ELEN E3084: Signals and Systems Lab
Lab II: Elementary Signals

1 Introduction

The purpose of this lab is to introduce MATLAB commands relevant to the representation and manipulation of elementary signals. We need to understand how MATLAB expresses complex numbers and vectors in order to be able to represent a signal in time and its fourier transform.

Our tool set wouldn't be complete if we didn't have a way to plot signals and visualize the time and frequency domains. We will complete this lab by presenting the commands necessary for plotting presentation-ready plots.

Before we begin we will start again the diary function which will capture all our commands. Go the `lab2` directory using the `cd` command the same way as in UNIX. Then type:

```
>> diary on;
```

Note that again you should try to reproduce some of the examples that will be presented below. By receiving your diary file at the end of the lab session we will have a simple way to see that you spent some time experimenting as this is not supposed to be a hard grading policy.

2 Complex numbers

It is very easy to handle complex numbers in MATLAB. The special characters `i` and `j` are built-in variables and hold the value of the imaginary unit or simply $\sqrt{-1}$. For example all following statements define the same complex number.

```
>> 2 + 3*i
ans =
   2.0000 + 3.0000i
>> 2 + 3*j
ans =
   2.0000 + 3.0000i
>> 2 + 3i
ans =
   2.0000 + 3.0000i
```

```
>> 2 + 3j
ans =
   2.0000 + 3.0000i
```

Notice that you don't need the multiplication symbol `*` as MATLAB can parse the expression either way. The `Ni` format for imaginary numbers is used in optimization of MATLAB scripts which is outside of the scope of this lab.

Once we enter a complex number then pretty much all the relevant elementary functions presented in the previous lab can be used. For example even the complex logarithm is defined as we see in:

```
>> x = 2 + 3*i;
>> log(x)
ans =
   1.2825 + 0.9828i
```

Moreover, given a complex number `z` then the functions `real(z), imag(z), conj(z), abs(z)` and `angle(z)` calculate the real part, the imaginary part, the conjugate, the absolute value and the angle of `z` respectively. One thing to note is that the angle is defined in the interval $(-\pi, \pi)$. This can cause confusion if the exact angle is needed. For example for $z = 1 - j$ we have $Re\{z\} = 1, Im\{z\} = -1, z^* = 1 + j, |z| = \sqrt{2}$ and $\angle z = 3\pi/4$ Now if we try to reproduce the same result with MATLAB we will get:

```
>> z = 1 - j;
>> disp(['real part: ',num2str(real(z))]);
real part: 1
>> disp(['imag part: ',num2str(imag(z))]);
imag part: -1
>> disp(['conjugate: ',num2str(conj(z))]);
conjugate: 1+1i
>> disp(['abs value: ',num2str(abs(z))]);
abs value: 1.4142
>> disp(['angle:     ',num2str(angle(z))]);
angle:     -0.7854
```

Everything is what we expect except from $\angle z$. Due to the constraint mentioned above, MATLAB calculates $-\pi/4$ instead of the real angle $3\pi/4$. Indeed, $-\pi/4 = 3\pi/4 - \pi$. If the real angle is needed then we need to correct the result ourselves. In addition the angle is calculated in radians. Conversion in degrees is also left to us.

3 Vectors

3.1 Definition of vectors

MATLAB is extremely fast and efficient in manipulating vectors and matrices. This is in fact what the name MATrix LABratory denotes. There are two types of vectors. Row vectors and column vectors. By convention MATLAB uses column vectors to represent signals. For displaying purposes we will use row vectors since they occupy less lines when printed.

The simplest way to create a row vector is by the bracket concatenation operator we first saw in string manipulation. We can use either a space or a comma to separate the different values. In fact typing:

```
>> x = [1 -12.3 pi sqrt(2)]
```

or

```
>> x = [1,-12.3,pi,sqrt(2)]
```

both give:

```
x =
    1.0000  -12.3000    3.1416    1.4142
```

If we want to get column version of the same vector `x` we can use the transpose operator {`.'`}. For example:

```
>> x.'
ans =
    1.0000
  -12.3000
    3.1416
    1.4142
```

Vectors can be real or complex but numerical and string values cannot be mixed in the same vector[1]. For example a complex vector can be defined as:

```
>> z = [1 -j 3+2i]
z =
   1.0000                  0 - 1.0000i   3.0000 + 2.0000i
```

[1]For that we have to use cell arrays which is outside of the scope of this lab

If we want to define vectors straight in the column form instead of using the transpose operator we can concatenate the values using semicolons. So

```
>> x = [1;-12.3;pi;sqrt(2)]
```

gives:

```
x =
    1.0000
  -12.3000
    3.1416
    1.4142
```

3.2 Manipulation of vectors

We can select the *k-th* element of a vector using subscripted indexing. For example:

```
>> a = [3;-5;12;0]
a =
     3
    -5
    12
     0
>> a(2)
ans =
    -5
>> a(4)
ans =
     0
```

Notice the use of parenthesis and that the index is *1-based* and *not 0-based*.

The keyword `end`, first seen in the `for` and `if` constructs, can be used to select the last element of a vector, row or column. For example:

```
>> b = [-5,-2,0,0,12]
b =
    -5    -2     0     0    12
>> b(end)
ans =
    12
```

Essential role in the manipulation of vectors plays the colon {:} operator. We have already seen the colon operator used in `for` loops but here we will explain what it does. In its simplest form the colon operator can generate an index vector. For example:

```
>> idx1 = [3:6]
idx1 =
     3     4     5     6
>> idx2 = [-3:1]
idx2 =
    -3    -2    -1     0     1
```

This means that we can use the colon operator to *slice* a vector which means to select a part of it. For example:

```
>> c = [-3,2,0,2,5,7,9]
c =
    -3     2     0     2     5     7     9
>> c(3:6)
ans =
     0     2     5     7
```

or using the above defined `idx1`

```
>> c(idx1)
ans =
     0     2     5     7
```

Notice that when we use the colon operator for indexing we don't have to use square brackets.

We can *find* the indices of the elements of a vector that match a certain logical condition using the `find` command. For example we can find the positive elements of the vector `a` using:

```
>> a = [-2,3,-1,5,7]
a =
    -2     3    -1     5     7
>> idx = find(a>0)
idx =
     2     4     5
```

Find returns that the 2nd 4th and 5th element of the vector `a` is positive. Now if we use the variable `idx` we can see the values of the elements that are positive:

```
>> a(idx)
ans =
     3     5     7
```

Finally we can selectively delete elements of a vector using the empty square bracket operator. We can use all the methods to select elements such us subscripted indexing or slicing and just assign the empty element which is the empty square brackets. For example:

```
>> b = [2,4,2,1,6,7,4,6]
b =
     2     4     2     1     6     7     4     6
>> idx = find(b<=2)
idx =
     1     3     4
>> b(idx) = []
b =
     4     6     7     4     6
```

As you see we found the indices of the elements of `b` that are smaller or equal to 2 and we deleted them from `b`.

3.3 Special vectors

We often need to create special long vectors automatically since explicitly typing every element would be tedious. Vectors consisting of only zeros or ones turn out to be especially useful in MATLAB programming. We can generate a row vector of zeros or ones using the commands:

```
>> disp(zeros(1,5))
     0     0     0     0     0
>> disp(ones(1,3))
     1     1     1
```

If we need column versions we can get them either with transposition or by using:

```
>> disp(zeros(5,1))
     0
     0
     0
     0
     0
```

```
>> disp(ones(3,1))
     1
     1
     1
```

It is often required to generate vectors consisting of equally spaced numbers. In order to do that we can use the general form of the colon operator that we introduced before. For example typing:

```
>> [-0.3 : 0.1 : 1.5]
ans =
  Columns 1 through 10
   -0.3000   -0.2000   -0.1000    0.0000    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 11 through 19
    0.7000    0.8000    0.9000    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000
>> [7 : -2 : -1]
ans =
     7     5     3     1    -1
```

gives us a first vector with the elements starting from -0.3, increasing with step 0.1 and ending at 1.5 and second vector starting at 7 decreasing by step -2 down to -1.

Another way to create equally spaced numbers is by using the `linspace` command:

```
>> linspace(3,10,5)
ans =
    3.0000    4.7500    6.5000    8.2500   10.0000
```

which gives us 5 equally spaced points between 3 and 10 inclusive.

3.4 Arithmetic operations with vectors

Arithmetic operations are defined either between vectors of the same length or between a scalar and a vector. For example addition and subtraction is defined as an element-by-element operation:

```
>> a = [1,3,-2];
>> b = [0,2,1];
>> a + b
ans =
     1     5    -1
>> a - b
ans =
     1     1    -3
```

The multiplication, division and power operators, however need special attention. They have to be preceded by a dot in order to be performed element-by-element as illustrated below:

```
>> a = [1,3,-2];
>> b = [0,2,1];
>> b .* a
ans =
     0     6    -2
>> b ./ a
ans =
         0    0.6667   -0.5000
>> b .^ a
ans =
     0     8     1
```

In the case of a scalar and a vector:

```
>> a = [1,3,-2];
>> b = 2;
>> a + b
ans =
     3     5     0
>> a - b
ans =
    -1     1    -4
>> a .* b
ans =
     2     6    -4
>> a ./ b
ans =
    0.5000    1.5000   -1.0000
>> a .^ b
ans =
     1     9     4
```

3.5 Functions on vectors

Most of the functions we have seen so far are *vectorized* which means that they can accept a vector as an input argument and perform their operation on each element separately. So for example:

```
>> x = [1,3,2.1,exp(1)]
```

```
x =
    1.0000    3.0000    2.1000    2.7183
>> log(x)
ans =
         0    1.0986    0.7419    1.0000
>> sin(x)
ans =
    0.8415    0.1411    0.8632    0.4108
>> imag(x)
ans =
     0     0     0     0
```

We have to note though that there are some functions which have special meaning when operate on vectors. Assuming a vector `a` then the `length(a)` gives the number of elements whereas the `sum(a)` and `prod(a)` calculate the sum and product of its elements. The function `diff(a)` calculates the difference between each consecutive element. Finally `mean(a)` and `var(a)` calculate the mean and variance respectively. Assuming a column vector then `flipud(a)` reverses it upside-down and assuming a row vector then `fliplr(a)` reverses it left-right. For example:

```
>> a = [1,3,2,5];
>> disp(length(a))
     4
>> disp(sum(a))
    11
>> disp(prod(a))
    30
>> disp(diff(a))
     2    -1     3
>> disp(mean(a))
    2.7500
>> disp(var(a))
    2.9167
>> disp(fliplr(a))
     5     2     3     1
```

Notice that the `diff` command returns a vector with length one element less than the original length (3 instead of 4 elements in this case).

To Do 1

Generate vectors.

We will now make use of the commands we just learned in order to generate some specific vectors. Note that you shouldn't use any loops such as `for` or `while` just the colon operator

and elementary functions. Show the LA or TA the code that generates each of the row vectors below:

- `1, 3, 5, 7, 9, 11`
- `5, 5, 5, 5, 5`
- `4 - 3i, 2 - 2i, 0 - 1i, -2`
- `4, 16, 64, 256`

4 Plotting

4.1 Plots and sublots

MATLAB makes it possible to create sophisticated plots. Using simple commands we are able to visualize elementary signals and even include them in our reports.

The basic idea of plotting in MATLAB is very simple. Assuming that the independent `x` and dependent `y` variables are stored in vectors of the same length then the `plot(x,y)` command plots the pairs of points in the two-dimensional plane. By default `plot` connects the points by straight line segments but options like dashed lines or no lines at all are possible. If the plotting is successful, a window pops up containing the plot. Finally, using the menus on the window the plot can be printed or saved.

If `y` is a vector, then the simplest plotting command is `plot(y)`. In this case the vector elements are the dependent variables and the indices of the vector (in increasing order) are the independent variables. Here is an example with the resulting plot shown in figure 1:

```
>> x = linspace(0,4*pi,200);
>> y = sin(x);
>> plot(y)
```

Independent variables are included using the command `plot(x,y)` where the vector `x` contains the independent variables and the vector `y` (of the same length) contains the dependent variables. In this fashion more plots can be included in one figure by putting the corresponding pairs of vectors inside the parenthesis of the `plot` command. This is illustrated in the figure 2 and the code below:

```
>> x = linspace(0,4*pi,300);
>> y1 = sin(x);
>> y2 = cos(x);
```

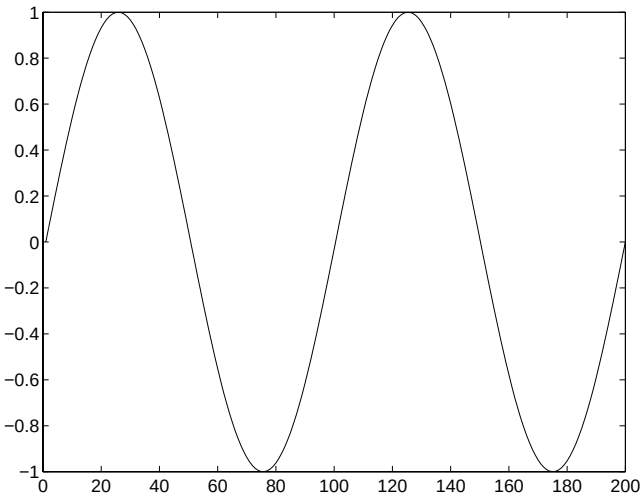

Figure 1: A simple plot of a sine where the independent variables are the vector indices

```
>> y3 = y1.^2;
>> plot(x,y1,x,y2,x,y3)
>> axis tight
```

More plots in the same figure look often neater by using the subplotting option of MATLAB. Here each plot command is preceded by the `subplot` command. The `subplot` and `plot` commands are separated by a comma. We will use a simple `subplot` command we can place `plots` vertically. The command is given by `subplot(m,i,k)` where `m` and `k` are integers and (for us) the middle number is always 1. The number `m` refers to the total number of plots and `k` refers to the indices of particular plots. Here is an example of the functions plotted in figure 2 but this time in figure 3 using subplots.

```
>> x = linspace(0,4*pi,300);
>> y1 = sin(x);
>> y2 = cos(x);
>> y3 = y1.^2;
>> subplot(3,1,1)
>> plot(x,y1)
>> axis tight
>> subplot(3,1,2)
>> plot(x,y2)
>> axis tight
>> subplot(3,1,3)
>> plot(x,y3)
>> axis tight
```

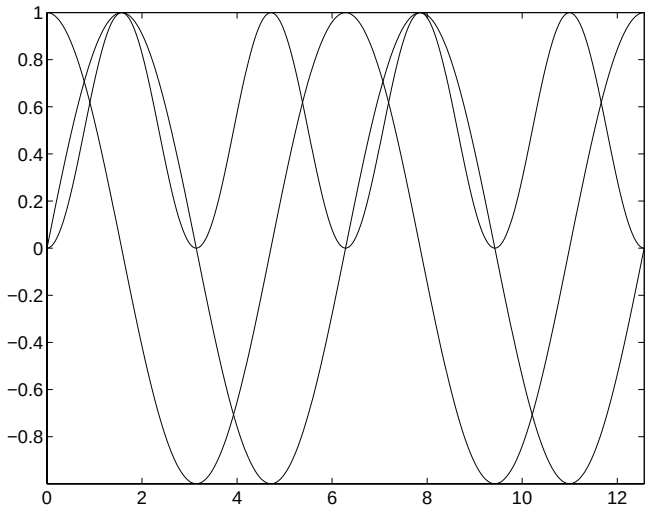

Figure 2: A plot of three functions with independent variables

4.2 Miscellaneous commands

Annotating our plots makes them easier to read. Adding a title, x- and y-axis labels is easy to do using the commands `title, xlabel` and `ylabel` respectively. Adding a legend with names is also easy using the command `legend`. We can also manipulate the shape of the axis by using the `axis` command using a variety of arguments (see `help axis` for more). For better readability we can also make visible a grid using the `grid on` command. We can see those commands in action the following snippet of code which generates the figure 4:

```
>> t = linspace(0,2,500);
>> v = exp(-3*t);
>> plot(t,v)
>> grid on
>> title('Exponential decay')
>> xlabel('Time (sec)')
>> ylabel('Voltage (mV)')
>> axis square
```

Lastly two other useful commands are the `figure` command and the `close` command. With `figure` we tell MATLAB to create a new figure so that we don't overwrite any existing one. This way we can have multiple figures displayed on the screen at once. Using `close` we tell MATLAB to close the current figure and with `close all` we can close all the open figures.

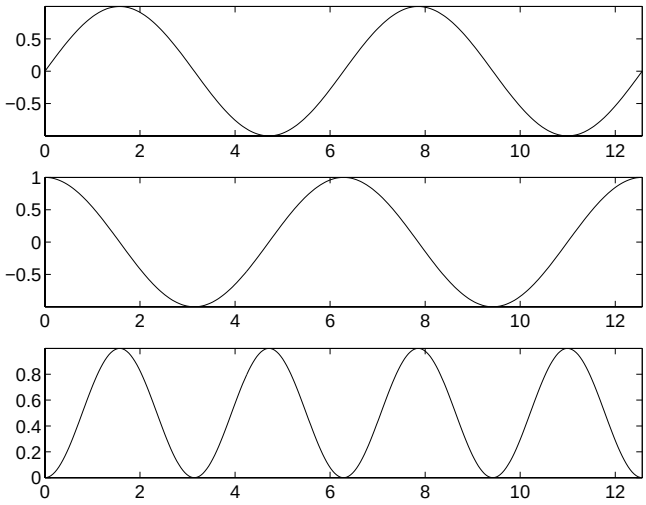

Figure 3: A plot of three functions with independent variables using three subplots

4.3 Exporting plots for publications

Even though MATLAB is a powerful plotting tool its built-in functions fall short when it comes to creating plots for high-quality publications such as conference papers. To overcome those limitations Mathworks released a set of functions that make this task easier and it is certainly worth mentioning here. Following the link:

`http://www.mathworks.com/company/digest/december00/export.shtml`

you should download the four functions `exportfig, previewfig, applytofig` and `restorefig` and save them under the `/lab2` directory. Now we will go through an example which will generate the plot on figure 5.

To Do 2

Export a plot.
We will now revisit the plot of figure 2 but this time we will export it using the `exportfig` function. Type the following code on the command prompt:

```
>> x = linspace(0,4*pi,300);
>> y1 = sin(x);
>> y2 = cos(x);
>> y3 = y1.^2;
>> plot(x,y1,x,y2,x,y3)
>> grid on
>> axis tight
>> axis([0 4*pi -1.5 2.5])
>> legend('sin','cos','sin^2');
>> title('Plot of sin, cos and sin^2')
```

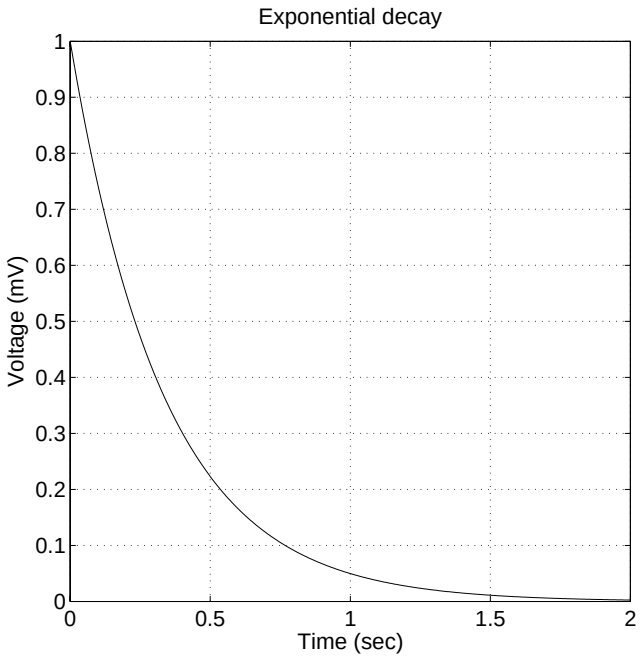

Figure 4: Demonstrating miscellaneous plotting commands

Using the mouse resize the plot window so that it looks shorter in height as in figure 5. Then go the `/lab2` directory using the `cd` command. Export the plot using the following commands and submit the `expfig.eps` file.

```
>> set(gcf,'paperunits','centimeters');
>> exportfig(gcf,'expfig.eps','bounds','tight','width',8.5,'linestylemap','bw');
```

The benefits of using this approach might not be apparent now but one can appreciate them when including plot in papers using LaTeX. For example this lab manual is written in LaTeX and figure 2 was created using the standard MATLAB exporting mechanism while figure 5 was created using `exportfig`.

Notice that in figure 5 we managed to preserve the aspect ratio of the plot and that the line widths and font sizes are not scaled down. Also notice that the colors have been automatically substituted by different line types to make them legible in black and white. Lastly using the `'width'` parameter we were able to specify the exact width of the figure to 8.5cm which is the standard width for a two-column conference paper.

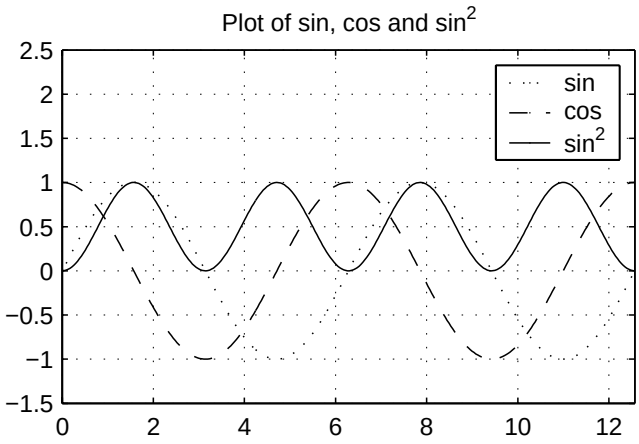

Figure 5: Example using the `expfig` function

5 Elementary signals

5.1 Unit impulse, unit step and ramp functions

The theory presented at the Signals & Systems course, examines the behavior of continuous-time signals and systems. MATLAB can use special, so-called *symbolic* variables to represent and manipulate continuous signals and systems. Their real-life MATLAB use though is limited and in fact another computing package called Mathematica is better suited for that purpose. Instead of using symbolic representation we will now use discretized versions.

Using the vector notation and the plotting commands we introduced it is now easy to create and visualize signals. In fact we have already done that when we created sines and cosines for our plotting examples. Now we will write three simple functions to generate the impulse, unit step and ramp signals.

To Do 3

Elementary signal functions.
Write and save in m-files under the `/lab2` directory the following three functions (remember that the m-file should have the same name as the function definition plus the extension `.m`):

The unit *impulse* function is defined as[2]:

$$ui[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

```
function y = ui(n)
%UI Unit impulse
```

[2]Note that there is no discretized version of the dirac delta function as delta is not defined at $t = 0$

```
% y = ui(n)
%   n:   time index
%   y:   signal

y = (n == 0);
```

The unit *step* function is defined as:

$$us[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

```
function y = us(n)
%US Unit step
% y = us(n)
%   n:   time index
%   y:   signal

y = (n >= 0);
```

And the unit *ramp* function is defined as:

$$ur[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

```
function y = ur(n)
%UR Unit ramp
% y = ur(n)
%   n:   time index
%   y:   signal

n(n<0) = 0;
y = n;
```

First try to understand how each function is implemented. Each function is essentially one vectorized command. Now it is easy to create other signals using the functions we just wrote. For example:

```
>> t = -5:0.001:5;
>> y = us(-t+2) .* ur(t);
>> plot(t,y)
>> grid on
>> axis([-5 5 -.5 2.5])
>> set(gcf,'paperunits','centimeters');
>> exportfig(gcf,'elem1.eps','bounds','tight','width',8.5,'linestylemap','bw');
```

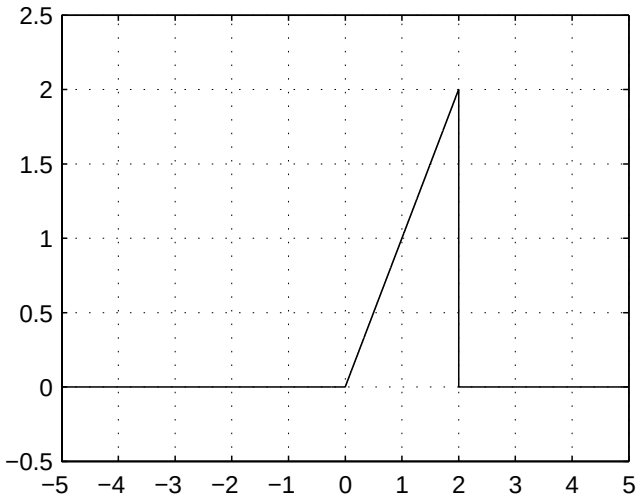

Figure 6: A signal created using the unit step and unit ramp functions

gives the plot in figure 6. Using our functions it's now easy to visualize time shifting and time reversal as the matlab command `y = us(-t+2) .* ur(t);` we used above is very intuitive.

Finally following the same procedure write the code that recreates the figures 1.16(a)/p68, 1.18/p69 in Lathi's book. Submit the code and the corresponding plots in `.eps` format.

To Do 4

Average length of ones.

Now we will write a function which will test our knowledge on vector manipulation. Assuming a random sequence of zeros and ones we should calculate the average length of contiguous ones. For example given a vector:

```
>> a = [0,0,0,0,1,1,0,1,0,0];
```

our function should be named `avgones` and give the following result:

```
>> avgones(a)
ans =
    1.5000
```

As we see the vector has two consecutive ones at indices 5,6. This segment thus has length 2. It also has a one at index 8 which obviously has length 1. This means that the average length of consecutive ones in the vector `a` is $(2+1)/2 = 1.5$.

The trick here is to use the `diff` and `find` commands. Obviously the cases of a vector full of ones should give as result the length of the vector and the case of a vector full of zeros

should give as result zero. Lastly notice that using the same function we can calculate the average length of zeros using the command:

```
>> a = [0,0,0,0,1,1,0,1,0,0];
>> avgones(~a)
ans =
    2.3333
>> (4+1+2)/3
ans =
    2.3333
```

Save the function using the filename `avgones.m` in the directory `/lab2` and submit it.

At this point you should turn off the diary function we started in the beginning of this lab session. You can do that by typing:

```
>> diary off;
```

This will create the diary file that contains all the commands you typed in the command prompt as well as all the output they generated. You should show the LA or TA the diary file.