Programming in Scilab
Files in Scilab and Maxima

April 20, 2007

In the directory for this Lecture you will find scilab.pdf which is a reference on Scilab I prepared for 2nd year students.

Contents

12.1 Programming in Scilab ............................................ 2
  12.1.1 FOR loops ............................................. 2
  12.1.2 Functions .............................................. 4
  12.1.3 Functions and Dot Operators ............................. 5
  12.1.4 Comparison Operators ................................. 6
  12.1.5 WHILE Loops ............................................ 6
  12.1.6 IF Statements .......................................... 6
12.2 Working with Files in Scilab ................................. 7
  12.2.1 Function Files .......................................... 8
  12.2.2 Exporting Data ....................................... 8
  12.2.3 Importing Data ....................................... 9
  12.2.4 Saving and Restoring Data ............................ 11
12.3 Working with Files in Maxima .............................. 11
  12.3.1 Saving Output ......................................... 11
  12.3.2 Saving and Restoring Work ............................ 12
12.1 Programming in Scilab

In later lectures we will need to know about Scilab functions and a few simple programming constructs.

12.1.1 FOR loops

In Scilab for loops are used to iterate over a set of values. Here is a simple example of a for loop which should be easy to understand:

```scilab
-->v = zeros(1,10);
-->for i = 1:10
    --> v(i) = i;
-->end

-->v
v =

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

For each i from 1 to 10, we have set the ith component of the vector v to the value i.

The general form of a for loop is:

```scilab
for variable = row_vector
    statement
    
    statement
end
```

Notes:

1. In the example above the vector v was initialized to vector of zeros before performing the loop. This is not strictly necessary, but is good programming practice. If v wasn’t initialized, then on each pass through the loop the size of v would have to increase. for loops can be pretty slow and having to reallocate memory on each pass through the loop could slow things down even more.

2. The row vector used for iteration in a for loop is almost always constructed using the colon operator, e.g. the range 1:10 in the example above.
3. \texttt{for} loops are one place you almost always want to terminate statements with semicolons; otherwise the a result would be printed on every pass through the loop.

Here is an example producing the 5 by 5 identity matrix:

\begin{verbatim}
-->ident = zeros(5, 5);

-->for i = 1:5
-->   ident(i,i) = 1;
-->end

-->ident
ident =

! 1. 0. 0. 0. 0. !
! 0. 1. 0. 0. 0. !
! 0. 0. 1. 0. 0. !
! 0. 0. 0. 1. 0. !
! 0. 0. 0. 0. 1. !
\end{verbatim}

For each \texttt{i} from 1 to 5, we have set the corresponding diagonal component of the matrix, \texttt{ident(i,i)}, to 1.

It is possible to have \texttt{for} loops within \texttt{for} loops. These are called \textit{nested loops} and are useful in constructing matrices conforming to some pattern.

Recall that the \textbf{Hilbert matrix} is the $n \times n$ matrix

$$H = \begin{bmatrix}
1 & \frac{1}{2} & \cdots & \frac{1}{n} \\
\frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{1}{n} & \frac{1}{n+1} & \cdots & \frac{1}{2n-1}
\end{bmatrix}$$

Here is how to produce the $5 \times 5$ Hilbert matrix:

\begin{verbatim}
-->h = zeros(5,5);

-->for i = 1:5
-->   for j = 1:5
-->      h(i,j) = 1/(i+j-1);
-->   end
-->end
\end{verbatim}
--->h
h =

! 1. 0.5 0.3333333 0.25 0.2 !
! 0.5 0.3333333 0.25 0.2 0.1666667 !
! 0.3333333 0.25 0.2 0.1666667 0.1428571 !
! 0.25 0.2 0.1666667 0.1428571 0.125 !
! 0.2 0.1666667 0.1428571 0.125 0.1111111 !

Note that each for statement has a corresponding end statement. Indenting for loops as above makes it clear which end statement matches which for statement.

12.1.2 Functions

To produce the Hilbert matrix \( H_n \) for any value of \( n \) it is best to define a function to perform the task:

-->function h = hilbert(n)
--> h = zeros(n,n)
--> for i = 1:n
--> for j = 1:n
--> h(i,j) = 1/(i + j - 1)
--> end
--> end
-->endfunction

-->hilbert(5)
ans =

! 1. 0.5 0.3333333 0.25 0.2 !
! 0.5 0.3333333 0.25 0.2 0.1666667 !
! 0.3333333 0.25 0.2 0.1666667 0.1428571 !
! 0.25 0.2 0.1666667 0.1428571 0.125 !
! 0.2 0.1666667 0.1428571 0.125 0.1111111 !

-->hilbert(3)
ans =

! 1. 0.5 0.3333333 !
! 0.5 0.3333333 0.25 !
! 0.3333333 0.25 0.2 !
In this example

1. **hilbert** is the name of the function.

2. **n** is the argument to the function. Functions can any number of arguments.

3. **h** is the value returned by the function. The actual value returned is the value of **h** immediately before the `endfunction` statement is reached.

Here is another example, the factorial function:

\[ n! = 1 \times 2 \times \ldots \times n \]

```plaintext
-->function fact = factorial(n)
--> fact = 1
--> for k = 1:n
--> fact = k*fact
--> end
-->endfunction

-->factorial(5)
ans =

120.

-->factorial(100)
ans =

9.333+157
```

### 12.1.3 Functions and Dot Operators

We have seen that built-in functions like *sin* can take vectors or matrices as arguments and then act element-by-element on that argument. We usually want the same thing to happen when we define mathematical functions in Scilab, for example, when we want to graph the function. This requires careful attention to the use of **dot** operators.

Consider, for example, the function

\[ f(x) = \sin x \cos x \]

Here is how we would write it as a Scilab function:

```plaintext
-->function y = f(x)
--> y = sin(x).*cos(x)
-->endfunction
```
If x is a vector, then \( \sin(x) \) and \( \cos(x) \) are both vectors, and we want our function to return the vector of values obtained by element-by-element multiplication of these two vectors.

### 12.1.4 Comparison Operators

These are used to compare values:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equal</td>
</tr>
<tr>
<td><code>~=</code></td>
<td>not equal</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

They are typically used in `while` and `if` statements (see below).

### 12.1.5 WHILE Loops

Usually `for` loops are used to repeat a series of statements a fixed number of times. In contrast `while` loops repeat a series of statements until a given condition is satisfied.

The following example illustrates how to find \( \varepsilon_{\text{mach}} \) without knowing the precision of arithmetic we are using. Recall that \( \varepsilon_{\text{mach}} \) is the smallest floating point number such that

\[
1 + \varepsilon_{\text{mach}} \neq 1
\]

Start with \( \text{eps} = 1 \) and repeatedly halve it until \( 1 + \text{eps} = 1 \). Then the value of \( \text{eps} \) we finish up with is twice \( \varepsilon_{\text{mach}} \), since the previous value \( \text{eps} \) must have been the last value which satisfied \( 1 + \text{eps} \neq 1 \).

```matlab
-->eps = 1;

-->while (1 + eps ~= 1)
-->eps = eps/2;
-->end

-->2*eps
ans =

2.220E-16
```

### 12.1.6 IF Statements

`if` statements allow us to perform alternative actions depending on the result of a test. The general form of the `if` statement is:
if (test1) then
    statements
elseif (test2) then
    statements
.
.
.
else
    statements
end

You can have any number of elseif clauses. On the other other hand, you don’t have to have an elseif clause nor, indeed, an else clause.

Here is a function which returns the sign of a number:

-->function s = signum(x)
-->    if (x > 0) then
-->        s = 1
-->    elseif (x < 0) then
-->        s = -1
-->    else
-->        s = 0
-->    end
-->endfunction

-->signum(12345)
ans =

   1.

-->signum(-12345)
ans =

- 1.

-->signum(0)
ans =

   0.

12.2 Working with Files in Scilab

We looked at Scilab script files, which are used to perform a sequence of Scilab commands, in the previous lecture.
12.2.1 Function Files

Function files are much like script files except that they contain one or more function definitions like the functions hilbert and factorial in §12.1.2. It is common to put functions in files rather than enter them directly into Scilab since (a) they are then saved away for further use, and (b) it is easy to correct or modify a function by editing the file.

Create a file, say hilbert.sci, containing the function defined earlier. (It is usual but not mandatory to end function files with the suffix .sci, SciPad will recognize a function file and supply the suffix.)

```scilab
function h = hilbert(n)
    h = zeros(n,n)
    for i = 1:n
        for j = 1:n
            h(i,j) = 1/(i + j - 1)
        end
    end
endfunction
```

Function files, like script files, are loaded with the exec command or from the Execute menu.

```scilab
-->exec("hilbert.sci");
-->hilbert(4)
```

ans =

! 1.  0.5  0.3333333  0.25  !
! 0.5  0.3333333  0.25  0.2  !
! 0.3333333  0.25  0.2  0.1666667 !
! 0.25  0.2  0.1666667  0.1428571 !

12.2.2 Exporting Data

The write command writes Scilab data to external files, which can then be used by other programs. Here is an example:

```scilab
-->z = rand(8,4)
z =

! 0.2113249  0.8782165  0.2312237  0.3616361 !
! 0.7560439  0.0683740  0.2164633  0.2922267 !
! 0.0002211  0.5608486  0.8833888  0.5664249 !
! 0.3303271  0.6623569  0.6525135  0.4826472 !
! 0.6653811  0.7263507  0.3076091  0.3321719 !
```
If you examine the file out1.dat it will look something like:

0.211324865 0.878216481 0.23122372 0.361636101
0.756043854 0.0683740368 0.216463263 0.292226664
0.000221134629 0.560848606 0.883388781 0.566424882
0.330327092 0.662356937 0.652513495 0.482647197
0.665381104 0.726350677 0.307609074 0.332171891
0.628391788 0.198514384 0.932961621 0.59350947
0.849745236 0.544257316 0.214600786 0.50153416
0.68573102 0.23207479 0.312642 0.436858758

Note that only one matrix can be written to a file at any one time.

12.2.3 Importing Data

The read command is used to read data from external files into a Scilab matrix.

We can read the data we wrote to a file in the previous example:

-->z1 = read("out1.dat", 8, 4)
   z1 =
   ! 0.2113249 0.8782165 0.2312237 0.3616361 !
   ! 0.7560439 0.0683740 0.2164633 0.2922267 !
   ! 0.0002211 0.5608486 0.8833888 0.5664249 !
   ! 0.3303271 0.6623569 0.6525135 0.4826472 !
   ! 0.6653811 0.7263507 0.3076091 0.3321719 !
   ! 0.6283918 0.1985144 0.9329616 0.5935095 !
   ! 0.8497452 0.5442573 0.2146008 0.5015342 !
   ! 0.6857310 0.2320748 0.312642 0.4368588 !

The matrix z1 is not the same as the matrix z since we saw that the data from z was written with only about 9 digits (or 15 digits in Scilab for Windows):
The `read` statement has the general form

\[ x = \text{read(filename, nrows, ncols)} \]

and `read` assumes that the data in the file it is reading is organized in columns. However the number of rows, `nrows`, and number of columns, `ncols` in the `read` statement doesn't have to match the layout of the data in the file.

Here are some examples:

```plaintext
-->z2 = read("out1.dat", 3, 3)
z2 =

! 0.2113249  0.8782165  0.2312237 !
! 0.7560439  0.0683740  0.2164633 !
! 0.0002211  0.5608486  0.8833888 !

-->z3 = read("out1.dat", 2, 5)
z3 =

! 0.2113249  0.8782165  0.2312237  0.3616361  0.7560439 !
! 0.0002211  0.5608486  0.8833888  0.5664249  0.3303271 !

-->z4 = read("out1.dat", -1, 4)
z4 =

! 0.2113249  0.8782165  0.2312237  0.3616361 !
! 0.7560439  0.0683740  0.2164633  0.2922267 !
! 0.0002211  0.5608486  0.8833888  0.5664249 !
! 0.3303271  0.6623569  0.6525135  0.4826472 !
! 0.6653811  0.7263507  0.3076091  0.3321719 !
```
In the first example above, we just read the first three rows and columns of the data. In the second example the first ten data values were read into a $2 \times 5$ matrix. If you know the number of columns in a data file, you can simply use -1 for the number of rows and all rows of the data will be read.

### 12.2.4 Saving and Restoring Data

Saving and restoring the environment between Scilab sessions can be done with the commands `save` and `load`. For example the current environment can be saved in file `work.dat` with the command

```
-->save("swork")
```

and then later restored with

```
-->load("swork")
```

The file produced by the `save` is a binary file which cannot be used in any sensible way by other programs.

If you only want to save a few variables or functions, say `a`, `b` and `c`, then the variation

```
-->save("swork", a, b, c)
```

saves only the named objects.

### 12.3 Working with Files in Maxima

We looked at Maxima batch files, which are like Scilab script files, in the previous lecture.

#### 12.3.1 Saving Output

The `writefile` command continues to write a transcript of your Maxima session to a file until the `closefile` command is given. Here is an example using the batch file `e11-3.mac` from the previous lecture:

```
(\%i2) writefile("l12");

(\%o2) FALSE
(\%i3) display2d : false;

(\%o3) FALSE
(\%i4) batch("e11-3.mac");
```
batching /home/gbunting/amth142/e11-3.mac

(%i5) e1:3*x-5*y = 6
(%o5) 3*x-5*y = 6
(%i6) e2:-3*y+y^2+x^2 = 7
(%o6) y^2-3*y+x^2 = 7
(%i7) s:SOLVE([e1,e2],[x,y])
(%o7) [[x = 49/17,y = 9/17],[x = -1/2,y = -3/2]]

(%i8) display2d : true;

(%o8) TRUE
(%i9) closefile();

(%o9) FALSE

The file 112 then gives a record of our interaction with Maxima:

(%o2) FALSE
(%i3)
(%o3) FALSE
(%i4)

batching /home/gbunting/amth142/e11-3.mac

(%i5) e1:3*x-5*y = 6
(%o5) 3*x-5*y = 6
(%i6) e2:-3*y+y^2+x^2 = 7
(%o6) y^2-3*y+x^2 = 7
(%i7) s:SOLVE([e1,e2],[x,y])
(%o7) [[x = 49/17,y = 9/17],[x = -1/2,y = -3/2]]

(%i8) (%o8) TRUE
(%i9)

By setting display2d to false the output is in a form which can be read by Maxima. This would allow, for example, the output file 112 to be edited and used as part of batch file for input.

12.3.2 Saving and Restoring Work

You can save and then restore your work with the pair of commands save and loadfile. These are much like save and load in Scilab. For example

(%i10) save("mwork",e1,e2);
(%o10) mwork
saves the variables \texttt{e1} and \texttt{e2} in the file \texttt{mwork}. These variables can be reloaded into a subsequent Maxima session by

\begin{verbatim}
(%i2) loadfile("mwork");

(%o2) mwork
\end{verbatim}

Files created with \texttt{save} contain Lisp, rather than Maxima, commands.