http://www.network-theory.co.uk/docs/gccintro/
```

Some reference books on C are given in Ref. [3]. As an example of using `gcc`, consider the following main program `sines.c`, which generates two noisy sinusoids and saves them (in ASCII format) into the data files `y1.dat` and `y2.dat`:

```
/* sines.c - noisy sinusoids */

#include <stdio.h>
#include <math.h>
```

```
#define L   100
#define f1  0.05
#define f2  0.03
#define A1  5
#define A2  A1

double gran();                     /* gaussian random number generator */

void main()
{
    int n;
    long iseed=2001;               /* gran requires iseed to be long int */
    double y1, y2, mean = 0.0, sigma = 1.0, pi = 4 * atan(1.0);
    FILE *fp1, *fp2;

    fp1 = fopen("y1.dat", "w");           /* open file y1.dat for write */
    fp2 = fopen("y2.dat", "w");           /* open file y2.dat for write */

    for (n=0; n<L; n++) {                /* iseed is passed by address */
        y1 = A1 * cos(2 * pi * f1 * n) + gran(mean, sigma, &iseed);
        y2 = A2 * cos(2 * pi * f2 * n) + gran(mean, sigma, &iseed);
        fprintf(fp1, "%12.6f\n", y1);
        fprintf(fp2, "%12.6f\n", y2);
        }

    fclose(fp1);
    fclose(fp2);
}
```

The noise is generated by calling the gaussian random number generator routine `gauss`, which is defined in the separate module `gran.c`:

```
/* gran.c - gaussian random number generator */

double ran();                             /* uniform generator */

double gran(mean, sigma, iseed)           /* x = gran(mean,sigma,&iseed) */
double mean, sigma;                       /* mean, variance = sigma^2 */
long *iseed;                              /* iseed passed by reference */
{
    double u = 0;
    int i;

    for (i = 0; i < 12; i++)              /* add 12 uniform random numbers */
        u += ran(iseed);

    return sigma * (u - 6) + mean;        /* adjust mean and variance */
}
```

In turn, `gran` calls a uniform random number generator routine, which is defined in the file `ran.c`:

```
/* ran.c - uniform random number generator in [0, 1) */

#define  a    16807                            /* a = 7^5 */
#define  m    2147483647                       /* m = 2^31 - 1 */
#define  q    127773                           /* q = m / a = quotient */
#define  r    2836                             /* r = m % a = remainder */
```

```
double ran(iseed)                               /* usage: u = ran(&iseed); */
long *iseed;                                    /* iseed passed by address */
{
   *iseed = a * (*iseed % q) - r * (*iseed / q);         /* update seed */

   if (*iseed < 0)                              /* wrap to positive values */
         *iseed += m;

   return (double) *iseed / (double) m;
}
```

The three programs can be compiled and linked into an executable file by the following command-line call of `gcc`:

```
gcc sines.c gran.c ran.c -o sines -lm           (unix version of gcc)
gcc sines.c gran.c ran.c -o sines.exe -lm       (MS-DOS version of gcc)
```

The command-line option `-lm` links the math library and must always be last. The option `-o` creates the executable file `sines` (or, `sines.exe` for MS-DOS.) If this option is omitted, the executable filename is `a.out` (or, `a.exe`) by default. Another useful option is the warning message option `-Wall`:

```
gcc -Wall sines.c gran.c ran.c -o sines -lm
```

If the command line is too long and tedious to type repeatedly, one can use a so-called response file, which may contain all or some of the command-line arguments. For example, suppose the file `argfile` contains the lines:

```
-Wall
sines.c
gran.c
ran.c
-o sines
-lm
```

Then, the following command will have the same effect as before, where the name of the response file must be preceded by the at-sign character `@`:

```
gcc @argfile
```

To compile only, without linking and creating an executable, we can use the command-line option `-c`:

```
gcc -c sines.c gran.c ran.c
```

This creates the object-code modules `*.o`, which can be subsequently linked into an executable as follows:

```
gcc -o sines sines.o gran.o ran.o -lm
```

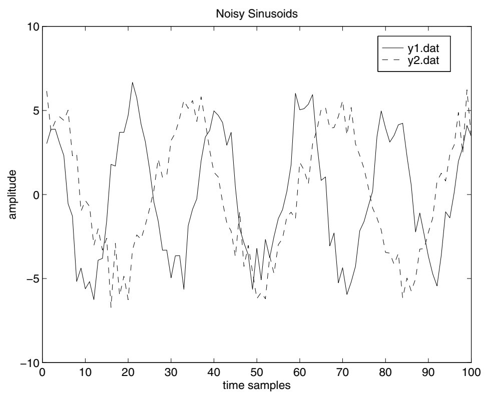
0.3. Using MATLAB

The plotting of data created by C or MATLAB programs can be done using MATLAB's extensive plotting facilities. Here, we present some examples showing how to load and plot data from data files, how to adjust axis ranges and tick marks, how to add labels, titles, legends, and change the default fonts, how to add several curves on the same graph, and how to create subplots.

Suppose, for example, that you wish to plot the noisy sinusoidal data in the files `y1.dat` and `y2.dat` created by running the C program `sines`. The following MATLAB code fragment will load and plot the data files:

```
load y1.dat;                              % load data into vector y1
load y2.dat;                              % load data into vector y2

plot(y1);                                 % plot y1
hold on;                                  % add next plot
plot(y2, 'r--');                          % plot y2 in red dashed style

axis([0, 100, -10, 10]);                  % redefine axes limits
set(gca, 'ytick', -10:5:10);              % redefine yticks
legend('y1.dat', 'y2.dat');               % add legends
xlabel('time samples');                   % add labels and title
ylabel('amplitude');
title('Noisy Sinusoids');
```

The resulting plot is shown below. Note that the command `load y1.dat` strips off the extension part of the filename and assigns the data to a vector named y1.



The command `hold on` leaves the first plot on and adds the second plot. The axis command increases the y-range in order to make space for the legends. The legends, labels, and title are in the default font and default size (e.g., Helvetica, size 10 for the Windows version.)

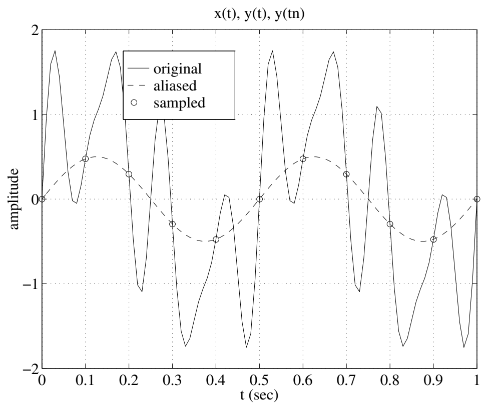
A more flexible and formatted way of reading and writing data from/to data files is by means of the commands `fscanf` and `fprintf`, in conjunction with `fopen` and `fclose`. They have similar usage as in C. See Ref. [2] for more details.

The next example is similar to what is needed in Lab-1. The example code below generates two signals $x(t)$ and $y(t)$ and plots them versus t. It also generates the time-samples $y(t_n)$ at the time instants $t_n = nT$. All three signals $x(t)$, $y(t)$, $y(t_n)$ span the same total time interval $[0, t_{\max}]$, but they are represented by arrays of different dimension ($x(t)$ and $y(t)$ have length 101, whereas $y(t_n)$ has length 11). All three can be placed on the same graph as follows:

```
tmax = 1;                                 % max time interval
Nmax = 100;                               % number of time instants
Dt = tmax/Nmax;                           % continuous-time increment
T = 0.1;                                  % sampling time interval

t  = 0:Dt:tmax;                                        % continuous t
x = sin(4*pi*t) + sin(16*pi*t) + 0.5 * sin(24*pi*t);    % signal x(t)
y = 0.5 * sin(4*pi*t);                                 % signal y(t)

tn = 0:T:tmax;                            % sampled version of t
yn = 0.5 * sin(4*pi*tn);                  % sampled version of y(t)

plot(t, x, t, y, '--', tn, yn, 'o');      % plot x(t), y(t), y(tn)

axis([0, 1, -2, 2])                       % redefine axis limits
```

```
set(gca, 'xtick', 0:0.1:1);              % redefine x-tick locations
set(gca, 'ytick', -2:1:2);               % redefine y-tick locations
set(gca, 'fontname', 'times');           % Times font
set(gca, 'fontsize', 16);                % 16-point font size
grid;                                    % default grid

xlabel('t (sec)');
ylabel('amplitude');
title('x(t), y(t), y(tn)');

axes(legend('original', 'aliased', 'sampled'));   % legend over grid
```

The following figure shows the results of the above commands. Note that the *x*-axis tick marks have been redefined to coincide with the sampled time instants $t_n = nT$.



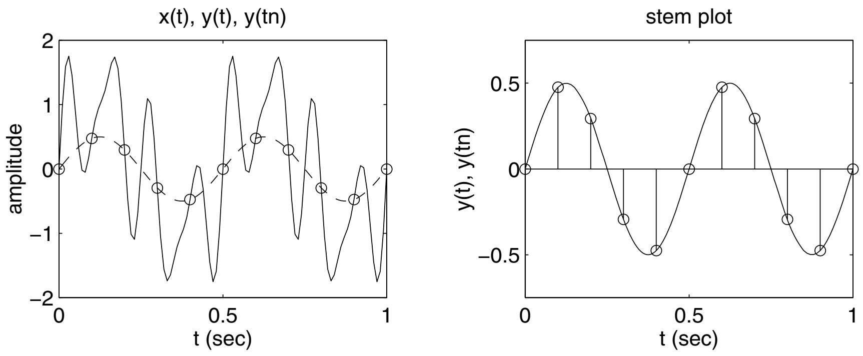
The `'o'` command plots the sampled signal $y(t_n)$ as circles. Without the `'o'`, the plot command would interpolate linearly between the 11 points of $y(t_n)$.

The font has been changed to Times-Roman, size 16, in order to make it more visible when the graph is scaled down for inclusion in this manual. The command `axes` creates a new set of axes containing the legends and superimposes them over the original grid (otherwise, the grid would be visible through the legends box.)

The next program segment shows the use of the command `subplot`, which is useful for arranging several graphs on one page. It also illustrates the `stem` command, which is useful for plotting sampled signals.

```
subplot(2, 2, 1);                        % upper left subplot

plot(t, x, t, y, '--', tn, yn, 'o');     % plot x(t), y(t), y(tn)

xlabel('t (sec)');
ylabel('amplitude');
title('x(t), y(t), y(tn)');

subplot(2, 2, 2);                        % upper right subplot

plot(t, y);                              % plot y(t)
hold on;                                 % add next plot
stem(tn, yn);                            % stem plot of y(tn)

axis([0, 1, -0.75, 0.75]);               % redefine axis limits

xlabel('t (sec)');
ylabel('y(t), y(tn)');
title('stem plot');
```

The resulting graph is shown below. Note that a 2×2 subplot pattern was used instead of a 1×2, in order to get a more natural aspect ratio.



Finally, we mention some MATLAB resources. Many of the MATLAB functions needed in the experiments are included in Appendix D of the text [1]. Many MATLAB on-line tutorials can be found at the following web sites:

```
http://www.mathworks.com/academia/student_center/tutorials/index.html
http://www.eece.maine.edu/mm/matweb.html
```

0.4. References

[1] S. J. Orfanidis, *Introduction to Signal Processing*, online book, 2010, available from: `http://www.ece.rutgers.edu/~orfanidi/intro2sp/`

[2] MATLAB Documentation: `http://www.mathworks.com/help/techdoc/`

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

S. P. Harbison and G. L. Steele, *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1984.

A. Kelly and I. Pohl, *A Book on C*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1990.