This practical assumes that you know how to create and edit text files and make them accessible to Scilab.

Contents

1 Script Files 2
2 Function Files 4
3 for Loops 7
4 while Loops 9
5 if Statements 10
1 Script Files

Script are files containing Scilab commands which are executed as if they were typed directly into Scilab. For example, create a file, say prac4-1, containing the commands

\[ k = -16:-1; \]
\[ h = 10.^(k); \]
\[ approx = (\sin(1+h) - \sin(1))./h; \]
\[ err = abs(approx - \cos(1)); \]
\[ plot2d(log10(h), log10(err)); \]

which is the set of commands that were used in Lecture 7 to examine the truncation error for the finite difference approximation. A semicolon at the end of line stops the result of that line being printed, which is usually what you want in script files.

Now at the Scilab prompt type exec('prac4-1') and the graph we created in Lecture 7 will be reproduced. (Note that although none of the results of the intermediate calculations were printed, these results are still available. For example, typing err will print the vector of errors.)

The main use of script files is to repeat a series of commands. We can make the above example more flexible by allowing different functions. For example, edit prac4-1:

\[ k = -16:-1; \]
\[ h = 10.^(k); \]
\[ approx = (f(x+h) - f(x))./h; \]
\[ err = abs(approx - df(x)); \]
\[ plot2d(log10(h), log10(err)); \]

You will note there are three names, f, df and x which have not been assigned values. These are respectively the function whose derivative we are approximating, the derivative of the function and the point at which the derivative is approximated. These have to be given values before the new prac4-1 script can be executed.

Setting \( x = 1 \) is easy:

\[ \rightarrow x = 1 \]
\[ x = 1. \]
Defining functions is different:

-->function y = f(x)
--> y = sin(x)
-->endfunction

-->function y = df(x)
--> y = cos(x)
-->endfunction

and we have defined two functions f and df. Now exec(‘prac4-1’) gives the same graph as before.

Changing x to 2 and executing the script:

-->x = 2
x =

2.

-->exec(‘prac4-1’)

gives a graph very like the previous one.

Now try a different function:

-->x = 1
x =

1.

-->function y = f(x)
--> y = exp(x)
-->endfunction
Warning :redefining function: f

-->function y = df(x)
--> y = exp(x)
-->endfunction
Warning :redefining function: df
Note that the new functions we defined had to be called $f$ and $df$ because these are the names that the script file expected and which led to the warnings.

```bash
-->exec('prac4-1')
```

Again we get a very similar graph, which indicates that errors in the finite difference approximation have the same sort of pattern irrespective of the function whose derivative we are approximating.

This example shows one of the important uses of script files — to perform numerical experiments where we want to vary the input to the experiment, in this case to examine the errors in a finite difference approximation for various functions and evaluation points.

## 2 Function Files

Function files are files containing a collection of function definitions similar to the functions we defined above. Function files generally serve a different purpose to script files in that they are usually collections of functions which either (a) we want to use repeatedly, or (b) are one-off functions for which it more convenient to enter via a file than via the terminal because it allows easier correction of errors.

Let us write a function that computes the error in the forward difference approximation we have been examining. We will make this even more flexible by having the function take as input a function $f$, its derivative $df$, the evaluation point $x$ and a vector of step-sizes $h$ and which returns the vector of errors $err$.

Create a file, say `fdiff.sci` (It is usual, but not mandatory, to end function files with the suffix `.sci`):

```scientific
// fdiff.sci

// Errors in the forward difference approximation

function err = fdiff(f, df, x, h)
    approx = (f(x+h) - f(x))./h
    err = abs(approx - df(x))
endfunction
```
The first three lines are comments, ignored by scilab. To load this function file use:

```scilab
-->getf('fdiff.sci')
```

We have already defined \( f \), \( df \), \( x \) and \( h \) and if we call our function \( fdiff \) with these arguments:

```scilab
-->e = fdiff(f, df, 1, h)
e =
```

<table>
<thead>
<tr>
<th>column 1 to 5</th>
<th>!</th>
<th>2.7182818</th>
<th>0.3903426</th>
<th>0.0093376</th>
<th>0.0004559</th>
<th>0.0004323</th>
<th>!</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>column 6 to 10</td>
<td>!</td>
<td>0.0000326</td>
<td>0.0000015</td>
<td>2.154E-07</td>
<td>6.603E-09</td>
<td>1.399E-07</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>column 11 to 15</td>
<td>!</td>
<td>0.0000014</td>
<td>0.0000136</td>
<td>0.0001359</td>
<td>0.0013596</td>
<td>0.0136368</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>column 16</td>
<td>!</td>
<td>0.1405601</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

we get the same results as before:

```scilab
-->err=e
ans =
```

| column 1 to 11 |    ! | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | ! |          |
| column 12 to 16 |    ! | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | ! |          |
We will use our function \texttt{fdiff} for one more example:

\begin{verbatim}
frctn y = g(x)
  y = log(x)
endfunction

frctn y = dg(x)
  y = x.^(-1)
endfunction

err = fddiff(g, dg, 3, h)
err =

 Column 1 to 5

|   0.333333 | 0.112887 | 0.0002664 | 0.0002664 | 0.0000444 |

 Column 6 to 10

| 2.758E-08 | 2.758E-08 | 2.758E-08 | 1.683E-08 | 7.947E-09 |

 Column 11 to 15

| 5.569E-08 | 5.556E-07 | 0.0000056 | 0.0000555 | 0.0000543 |

 Column 16

| 0.0054351 |

plot2d(log10(h), log10(err))
\end{verbatim}

and we get once more exactly the same sort of behaviour in the errors.

\textbf{Important Note:} I have already mentioned a few times the importance of understanding the dot operator, \texttt{.}, in Scilab. You will notice that the function \texttt{dg} defined \( y = x.\cdot(-1) \) rather than simply \( y = x\cdot(-1) \); the former allows \( x \) to be a vector, or even a matrix, while the latter will only work if \( x \) is a number.
3 for Loops

Here is a simple example of a for loop:

```matlab
-->
v = zeros(1, 10)
v =

! 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. !

-->for i = 1:10
--> v(i) = i;
-->end

-->v
v =

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

for loops are one place you generally want to terminate statements with semicolons to suppress the printing of intermediate results. Try

```matlab
-->v = zeros(1, 10);

-->for i = 1:10
--> v(i) = i
-->end
```

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

```matlab
-->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. !
```
for loops are often used in functions. Here is a factorial function:

```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
```

It is possible to have for loops within for loops. These are called nested loops and are common when dealing with matrices. The 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 a Scilab function to produce the Hilbert matrix:

```plaintext
-->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
-->end
```
-->endfunction

-->hilbert(5)
ans =

! 1.   0.5   0.333333   0.25   0.2   !
! 0.5  0.333333   0.25   0.2  0.166667   !
! 0.333333   0.25   0.2  0.166667  0.1428571   !
! 0.25   0.2  0.166667  0.1428571  0.125   !
! 0.2  0.166667  0.1428571  0.125  0.1111111   !

4 while Loops

for loops 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 the finite precision of computer arithmetic; we start with \( \text{eps} = 1 \) and repeatedly halve it until \( 1 + \text{eps} \neq 1 \), n.b. \( \neq \) mean 'not equal'.

-->eps = 1;

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

-->eps
eps =

1.110E-16

--> (1 + eps) - 1
ans =

0.

--> (1 + 2*eps) - 1
ans =
2.220E-16

5 if Statements

if statements allow us to perform alternative actions depending on the result of a test. Here is a function which returns the sign of a number:

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

-->signum(12345)
ans =
   1.

-->signum(-12345)
ans =
  -1.

-->signum(0)
ans =
   0.
```