

Experiment 4: Representation of signals and computing the signal energy.

1. Editing files

MATLAB offers the possibility to store commands in special files - called M-files – with the ending .m. There are two types of M-files: Scripts and Functions. To edit an M-file, select ‘New’ or ‘Open’ from the file-menu. In each case the MATLAB Editor window will be opened and you can continue by editing either a new file or, after opening, an existing file. When the MATLAB Editor is open, you can load further files or save and close opened files by selecting the appropriate item from the file menu of the Editor. Select ‘Save As’ and the browser if you save a new file or if you want to change the name of an existing file, or ‘Save’ if you have edited an existing file whose name you want to leave unchanged. The directory in which the files should be saved is usually the working directory.

1.1 Script M-files

In a Script M-file you just write a sequence of commands separated by semicolons (result not displayed) or commas (result displayed). A Script M-file is treated as if the commands were written in the Command window. All variables defined in the workspace are accessed and the computations are carried out in the workspace.

1.2 Function M-files

These files differ from Script M-files in so far that the commands are not executed in the present workspace, but in a separate workspace which is created whenever the function is called. As a consequence, you have to transfer variables from your workspace to the function workspace and specify what has to be returned, i.e. you have to specify input and output arguments.

The first (command) line of a Function M-file MUST be a function declaration line specifying input and output variables. Below this line any sequence of MATLAB commands can appear, in which the inputs are processed and the outputs should be defined. You have to make sure that no undefined variable occurs as input to these commands!

You can use Function M-files for programming in MATLAB as well as in the mathematical sense, i.e. for numerical implementation of formulae.

Functions defined by m files start with the line

Function [assignment_variable(s)] = function_name(arguments)

- function : is a required Matlab particle to identify the file as a function,
- assignment_variable(s) : is an optional dummy variable,
- function_name : is the name given to the function,
- arguments: are optional values passed on to the function from the main Matlab interface or from within another function.

Example: Let's start with a simple example, the (mathematical) function $y(x)=x^2$. To implement this function through a Function M-file, simply type

```
function y=f(x)

y=x.^2;
```

in a file and save the file as **square.m**. In the workspace (or a Script M-file) you can then evaluate the function by

```
» square(2.1)
ans =
    4.4100
```

or by square(varname) if varname is assigned to a numerical value. By writing .^ instead of just ^, the function is "array smart", i.e. one can pass arrays as argument as well. For example,

```
» square([1 2 3])
ans =
     1     4     9
```

returns the array consisting of the squares of the the input array [1 2 3].

Example :

```
function [x,y] = car2pol(r,theta);
% Conversion from polar to Cartesian coordinates
x = r.*cos(theta);
y = r.*sin(theta);
```

From Lathi)MATLAB SESSION 1: Working with functions

Working with functions is fundamental to signals and systems applications. MATLAB provides several methods of defining and evaluating functions. An understanding and proficient use of these methods is therefore necessary and beneficial.

1. Inline Function

Many simple functions are most conveniently represented by using MATLAB inline objects . An inline object provides a symbolic representation of a function defined in terms of MATLAB operator and functions. For example, consider defining the exponentially damped sinusoid

$$f(t) = e^{-t}\cos(2\pi t)$$

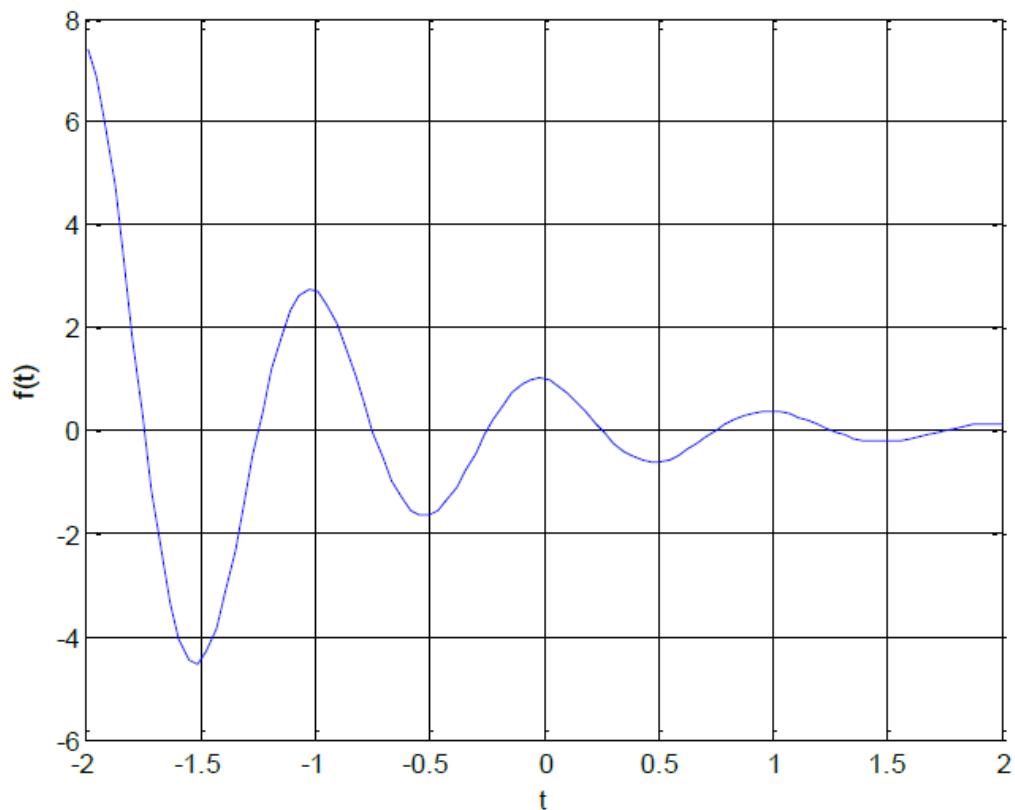
```
f = inline('exp(-t).*cos(2*pi*t)','t')
f =
Inline function:
f(t) = exp(-t).*cos(2*pi*t)
```

The second argument to the *inline* command identifies the function's input argument as t . Input argument such as t , are local to the inline object and are not related to any workspace variables with the same names.

Once defined, $f(t)$ can be evaluated simply by passing the input values of interest, for example

```
>> t = 0;  
f(t)  
ans =  
    1  
>> f(0)  
ans =  
    1  
>> t=(-2:2)  
t =  
   -2   -1    0    1    2  
>> f(t)  
ans =  
    7.3891    2.7183    1.0000    0.3679    0.1353
```

```
t=linspace(-2,2);  
plot(t,f(t))  
xlabel('t'); ylabel('f(t)'); grid;
```



2. Relational Operators and the Unit Step Function

The unit step function $u(t)$ arises naturally in many practical situations. For example, a unit step can model the act of turning on a system. With the help of relational operators, inline objects can represent unit step function.

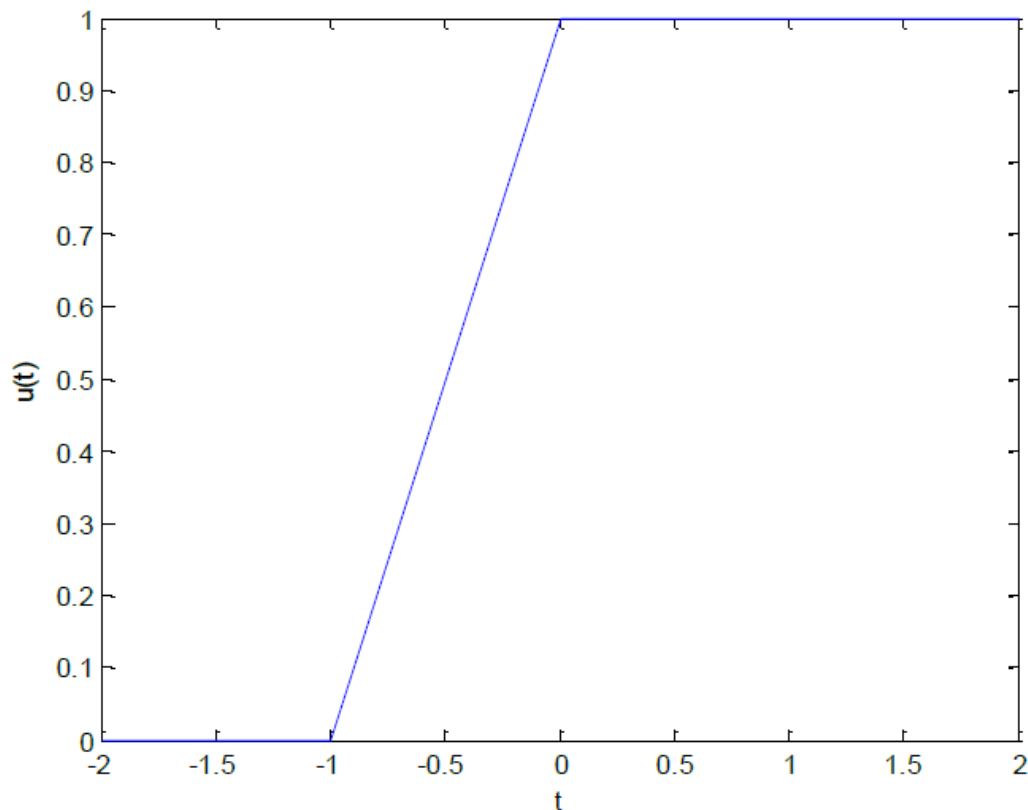
The unit step function is readily defined using the $\geq$ relational operator.

```
>> u = inline('t>=0','t')  
u =  
Inline function:  
u(t) = (t>=0)
```

Any function with a jump discontinuity, such as the unit step, is difficult to plot. Consider

Plotting $u(t)$ by using $t = (-2:2)$

```
t = (-2:2);  
u = (t>=0);  
plot(t,u);  
xlabel('t'); ylabel('u(t)');
```

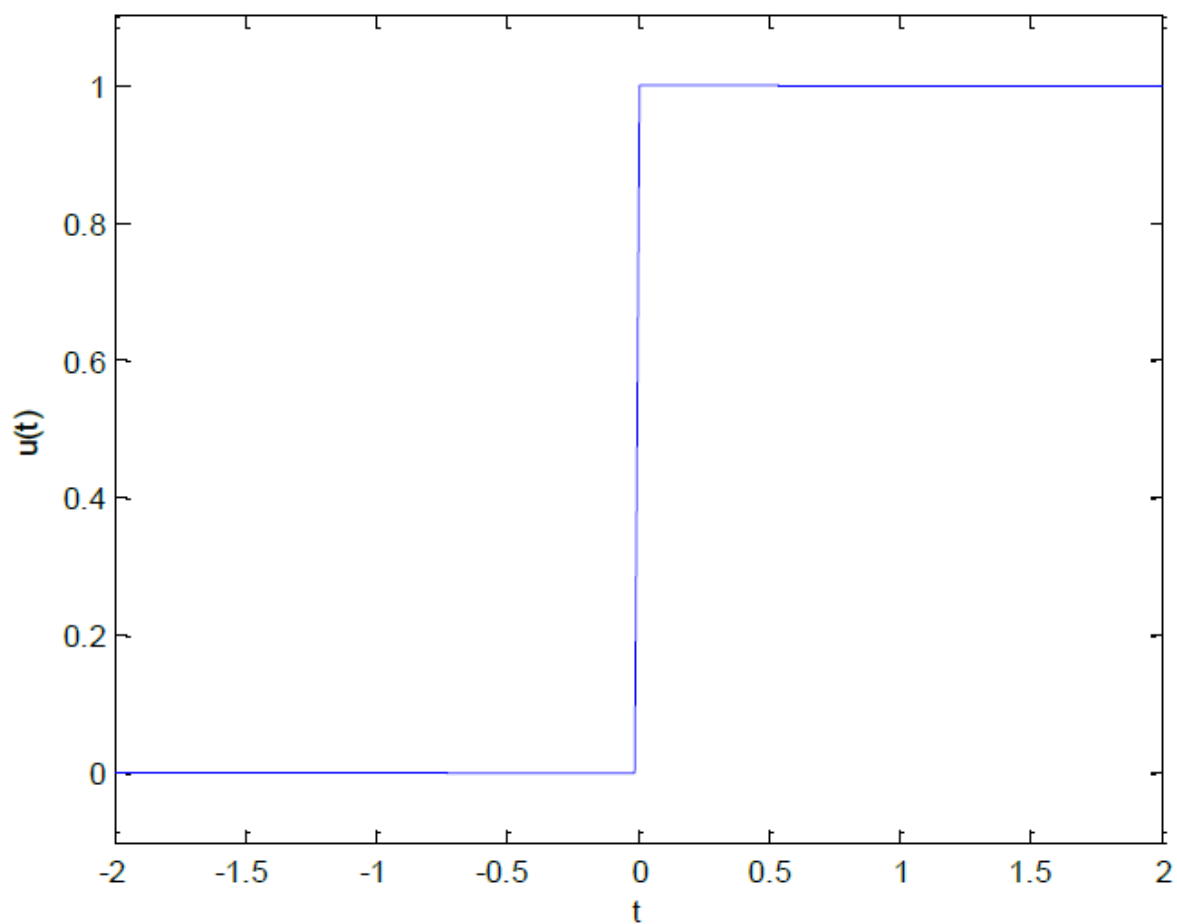


Two significant problems are apparent in the resulting plot, shown in Fig2. First MATLAB automatically scales plot axes to tightly bound the data. Second, MATLAB connect plot data with lines ,making a true jump discontinuity difficult to achieve .

```
t = (-2:0.01:2);% solve the second problem  
u = (t>=0);  
plot(t,u);  
xlabel('t'); ylabel('u(t)');
```

```
axis([-2 2 -0.1 1.1]);%solve the first problem
```

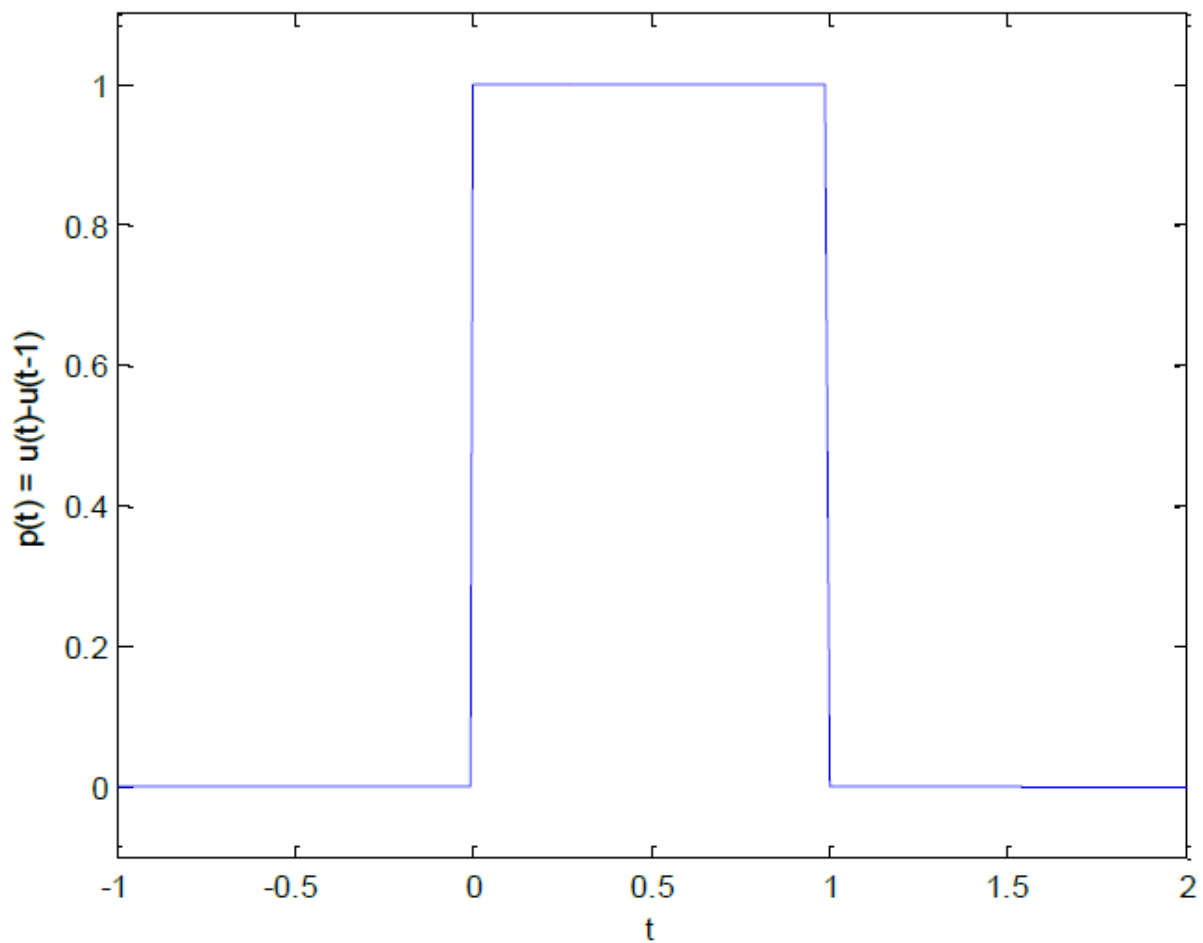
axis([x-axis min x-axis max y-axis min y-axis max])



Relational operators can be combined using logical AND, logical OR, and logical negation. For example

$$p(t) = u(t) - u(t - 1),$$

```
t = (-1:0.01:2);  
p = ((t>=0)&(t<1)); % p = (~((t<=0)|(t>=1)))  
plot(t,p);  
xlabel('t'); ylabel('p(t) = u(t)-u(t-1)');  
axis([-1 2 -0.1 1.1]);
```



3. Impulse function - Dirac Delta

The 'Impulse function' is also known as the 'Dirac delta' function, or δ function (it was introduced by Paul Dirac). In the context of digital signal processing (DSP) it's often referred to as the 'unit impulse function'.

It's defined as follows:

$$\delta(n) = \begin{cases} \infty & \text{if } n = 0 \\ 0 & \text{if } n \neq 0 \end{cases}$$

and has to satisfy the identity:

$$\int_{-\infty}^{\infty} \delta(n) dn = 1$$

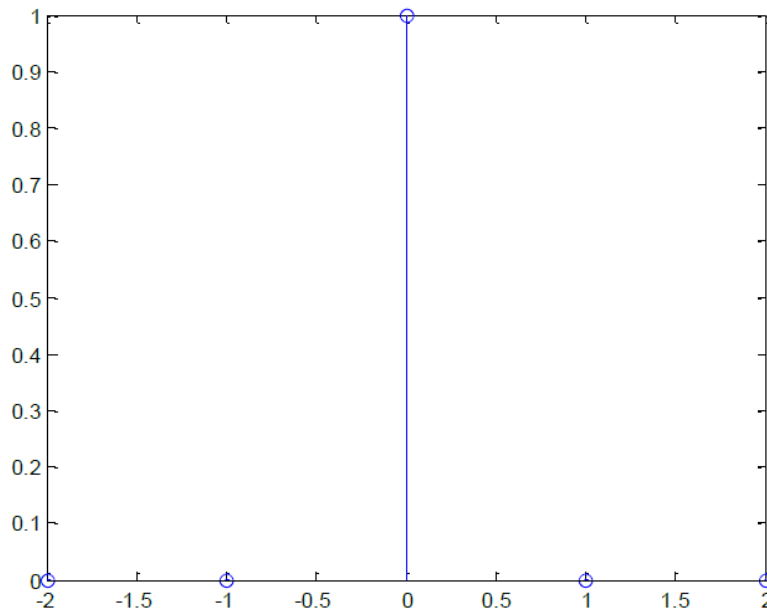
The function depends on real input parameters. The function output is infinite when the input is exactly 0. The output is zero for any other input value.

The Dirac delta is not strictly a function, because any real function that is equal to zero everywhere but at a single point must have a total integral equal to zero, but for many purposes this definition can be manipulated as a function.

Let's create some discrete plots using Matlab function 'stem'.

For our purposes, we are going to define the function as 1 when the argument of the Dirac function is 0, and the output will be 0 for any other value of the input argument.

```
delta = inline('n == 0');  
n = -2:2;  
x = delta(n) ;  
stem(n, x)
```



Examples

dirac has the property that

$$\int_{-\infty}^{\infty} \text{dirac}(x - a) * f(x) = f(a)$$

for any function f and real number a . For example:

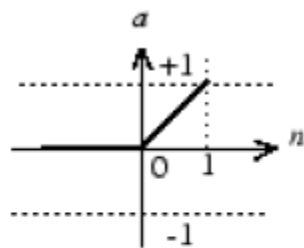
$$\int_{-\infty}^{\infty} \delta(x - 5) * \sin(x) dx$$

```
syms x a
a = 5;
int(dirac(x-a)*sin(x),-inf, inf)
ans =
sin(5)
```

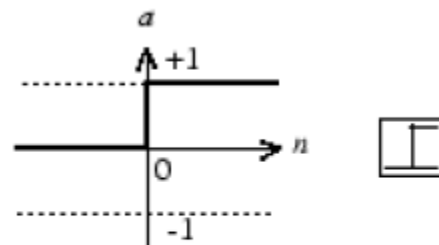
Note: the step and the ramp are ready implemented functions in Matlab

Hardlim (t) → step function

Poslin (t) → ramp function



$$a = \text{poslin}(n)$$



$$a = \text{hardlim}(n)$$

Positive Linear Transfer Function Hard-Limit Transfer Function