Programming with SCILAB

By Gilberto E. Urroz, September 2002

What is programming?

Programming the computer is the art of producing code in a standard computer language, such as Fortran, BASIC, Java, etc., to control the computer for calculations, data processing, or controlling a process. For details on programming structures, use of flowcharts and pseudo code, and some simple coding in SCILAB and Visual Basic, see document [16] in the References section of this document.

What do I need to program the computer?

To type code you need an editor, which is simply a program where you can edit text files. Some modern computer programs such as Visual Basic include the editor within an Integrated Development Environment that allows the user not only the editor for typing code, but also access and control over visual components of the program. If using SCILAB, or Fortran g77, you can use the free editor PFE (see reference [3]) to type code.

Once the code is typed and the corresponding file saved, you need to have either a compiler (as in the case of Fortran g77) or an interpreter as in the case of Visual Basic (although, compilation is also allowed). A compiler translates the contents of a program's code into machine language, thus making the operation of the program faster than interpreting it. An interpreter collects as much code as necessary to activate an operation and performs that portion of the code before seeking additional code in the source file. SCILAB functions are compiled within the SCILAB environment as soon as you load them with *getf*, but cannot be run separated from it as are Visual Basic compiled programs.

To produce code that works, you need, of course, to know the basic language syntax (see references [1], [2], [4]). If using SCILAB, you need to be familiar with the materials from, at least, Chapters 1 through 5 in the textbook *"Numerical and Statistical Methods with SCILAB for Science and Engineering*," i.e., references [1] and [2]. (The material for Chapter 1 corresponding to SCILAB version 2.6 is available in reference [4]). You also need some knowledge of SCILAB elementary functions (see reference [7]), of basic program structures (`if … then … else … end`, `for … end`, `while … end`, `case … select`, etc.), of string manipulation (see Chapter 2 in reference [1]), and input and output (see reference [9]). Knowledge of the materials from Chapter 4 (vectors) and, more importantly, Chapter 5 (matrices), both in reference [1], is imperative when programming with SCILAB since the numerical operation of the software is based on matrix manipulation.

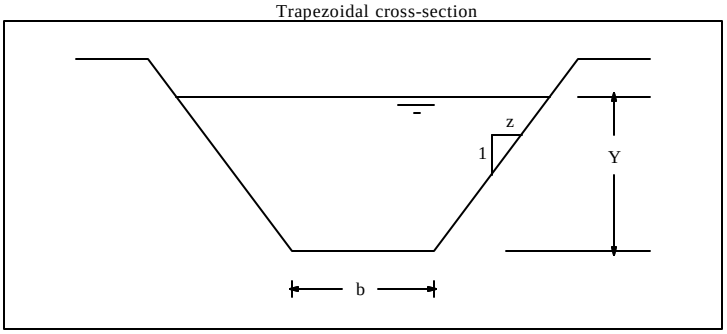
Scripts and functions

For details on the differences between scripts and functions in SCILAB see reference [6]. A script is a series of interactive SCILAB commands listed together in a file. Script files are typically given the suffix *.sce*, and are executed with the command *exec*. Scripts are not programs properly. SCILAB functions, on the other hand, can are self-contained programs that require arguments (input variables) and, many a time, an assignment to output variables. Some SCILAB functions can be called from other SCILAB functions (thus acting like subroutine or function subprograms in Fortran, or Sub or Function procedures in Visual Basic). Function files are typically given the suffix *.sci*, and are loaded into SCILAB (i.e., compiled) through the command *getf*. To call a function, we use the name of the function followed by either an empty set of

parentheses or parentheses containing the function arguments in the proper order. The function call can be done with or without assignment. Many examples of user-defined functions (as opposite to those already loaded with SCILAB, e.g., *sin*, *exp*) are presented through the SCILAB textbook (references [1] and [2]).

Examples of simple SCILAB functions

Refer to reference [6] for details about the basic structure of a function. Herein we present an example of a simple function that calculates the area of a trapezoidal cross-section in an open channel, given the bottom width, *b*, the side slope *z*, and the flow depth *y* (see the figure below).




```
function [A] = Area(b,y,z)
//|-----------------------------------------|
//| Calculation of the area of a trapezoidal    |
//| open-channel cross-section.                 |
//|                                             |
//| Variables used:                             |
//| ===============                             |
//|      b = bottom width (L)                   |
//|      y = flow depth   (L)                   |
//|      z = side slope (zH:1V, dimensionless)  |
//|      A = area (L*L, optional in lhs)        |
//|-----------------------------------------|
A = (b+z*y)*y
```

The file function turns out to have 13 lines, however, it contains only one operational statement, `A = (b+z*y)*y`. The remaining 12 lines are a fancy set of comments describing the function. A function like this could be easily defined as a SCILAB inline function through the command *deff*, e.g.,

```
deff('[A]=Area(b,y,z)','A = (b+z*y)*y')
```

An inline function becomes active as soon as it is defined with a *deff* command. Use inline functions only for relatively simple expressions, such as the case of function *Area*, shown above. An example of application of the function *Area* is shown next:

```
-->b = 2; z = 1.5; y = 0.75; A = Area(b,y,z)
 A  = 2.34375
```

If the inputs to the function will be vectors and you expect the function to return a vector of values, make sure that operations such as multiplication and division are term-by-term operations (`.*`, `./`), e.g.,

```
deff('[A]=Area(b,y,z)','A = (b+z.*y).*y')
```

An application of this function with vector arguments is shown next:

```
-->b = [1, 2, 3]; z = [0.5, 1.0, 1.5]; y = [0.25, 0.50, 0.75];
-->A = Area(b,y,z)
 A  =
!    .28125    1.25    3.09375 !
```

An inline function can return more than one result, as illustrated next with a function that calculates the Cartesian coordinates (*x,y,z*) corresponding to the Cylindrical coordinates (r,q,f):

```
-->deff('[x,y,z]=CC(rho,theta,phi)',['x=rho.*sin(phi).*cos(theta)', ...
-->     'y=rho.*sin(phi).*sin(theta)','z=rho.*cos(phi)'])
```

Notice the use of the continuation characters (...) to avoid typing an extra long line when using the command *deff*. Also, notice that since three different expressions are required to define the three results (*x,y,z*) in the function, the function statements are enclosed in brackets. Here is an example of function *CC* applied to the cylindrical coordinates $r = 2.5$, $q = p/6$, and $f = p/8$.

```
-->[x,y,z] = CC(2.5,%pi/6,%pi/8)
 z  =  2.3096988
 y  =    .4783543
 x  =    .8285339
```

If no assignment is made to a vector of three elements, the result given by *CC* is only one value, that belonging to *x*, e.g.,

```
-->CC(2.5,%pi/6,%pi/8)
 ans  =  .8285339
```

Notice that in the definition of the function term-by-term multiplications were provided. Thus, we could call the function using vector arguments to generate a vector result, e.g.,

```
-->[x,y,z] = CC([1,2,3],[%pi/12,%pi/6,%pi/3],[%pi/10,%pi/20,%pi/30])
 z  =   !    .9510565    1.9753767    2.9835657 !
 y  =   !    .0799795     .1564345     .2715729 !
 x  =   !    .2984875     .2709524     .1567927 !
```

Function *CC*, defined above, could also be put together as a file function, e.g.,

```
function [x,y,z] = CC(rho,theta,phi)
//|--------------------------------------|
//|This function converts the cylindrical |
//|coordinates (rho, theta, phi) into the |
//|Cartesian coordinates (x,y,z).         |
//|--------------------------------------|
//Check consistency of vector entries
nr = length(rho)
nt = length(theta)
np = length(phi)
```

```
if nr <> nt | nr <> np | nt <> np then
   error("Function CC - vectors rho, theta, phi have incompatible
dimensions")
   abort
end

//Calculate coordinates if vector lengths are consistent
x=rho.*sin(phi).*cos(theta)
y=rho.*sin(phi).*sin(theta)
z=rho.*cos(phi)'
```

Consider now a vector function $\mathbf{f}(\mathbf{x}) = [f_1(x_1,x_2,x_3)\ f_2(x_1,x_2,x_3)\ f_3(x_1,x_2,x_3)]^T$, where $\mathbf{x} = [x_1,x_2,x_3]^T$ (The symbol $[]^T$ indicates the transpose of a matrix). Specifically,

$$f_1(x_1,x_2,x_3) = x_1\ cos(x_2) + x_2\ cos(x_1) + x_3$$

$$f_2(x_1,x_2,x_3) = x_1x_2 + x_2x_3 + x_3x_1$$

$$f_3(x_1,x_2,x_3) = x_1^2 + 2x_1x_2x_3 + x_3^2$$

A function to evaluate the vector function $\mathbf{f}(\mathbf{x})$ is shown below.

```
function [y] = f(x)
//|----------------------------------|
//| This function calculates a vector |
//| function f = [f1;f2;f3] in which  |
//| f1,f2,f3 = functions of (x1,x2,x3)|
//|----------------------------------|
y = zeros(3,1)  //create output vector
y(1) = x(1)*cos(x(2))+x(2)*cos(x(1))+x(3)
y(2) = x(1)*x(2) + x(2)*x(3) + x(3)*x(1)
y(3) = x(1)^2 + 2*x(1)*x(2)*x(3) + x(3)^2
```

Here is an application of the function:

```
-->x = [1;2;3]
 x  =

!   1. !
!   2. !
!   3. !

-->f(x)
 ans  =

!   3.6644578 !
!   11.       !
!   22.       !
```

Function *f*, shown above, can be written as an inline function as follows:

```
-->deff('[y]=f(x)',['y=zeros(3,1)', ...
-->'y(1) = x(1)*cos(x(2))+x(2)*cos(x(1))+x(3)', ...
-->'y(2) = x(1)*x(2) + x(2)*x(3) + x(3)*x(1)', ...
-->'y(3) = x(1)^2 + 2*x(1)*x(2)*x(3) + x(3)^2'])
```

You can check that `f(x)` will produce the same result as above.

A function can return a matrix, for example, the following function produces a 2x2 matrix, defined by

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} x_1 + x_2 & (x_1 + x_2)^2 \\ x_1 x_2 & \sqrt{x_1 + x_2} \end{bmatrix},$$

with

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
function [J] = JMatrix(x)
//|-----------------------------------|
//| This function returns a 2x2 matrix |
//| J = [f1, f2; f3, f4], with f1, f2, |
//| f3, f4 = functions of (x1,x2)      |
//|-----------------------------------|
J = zeros(2,2)
J(1,1) = x(1)+x(2)
J(1,2) = (x(1)+x(2))^2
J(2,1) = x(1)*x(2)
J(2,2) = sqrt(x(1)+x(2)
```

An application of function *JMatrix* is shown below:

```
-->x = [2;-1]
 x  =

!   2. !
! - 1. !

-->JMatrix(x)
 ans  =

!   1.    1. !
! - 2.    1. !
```

Examples of functions that include decisions

Functions including decision may use the structures `if … then … end`, `if … then … else … end`, or `if … then … elseif … then … else … end`, as well as the `case … select` structure. Some examples are shown below.

Consider, for example, a function that classifies a flow according to the values of its Reynolds (*Re*) and Mach (*Ma*) numbers, such that if *Re* < *2000*, the flow is laminar; if *2000* < *Re* < *5000*, the flow is transitional; if *Re* > *5000*, the flow is turbulent; if *Ma* < *1*, the flow is sub-sonic, if *Ma* = *1*, the flow is sonic; and, if *Ma* >*1*, the flow is super-sonic.

```
function [] = FlowClassification(Re,Ma)
//|-------------------------------------|
//| This function classifies a flow     |
//| according to the values of the      |
//| Reynolds (Re) and Mach (Ma).        |
//| Re <= 2000, laminar flow            |
//| 2000 < Re <= 5000, transitional flow|
//| Re > 5000, turbulent flow           |
//| Ma < 1, sub-sonic flow              |
//| Ma = 1, sonic flow                  |
//| Ma > 1, super-sonic flow            |
//|-------------------------------------|
//Classify according to Re
if Re <= 2000 then
   class = "The flow is laminar "
elseif Re > 2000 & Re <= 5000 then
   class = "The flow is transitional "
else
   class = "The flow is turbulent "
end
//Classify according to Ma
if Ma < 1 then
   class = class + "and sub-sonic."
elseif Ma == 1 then
   class = class + "and sonic."
else
   class = class + "and super-sonic."
end
//Print result of classification
disp(class)
```

Notice that the function statement has no variable assigned in the left-hand side. That is because this function simply produces a string specifying the flow classification and it does not returns a value. Examples of the application of this function are shown next:

```
-->FlowClassification(1000,0.5)
 The flow is laminar and sub-sonic.

-->FlowClassification(10000,1.0)
 The flow is turbulent and sonic.

-->FlowClassification(4000,2.1)
 The flow is transitional and super-sonic.
```

An alternative version of the function has no arguments. The input values are requested by the function itself through the use of the *input* statement. (Comment lines describing the function were removed to save printing space).

```
function [] = FlowClassification()
//Request the input values of Re and Ma
Re = input("Enter value of the Reynolds number: \n")
Ma = input("Enter value of the Mach number:\n")
//Classify according to Re
if Re <= 2000 then
   class = "The flow is laminar "
elseif Re > 2000 & Re <= 5000 then
   class = "The flow is transitional "
else
```

```
   class = "The flow is turbulent "
end
//Classify according to Ma
if Ma < 1 then
   class = class + "and sub-sonic."
elseif Ma == 1 then
   class = class + "and sonic."
else
   class = class + "and super-sonic."
end
//Print result of classification
disp(class)
```

An example of application of this function is shown next:

```
-->FlowClassification()
Enter value of the Reynolds number:
-->14000
Enter value of the Mach number:
-->0.25

 The flow is turbulent and sub-sonic.
```

The structures `if … then … end`, `if … then … else … end`, or `if … then … elseif … then … else … end` are useful in the evaluation of multiply defined functions. Refer to reference [13] for more details.

The following example presents a function that orders a vector in increasing order:

```
function [v] = MySort(u)
//|-------------------------------|
//|This function sorts vector u in |
//|increasing order, returning the |
//|sorted vector as v.             |
//|-------------------------------|
//Determine length of vector
n = length(u)
//Copy vector u to v
v = u
//Start sorting process
for i = 1:n-1
    for j = i+1:n
        if v(i)>v(j) then
           temp = v(i)
           v(i) = v(j)
           v(j) = temp
        end
     end
end
```

An application of this function is shown next:

```
-->u = int(10*rand(1,10))
 u  =

!   5.    4.    2.    6.    4.    9.    0.    4.    2.    4. !

-->MySort(u)
 ans  =
```

```
!   0.    2.    2.    4.    4.    4.    4.    5.    6.    9. !
```

Of course, you don't need to write your own function for sorting vectors of data since SCILAB already provides function *gsort* for this purpose, e.g.,

```
-->gsort(u,'g','i')
 ans  =

!   0.    2.    2.    4.    4.    4.    4.    5.    6.    9. !
```

However, a function like *MySort*, shown above, can be used for programming an increasing-order sort in other computer languages that do not operate based on matrices, e.g., Visual Basic or Fortran.

Notice that the function *gsort* can produce a decreasing order sort by using simply:

```
-->gsort(u)
 ans  =

!   9.    6.    5.    4.    4.    4.    4.    2.    2.    0. !
```

Function MySort can be modified to perform a decreasing order sort if the line `if v(i)>v(j) then` is modified to read `if v(i)<v(j) then`. Alternatively, you can modify the function *MySort* to allow for increasing or decreasing sort by adding a second, string, variable which can take the value of '*d*' for decreasing order or '*i*' for increasing order. The function file is shown below:

```
function [v] = MySort(u,itype)
//|--------------------------------|
//|This function sorts vector u in  |
//|increasing order, returning the  |
//|sorted vector as v.              |
//|If itype = 'd', decreasing order |
//|If itype = 'i', increasing order |
//|--------------------------------|
//Determine length of vector
n = length(u)
//Copy vector u to v
v = u
//Start sorting process
if itype == 'i' then
   for i = 1:n-1
       for j = i+1:n
           if v(i)>v(j) then
              temp = v(i)
              v(i) = v(j)
              v(j) = temp
           end
       end
   end
else
   for i = 1:n-1
       for j = i+1:n
          if v(i)<v(j) then
             temp = v(i)
             v(i) = v(j)
             v(j) = temp
          end
       end
   end
end
```

The following is an application of this function

```
-->u = int(10*rand(1,10))
 u  =

!   2.    7.    0.    3.    6.    6.    8.    6.    8.    0. !

-->MySort(u,'i')
 ans  =

!   0.    0.    2.    3.    6.    6.    6.    7.    8.    8. !

-->MySort(u,'d')
 ans  =

!   8.    8.    7.    6.    6.    6.    3.    2.    0.    0. !
```

In the following example, the function is used to swap the order of the elements of a vector. Thus, in a vector *v* of *n* elements, v_1 is swapped with v_n, v_2 is swapped with v_{n-1}, and so on. The general formula for swapping depends on whether the vector has an even or an odd number of elements. To check whether the vector length, *n*, is odd or even we use the function *modulo*. If *modulo(n,2)* is equal to zero, *n* is even, otherwise, it is odd. If *n* is even, then the swapping occurs by exchanging v_j with v_{n-j+1} for $j = 1, 2, \ldots, n/2$. If *n* is odd, the swapping occurs by exchanging v_j with v_{n-j+1} for $j = 1, 2, \ldots, (n-1)/2$. Here is the listing of the function:

```
function [v] = swapVector(u)
//|---------------------------------|
//| This function swaps the elements |
//| of vector u, v(1) with v(n), v(2)|
//| with v(n-1), and so on, where n  |
//| is the length of the vector.     |
//|---------------------------------|
n = length(u)  //length of vector
v = u          //copy vector u to v
//Determine upper limit of swapping
if modulo(n,2) == 0 then
   m = n/2
else
   m = (n-1)/2
end
//Perform swapping
for j = 1:m
    temp = v(j)
    v(j) = v(n-j+1)
    v(n-j+1) = temp
end
```

Applications of the function, using first a vector of 10 elements and then a vector of 9 elements, is shown next:

```
-->u = [-10:1:-1]
 u  =

! - 10.  - 9.  - 8.  - 7.  - 6.  - 5.  - 4.  - 3.  - 2.  - 1. !
```

```
-->swapVector(u)
 ans  =

! - 1.  - 2.  - 3.  - 4.  - 5.  - 6.  - 7.  - 8.  - 9.  - 10. !

-->u = [1:9]
 u  =

!   1.    2.    3.    4.    5.    6.    7.    8.    9. !

-->swapVector(u)
 ans  =

!   9.    8.    7.    6.    5.    4.    3.    2.    1. !
```

The `case ... select` structure can be used to select one out of many possible outcomes in a process. In the following example, the function *Area* calculates the area of different open-channel cross-sections based on a selection performed by the user:

```
function [] = Area()
//|-----------------------------------|
disp("============================")
disp("Open channel area calculation")
disp("============================")
disp("Select an option:")
disp("    1 - Trapezoidal")
disp("    2 - Rectangular")
disp("    3 - Triangular")
disp("    4 - Circular")
disp("============================")
itype = input("")
select itype,
  case 1 then
        id = "trapezoidal"
              b = input("Enter bottom width:")
        z = input("Enter side slope:")
        y = input("Flow depth:")
             A = (b+z*y)*y
  case 2 then
        id = "rectangular"
        b = input("Enter bottom width:")
        z = input("Enter side slope:")
        y = input("Flow depth:")
             A = (b+z*y)*y
  case 3 then
        id = "triangular"
              b = input("Enter bottom width:")
        z = input("Enter side slope:")
        y = input("Flow depth:")
              A = (b+z*y)*y
  else
        id = "circular"
              D = input("Enter pipe diameter:")
              y = input("Enter flow depth (y<D):")
              beta = acos(1-2*y/D)
              A = D^2/4*(beta+sin(beta)*cos(beta))
 end
//Print calculated area
printf("The area for a " + id + " cross-section is %10.6f .",A)
```

An example of application of this function is shown next:

```
-->Area()

 ============================

 Open channel area calculation

 ============================

 Select an option:

     1 - Trapezoidal

     2 - Rectangular

     3 - Triangular

     4 - Circular

 ============================

-->1
Enter bottom width:-->1.2
Enter side slope:-->0.2
Flow depth:-->1.0
The area for a trapezoidal cross-section is   1.400000 .
```

Simple output in SCILAB functions

Notice that we use function *disp* in function *Area,* above, to print the table after the function is invoked. On the other hand, to print the value of the resulting area we use function *printf*. This function is borrowed from C, while *disp* is a purely SCILAB function. Function *disp* will print a string used as the argument, as in the examples listed in the function *Area*. The following are a few examples of using function *disp* interactively in SCILAB:

- Displaying a string:

```
-->disp("Hello, world")

 Hello, world
```

- Displaying a string identifying a value and the corresponding value:

```
-->A = 2.3; disp(A," A = ")

  A =

    2.3
```

- Displaying more than one value and an identifying string

```
-->A = 2.3; b = 1.2; disp(b," b = ",A, " A = ")

  A =

    2.3
```

```
  b =

    1.2
```

- Displaying a value and its identifying string in a single line by converting the value into a string:

```
-->A = 34.5; disp(" A = " + string(A))

  A = 34.5
```

The function *Area* also utilizes the function *printf* to produce output. This function includes as arguments a string and, optionally, a list of variables whose values will be incorporated in the output string. To incorporate those values in the output string we use *conversion strings* such as *%10.6f*. The conversion string *%10.6f* represents an output field of for a floating-point result, thus the *f*, with 10 characters including 6 decimal figures. An integer value can be listed in a conversion string of the form *%5d*, which will reserve 5 characters for the output. The following interactive example illustrates the use of *printf* to print a floating point and an integer value:

```
-->A = 24.5; k = 2;

-->printf("The area is %10.6f for iteration number %5d.",A,k)
The area is  24.500000 for iteration number     2.
```

The characters \n in the output string of function *printf* produce a new line. For example:

```
-->printf("The area is %10.6f \nfor iteration number %5d.",A,k)
The area is  24.500000
for iteration number     2.
```

Function *printf* can be used to produce an output table, as illustrated in the following function:

```
function [] = myTable()
printf("====================================")
printf("    a        b        c        d   ")
printf("====================================")
for j = 1:10
    a = sin(10*j)
    b = a*cos(10*j)
    c = a + b
    d = a - b
    printf("%+6.5f  %+6.5f  %+6.5f  %+6.5f",a,b,c,d)
end
printf("====================================")
```

The application of this function is shown next:

```
-->myTable()
```

```
====================================
    a         b         c         d
====================================
-0.54402  +0.45647  -0.08755  -1.00049
+0.91295  +0.37256  +1.28550  +0.54039
-0.98803  -0.15241  -1.14044  -0.83563
+0.74511  -0.49694  +0.24817  +1.24206
-0.26237  -0.25318  -0.51556  -0.00919
-0.30481  +0.29031  -0.01451  -0.59512
+0.77389  +0.49012  +1.26401  +0.28377
-0.99389  +0.10971  -0.88418  -1.10360
+0.89400  -0.40058  +0.49342  +1.29457
-0.50637  -0.43665  -0.94301  -0.06972
====================================
```

Notice that the conversion string was changed to `%+6.5f`, which allows for the inclusion of a sign in the values printed.

The function *myTable* is modified next to produce output using scientific notation. The conversion string used here is `%+6.5e`.

```
function [] = myTable()
printf("============================================================")
printf("          a             b            c             d   ")
printf("============================================================")
for j = 1:10
    a = sin(10*j)
    b = a*cos(10*j)
    c = a + b
    d = a - b
    printf("%+6.5e  %+6.5e  %+6.5e  %+6.5e",a,b,c,d)
end
printf("============================================================")
```

The application of the function with the new conversion strings produces the following table:

```
-->myTable()
============================================================
          a             b             c             d
============================================================
-5.44021e-001  +4.56473e-001  -8.75485e-002  -1.00049e+000
+9.12945e-001  +3.72557e-001  +1.28550e+000  +5.40389e-001
-9.88032e-001  -1.52405e-001  -1.14044e+000  -8.35626e-001
+7.45113e-001  -4.96944e-001  +2.48169e-001  +1.24206e+000
-2.62375e-001  -2.53183e-001  -5.15558e-001  -9.19203e-003
-3.04811e-001  +2.90306e-001  -1.45050e-002  -5.95116e-001
+7.73891e-001  +4.90120e-001  +1.26401e+000  +2.83771e-001
-9.93889e-001  +1.09713e-001  -8.84176e-001  -1.10360e+000
+8.93997e-001  -4.00576e-001  +4.93420e-001  +1.29457e+000
-5.06366e-001  -4.36649e-001  -9.43014e-001  -6.97170e-002
============================================================
```

The table examples presented above illustrate also the use of the *for … end* structure to produce programming loops. Additional examples of programming loops are presented in the following examples. (For additional information on input-output, see reference [9]).

Examples of functions that include loops

SCILAB provides with the *while … end* and the *for … end* structures for creating programming loops. Programming loops can be used to calculate summations or products. For example, to calculate the summation

$$S(n) = \sum_{k=0}^{n} \frac{1}{k^2 + 1},$$

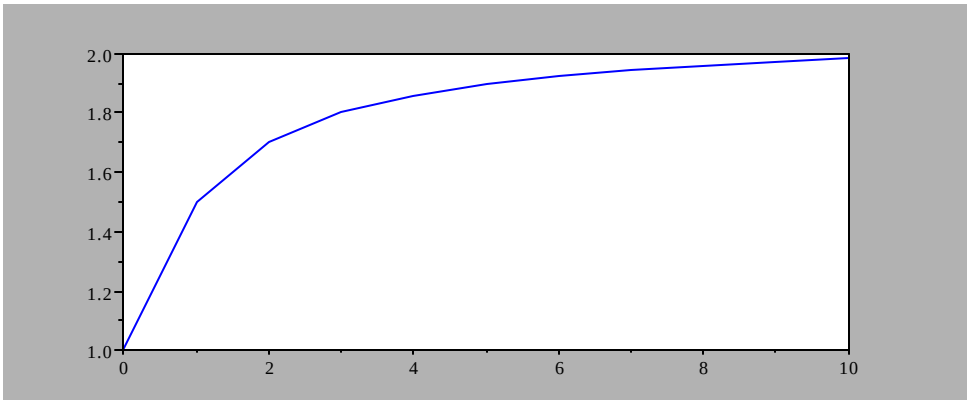
we can put together the following function:

```
function [S] = sum1(n)
//|----------------------------------------|
//| This function calculates the summation |
//|                                        |
//|                        n               |
//|                  \---------            |
//|                   \    1               |
//|        S(n) =      \ ------            |
//|                    /   2               |
//|                   /   k + 1            |
//|                  /---------            |
//|                      k = 0             |
//|                                        |
//|----------------------------------------|
S = 0        //Initialize sum to zero
k = 0        //Initialize index to zero
//loop for calculating summation
while k <= n
   S = S + 1/(k^2+1)  //add to summation
   k = k + 1          //increase the index
end
```

Within the SCILAB environment we can produce a vector of values for the sum with values of *n* going from *0* to *10*, and produce a plot of values of the sum versus *n*:

```
-->nn = [0:10];                //vector of values of n (nn)

-->m  = length(nn);            //length of vector nn

-->SS = [];                    //initialize vector SS as an empty matrix

-->for j = 1:m                 //this loop fills out the vector SS
-->    SS = [SS sum1(n(j))];   //with values calculated from function
-->end                         //sum1(n)

-->plot(nn,SS)                 //show results in graphical format
```

The resulting graph is shown below:

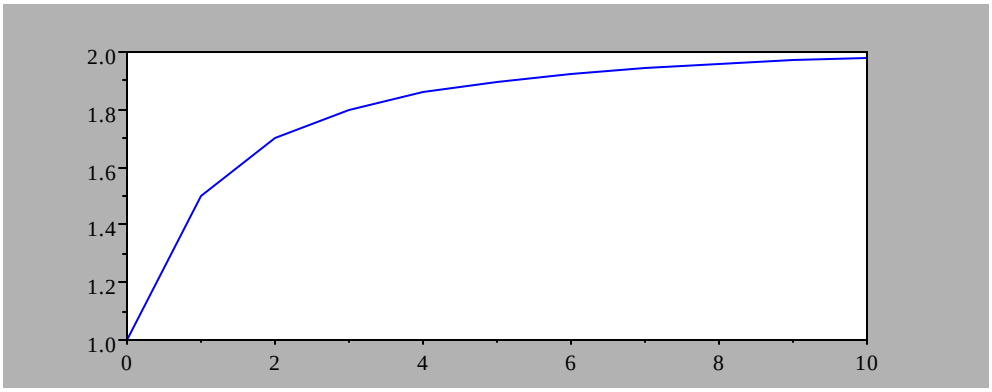


If we need to calculate a vector, or matrix, of values of the summation we can modify the original function *sum1* to handle matrix input and output as follows (the function is now called *sum2*):

```
function [S] = sum2(n)
//|----------------------------------------|
//| This function calculates the summation |
//|                                        |
//|                        n               |
//|                   \---------           |
//|                    \    1              |
//|        S(n) =       \ ______           |
//|                     /   2              |
//|                    /   k + 1           |
//|                   /---------           |
//|                       k = 0            |
//|                                        |
//| Matrix input and output is allowed.    |
//|----------------------------------------|
[nr,nc] = size(n) //Size of input matrix
S = zeros(nr,nc)  //Initialize sums to zero
//loop for calculating summation matrix
for i = 1:nr               //sweep by rows
    for j = 1:nc           //sweep by columns
        k = 0              //initialize index to zero
        while k <= n(i,j)
            S(i,j) = S(i,j) + 1/(k^2+1)
            k = k + 1
        end
        //disp(i,"i=",j,"j=",S(i,j),"S=")
    end
end
```

Notice that both *n* and *S* are now treated as matrix components within the function. The following SCILAB commands will produce the same results as before:

```
-->nn = [0:10]; SS = sum2(nn); plot(nn,SS)
```



Notice that function *sum2* illustrates the use of the two types of loop control structures available in SCILAB:

(1) *while … end* (used for calculating the summations in this example) and,
(2) *for … end* (used for filling the output matrix *S* by rows and columns in this example).

Nested *for … end* loops can be used to calculate double summations, for example,

$$S(n,m) = \sum_{i=0}^{n} \sum_{j=0}^{m} \frac{1}{(i+j)^2 + 1}.$$

Here is a listing of a function that calculates *S(n,m)* for single values of *n* and *m*:

```
function [S] = sum2x2(n,m)
//|--------------------------------------|
//| This function calculates the double  |
//| summation:                           |
//|            n        m               |
//|         \------  \------             |
//|          \        \          1       |
//| S(n,m) =  \        \    ----------   |
//|           /        /          2      |
//|          /        /     (i+j) + 1     |
//|         /------  /------              |
//|           i=0      j=0               |
//|--------------------------------------|
//|Note: the function only takes scalar  |
//|values of n and m and returns a single |
//|value.  An error is reported if n or m |
//|are vectors or matrices.              |
//|--------------------------------------|
//Check if m or n are matrices
if length(n)>1 | length(m)>1 then
   error('sum2 - n,m must be scalar values')
   abort
end
//Calculate summation if n and m are scalars
S = 0                    //initialize sum
for i = 1:n              //sweep by index i
    for j = 1:m          //sweep by index j
        S = S + 1/((i+j)^2+1)
    end
end
```

A single evaluation of function *sum2x2* is shown next:

```
-->sum2x2(3,2)
 ans  =

     .5561086
```

The following SCILAB commands produce a matrix *S* so that element S_{ij} corresponds to the *sum2x2* evaluated at *i* and *j*:

```
-->for i = 1:3
-->    for j = 1:4
-->        S(i,j) = sum2x2(i,j);
-->    end
-->end

-->S
 S  =

!    .2           .3           .3588235     .3972851 !
!    .3           .4588235     .5561086     .6215972 !
!    .3588235     .5561086     .6804207     .7659093 !
```

As illustrated in the example immediately above, nested *for … end* loops can be used to manipulate individual elements in a matrix. Consider, for example, the case in which you want to produce a matrix $z_{ij} = f(x_i,y_j)$ out of a vector *x* of *n* elements and a vector *y* of *m* elements. The function *f(x,y)* can be any function of two variables defined in SCILAB. Here is the function to produce the aforementioned matrix:

```
function [z] = f2eval(x,y,f)
//|-------------------------------------|
//| This function takes two row vectors |
//| x (of length n) and y (of length m) |
//| and produces a matrix z (nxm) such  |
//| that z(i,j) = f(x(i),y(j)).         |
//|-------------------------------------|
//Determine the lengths of vectors
n = length(x)
m = length(y)
//Create a matrix nxm full of zeros
z = zeros(n,m)
//Fill the matrix with values of z
for i = 1:n
    for j = 1:m
        z(i,j) = f(x(i),y(j))
    end
end
```

Here is an application of function *f2solve*:

```
-->deff('[z]=f00(x,y)','z=x.*sin(y)')

-->x = [1:1:3]
 x  =

!   1.    2.    3. !

-->y = [2:1:5]
```

```
 y  =

!   2.    3.    4.    5. !

-->z = f2eval(x,y,f00)
 z  =

!    .9092974     .1411200  -  .7568025  -  .9589243 !
!   1.8185949     .2822400  - 1.513605   - 1.9178485 !
!   2.7278923     .4233600  - 2.2704075  - 2.8767728 !
```

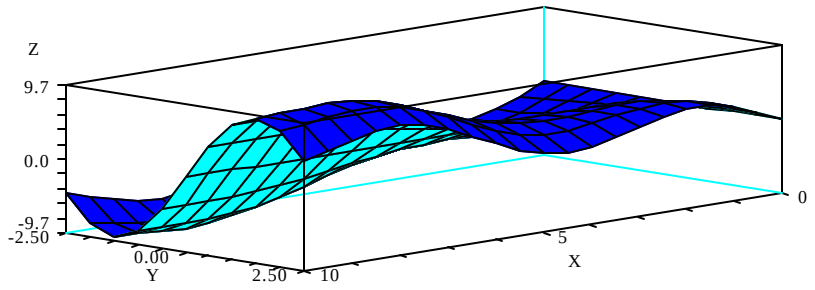
Notice that SCILAB already provides such a function (it is called *feval*):

```
-->feval(x,y,f00)
 ans  =

!    .9092974     .1411200  -  .7568025  -  .9589243 !
!   1.8185949     .2822400  - 1.513605   - 1.9178485 !
!   2.7278923     .4233600  - 2.2704075  - 2.8767728 !
```

SCILAB function *feval* (or the user-defined function *f2eval*) can be used for evaluating matrix of functions of two variables to produce surface plots. Consider the following case:

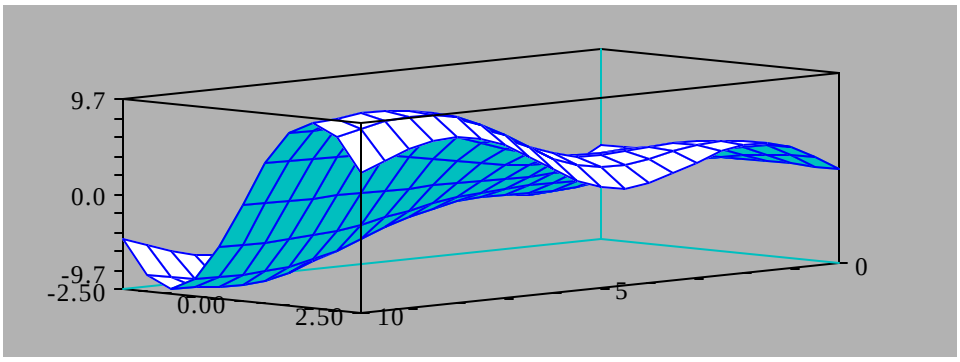
```
-->deff('[z]=f01(x,y)','z = x.*sin(y)+y.*sin(x)')
-->x = [0:0.5:10]; y = [-2.5:0.5:2.5];
-->z = f2eval(x,y,f01);
-->plot3d(x,y,z)
```

The resulting graph is shown next:



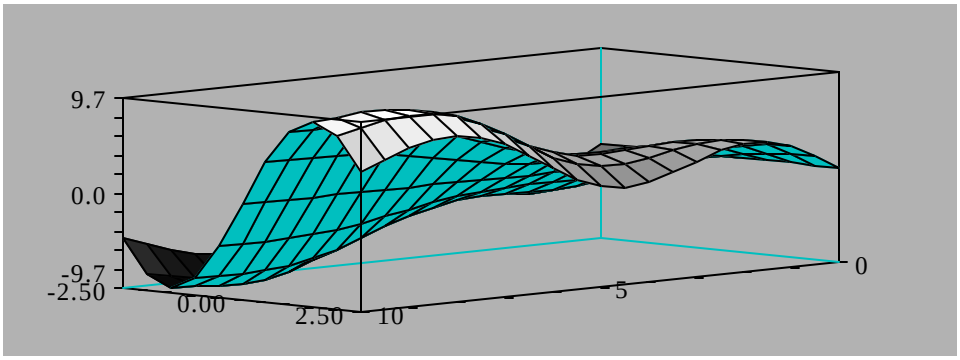
Or, if you have loaded the PLOTLIB library (see, for example, reference [11]), you can use, either

```
--> mesh(x,y,z')
```



or,

```
--> surf(x,y,z')
```



Nested loops can be used for other operations with matrices. For example, a function that multiplies two matrices is shown next:

```
function [C] = matrixmult(A,B)
//|--------------------------------------|
//| This function calculates the matrix  |
//| C(nxm) = A(nxp)*B(pxm)               |
//|--------------------------------------|
//First, check matrices compatibility
[nrA,ncA] = size(A)
[nrB,ncB] = size(B)
if ncA <> nrB then
   error('matrixmult - incompatible matrices A and B')
   abort
end
//Calculate matrix product
C = zeros(nrA,ncB)
for i = 1:nrA
    for j = 1:ncB
        for k = 1:ncA
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end
    end
end
```

An application of function *matrixmult* is shown next. Here we multiply $\mathbf{A}_{3x4}$ with $\mathbf{B}_{4x6}$ resulting in $\mathbf{C}_{3x6}$. However, the product $\mathbf{B}_{4x6}$ times $\mathbf{A}_{3x4}$ does not work.

```
-->A = int(10*rand(3,4))
 A  =

!   2.    3.    8.    0. !
!   7.    6.    6.    5. !
!   0.    6.    8.    6. !

-->B = int(10*rand(4,6))
 B  =

!   7.    2.    3.    3.    3.    2. !
!   1.    2.    9.    2.    5.    6. !
!   5.    8.    2.    5.    5.    4. !
!   2.    6.    3.    4.    4.    9. !
-->C = matrixmult(A,B)
```

```
 C  =

!   57.    74.     49.     52.    61.     54.  !
!   95.    104.    102.    83.    101.    119. !
!   58.    112.    88.     76.    94.     122. !

-->D = matrixmult(B,A)
 !--error  9999
matrixmult - incompatible matrices A and B
at line      10 of function matrixmult              called by :
D = matrixmult(B,A)
```

Of course, matrix multiplication in SCILAB is readily available by using the multiplication sign (*), e.g.,

```
-->C = A*B
 C  =

!   57.    74.     49.     52.    61.     54.  !
!   95.    104.    102.    83.    101.    119. !
!   58.    112.    88.     76.    94.     122. !

-->D = B*A
        !--error    10
inconsistent multiplication
```

Reading a table from a file

The first example presented in this section uses the command *file* to open a file for reading, and the function *read* for reading a matrix out of a file. This application is useful when reading a table into SCILAB. Suppose that the table consists of the following entries, and is stored in the file *c:\table1.dat*:

```
 2.35    5.42     6.28     3.17     5.23
 3.19    3.87     3.21     5.18     6.32
 1.34    5.17     8.23     7.28     1.34
 3.21    3.22     3.23     3.24     3.25
 8.17    4.52     8.78     2.31     1.23
 9.18    5.69     3.45     2.25     0.76
```

The following function reads the table out of file *c:\table1.dat* and separates the 5 columns into a number of variables *x1, x2, x3, x4, x5*:

```
function [x1,x2,x3,x4,x5] = readtable()
//|--------------------------------------|
//| This is an interactive function that  |
//| requests from the user the name of a  |
//| file containing a table and reads the |
//| table out of the file.  The table has |
//| exactly 5 columns.                    |
//|--------------------------------------|
printf("Reading a table from a file\n")
printf("===========================\n")
printf(" ")
filname = input("Enter filename between quotes:\n")
u = file('open',filname,'old')  //open input file
```

```
Table = read(u,-1,5)
[n m] = size(Table)
printf(" ")
printf("There are %d columns in the table",m)
for j = 1:m
    execstr('x'+string(j)+' = Table(:,j) ')
end
close(u)                          //close input file
```

The *file* command in function *readtable* uses three arguments, the first being the option *'open'* to indicate that a file is being opened. The second argument is the file name, and the third is a qualifier indicating that the file already exists (required for input files). Also, the left-hand side of this statement is the variable *u* which will be given an integer value by SCILAB. The variable *u* represents the *logical input-output unit* assigned to the file being opened.

Function *readtable* uses the command *read* with the second argument being *-1* which allows for not knowing the number of rows in the matrix to be read. With this argument set to *-1*, SCILAB will read all rows in the matrix. The number of columns, however, needs to be known (This is not too difficult to find out since we have access to the input file -- a text file -- through a text editor program, e.g., *PFE*).

Notice the line `execstr('x'+string(j)+' = Table(:,j) ')` in function *readtable*. This line puts together a string that becomes the argument of function *execstr*. For example, if *j=1*, the argument becomes `'x1 = Table(:,1)'`. Function *execstr* (*exec*ute *str*ing) then translate the resulting string into a SCILAB command. Thus, for this example, it will be equivalent to writing: `x1 = Table(:,1)`

As shown in function *readtable*, it is always a good idea to insert a *close* statement to ensure that the file is closed and available for access from other programs.

Function *readtable* assumes that all the files to be read have only 5 columns. The columns are returned into variables *x1, x2, x3, x4, x5*. Here is an example of application of function *readtable*. User response is shown in bold face.

```
-->[x1,x2,x3,x4,x5] = readtable()
Reading a table from a file
===========================

Enter filename between quotes:
-->'c:\table1.dat'

There are 5 columns in the table
 x5  =
!   5.23 !
!   6.32 !
!   1.34 !
!   3.25 !
!   1.23 !
!    .76 !
 x4  =
!   3.17 !
!   5.18 !
!   7.28 !
!   3.24 !
!   2.31 !
!   2.25 !
```

```
 x3  =
!   6.28 !
!   3.21 !
!   8.23 !
!   3.23 !
!   8.78 !
!   3.45 !
x2  =
!   5.42 !
!   3.87 !
!   5.17 !
!   3.22 !
!   4.52 !
!   5.69 !
 x1  =
!   2.35 !
!   3.19 !
!   1.34 !
!   3.21 !
!   8.17 !
!   9.18 !
```

A large amount of examples for input-output and file manipulation are contained in reference [9]. A few examples are also presented in Chapter 2 of reference [1].

Writing to a file

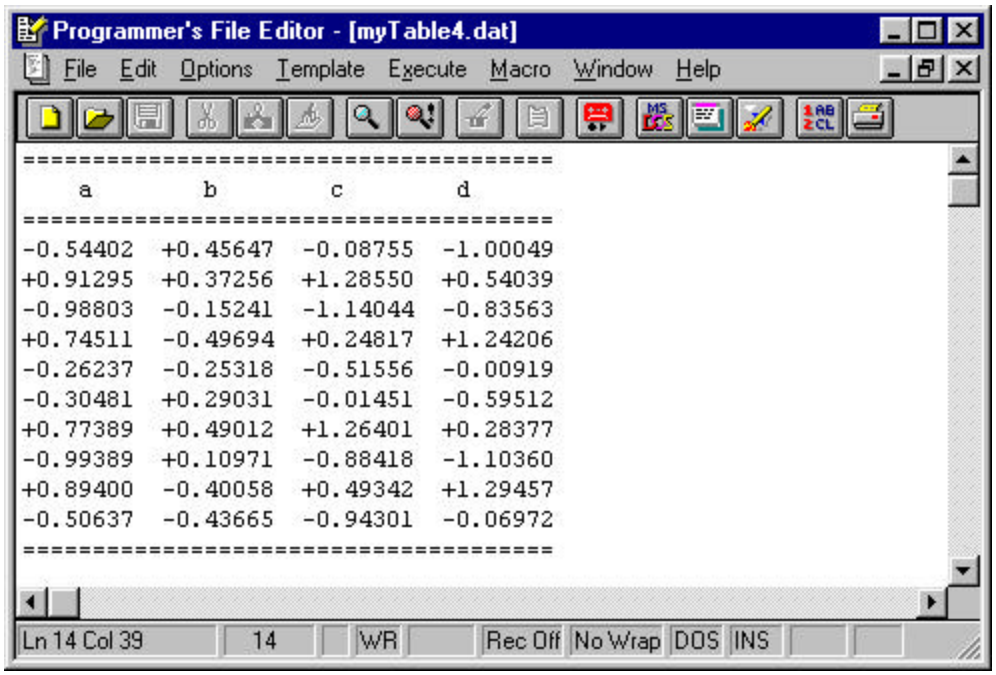
The simplest way to write to a file is to use the function *fprintf* in a similar fashion as function *printf*. As an example, we re-write function *myTable* (shown earlier) to include *fprintf* statements that will write the results to a file as well as to the screen:

```
function [] = myTableFile()
printf("Printing to a file")
printf("==================")
filname = input("Enter file to write to (between quotes):\n")
u = file('open',filname,'new')  //open output file
printf("=======================================")
fprintf(u,"=======================================")
printf("    a        b        c        d   ")
fprintf(u,"    a        b        c        d   ")
printf("=======================================")
fprintf(u,"=======================================")
for j = 1:10
    a = sin(10*j)
    b = a*cos(10*j)
    c = a + b
    d = a - b
    printf("%+6.5f  %+6.5f  %+6.5f  %+6.5f",a,b,c,d)
    fprintf(u,"%+6.5f  %+6.5f  %+6.5f  %+6.5f",a,b,c,d)
end
printf("=======================================")
fprintf(u,"=======================================")
close(u)       //close output file
```

Here is an application of function *myTableFile*. The function prints a table in SCILAB, while simultaneously printing to the output file. The user response is shown in bold face:

```
-->myTableFile()
Printing to a file
==================
Enter file to write to (between quotes):
-->'c:\myTable4.dat'
=====================================
    a         b         c         d
=====================================
-0.54402  +0.45647  -0.08755  -1.00049
+0.91295  +0.37256  +1.28550  +0.54039
-0.98803  -0.15241  -1.14044  -0.83563
+0.74511  -0.49694  +0.24817  +1.24206
-0.26237  -0.25318  -0.51556  -0.00919
-0.30481  +0.29031  -0.01451  -0.59512
+0.77389  +0.49012  +1.26401  +0.28377
-0.99389  +0.10971  -0.88418  -1.10360
+0.89400  -0.40058  +0.49342  +1.29457
-0.50637  -0.43665  -0.94301  -0.06972
=====================================
```

With a text editor (e.g., *PFE*) we can open the file *c:\myTable4.dat* and check out that the same table was printed to that text file.



Once more, the reader is invited to review the examples for input-output and file manipulation contained in reference [9] as well as in Chapter 2 of reference [1].

Summary

This document shows a number of programming examples using decision and loop structures, input-output, and other elements of programming. Review also the many functions presented in the references listed below. Advanced SCILAB programming includes the use of menus and dialogs. For examples of these applications, the user is referred to reference [14].

REFERENCES

The references included herein are of four types: (1) SCILAB books; (2) SCILAB documents; (3) Computer programming documents; and (4) Online programming examples. Information on the location of those documents is provided below.

SCILAB books by G. Urroz

[1] Numerical and Statistical Methods with SCILAB for Science and Engineering - Volume 1
[2] Numerical and Statistical Methods with SCILAB for Science and Engineering - Volume 2

Find additional information about these books at the web site:

http://www.engineering.usu.edu/cee/faculty/gurro/myBooks.htm

SCILAB documents by G. Urroz

All these documents can be downloaded from the web site (note: it is a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/SCILAB_Notes&Functions.htm

The web site contains additional files besides the ones listed below. The reference numbers shown below apply only to the references in this document, and do not represent any numbering of the documents in the web site.

[3] Getting started with SCILAB
[4] Introduction to SCILAB 2.6 - Replaces Ch. 1 in [1]
[5] Summary of SCILAB book - Volume 1 – Summary of chapters 1 through 9 in [1]
[6] Script, function and diary files with SCILAB
[7] Elementary mathematical functions with SCILAB
[8] Operations with logical vectors
[9] Input/Output and file manipulation
[10] Debugging SCILAB programs - zip file
[11] PLOTLIB examples - Complements Ch. 3 (Graphics) in [1]
[12] Additional notes on SCILAB graphics - Complements Ch. 3 (Graphics) in [1]
[13] Multiple-defined functions in SCILAB
[14] Menus and dialogs in SCILAB

Computer programming documents by G. Urroz

All these documents can be downloaded from the web site (note: it is a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Classes/Classes_Fall2002/ENGR2200/ENGR2200_IntroToComputers.htm

Look under the heading ***My class notes*** for the following documents. The web site contains additional files besides the ones listed below. The reference numbers shown

below apply only to the references in this document, and do not represent any numbering of the documents in the web site.

[15] Computer terminology and definitions
[16] Computer programming basics
[17] Flowchart examples for non-linear equation solution

Online SCILAB programming examples by G. Urroz

The functions accompanying the textbooks (see references [1] and [2], above) are available for download at the web site (a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/SCILAB_Notes&Functions.htm

Click on the link "**SCILAB functions**" at the top of the page for instructions and downloading the functions.

Additional programming examples are available in the following web site (a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Classes/Classes_Fall2001/CEE6510/CEE6510_SCILABExamples.htm

Programs are available in this web site that show how to drive SCILAB functions from scripts.

NOTE

All the web sites listed in this REFERENCES section are accessible through the web site:

http://www.engineering.usu.edu/cee/faculty/gurro

Follow the links *Software* > *SCILAB* for additional information on SCILAB, or the links *My Classes* > *ENGR2200 – Numerical Methods in Engineering I* or *My Classes* > *CEE6510 – Numerical and Statistical Methods in Civil Engineering* to access the web sites corresponding to these two courses that use SCILAB.