Matlab Laboratory 3. Program Control Statements in Matlab

Aim of the laboratory

- Understanding the Matlab computation engine
- Write simple Matlab code using program control statements

Necessary equipment

- Workstations with MATLAB

Theoretical Approach

A Matlab program consists basically of expressions and program control statements.

A Matlab expression consists of operands and operators. Matlab operands are either numeric structures, such as scalars and arrays, small expressions or functions. Each operand allows specific operators in order to write and evaluate expressions. One can distinguish between numeric and logical expressions in Matlab. Numeric expressions are those which evaluate to a numeric value. Logical expressions on the other hand evaluate to a logical value. Numeric expressions have been covered in the previous laboratory, so this laboratory will further on detail logical expressions in Matlab.

Logical expressions in Matlab

Logic expressions in Maltab consist of specific operands and operators and evaluate to a logical value. The operands of a logical expression can be numeric structures, such as scalars and arrays, small expressions or logic functions. In order to evaluate to a logical value, the operators of a logical expression are either logical operators or relational operators.

Logical operators in Matlab are:

- complement: *~operand*;
- logical AND: *operand1 && operand2*
- logical OR: *operand1 || operand2*

Evaluation of the logical operators listed above is performed according to the corresponding truth table. For reduced execution time however, one can use the element-wise logical operators. For example

```
operand1 & operand2
```

is evaluated form left to right. If *operand1* is equal to logical 0, the result of the AND operation is definitely logical ‘0’ and evaluation stops here. Evaluation is only continued if *operad1* is logical 1 and operand2 is required for evaluation of the overall AND function. The same is true for

```
operand1 | operand2
```

Relational operators in Matlab are:

- equality: *operand1 == operand2*;
- inequality: *operand1 ~= operand2*
- greater than: *operand1 > operand2*
- greater than or equal: *operand1 >= operand2*
- smaller than: *operand1< operand2*
- smaller than or equal: *operand1 <= operand2*

Next, the Matlab program control statements are very similar to those from other programming languages. The statements further studied in this laboratory are **for**, **while**, **if** and **switch**.

The for statement

The **for** statement executes a program section for a specified number of times [Matlab help]. The repetitive execution of the program section is called a loop.

The generic syntax of the **for** loop is

```
for index = values
   statements
          :
end
```

where *index* is the counter of the loop, *values* is the range where *index* takes values from, and *statements* is the program section which is repeatedly executed within the loop.

The **for** statement allows three approaches to define the loop range:

1. *values* is defined as an array; in this case *index* takes the values of the elements within the array, as for example

```
index = [1 3 6 8 9];
```

2. *values* is defined as an interval; in this case *index* is varied from an initial value to an end value with an implicit unitary increment, as for example

```
index = 1:10;
```

3. *values* is defined as an interval with a specified increment; in this case *index* is varied from an initial value to an end value with the given increment, as for example

```
index = 2:2:20;
```

For exemplification, consider the following code sequence to generate the first 10 Fibonacci numbers.

```
fib = ones(1,10);      % Preallocation of a row vector with 10 elements.
                       % Unity elements are considered because fub(1)=fib(2)=1

for i = 3:10           % Fibbonacci elements are computed starting with the
                       % 3rd index
    fib(i) = fib(i-2) + fib(i-1);
end;

fib
```

This code sequence will return

```
fib =

     1     1     2     3     5     8    13    21    34    55
```

As a rule of thumb, avoid using the index variable within the program statements, as changing the loop index within the program statements will override the loop counter. On the other hand, to change loop indexing on purpose, one can use the following instructions [Matlab help]:

- **break** – terminates the execution of the **for** loop,
- **return** – terminates execution of the **for** loop and returns to the invoking function,
- **continue –** skips one iteration of the **for** loop and passes control to the next iteration.

To be noticed is that, in order to skip several iterations of the **for** loop, one must increment the loop index accordingly within the program statements and then write the **continue** instruction.

The while statement

The **while** statement executes a program section repeatedly as long as a Boolean condition is true [Matlab help]. The repetitive execution of the program section is called a loop.

The generic syntax of the **while** loop is

```
while expression
   statements
          :
end
```

The logical state of *expression* is evaluated at every iteration. The **while** loop is entered provided the logical state is 1 and the *statements* are executed. The **while** loop is exited once the logical state of expression becomes 0.

Interference with the **while** loop is similar to the **for** loop, and can be done using the **break**, **return** and **continue** instructions.

For exemplification, consider the next code sequence which generates the first 10 Fibonacci numbers using this time the **while** statement.

```
fib = ones(1,10);      % Preallocation of a row vector with 10 elements.
                       % Unity elements are considered because fub(1)=fib(2)=1
i = 3;                 % Fibbonacci elements are computed starting with the
                       % 3rd index
while i <= 10
    fib(i) = fib(i-2) + fib(i-1);
    i = i + 1;         % increment counter
end;

fib
```

As in the previous example, this code sequence will return

```
fib =

     1     1     2     3     5     8    13    21    34    55
```

The if statement

The **if** statement executes a program section provided a Boolean condition is true [Matlab help]. The generic syntax of the **if** statement is

```
if expression1
    statements1
[elseif expression2
    statements2]
[else
    statements3]
end
```

In the simplest form of the **if** statement, namely

```
if expression1
    statements1
end
```

Matlab evaluates the logical state of *expression1*. Then, the program section *statements1* is executed only if the evaluation result is logical 1. Otherwise, Matlab runs nothing. Alternatively, to run another program section if the evaluation result is logical 0, one must consider an **else** branch as in

```
if expression1
    statements1
else
    statements2
end
```

In this case, Matlab takes some action for either logical value of expression 1. Additionally, one can consider several conditions within the same **if** statement, namely

```
if expression1
    statements1
elseif expression2
    statements2
elseif expression3
    statements3
elseif expression4
    statements4
       :
[else
    statements5]
end
```

In this case, Matlab runs *statements1* provided *expression1* is true. Alternatively, Matlab runs *statements2* provided *expression2* is true, and so on. Finally, if an **else** branch is present, statements 5 will be run only if neither logical expression is evaluated to a logical 1.

For exemplification of the **if** statement, consider the implementation of the numeric comparator from figure 1, which takes as input two real scalars a and b and outputs the relation between the two scalars as follows:

- $q_{a<b}$ is logical 1 if $a < b$,
- $q_{a=b}$ is logical 1 if $a = b$,
- $q_{a>b}$ is logical 1 if $a > b$,

the remainder outputs being logical 0 respectively.

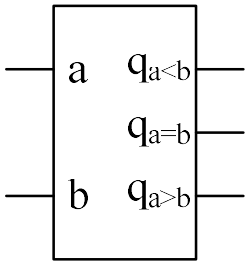

Figure 1.

The Matlab function which implements the numeric comparator is listed as follows.

```
function [q_greater,q_equal,q_smaller] = numcomp(a,b)

if a > b
    q_greater = 1;
    q_equal = 0;
    q_smaller = 0;
elseif a == b
    q_greater = 0;
    q_equal = 1;
```

```
    q_smaller = 0;
else
    q_greater = 0;
    q_equal = 0;
    q_smaller = 1;
end;
```

Calling the function from the command line for two arbitrary scalars, for example a = 20 and b = = 13, will return

```
[q_greater,q_equal,q_smaller] = numcomp(20,12)

q_greater =

     1


q_equal =

     0


q_smaller =

     0
```

The switch statement

The **switch** statement selects among several cases and executes a specific program section depending on the value of an expression [Matlab help]. The generic syntax of the **switch** statement is

```
switch expression
  case scalar1
    statements1
  case {scalar2, scalar3, ...}
    statements2
  otherwise
    statements3
end
```

The **switch** statement evaluates the scalar value of *expression* and compares the evaluation result to the case branches. If the value of *expression* is equal to scalar1 Matlab will execute *statements1*. Otherwise, if the value of *expression* is equal to either scalar from the array {scalar2, scalar3, …} Matlab will execute *statements2*. If none of the above is met Matlab will execute *statements3* from the default branch.

There are two main differences between the **if** and **switch** statements. While the expression of the **if** statement must evaluate to a Boolean value, the expression of the **switch** statement evaluates to a scalar and consequently allows several branches. On the other hand, the **switch** statement takes

one single expression for evaluation, whereas the **if** statement may evaluate several independent expressions.

For exemplification of the **switch** statement, consider the following code sequence which calls the numeric comparator function defined in the previous example and displays to the command line the relationship between the two scalar a and b.

```
a = 15;
b = 21;

[q_greater,q_equal,q_smaller] = bincomp(a,b);

switch q_greater
    case 1
        disp('a > b');
    otherwise
        disp('a <= b');
end;
```

This code sequence will return

```
a <= b
```

Exercise 1. Plot the input and output signals of the DR circuit from figure 2.

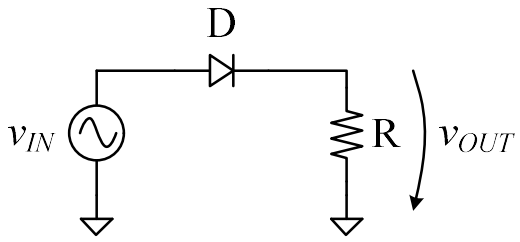

Figure 1.

For simplicity, let's consider an ideal diode model. Accordingly, a positive diode voltage drop brings the diode to conduction whereas a negative diode voltage drop blocks the diode. The output voltage is then expressed as

$$v_{OUT} = \begin{cases} v_{IN}, if\ v_{IN} \geq 0 \\ 0, if\ v_{IN} < 0 \end{cases}$$

To be noticed is that a sweep of the time domain is required in order to determine the output signal in each point along the time axis. For this purpose, the variation range of the **for** instruction is from origin, i.e. index 1 for $t = 0$, to the length of the time domain. The ***length()*** function is used to determine the index of the time domain limit.

The Matlab script which plots the input and output signals of the DR circuit is listed as follows

```
clear all;
close all;
clc;

A = 2;
f = 50;

per = 1/f;
t = 0:per/100:2*per;

vin = A * sin(2*pi*f*t);
vout = zeros(size(vin));  %prealocation of vout with same dimensions as vin

for i = 1:length(t)         %sweep range
    if vin(i) >= 0          %diode is in conduction
        vout(i) = vin(i);
    end;
end;

figure();
subplot(2,1,1);
plot(t,vin);
grid on;
subplot(2,1,2);
plot(t,vout);
grid on;
```

The input and output voltage of the DR circuit are plotted in figure 3.

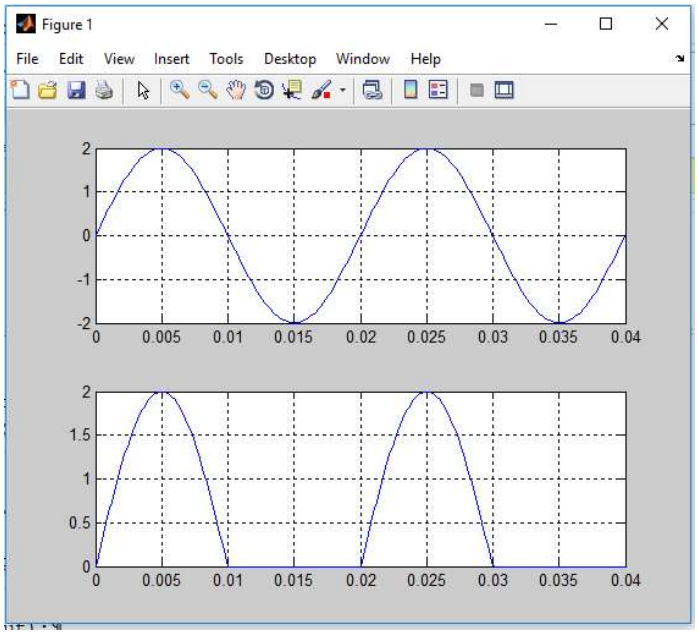

Figure 3.

To do…

1. Study the ***length*** function using the MATLAB help. Why was the ***length*** function is used to define the loop range of the **for** instruction?
2. Repeat the exercise, considering this time a 0.7V constant voltage drop diode model. In this case, the expression of the output voltage changes to

$$v_{OUT} = \begin{cases} v_{IN} - 0.7, if\ v_{IN} \geq 0.7 \\ 0, if\ v_{IN} < 0.7 \end{cases}$$

3. Repeat the exercise considering this time a V_{Th} = 1V bias voltage source in series with the resistance, as illustrated in figure 4. In this case, the expression of the output voltage changes to

$$v_{OUT} = \begin{cases} v_{IN}, if\ v_{IN} \geq V_{Th} \\ V_{Th}, if\ v_{IN} < V_{Th} \end{cases}$$

4. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the input signal amplitude, frequency and phase respectively, as well as for the threshold voltage.

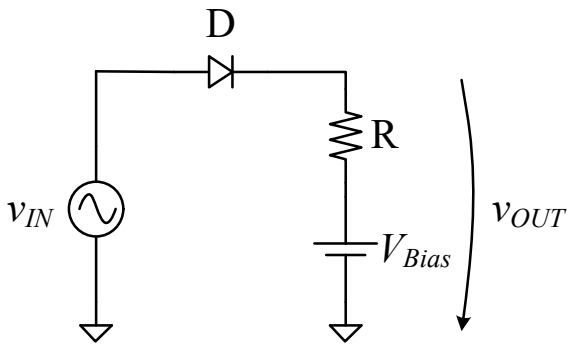

Figure 4.

Exercise 2. Plot the input and output signals of the spatial-maximum DR multiport network from figure 5.

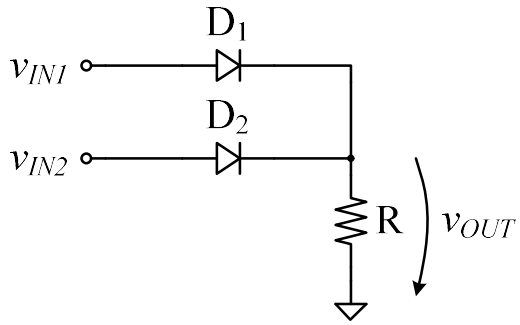

Figure 5.

For simplicity, let's consider an ideal diode model. The operation of the spatial-maximum DR multiport network is expressed by

$$v_{OUT} = \max(v_{IN}\ , v_{IN}\ , 0)$$

This expression can be detailed as

$$v_{OUT} = \begin{cases} v_{IN1}, if\ v_{IN1} > v_{IN2}\ and\ v_{IN}\ > 0 \\ v_{IN}\ , if\ v_{IN2} > v_{in}\ \ and\ v_{IN2} > 0 \\ 0, if\ v_{IN1} < 0\ and\ v_{IN2} < 0 \end{cases}$$

To do…

1. Write a Matlab script which plots the input and output signals of the spatial-maximum DR multiport network. Consider as input signals two sine waves with different frequencies and amplitudes.
2. Repeat the exercise considering a triangle wave and a sine wave as input signals respectively.
3. Repeat the exercise considering a rectangular wave and a sine wave as input signals respectively.
4. Repeat the exercise, considering this time a 0.7V constant voltage drop diode model.
5. Repeat the exercise considering this time a V_{Th} = 0.5V bias voltage source in series with the resistance, as illustrated in figure 4.
6. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the input signal amplitude, frequency and phase respectively, as well as for the threshold voltage.
7. Extend the DR multiport network, and consequently the Matlab code, in order to determine the spatial-maximum of multiple input signals.

Exercise 3. Plot the input and output signals of the non-inverting OpAMP comparator from figure 6.

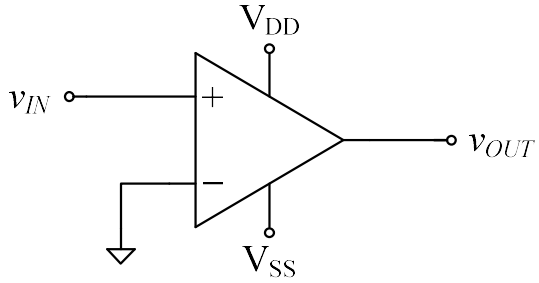

Figure 6

This output signal of the non-inverting OpAMP comparator is expressed as

$$v_{OUT} = \begin{cases} V_{DD}, if\ v_{IN} \geq 0 \\ V_{SS}, if\ v_{IN} < 0 \end{cases}$$

To do…

1. Write a Matlab script which plots the input and output signals of the non-inverting OpAMP comparator.
2. Repeat the exercise, considering this time a 3V voltage source applied to the OpAMP inverting input.
3. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the input signal amplitude, frequency and phase respectively, as well as for the DC voltage level.

Exercise 4. Write a Matlab function which determines the state of the LEDs in figure 7.

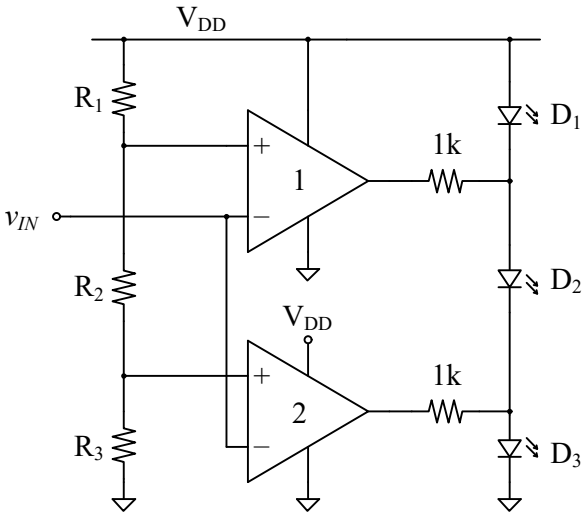

Figure 7

The input voltages to the first OpAMP are expressed as

$$\begin{cases} v_1^+ = V_{Th1} = \dfrac{R_2 + R_3}{R_1 + R_2 + R_3} \cdot V_{DD} \\ v_1^- = v_{IN} \end{cases}$$

The input voltages to the second OpAMP are expressed as

$$\begin{cases} v_2^+ = V_{Th2} = \dfrac{R_3}{R_1 + R_2 + R_3} \cdot V_{DD} \\ v_2^- = v_{IN} \end{cases}$$

Operation of the circuit from figure 7 is then detailed in table 1.

	v_{O1}	v_{O2}	D_1	D_2	D_3
$v_1^+ > v_2^+ > v_{IN}$	V_{DD}	V_{DD}	off	off	ON
$v_1^+ > v_{IN} > v_2^+$	V_{DD}	V_{SS}	off	ON	off
$v_{IN} > v_1^+ > v_2^+$	V_{SS}	V_{SS}	ON	off	off

To do…

1. Write a Matlab script which plots the input and output signals of the circuit in figure 7.
2. Depending in the state of the OpAMP output voltages, decide upon the state of the LEDs.

The second task assumes two steps. First, the OpAMP output voltages should be binarized:

```
vout1_bin = vout1 > ADC;
vout2_bin = vout2 > ADC;
```

Second, depending on the binary word at the OpAMP outputs, the LED states can be determined using the **switch** instruction.

```
for i = 1:length(t)
    switch 10*vout1_bin(i)+vout2_bin(i)
        case 11
            D1(i) = 0;
            D2(i) = 0;
            D3(i) = 1;
        case 10
            D1(i) = 0;
            D2(i) = 1;
            D3(i) = 0;
        case 0
            D1(i) = 1;
            D2(i) = 0;
            D3(i) = 0;
        otherwise
            D1(i) = 0;
            D2(i) = 0;
            D3(i) = 0;
    end;
end;
```

Exercise 5. Plot the input and output signals of the non-inverting hysteresis OpAMP comparator from figure 8.

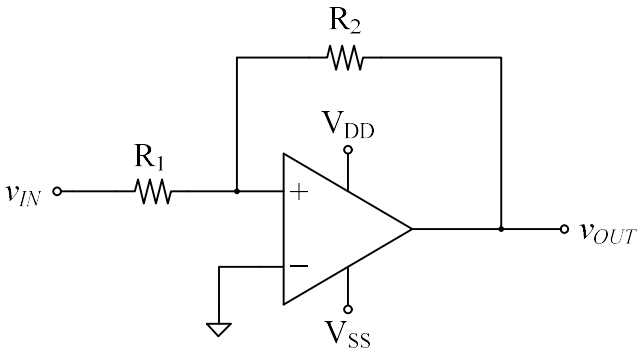

Figure 8

The hysteresis parameters are given by:

- high and low output voltage levels:

$$\begin{cases} V_{OH} = V_{DD} \\ V_{OL} = V_{SS} \end{cases}$$

- high and low threshold

$$\begin{cases} V_{ThH} = -\dfrac{R_1}{R_2} \cdot V_{OL} \\ V_{ThL} = -\dfrac{R_1}{R_2} \cdot V_{OH} \end{cases}$$

The voltage transfer characteristic of the non-inverting hysteresis OpAMP comparator is illustrated in figure 9.

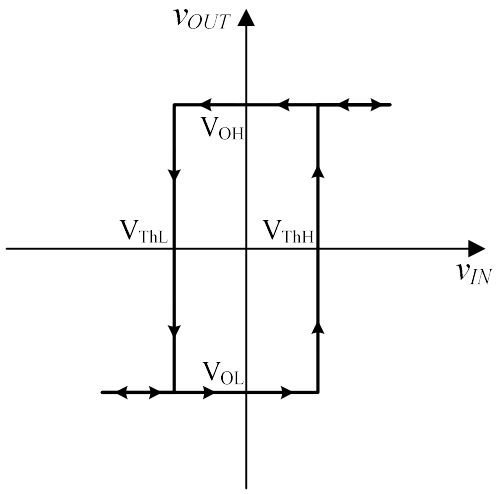

Figure 9

The comparator switches according to the voltage transfer characteristics as follows,

$$v_{OUT} = \begin{cases} V_{OH}, if\ v_{IN} > V_{ThH}\ and\ v_{IN} \uparrow \\ V_{OL}, if\ v_{IN} < V_{ThL}\ and\ v_{IN} \downarrow \end{cases}$$

expressing that the output switches to V_{OH} if the input voltage is increasing above the higher threshold value, whereas the output switches to V_{OL} if the input voltage is increasing above the lower threshold value. If none of the above is met, the output keeps value. The Matlab code which performs this operation is listed as follows:

```
if (vin(i)>vthH && vout(i-1)==voL)
    vout(i) = voH;
elseif (vin(i)<vthL && vout(i-1)==voH)
    vout(i) = voL;
else vout(i) = vout(i-1);
end;
```

To be noticed is that the **if** instruction listed above assumes a known value of the output signal in the previous sample. Accordingly, the time domain can only be swept from index 2 to the limit of the time domain. Initialization of the initial output voltage sample is then required.

To do…

1. Write a Matlab script which plots the input and output signals of the non-inverting hysteresis OpAMP comparator.
2. Repeat the exercise, considering this time a 3V voltage source applied to the OpAMP inverting input.
3. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the input signal amplitude, frequency and phase respectively, as well as for the DC voltage level.

Exercise 6. Plot the input and output signals of the inverting OpAMP amplifier from figure 10.

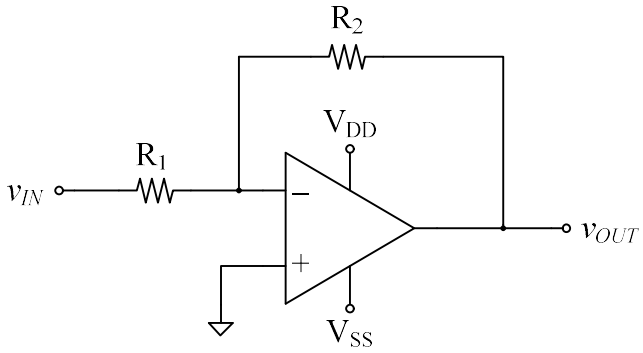

Figure 10.

This output signal of the inverting OpAMP amplifier is expressed as

$$v_{OUT} = A_V \cdot v_{IN}$$

where the amplifier voltage gain A_V is expressed as

$$A_V = -\frac{R_2}{R_1}$$

To do…

1. Write a Matlab script which plots the input and output signals of the inverting OpAMP amplifier.
2. Repeat the exercise, considering this time a 3V voltage source applied to the OpAMP non-inverting input.
3. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the input signal amplitude, frequency and phase respectively, as well as for the DC voltage level.

Exercise 7. Write a Matlab function which implements a numeric comparator with an enable input. The aim of the enable input is to validate the operation of the numeric comparator as long as an enable signal is logical 1. Thus, the *bincomp* function presented in the introduction must be changed accordingly.

Validation of the numeric comparator is performed within a **while** loop as follows:

```
i = 1;
while en(i)
    if a(i) > b(i)
        q_greater(i) = 1;
        q_equal(i) = 0;
        q_smaller(i) = 0;
    elseif a(i) == b(i)
        q_greater(i) = 0;
        q_equal(i) = 1;
        q_smaller(i) = 0;
    else
        q_greater(i) = 0;
        q_equal(i) = 0;
        q_smaller(i) = 1;
    end;
    i = i+1;
end;
```

The *bincomp* function is then called from a Matlab script:

```
a = [14 1 3 6 13 19];
b = [9 3 3 7 10 11];
en = [1 1 1 0 0 0];

[q_greater,q_equal,q_smaller] = bincomp(a,b,en);

subplot(5,1,1);
plot(a);
subplot(5,1,2);
plot(b);
subplot(5,1,3);
plot(q_greater);
subplot(5,1,4);
plot(q_equal);
subplot(5,1,5);
plot(q_smaller);
```

To do…

1. Write a Matlab script which plots the input and output signals of the numeric comparator.
2. Using the ***uicontrol*** function, extend the Matlab code in order to add control fields for the numeric input signals and the binary enable signal.