Instructor(s): Dr S. Roberts/Dr L. Stals
Department of Mathematics
Australian National University

2007

ANU Teaching Modules
Scilab Tutorials

Graeme Chandler and Stephen Roberts

Contents

Chapter 1

Introduction to Scilab

1.1 Introduction

These exercises are intended to introduce you to the computer environment SCILAB.

By the end of this tutorial you should have checked that your computer account is working and you will have practiced using an editor and SCILAB.

One way or another you should try to get though all of this tutorial before the next tutorial starts, as it will require some programming work, so do not hesitate to ask for help.

In this session you will mostly gain some experience with the SCILAB package. We will consider some simple SCILAB expressions and use them to consider some problems related to the use of floating point arithmetic.

Along the way, questions will be asked. You should provide answers to these questions in your lab book. Marks for the tutorial will be awarded based on the work presented in your lab book.

1.2 Login

The startup procedure for SCILAB is a little different depending on what type of machine you are on and how it has been setup. But essentially you need to startup a SCILAB session, as well as an editor. For small calculations you can just use the SCILAB interactive command line, i.e. just type commands straight into SCILAB. For anything more complicated it is much more efficient to type your commands into a file (using your favorite text editor), save the file and then execute the commands from the editor (or using the `exec filename` command from the SCILAB command line).

I will go through the procedure in a little more detail for Windows and Mac machines. So choose your system:

1.3 Windows Login

The integration of SCILAB into Windows is quite good. For this operating system SCILAB has an integrated editor. SCILAB is simply run from the application menu off the start menu. At the ANU, SCILAB can be found under Mathematics Software menu item.

Once SCILAB is running you are met with the command window. Startup the editor (from the menu bar) and you are ready to start playing with SCILAB, either from the command line or from the editor.

1.4 Mac Login

Unfortunately the integration of SCILAB into the Mac world is still quite primitive. An editor is not integrated into SCILAB. You must choose your own. And SCILAB itself needs the Xserver running.

At ANU we have setup a text editor, "text commander" in the application folder to help with the integration. If you use this application, you can edit files, and also startup a SCILAB session. This is not as good as the windows environment, but it is a start.

If you don't have access to the ANU system, the procedure for starting SCILAB is a little more complicated. You will first need to ensure that an Xserver is running. This will depend on how you have setup your system. Once you have your Xserver running you will be able to start up an xterm. From the xterm command line, type `scilab&`. This will startup SCILAB. Startup your favorite editor and you are ready to go.

1.5 Talking between Scilab and the Editor

From the SCILAB command window you can run SCILAB commands. In particular the command `pwd` will tell you the directory you are present working in. This is were SCILAB will look for files to execute. You can change this using CHANGE DIRECTORY item under the file menu or from the command line using the command `chdir directory`.

From your text editor create some SCILAB commands such as

```
A = [ 1 2 3 ; 4 5 6 ; 7 8 9]
```

and then save the file (say into file `test-01.sce`) into the same directory in which SCILAB is looking (as shown by the `pwd` command).

Now we can run the commands in the file via the command at the SCILAB prompt

```
exec test-01.sce
```

As you make changes to your file, you will need to save those changes and then run the `exec` command again (you can save typing by using the up arrow to recall previous commands).

If you are using the editor integrated into SCILAB on the windows system, then the editor automatically asks to save the file you are working on, and automatically executes the file.

1.6 Basic Commands

Now let us play with SCILAB.

Move the pointer to the SCILAB window and select the window (press the left mouse button). Now you should be able to type commands into the SCILAB window. SCILAB is made to easily work with vectors and matrices. First input a vector

```
x = [ 0 ; 2 ; 5]
```

and then a matrix

$$A = \begin{bmatrix} 1 & 3 & 4 \\ -1 & 2 & 5 \\ 4 & -3 & 5 \end{bmatrix}$$

via the command

```
A = [ 1  3  4 ; -1  2  5 ; 4  -3  5 ]
```

The semi-colons may be replaced by returns.

SCILAB is case sensitive so that for example `a` and `A` will represent different variables. All the built-in functions in SCILAB like the rank or the eigenvalues for a matrix, are denoted by lower-case names, as in `rank(A)` or `spec(A)`. If you cannot remember how to use a particular SCILAB command try the `help` command in SCILAB. Try

```
help spec
```

SCILAB provides many vector and matrix operations. For instance we can multiply matrices and vectors. Try the simple commands

```
A*x
A*A
```

Lab Book: Record the results of these calculations.

SCILAB does all its calculation in double precision but formats the output using a short format. The format can be changed to show full precision with the SCILAB command

```
format(20,'e')
```

Try some calculations to see the different format.

The format can be changed back using

```
format(10,'v')
```

A semi-colon at the end of a line will stop SCILAB output for that line. This is useful if you are working with large vectors or matrices. Try

```
y = A*x ;
```

Note that the answer is not printed out. To see `y` just type

```
y
```

Let's make some larger vectors. It is easy to produce vectors that have components which increment nicely. Try

```
w = 0:0.1:7
```

This will produce a vector starting at 0 and increasing by 0.1 until 7 is reached. Note that a semi-colon at the end of the line would be helpful here.

We can take functions of vectors (or matrices). For instance try

```
z = sin(w)
```

Now we have two vectors `w` and `z` of the same length. We can plot the values of `w` versus the values of `z`. Try

```
plot2d(w,z)
```

A plot of `w` versus `z` should be produced. You can save the graph to a file by using the file—export menu item on the graphics window.

Lab Book: Print this plot and paste into your lab book

1.7 Matrix Vector problems

We can also solve matrix problems. Try

```
y = A \ x
```

The vector `y` should solve the linear equation `x = A*y` (check this). The inverse of a matrix can also be calculated using the `inv` command. Use the `inv` command to solve the matrix equation $A\mathbf{y} = \mathbf{x}$.

Lab Book: Compare the results of `A\x` and `inv(A)*x`. Are they equal? Why?

The Scilab command `rand(n,m)` produces a random $n \times m$ matrix. Redefine `A` to be a random 5×5 matrix and `x` a random 5×1 vector.

The `\` command is more general and can be used to solve over-determined systems (systems with more equations than unknowns) by finding a "least squares solution". In fact under-determined systems can also be accommodated.

Lab Book: Try solving an under-determined system (Matrix has less rows than columns). Try an over-determined system (more rows than columns). Document your experiments.

1.8 Loops and Conditionals

SCILAB provides `for`, `while` and `if` statements to control flow within a program. Consider the example program for calculating e^x.

```
ans = 0; n = 1; term = 1;
while( ans + term ~= ans )
        ans = ans + term;
        term = term*x/n;
        n = n + 1;
end
ans
```

Here `~=` means "not equal to".

We can avoid retyping by saving the program in a file. Open up a text editor. Type in the program. Save to a file using the file menu. You will be prompted for a file name. Use the name ex.sci, the `.sci` is a standard extension for SCILAB function files. Go back to the SCILAB window and test your program `ex`. Type in a value for `x` and then issue the command `exec('ex.sci')`. For instance try

```
x = 1.0
exec('ex.sci')
```

to let SCILAB run your program with `x = 1.0`.

Lab Book: Using your program, what is the approximation to e^1, e^{10} and e^{-10}.

We can make our program into a function. Return to the editing window and change the program as follows:

```
function y = ex(x)
// EX A simple function to calculate exp(x)
y = 0; n = 1; term = 1;
while( y + term ~= y )
        y = y + term;
        term = term*x/n;
        n = n + 1;
end
endfunction
```

The lines starting with `//` are comment lines.

Save your changes. Now you can use the command `ex(x)`. But you must load in your changes. Try

```
exec('ex.sci')
ex(1.0)
```

Alternatively if you are using the editor supplied with SCILAB you can just use the menu item "execute" to load in your file.

This algorithm is useless for large negative values of `x`. On the other hand, as in the lectures we can use this algorithm as a basis for a more robust algorithm. First though we need to become familiar with the `if` statement. The SCILAB `if` statement has the simple form

```
if expression then
   Scilab statements
else
   Scilab statements
end
```

By the way, SCILAB has a `for` statement of the form

```
for i=v
  Scilab statements
end
```

Here `i` is a dummy variable to be used in the loop, and `v` is a vector, usually a range of numbers defined `1:20`, that the dummy variable will iterate through. So

```
for j=-4:2:6
  disp(j**2)
end
```

will print out the squares of the even numbers from -4 to 6.

Use your function `ex` and the `if` statement to produce a new function `newex` which produces reasonable approximations of `e^x` for `x` positive or negative (as demonstrated in the lectures). You can use the same file to define both the `ex` and the `newex` functions, but you will have to use `exec` again to load in the new function `newex`.

Lab Book: Produce a table of the results of the ex, and newex functions compared to the inbuilt exp SCILAB function. Do the table for a range of large negative through to large positive numbers.

1.9 The Help Command

The help command is the easiest way to find out more about specific Scilab commands. If you have forgotten small details, for example. The command `help name` gives information about the Scilab command `name`.

```
help sin     // Information about sin.
help +       // Gives links to help on Scilab operator names
help log     // This is enough information about log
             //     to show  log  means log to the base  e .
```

Note that often the help information provides an example of how the command is used. This can be particularly useful when experimenting with a new command. You can use cutting and pasting to run the example commands.

In Scilab the `apropos` command can also be used to search for relevant information.

```
apropos logarithm  // Provides a list of
                   // functions related to logarithms
```

1.10 The Help Window

It is also possible to get help by clicking on the HELP menu above the command window. From the HELP menu, select the HELP DIALOG item and the list of help topics is displayed. The advantage of this method is that it is possible to navigate around different topics and zero in on useful commands. For example, in the help dialog window click on the chapter 'Linear Algebra'. Then find the item 'linsolve'. Once found you click on 'show' to see the relevant help page. Note that double clicking doesn't work in windows Scilab version 2.6 and there is a slight bug when you change chapters and scroll to new items. Choosing the item a second time seems to work.

Generally the help window is a good way to explore Scilab's commands.

Exercise 1.

1. *Is the inverse sine function one of Scilab's elementary functions? (The inverse sine is also known as sin^{-1}, arcsin, or asin.)*
   (a) *How do you find* $\sin^{-1}(.5)$ *in Scilab? Use help to find out.*
   (b) *If* $x = .5$, *is* $\sin(\sin^{-1}(x)) - x$ *exactly zero in Scilab?*
   (c) *If* $x = \pi/3$, *is* $\sin^{-1}(\sin(x)) - x$ *exactly zeros in Scilab? What about* $x = 5\pi/11$*?*
2. *Does* SCILAB *have a function to convert numbers to base 16, i.e. to hexadecimal form? (Hint: Use* `apropos` *to find a way to* ***convert*** *a* ***decimal*** *number to* ***hexadecimal****.) What is 61453 in base 16? Computers almost always represent numbers internally as hexadecimals.*
3. *Look for information about logarithms. Note that searching for logarithms fails where as logarithm succeeds. There are 8 entries, including the command,* `logm`*, for calculating the log* ***of a matrix****!*
4. *To wind down, use the* SCILAB *menu command* DEMOS*. This brings a menu of demonstrations and examples to explore.*
   (a) *Visit the graphics to see a number of attractive images.*
   (b) *I also like the car and trailer parking demo.*

Lab Book: Record your results in your lab book.

1.11 Summary

We have tried to give a very quick idea of the facilities available in SCILAB and to the give an idea of using the SCILAB program in conjunction with a text editor to efficiently experiment with matrix problems. Also the demos available from the menu gives some good ideas.

But now see if you can tackle the following problem.

1.12 Exercises

Exercise 2. *Write a* SCILAB *program* `quadroots` *to compute and print the roots of a quadratic equation* $ax^2 + bx + c = 0$ *using the quadratic formula*

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

It should run with a command like

```
quadroots(1,3,2)
```

for the case of $x^2 + 3x + 2 = 0$. *This program can be made to return both roots in a vector by calling them, for example, root1 and root2 in the function file and starting the function definition with the line*

```
function [root1, root2] = quadroots(a,b,c)
```

These can then be stored into two variables such as r1 and r2 by running the function with the command

```
[r1 r2] = quadroots(1,3,2)
```

The program should avoid division by zero, printing an appropriate message in that case. Make your own decision about what to do in the case of complex roots, since SCILAB *can compute and print complex values. Test* `quadroots` *on the following examples, checking the results in each case, and pay particular attention to the last example.*

1. $x^2 + 1 = 0$
2. $0x^2 + 2x + 1 = 0$
3. $x^2 + 3x + 2 = 0$
4. $4x^2 + 24x + 36 = 0$
5. $10^{18}x^2 - x - 1 = 0$
6. $10^{-18}x^2 - x + 1 = 0$ *(The roots are approximately* $1 + 10^{-18}$ *and* $10^{18} - 1$*), which you can check by hand.)*

The last example above illustrates the problem of catastrophic cancellation. Explain why it makes one root highly inaccurate but not the other, and devise a better algorithm for computing both roots accurately.

You can use the fact that the roots x_1 *and* x_2 *of* $ax^2 + bx + c = 0$ *always satisfy* $x_1 x_2 = c/a$.

Write a modified program `quadroots2` *using this improved method, and test it on the same cases as above.*

Unit Testing

It is very useful to write a test program which will test a program against known results. Here is a program `test_quadroots` *together with some auxiliary functions which takes a quadroots function (*`quadroots` *or* `quadroots2`*) and tests the program with the examples given above. This is known as unit testing. Copy the following code into your* Scilab *environment, and test your* `quadroots` *functions with the command* `test_quadroots(quadroots)`.

```
//-------------------------------------------------
//
// Test program for the quadroots function
//
// Test a range of inputs.
// Look at the expected results in the table
// to see how to code your function
//-------------------------------------------------
function test_quadroots(quadroots)


n=0
table = zeros(1,5);

//-------------------------------------
// Setup tests, a,b,c r1,r2 in a table
// r1 and r2 are expected roots. We are using
// %nan for non real roots
//-------------------------------------
// x^2 + 1 = 0
n=n+1; table(n,:) = [ 1.0, 0.0, 1.0 %nan, %nan]

// 0x^2 + 2x + 1 = 0
n=n+1; table(n,:) = [ 0.0, 2.0, 1.0, -0.5, %nan]

// x^2 + 3x + 2 = 0
n=n+1; table(n,:) = [ 1.0, 3.0, 2.0, -1.0, -2.0]
```

```
// 4x^2 + 24x + 36 = 0
n=n+1; table(n,:) = [ 4.0, 24.0, 36.0, -3.0, -3.0]

//10^18 x^2 - x - 1 = 0
n=n+1; table(n,:) = [ 1.0e18, -1.0, -1.0, 1.0000000005e-9, -9.999999995e-10]

//10^-18 x^2 - x + 1 = 0 (The roots are approximately 1 + 10e-18 and 10^18 - 1))
n=n+1; table(n,:) = [ 1.0e-18, -1.0, 1.0, 1.0e18, 1.0]


//--------------------------------------
// Test each problem.
//--------------------------------------
n = size(table,1)
success = 0.0
for i=1:n,
  success = success + ...
     print_test(table(i,1),table(i,2),table(i,3),table(i,4),table(i,5),quadroots)
end
printf('================================================\n')
printf('%g successful tests out of %g\n',success,n)

endfunction

//-------------------------------------------
// Print whether specific quadtest is correct.
// Returns 1.0 if correct, 0.0 otherwise
//-------------------------------------------
function success = print_test(a,b,c,r1,r2,quadroot)
success = 0.0

printf('================================================\n')
printf('TEST: %g x^2 + %g x + %g = 0\n',a,b,c)
printf('Expect   r1 = %20.15e, r2 = %20.15e \n',r1,r2)
[c1,c2]= quadroots(a,b,c)
printf('Obtained r1 = %20.15e, r2 = %20.15e\n',c1,c2)

if( (isnan(r1) & isnan(c1)) & (isnan(r2) & isnan(c1))) then
    printf('TEST PASSED\n')
    success = 1.0
    return
end
```

```
if( (isnan(r2) & isnan(c2)) & close(r1,c1) ) then
    printf('TEST PASSED\n')
    success = 1.0
    return
end

if (close(r1,c1)&close(r2,c2)) then
    printf('TEST PASSED\n')
    success = 1.0
    return
end

if (close(r1,c2)&close(r2,c1)) then
    printf('TEST PASSED\n')
    success = 1.0
    return
end

success = 0.0
printf('TEST FAILED\n')

endfunction

//-------------------------------------
// Check whether two values are relatively
// close.
//-------------------------------------
function r = close(x,y)

xx = max(abs(x),abs(y))

if xx ~= 0.0 then
    r = (abs(x-y)/xx)<%eps*2
else
    r = abs(x-y)<%eps*2
end
return


endfunction
```

Evolve your `quadroots` *program to a stage in which it successfully passes all of the* `test_quadroots` *tests.*

Lab Book: Add to the `test_quadroots` function a test for the quadratic $10^{-18}x^2 - x + 1 = 0$ which should have roots approximately $1 + 10^{-18}$ and $10^{18} - 1$. Copy your commented final code, with the functions `quadroots`, `quadroots2` and your amended `test_quadroots` function into your lab book. Present the results of the `test_quadroots` function showing the success rate of your various implementations of `quadroots`.

Chapter 2

Matrix Calculations

2.1 Introduction

This tute is designed to familiarise you with matrix calculations in SCILAB.

Although SCILAB is a useful calculator, its main power is that it gives a simple way of working with matrices and vectors. Indeed we have already seen how vectors are used in graphs.

Remember to keep typing in the commands as they appear here, and observe and understand the SCILAB response. If you mistype, it is easy to correct using the arrows and the [Del] key. Try to use the help facility to find out about unfamiliar commands. Otherwise ask another student or a tutor.

2.2 Solving Equations

Most people will have seen systems of equations from school. For example, we may need to find x_1, x_2, and x_3 so that

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 1 \\ -2x_1 - 6x_2 + 4x_3 &= -2 \\ -x_1 - 3x_2 + 3x_3 &= 1 \end{aligned}$$

Although these problems can be solved manually by eliminating unknowns, this is unpleasant. Besides errors usually occur.

In first year Mathematics the problem is rewritten in matrix-vector notation. We introduce a matrix A and a vector b by

$$A = \begin{bmatrix} 1 & 2 & -1 \\ -2 & -6 & 4 \\ -1 & -3 & 3 \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}.$$

Now we want to find the solution vector $x = [x_1, x_2, x_3]$ so that

$$Ax = b.$$

In spite of the new notation, it is still just as unpleasant to find the solution. In SCILAB, we can set up the equations and find the solution x using simple commands.

```
                                    // Set up a system
A = [ 1 2 -1; -2 -6 4 ; -1 -3 3 ]   // of equations.
b = [ 1; -2; 1 ]
x = A\b                             // Find x with A x = b.
```

SCILAB should give the solution

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}.$$

A sound idea from manual computations is to substitute the computed solution back into the system, and make sure all equations are indeed satisfied. In SCILAB we can do this by checking that the matrix vector product Ax gives the vector b, or better still, we check that $b - Ax$ is exactly the zero vector.

```
A*x          // Check that  Ax  and  b  are the same.
b
b - A*x ;    // So that we do not miss small differences,
             //  it is better to check that  b - A x = 0 .
```

The vector $b - Ax$ is known as the *residual*.

Lab Book: Change the middle element of A from -6 to 5 (i.e. $a_{22} = 5$). What is the solution to this new system? What is the new residual? Why is the residual not exactly 0?

Hint: In SCILAB the number 1.23×10^{-5} is displayed as `1.23e-05`. The column vector $[1.23 \times 10^{-5}; 4.44 \times 10^{-6}]$ could be displayed as

```
1.0e-004 *                          1.2300e-05
    0.1234          or              4.4400e-06
    0.0444
```

The user can control which form is used by using the command `format`. Essentially the full accuracy can be seen using `format('e',20)`.

Lab Book: Suppose the middle element is changed from -6 to -5. Can SCILAB solve this third system? Explain what has happened.

2.3 Matrices and Vectors

Much scientific computation involves solving very large systems of equations with many millions of unknowns. This section gives practice with the commands that are needed to work larger matrices.

The following code sets up a 4×4 matrix, A, and then finds some of its important properties. When `[ ]` is used to set up matrices, either blanks or commas (`,`) can be used to separate entries in a row. Semicolons (`;`) are used to begin a new row.

```
A =  [1 2 3 4 ; 1 4 9 16 ; 1 8 27 64 ; 1 16 81 256 ]
A'
det(A)
spec(A)
[D, X] = bdiag(A)
inv(A)
```

Even in the simple 4×4 case, it is tedious to evaluate determinants and inverses. But in Scilab they can be quickly calculated and used.

Indices can be used to show parts of a larger matrix. For example try

```
A(2,3),  A(1:2,2:4),   A(:,2),  A(3,:),   A(2:$,$)
```

In general `A( i:j, k:l )` means the square sub–block of `A` between rows i to j and columns k to l . The ranges can be replaced by just ':' if all rows or columns are to be included. The last row or column is designated `$`.

Lab Book: How would you display the bottom left 2×3 corner of A? How would you find the determinant of the upper left 3×3 block of A?

Lab Book: Find the SCILAB command to calculate the row echelon form of a matrix. The row echelon form is never useful in practice, but it is handy for checking homework in other subjects!!

2.4 Creating Matrices

Scilab also provides quick ways to create special matrices and vectors.

```
c  =  ones(4,3)
d  =  zeros(20,1)
I  =  eye(5,5)
D  =  diag( [2 1 0 -1 -2] )
L  =  diag( [1 2 3 4], -1 )
U  =  rand(5,5)              // U is a  5 x 5 matrix
                             // of uniformly distributed random numbers.
R  =  rand(5,5,'normal');    // R is a  5 x 5 matrix
                             // of normally distributed random numbers.
```

(More information on each of these commands can be found with `help`.)

Matrix-matrix and matrix-vector multiplication work as expected, provided the dimensions agree. Matrices and vectors can both be multiplied by scalars. In the next example, `B` is another 4×4 matrix and `c` is a column vector of length 4 (i.e. a 4×1 matrix).

```
B = [1 1 0 0
     0 2 1 0
     0 0 3 1
     0 0 0 4 ]        // Rows can be separated by
                      // making a new line as well as using ';'.
c = [1; 0; 0; -1]
5*B                   // Multiply scalars and matrices.
B*c                   // Multiply matrices and vectors.
A*B                   // Matrix by matrix multiplication.
B*A                   // Note: AB is not the same as BA !
```

Some of the following commands do not work. There is no harm in trying them.

```
c*c        // Wrong dimensions for matrix multiplication.
c*A        // Wrong dimensions for matrix multiplication.
c'         // The transpose of c, i.e. a row vector.
c'*A       // Now matrix multiplication is permitted.
c*c'       // This multiplies a  4 x 1  by a  1 x 4  matrix.
c'*c       // What is this quantity usually called?
```

Lab Book: Calculate `inv(A)*A` and `A*inv(A)`. Do they give the expected result? (Use the help command if you can't guess what the `inv` command does.)

Lab Book: Verify for the matrices A and B above that $(AB)^{-1} = B^{-1}A^{-1}$.

Lab Book: Verify that the SCILAB command `A^(-1)` can also be used to form the inverse of A.

2.5 Systems of Equations

Consider again the problem of solving the system of equations $Ax = b$. One obvious way that appeals to Mathematicians is to calculate $A^{-1}b$. As in the previous 3x3 case, we should also check that the solution is correct. We do this by calculating the residual $Ax - b$, for our computed solution x. Because of roundoff errors this residual may not be exactly zero, but all its components should be small; say less than 1×10^{-15}.

```
x = A^(-1)*b        // Solve the equation  A x = b A*x , b
                    // Check  x  is correct by making
```

```
                    //   sure it satisfies the equations.
resid = b - A*x     // Check the residual is zero,
                    //   at least to within roundoff.
```

In fact it is far quicker, and usually more accurate, to solve equations using the backslash operator (\ ) introduced in the first section; rather than calculating with the inverse A^{-1}. The next lines use the backslash command to the solve the equations $Ax = b$, and check that it gives the same answers as those that were calculated using A^{-1}.

```
x1 = A \ b    // This is the fastest way to solve  A x = b
x - x1        // Here both answers are the same (up to roundoff error).
```

Lab Book: Try the following exercises.

Exercise 3.

1. *Solve the system of equations*

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix} x = \begin{bmatrix} 1/4 \\ 1/5 \\ 1/6 \end{bmatrix}$$

   using `\`. *Check that the computed solution does actually satisfy the equations (i.e. check that* $b - Ax = 0$, *apart from rounding errors).*

2. *Use* `rand(n,m,'normal')` *to generate a 700 x 700 matrix,* `A`, *and a column vector* `b` *of length 700. Solve the system of equations* $Ax = b$ *using the commands* `x = A\b` *and* `x = inv(A)*b`, *timing each command with the* `timer()` *command. Which command is the faster and why?*

   Hint: Before working with these large matrices use the command `stacksize(m)` *to increase the size of the memory used by* Scilab, *here* `m` *is the number of words used in memory. I would suggest using* `m = 1e7`.

3. *Set up the* 10×10 *Hilbert matrix. The* ij *component is given by* $1/(i + j - 1)$. *You can use* `feval` *to produce the matrix given the function*

```
function z=f(x,y)
z=1.0/(x+y-1)
endfunction
```

   Create a right hand side vector b *which is the first column of* A, `b = A(:,1)`. *We are now going to solve the equations* $Ax = b$. *This is a common test problem in linear algebra. (What is the true solution* x *to the equations* $Ax = b$*?)*

(a) *Solve the equations $Ax = b$ using the backslash command, and then check that the computed solution satisfies the equations by calculating the residual, $b - Ax$.*

(b) *Now solve the equations by the command* `A^(-1)*b` *and calculate the residual for this second solution.*

(c) *Which gives the better (i.e. smaller) residual?*

(d) *Which method gives the smaller error?*

Explanation: As b is the first column of A, you can work out the true value of x without computation. (Explain this briefly.) The error is the difference between Scilab's computed solution and the true solution. (The true solution, the one you work out by abstract thought, can be typed directly into Scilab.) The size of the error is the largest absolute value of the components of the error vector.)? (Also SCILAB *provides the exact inverse of the Hilbert matrix with the command* `testmatrix('hilb',n)`.

Chapter 3

Data and Function Plots

3.1 Introduction

This tutorial is designed to familiarise you with plotting data and graphing functions in SCILAB.

3.2 Plotting Lines and Data

This section shows how to produce simple plots of lines and data.

Suppose we wish to plot some points. For example we are given the following table of experimental results.

x_k	.5	.7	.9	1.3	1.7	1.8
y_k	.1	.2	.75	1.5	2.1	2.4

To work with the data in SCILAB set up two column vectors `x` and `y`.

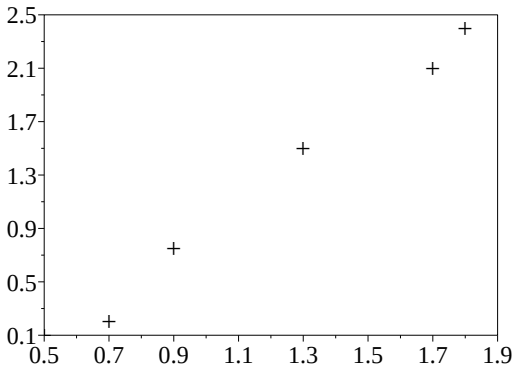
```
x = [.5 .7 .9 1.3 1.7 1.8 ]'
y = [.1 .2 .75 1.5 2.1 2.4 ]'
```

(Vectors are discussed in detail in the next lesson; but we can use them to draw graphs without knowing all the details.) To graph y against x use the plot command.

```
plot2d(x,y, style=-1)
```

The graph should now appear. (If not, it may be hidden behind other windows. Click on the icon 'Figure No. 1' on the Windows task bar to bring the graph to the front.)

This graph marks the points with an '+'. Other types of points can be used by changing the `style=-1` in the plot2d command. (Use `help plot2d` to find out the details.)



Plot showing data points

Lab Book: Experiment with different values for the style. Negative values give markers, positive values coloured lines.

As you experiment you should erase the previous plot. Use the command

```
xbasc()
```

Commands starting with `x` generally are associated with graphics commands (This comes from the X window system used on Unix machines). So `xbasc()`, is a **bas**ic graphi**x** command which **c**lears the graphics window.

From the graph it is clear that the data is approximately linear, whereas this is not so obvious just from the numbers. Good graphs quickly show what is going on!

When you have finished looking at the graph, just click on any visible part of the command window. More commands can then be typed in.

Lab Book: Plot the above data with the points shown as circles. You can either do this via the style argument, but also via the edit menu on the figure

Lab Book: Plot the above data with the points joined by lines.

Hint: Use `help plot2d`).

Lab Book: Find out how to add a title to the plot.

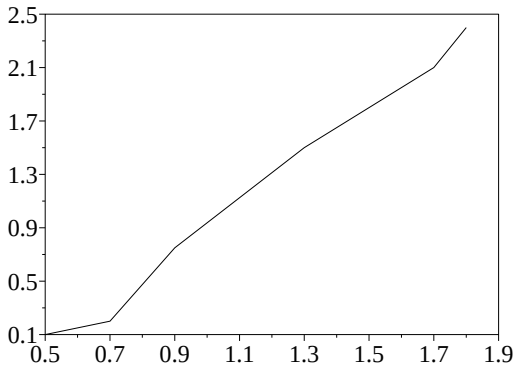
Hint: Use the command `apropos title`.)

3.3 Plot data as points

Often people use the very simplest command to plot data and automatically type just

```
plot2d( x, y )          // The simplest plot command
```

This simple command does not present the data here very well. It is hard to see how many points were in the original data. It is really better to plot just the points, as the lines between points have no significance; they just help us follow the set of measurements if there are several data sets on the one graph. If we insist on joining the points it is important to mark the individual points as well.



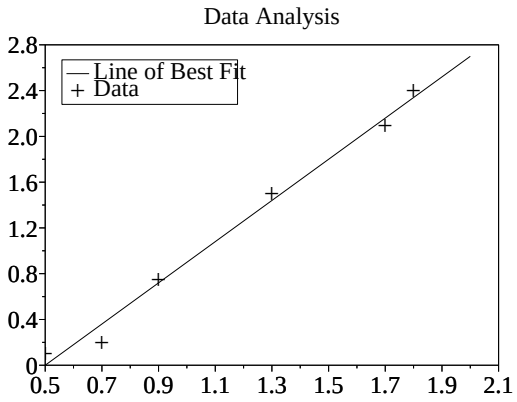
Simple `plot2d` command.

3.4 Adding a Line

From the graph it is clear that the points almost lie on a straight line. Perhaps the points are off the line because of experimental errors. A course in statistics will show how to calculate a 'line of best fit' for the data. But even without statistics, the line between the points $(.5, 0)$ and $(2, 3)$ is a good candidate for 'a line fitting the data'. Lets add this line to the plot and see how well it approximates the data. We do this by asking Scilab to plot the points $(.5, 0)$ and $(2, 2.7)$ joined by a line.

```
xbasc()
x = [.5 .7 .9 1.3 1.7 1.8 ]';
y = [.1 .2 .75 1.5 2.1 2.4 ]';
plot2d(x,y,style=-1)
x_vals = [ .5 2]';        //  The X-coords of the endpoints.
y_vals = [ 0 2.7 ]';      //  The Y-coordinates
plot2d(x_vals,y_vals)
legends(['Line of Best Fit';'Data'],[1,-1],5)
xtitle( 'Data Analysis')
```

Note that `x_vals` contains the x–coordinates, `y_vals` contains the y–coordinates, and the two points are joined by a line because we don't specify a style (default style is a line).



The line of best fit can be found using the `datafit` function (we will come back to this later in the course).

Lab Book: Produce this plot and add to your lab book

3.5 Hints for Good Graphs

There are many opinions on what makes a good graph. From the scientific viewpoint a simple test is to see whether we can recover the original data easily from the graph. A graph is supposed to add something, not remove information. For example if our data runs from 1 to 1.5 it is a bad idea to use an axis running from 0 to 10, as we will not be able to see the differences between the values. We give two more subtle recommendations.

Lab Book: Change the above plot commands to show the data more clearly by plotting the data points as small circles joined by dotted lines. Hint:

Use `help plot2d` to see the possible arguments.

3.6 Choose a good scale

In the example in this section, it was easy to see the relationship between x and y from the simple plot of x against y. In more complicated situations, it may be necessary to use different scales to show the data more clearly. Consider the following model results

n	3	5	9	17	33	65
s_n	.257	.0646	.0151	3.96×10^{-3}	9.78×10^{-4}	2.45×10^{-4}

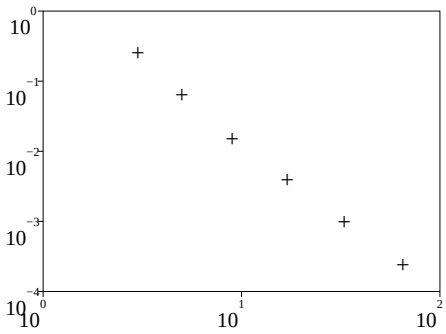
.

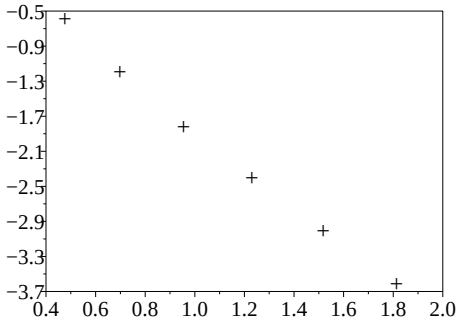
A plot of n against s_n directly shows no obvious pattern. (Note the dots, '. . .', in the next example mean the current line continues on the next line. Do not type these dots if you want to put the whole command on the one line.

```
n = [ 3 5 9 17 33 65 ]';
s = [ 2.57e-1  6.46e-2  1.51e-2 ...
             3.96e-3 9.78e-4  2.45e-4 ]' ;

xbasc()                        // clear the previous plot
plot2d( n, s, style=-1 )       // This is a poor plot!!
```

In fact it is hard to read the values of s_n back from the graph. However a plot of $\log(n)$ against $\log(s_n)$ is much clearer. To produce a plot on a log scale we can use either of the commands

```
xbasc(); plot2d( log10(n), log10(s), style=-1)
xbasc(); plot2d( n, s, style=-1 , logflag = 'll')
```





These plots reveal that there is almost a linear relationship between the logs of the two quantities.

Lab Book: Using `help plot2d` find a command which will use a log scale only for the s_n data. Is this better or worse than the log-log plot?

Lab Book: Add a title to the last plot above and label the X and Y axes.

3.7 Graphs

Consider the problem of graphing functions, for example the function

$$f(x) = x|x|/(1 + x^2)$$

over the interval $[-5, 5]$.

3.8 Function Plotting

Scilab provides a simple method for defining functions. For simple functions, the definition can be written at the interactive prompt "on line". For instance to define the function

$$f(x) = x|x|/(1+x^2)$$

we would write

```
function [y]=f(x)
y = x*abs(x)/(1+x^2);
endfunction
```

This will define a Scilab function with the name `f`. We can then evaluate the function as such

```
f(3)
```

To produce a plot of this function we can use the `fplot2d` command. For instance to plot the function f in the interval $[0, 1]$ we use

```
x = (-5:0.1:5)';
fplot2d(x,f)
```

The first command produces a column vector of x values from 0 to 1 in steps of .1. The second produces the plot of the function `f`.

3.9 Component Arithmetic

we would like to be able to do componentwise arithmetic. This is how we do it.

```
x = (-5:.1:5)' ;
y = x .* abs(x) ./ ( 1 + x.^2) ;
plot2d( x , y )
```

The first command produces a vector of x values from -5 to 5 in steps of .1. That is the column vector $\mathbf{x} = [-5, -4.9, \cdots, 0, \cdots 4.9, 5]'$. The vector $\mathbf{y}$ contains the values of f at these $\mathbf{x}$ values. As there are so many points the graph of $\mathbf{x}$ against $\mathbf{y}$ looks like a smooth curve.

The novelties here are the operators `.*` , `./` and `.^` in the second command. These are the so called *component–wise operators.* In the above example `x` is a column vector of length 101 and `abs(x)` is the column vector whose i^{th} entry is $|x_i|$. The formula `x*abs(x)` would be wrong, as this tries to multiply the 1×101 matrix `x` by the 1×101 matrix `abs(x)`. However the component-wise operation `x.*abs(x)` forms another vector of length 101 with entries $x_i|x_i|$. Continuing to evaluate the expression using component-wise arithmetic, we get the vector `y` of length 101 whose entries are $x_i|x_i|/(1+x_i^2)$. (Of course we should not

become overconfident. This rule for dividing by vectors cannot be used in Mathematics proper.)

Note that `p.*q` and `p*q` are *entirely different*. Even if `p` and `q` are matrices of the same size and both products can be legitimately formed, the results will be different.

Before more exercises, we show how to add further curves to the graph above. Suppose we want to compare the function we have already drawn, with the functions $x|x|/(5+x^2)$ and $x|x|/(\frac{1}{5}+x^2)$. This is done by the following three additional commands. (Remember `x` already contains the x–values used in the plot. These new commands are most easily entered by editing previous lines.)

```
y2 = x .* abs(x) ./ (5 + x.^2) ;
y3 = x .* abs(x) ./ ( 1/5 + x.^2) ;
plot2d(x,[y y2 y3])
```

Note that `x` needs to be a column vector for this to work.

Lab Book: Use SCILAB to graph the functions $\cos(x)$, $1/(1+\cos^2(x))$, and $1/(3+\cos(1/(1+x^2))$ on separate graphs.

Lab Book: Graph $1/(1+e^{\alpha x})$, for $-4 \le x \le 4$ and $\alpha = .5, 1, 2$, on the one plot.

Lab Book: Use `plot2d` to draw a graph that shows

$$\lim_{x\to 0} \frac{\sin(x)}{x} = 1.$$

Hint: Select a suitable range and a set of x values and plot $\sin(x)/x$. If you get messages about dividing by zero, it means you tried to evaluate $\sin(x)/x$ when x is exactly 0.

3.10 Printing Graphs

When the graph has been correctly drawn we will usually want to print it. As many graphs come out on the printer, it is a good idea to label the graph with your name. To do this use the `xtitle` command.

```
                                    \\ Be sure to put quotes
xtitle('Exercise 3.4 Kim Lee.')     \\     around the title.
```

Bring the plot window to the front. The title should appear on the graph. If everything is okay, click on the 'file' menu → 'print scilab' item in the graph window. If everything is set up correctly, the graph will appear on the printer attached to your computer.

After printing a graph, it is useful to take a pen and label the axes (if that was not done in SCILAB), and perhaps explain the various points or curves in the space underneath. Alternatively before printing the graph, use the SCILAB command `xtitle` with two optional arguments and the legend argument of the `plot2d` command for a more professional appearance. For instance

```
xbasc()
plot2d(x,[y y2 y3],leg='function y@function y2@function y3')
xtitle('Plot of three functions','x label','y label')
```

Another option is to use the graphics window file menu item, export, or the `xbasimp` command to print the current graph to a file so that it can be printed later. For example to produce a postscript file use the command

```
xbasimp(0,'mygraph.eps')
```

This will write the current graph (associated with graphics window 0) to the file `mygraph.eps`. This file can be stored and printed out later.

3.11 Saving Graphs

The easiest way to save a graph is to save the Scilab command you used to create the graph. These are usually only one or two lines. If the commands are more than one or two lines long, they should probably be saved in a file (see later chapters.)

Chapter 4

Polynomials and Interpolation

4.1 Introduction

This tutorial is designed to familiarise you with polynomial evaluation and interpolation and with least squares fitting of data with polynomials.

4.2 Evaluation of Polynomials

Use SCILAB to calculate the following functions in the obvious manner without rearranging or factoring

$$\begin{aligned} f(x) &= x^8 - 8\,x^7 + 28\,x^6 - 56\,x^5 + 70\,x^4 - 56\,x^3 + 28\,x^2 - 8\,x + 1 \\ g(x) &= ((((((((x-8)\,x + 28)\,x - 56)\,x + 70)\,x - 56)\,x + 28)\,x - 8)\,x + 1 \\ h(x) &= (x-1)^8 \end{aligned}$$

at the points `0.975:0.0001:1.025`.

Lab Book: Plot these points using SCILAB. Observe that the three functions are identical mathematically but that the numerical results are quite different. Can you account for this discrepancy.

SCILAB *Hint:* You might want to create a file where the three functions, f, g and h are defined. In SCILAB the command `x^4` will try to raise the matrix `x` to the 4th power (using matrix multiplication). On the other hand the "dot" operator `x.^4` will apply the operation `^` to each entry of the matrix `x`. It is recommended that you always use the "dot" notation when you define such functions.

4.3 Polynomials

We can use SCILAB procedures to deal with polynomials. The polynomial $a_1 + a_2 x + \cdots + a_m x^{m-1}$ can be represented in SCILAB symbolically.

In the following example, $-4 - 3x + x^2$ is represented by the vector of coefficients $[-4\ -3\ 1]$ and by a polynomial p. We setup the polynomial with the command

```
v = [-4,-3,1]
p = poly(v,'x','coeff')
```

We can also build up a polynomial in a more natural way.

```
x = poly(0,'x')       // Seed a Polynomial using variable 'x'
p = x^2 - 3*x - 4     // Represents x^2 - 3 x -4
```

There are many useful functions that can be applied to polynomials. We can find the roots (numerically) of a polynomial, evaluate a polynomial or find its derivative.

```
z = roots( p )
z(1)^2 -3*z(1) -4     //  Two ways to evaluate the poly.
horner(p,z(1))        //    at the first root.
derivat(p)            // Calculate the derivative of the polynomial
```

The Scilab polynomial methods generalise to rational functions. For instance

```
x = poly(0,'x')       // Seed a Polynomial using variable 'x'
p = x^2 - 3*x - 4     // Represents x^2 - 3 x -4
r = x/p               // Represents the rational
                      //    function x/(x^2 - 3 x -4)
horner(r,1.0)         // Evaluate r at x=1
```

4.4 Interpolating Runge's Function

Consider the function

$$f(t) = \frac{1}{1 + 25t^2}.$$

This function is known as Runge's function.

Define a Scilab function to calculate Runge's function.

```
function y=runge(t)
//
// Runge's Function
// Standard example of polynomial interpolation
// creating oscillations
//
y=(1.0)./(1+25*t.^2)
endfunction
```

Note we have put parentheses around the `(1.0)` to ensure that the `./` command is used. An expression of the form `1./(..)` would actually use the `/` matrix operator!

Consider a polynomial

$$p_n(t) = \sum_{j=1}^{n} x_j t^{j-1}$$

which interpolates (coincides with) Runge's function at n evenly spaced points between -1 and 1 (i.e. at the points `-1:2/(n-1):1|`. We have n unknowns x_i for the coefficients of the polynomial and n data points. This will provide us with a linear equation for the polynomial coefficients given the values of the data points.

Create a function called `polyfit`, with calling sequence `pp=polyfit(tdata, ydata)`, that fits a polynomial of degree $n-1$ to n data points `t`, `y` and returns a polynomial `pp`. You can easily add an extra argument to calculate the least square fit of an $m < n$ degree polynomial to the n data points. The result of `polyfit` should then be able to be evaluated using the command

```
tt = -1:0.01:1; yy = horner(pp,tt)
```

Hint: You can use the standard backslash operator to solve for the coefficients once you have the Vandermonde matrix. The `feval` command together with an appropriately defined function

```
function z=vandermonde(t,q)
z=t.^q;
endfunction
```

may be of help in producing the associated Vandermonde matrix.

You might like to use `deff` to define your function

```
deff('z=vandermonde(t,q)','z=t.^q')
```

This is useful for short functions.

Lab Book: Use your `polyfit` function to calculate the coefficients of the degree 5 and degree 10 polynomials which interpolate Runge's function at 6 and 11 evenly spaced points in the interval $[-1, 1]$.

Lab Book: How successful are the polynomial approximations about zero and the end points?

4.5 Taylor Polynomials

Note that the expression $1/(1+t) = 1 - t + t^2 - t^3 ...$ easily gives the Taylor polynomials centred about the origin of the Runge function to have the form

$$1 - 25t^2 + (25t^2)^2 - (25t^2)^3 ...$$

Lab Book: Use this to plot the Taylor polynomial of degree four centred at zero for the Runge function, over the interval [-1,1]. You should also plot the Runge function and the Taylor polynomial over a smaller interval, say [-0.2,0.2]. Try to explain its accuracy, and what you would expect the accuracy of the degree eight Taylor polynomial to be like.

4.6 Chebyshev Interpolation

Let us compute and graph polynomial approximations to the Runge function of degree four and eight using interpolation at the Chebyshev points: for the case of degree four, the Scilab expression to compute the five points needed is

```
tdata = cos((1:2:(2*5-1))*%pi/(2*5))
ydata = runge(tdata)
chebp = polyfit(tdata,ydata)
tt    = -1:.01:1;
rr    = runge(tt);
yy    = horner(chebp,tt);
xbasc();
plot2d(tt,yy,style=1);
plot2d(tt,rr,style=2);
plot2d(tdata,ydata,style=-1)
```

The results are more accurate than with equally spaced points or Taylor polynomials but still not very good: try to explain these observations.

Lab Book: Plot the result of using a range of higher order Chebyshev interpolants.

4.7 Spline Interpolation

The following Scilab commands use the command `splin` to compute the natural spline interpolate through the points specified by the vectors `x` and `y`. You use `interp` to evaluate the spline at the points `xx`.

```
tdata = -1:.5:1;
ydata = runge(tdata);
ddata = splin(tdata,ydata);
tt    = -1:.01:1;
rr    = runge(tt);
ss    = interp(tt,tdata,ydata,ddata);
```

Thus the spline approximation can then be graphed using

```
xbasc();
plot2d(tt,ss,style=1);
plot2d(tt,rr,style=2);
plot2d(tdata,ydata,style=-1);
```

Lab Book: Change the code to plot the cubic spline approximations of the Runge function using a set of nine equally spaced points, interpolating the Runge function.

4.8 Monotonic Cubic Interpolation

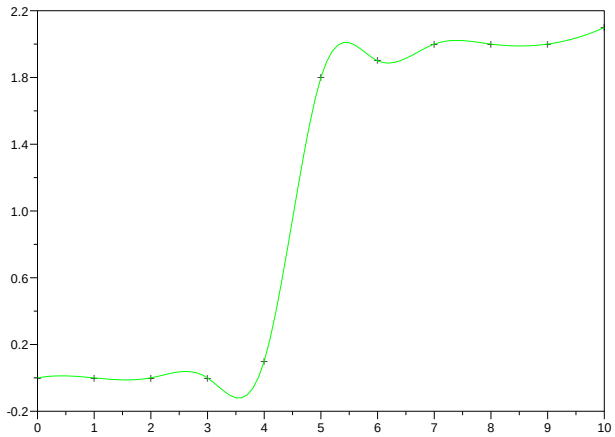
From the previous example we see that cubic spline interpolation can still introduce unwanted oscillations. There are many techniques which guarantee oscillation free interpolation (though at a cost of possible lower order accuracy), for instance a fairly common method is that of Fritsch and Carlson (SIAM J. Numer. Anal. 17, pp 238-246 1980). Scilab implements a monotonic interpolation method as an option to the `splin` function.

Here is some data

```
tdata = 0:1:10; ydata = [0 0 0 0 0.1 1.8 1.9 2.0 2.0 2.0 2.1];
```

Lab Book: Fit an ordinary spline function to this data set. Produce a plot

I obtained the following:



The result is quite oscillatory, maybe not really what we want.

Now re do the spline interpolation of this data, but use the extra argument `'monotone'`. I.e. use

```
ddata = splin(tdata,ydata, 'monotone');
```

Now you can use `tdata` and `ydata` values together with the new derivative values to produce an interpolant using the `interp` function.

Lab Book: Plot the data points, together with plots of the ordinary spline and monocubic interpolant. Comment on the respective interpolants.

Hopefully you can see that in some cases it is useful to try to maintain the shape implicit in your data sets. This is particularly true for coarse data sets.

Chapter 5

Sparse Matrices and Direct Solvers

5.1 Introduction

This tutorial is designed to familiarise you with sparse matrix calculations in Scilab.

5.2 Sparse Matrices

Scilab allows you to work with sparse matrices almost as easily as full matrices. Let us make some sparse matrices.

5.3 Convert Full to Sparse Matrices

We can use the `diag` command to create a tridiagonal matrix. Try

```
A = diag(ones(20,1),-1) -2*diag(ones(21,1)) + diag(ones(20,1),1);
```

This should produce a 21×21 tridiagonal full matrix. You can convert to sparse storage using the `sparse` command.

```
A = sparse(A);
```

With sparse matrices it is useful to look at the non-zero structure of the matrix.

The book "Engineering and Scientific Computing with Scilab" edited by Claude Gomez provides some very helpful hints on working with Scilab. In particular they have produced some code which mimic the `spy` command form Matlab. Here I have reproduced the code, encapsulated in a function.

```
//-----------------------------------------------------
function spy(A)
// Mimic the Matlab spy command
// Draw the non zero components of a matrix
```

```
// Taken from Engineering and Scientific Computing with Scilab''
// edited by Claude Gomez

[i,j] = find(A~=0)

[N,M] = size(A)

xsetech([0,0,1,1],[1,0,M+1,N])
xrects([j;N-i+1;ones(i);ones(i)],ones(i));
xrect(1,N,M,N);

endfunction
//-------------------------------------------------------
```

Copy this code into your SCILAB workspace and execute it on the matrix `A`.

```
xbasc();spy(A)
```

The `xbasc()` is there just to clear the graphics window if you have already used it.

Lab Book: Explain the plot you obtain

5.4 Sparse Matrix Creation

When we created A we first created a full matrix and then converted to a sparse matrix. Obviously we should avoid the full matrix. Use the `whos()` or `whos -name A` to see the storage requirements of A before and after the conversion to sparse storage format.

Now let us create the matrix as a sparse matrix without using full storage. We can use `diag` again but with an argument which is sparse. The following command will do the trick.

```
n=21; B = diag(sparse(ones(n-1,1)),-1) ...
       - 2*diag(sparse(ones(n,1))) + diag(sparse(ones(n-1,1)),1);
```

Check out the matrix by printing out its entries or even printing out a full representation of B.

```
B
full(B)
```

Don't do this with large matrices! Also use `spy` to check out the structure.

The sparse storage format is essentially a list of ij entries and corresponding matrix values v of the non-zero entries in the matrix.

For instance try out

```
//-------------------------------------------------------
dij  = [(1:n)', (1:n)'];          // i j coordinates of diagonal
dv = [-2*ones(n,1)];              // value of -2 down diagonal
udij = [ (1:n-1)',(2:n)'];        // i j coordinates of upper diagonal
udv  = [ ones(n-1,1) ];           // value of 1 for upper and lower diagonals
ldij = [ (2:n)',(1:n-1)'];        // i j coordinates of lower diagonal
ij = [dij ; udij ; ldij ];        // concatenate all the i, j coordinates
v  = [dv ; udv ; udv ];           // concatenate all the values
C=sparse(ij,v)                    // create sparse matrix
full(C)
//-------------------------------------------------------
```

Make sure you know what this code is doing. Try a smaller value of n and printout each step.

Lab Book: Compare the matrices `A, B, C`. Describe how each version has been created. Comment on the efficiency of each method (speed and storage).

5.5 Matrix Graph Example

The help file for the `mesh2d` function provides some very interesting matrices associated with triangulating a circular region. Here is a reproduction of the example given in the help file. Copy this code into your Scilab workspace.

```
//-------------------------------------------------------
function [a1,a2,a3,g1,g2,g3]=mesh_examples()
   //
   // Code taken from the help file for mesh2d. Produces 3
   // matrices and graphs associated with triangulating
   // a circular region with a hole
   //

   // FIRST CASE
    theta=0.025*[1:40]*2.*%pi;
    x=1+cos(theta);
    y=1.+sin(theta);
    theta=0.05*[1:20]*2.*%pi;
    x1=1.3+0.4*cos(theta);
    y1=1.+0.4*sin(theta);
    theta=0.1*[1:10]*2.*%pi;
    x2=0.5+0.2*cos(theta);
    y2=1.+0.2*sin(theta);
    x=[x x1 x2];
```

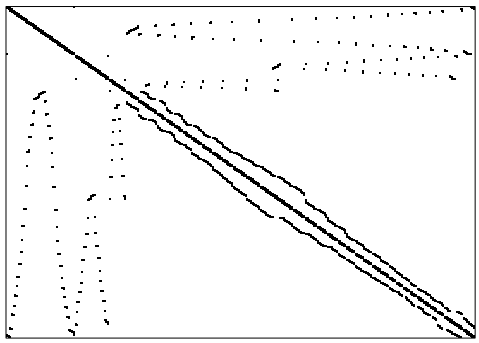
```
y=[y y1 y2];

[nu,a1]=mesh2d(x,y);

nbt=size(nu,2);
jj=[nu(1,:)' nu(2,:)';nu(2,:)' nu(3,:)';nu(3,:)' nu(1,:)'];
as=sparse(jj,ones(size(jj,1),1));
ast=tril(as+abs(as'-as));
[jj,v,mn]=spget(ast);
n=size(x,2);
g=make_graph('foo',0,n,jj(:,1)',jj(:,2)');
g('node_x')=300*x;
g('node_y')=300*y;
g('default_node_diam')=10;
g1 = g;

// SECOND CASE !!! NEEDS x,y FROM FIRST CASE
x3=2.*rand(1:200);
y3=2.*rand(1:200);
wai=((x3-1).*(x3-1)+(y3-1).*(y3-1));
ii=find(wai >= .94);
x3(ii)=[];y3(ii)=[];
wai=((x3-0.5).*(x3-0.5)+(y3-1).*(y3-1));
ii=find(wai <= 0.055);
x3(ii)=[];y3(ii)=[];
wai=((x3-1.3).*(x3-1.3)+(y3-1).*(y3-1));
ii=find(wai <= 0.21);
x3(ii)=[];y3(ii)=[];
xnew=[x x3];ynew=[y y3];
fr1=[[1:40] 1];fr2=[[41:60] 41];fr2=fr2($:-1:1);
fr3=[[61:70] 61];fr3=fr3($:-1:1);
front=[fr1 fr2 fr3];

[nu,a2]=mesh2d(xnew,ynew,front);

nbt=size(nu,2);
jj=[nu(1,:)' nu(2,:)';nu(2,:)' nu(3,:)';nu(3,:)' nu(1,:)'];
as=sparse(jj,ones(size(jj,1),1));
ast=tril(as+abs(as'-as));
[jj,v,mn]=spget(ast);
n=size(xnew,2);
g=make_graph('foo',0,n,jj(:,1)',jj(:,2)');
g('node_x')=300*xnew;
```

```
    g('node_y')=300*ynew;
    g('default_node_diam')=10;
    g2=g;


    // REGULAR CASE !!! NEEDS PREVIOUS CASES FOR x,y,front
    xx=0.1*[1:20];
    yy=xx.*.ones(1,20);
    zz= ones(1,20).*.xx;
    x3=yy;y3=zz;
    wai=((x3-1).*(x3-1)+(y3-1).*(y3-1));
    ii=find(wai >= .94);
    x3(ii)=[];y3(ii)=[];
    wai=((x3-0.5).*(x3-0.5)+(y3-1).*(y3-1));
    ii=find(wai <= 0.055);
    x3(ii)=[];y3(ii)=[];
    wai=((x3-1.3).*(x3-1.3)+(y3-1).*(y3-1));
    ii=find(wai <= 0.21);
    x3(ii)=[];y3(ii)=[];
    xnew=[x x3];ynew=[y y3];

    [nu,a3]=mesh2d(xnew,ynew,front);

    nbt=size(nu,2);
    jj=[nu(1,:)' nu(2,:)';nu(2,:)' nu(3,:)';nu(3,:)' nu(1,:)'];
    as=sparse(jj,ones(size(jj,1),1));
    ast=tril(as+abs(as'-as));
    [jj,v,mn]=spget(ast);
    n=size(xnew,2);
    g=make_graph('foo',0,n,jj(:,1)',jj(:,2)');
    g.node_x=300*xnew;
    g('node_y')=300*ynew;
    g('default_node_diam')=3;
    g3=g;

 endfunction


//-------------------------------------
// Now run the function to produce the
// example matrices and associated graphs
//-------------------------------------
```

```
[A1,A2,A3,g1,g2,g3] = mesh_examples();
//-------------------------------------------------------
```

If you have loaded the previous snippet of code into SCILAB there should now be 3 new matrices `A1, A2, A3` defined in your workspace. Try `spy` on these matrices.

I obtained the following spy plot of the matrix `A3`



Spy plot of `A3`.

Lab Book: Printout the spy plot of the `A2` matrix. Can you make any sense of this spy plot.

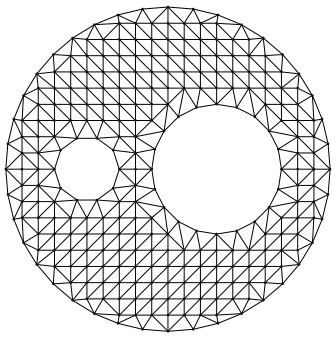
Now the previous code has also produced 3 graph objects associated with each sparse matrix.

Lets look at the graphs using the `show_graph` command. (If you want to do this on a windows machine and are using an earlier version (2.6) of SCILAB you will have to load in the *scigraph* package and use the `show_scigraph` command instead).

```
xbasc(); show_graph(g1);
```

Actually I think the `g2` and `g3` are more interesting. You can use the options submenu of the graph menu to number the nodes. This can help to understand the strange structure of the corresponding sparse matrix.

I obtained the following plot of the graph associated with matrix `A3`



Graph associated with matrix `A3`.

Lab Book: Printout the graph plot corresponding to the matrix `A2` matrix. Have a look at the first column of the matrix `A2`. Reconcile this with the graph `g2`

5.6 Sparse Matrix Operations

Most of the standard matrix operations have been implemented for sparse matrices. You can add, multiply, transform, invert and solve matrix equations.

For instance, you can find the inverse of `A1` the matrix created by the `mesh_examples` function.

```
A1inv = inv(A1);
```

Look at the `A1inv` with the `spy` function. The inverse is nearly full. This is typical, the inverse of a random sparse matrix is invariably full. The moral, avoid calculating the inverse of sparse matrices!

5.7 Solving Sparse Matrix Equations

There are basically 3 methods available for solving matrix equations

- The backslash operator \
- The combination of the `lufact` and `lusolve`
- And for positive definite matrices, the combination of the functions `chfact` and `chsolve`
- There is also a function `spchol`, but the `chfact` function seems to be more efficient.

Setup a test problem

```
[n,m]=size(A1);
x = ones(n,1);
b = A1*x;
```

So you have a right hand side **b** together with a solution vector **x**. Now lets see how well each of the solution methods work. Use the `help` command to see how to call the different factorisation commands. Use the `timer()` command to time each calculation. For instance

```
timer(); x1 = A1\b ; t1 =timer()
err1 = norm(x1-x)
```

5.8 Extensions

The `chfact`, `chsolve` combination of commands is quite efficient for positive definite matrices as it uses minimum degree reordering (the `lufact` and `spchol` do not reorder). Unfortunately the matrix $A1$ is not positive definite!

Lets see if we can improve the efficiency of the `lufact` command by reordering the matrix $A1$. The command `bandwr` will try to reorder a symmetric matrix to create a reordered matrix with smaller bandwidth. Applying this to $A1$ (which is symmetric) try

```
[iperm,mrepi]=bandwr(triu(A1));
RA1 = A1(mrepi,mrepi);
xbasc();spy(A1)
xbasc();spy(RA1)
```

Note that the reordered matrix has a much smaller bandwidth. Determine whether the solution methods work better on the reordered matrix.

Chapter 6

Iterative Methods Applied to Membrane Problem

6.1 Introduction

In this tutorial we will investigate the solution of matrix problems which are commonly found in the modelling of membranes, equilibrium temperature distributions and flow of fluids through porous media.

We will first define the problem and solve it using standard the Scilab sparse linear equation solver. We will then investigate means of speeding up our method. In particular we will see that the iterative method, the conjugate gradient method provides a good method for solving larger problems. The tutorial will also give you some experience with plotting functions of 2 variables (surface graphs).

There is a lot of code in this tutorial as we will be using quite sophisticated methods. I suggest you cut and paste the code into Scilab run it from there.

6.2 Setup Matrix

We will first setup a problem which models a membrane with loading, such as mass loads or pressure. The membrane will be approximated by a 2 dimensional grid.

Let's setup the grid

```
n = 10; m = 15;
x = 0:1/(n-1):1;
y = 0:1/(m-1):1;
u = x'*y;
```

You can plot the value of `u` using the `plot3d` command. Namely

```
xbasc(); xselect();
plot3d(x,y,u)
```

Rotate the plot (using the rotate button on the figure window) to get a better idea of the shape of the surface.

You might want to try a shaded plot using the `plot3d1` command.

```
xbasc(); xselect();
plot3d1(x,y,u)
```

The default colour map does not produce a very nice plot. SCILAB 3.0 has three in built colour maps, `jetcolormap`, `hotcolormap` and `graycolormap`. In SCILAB version 2.7 it seems that the function `jetcolormap` is not available. Here is the code for `jetcolormap` taken from the SCILAB 3.0 distribution. If you are using SCILAB 2.7 or earlier, load in this function so that you run the code given in tutorials.

```
function [cmap] = jetcolormap(nb_colors)
//   PURPOSE
//      to get the usual classic colormap which goes from
//      blue - lightblue - green - yellow - orange then red
//   AUTHOR
//      Bruno Pincon
//
   r = [0.000 0.125 0.375 0.625 0.875 1.000;...
        0.000 0.000 0.000 1.000 1.000 0.500]
   g = [0.000 0.125 0.375 0.625 0.875 1.000;...
        0.000 0.000 1.000 1.000 0.000 0.000]
   b = [0.000 0.125 0.375 0.625 0.875 1.000;...
        0.500 1.000 1.000 0.000 0.000 0.000]
   d = 0.5/nb_colors
   x = linspace(d,1-d, nb_colors)
   cmap = [ interpln(r, x);...
            interpln(g, x);...
            interpln(b, x) ]';
   cmap = min(1, max(0 , cmap))  // normally not necessary
endfunction
```

You can set the new color map using the `xset` command. Try

```
xset("colormap",jetcolormap(64))
xbasc(); xselect();
plot3d1(x,y,u)
```

Lab Book: Change the colour map. Produce a plot using the gray scale colour map, rotated to give a good view of the surface.

6.3 Setting Boundary Values

We want to model the situation in which our membrane is fixed around the edge. We will specify the height of the edge nodes using a function. In the first instance lets set the boundary to have a saddle shape given by the function

```
function u = saddle_function(x,y)
   //
   // Saddle point function to be used as
   // boundary and exact solution
   //
   u = (x-0.5)^2 - (y-0.5)^2
endfunction
```

Load this function into Scilab.

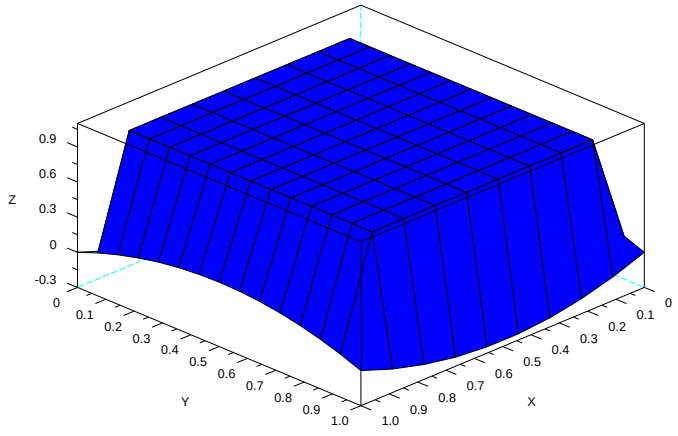
The boundary nodes correspond to the i, j values $i = 1$, $i = n$, $j = 1$ and $j = m$. So lets set those nodes to have the value given by the function `saddle_function`.

```
for i=1:n
   for j=1:m
      // If statement to determine boundary nodes
      if( i==1 | i==n | j==1 | j==m) then
         u(i,j) = saddle_function(x(i),y(j));
      else
         u(i,j) = 1.0;
      end
   end
end
```

Lab Book: Can you explain what the `for` loop is doing?

Lab Book: Plot the grid function u.

I obtained the following plot using the `plot3d` command.



6.4 Applying Tension

We want to apply a tension force to the membrane. We do this by specifying that

$$f_{ij} = \frac{u_{i+1j} - 2u_{ij} + u_{i-1j}}{\Delta x^2} + \frac{u_{ij+1} - 2u_{ij} + u_{ij-1}}{\Delta y^2}$$

Here f_{ij} is representing a vertical force applied to node i, j, and $\Delta x = \frac{1}{n-1}$ and $\Delta y = \frac{1}{m-1}$. The vertical force is most commonly interpreted as a gravitational force or a pressure force.

So we have a system of linear equations

$$\frac{u_{i+1j}}{\Delta x^2} + \frac{u_{i-1j}}{\Delta x^2} - 2\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) u_{ij} + \frac{u_{ij+1}}{\Delta y^2} + \frac{u_{ij-1}}{\Delta y^2} = f_{ij}$$

Note that if $\Delta x = \Delta y$ then the equations can be written in the simpler form

$$u_{i+1j} + u_{i-1j} - 4u_{ij} + u_{ij+1} + u_{ij-1} = f_{ij}\Delta x^2.$$

The number of nodes is nm, and so the corresponding matrix will be nm by nm.

Lets set up a sparse matrix which encodes all these equations. For the boundary nodes we will set the corresponding row of the matrix to have only a diagonal equal to one and set a right had side to be the boundary value. Initially we need to set up the sparse matrix using the following `create_matrix` function.

```
//-------------------------------------------
// Define Functions
//-------------------------------------------
function [A,b] = create_matrix_1(n,m)
   //
```

```
   // Create a matrix associated with a grid
   // nodes with x and y coordinates given as
   // input, connected by springs with nodes
   // on the boundary set the function boundary
   //
   dx = 1/(n-1);
   dy = 1/(m-1);

   dx2 = 1/dx^2;
   dy2 = 1/dy^2;

   x = 0:dx:1;
   y = 0:dy:1;


   A = speye(n*m,n*m);
   b = zeros(n*m,1);
   for i=1:n
      for j=1:m
         index = i + (j-1)*n;
         if( i==1 | i==n | j==1 | j==m) then      // Boundary nodes
            b(index) = saddle_function(x(i),y(j));
            A(index,index) = 1.0;
         else                                    // Interior Nodes
            b(index) = force;
            A(index,index) = -2.0*(dx2 + dy2);
            A(index,index+1) = dx2;
            A(index,index-1) = dx2;
            A(index,index+n) = dy2;
            A(index,index-n) = dy2;
         end
      end
   end

endfunction
```

Load this function into Scilab, and create the matrix A and right hand side $\mathbf{b}$ corresponding to a 5 by 5 grid by using the command

```
[A,b]=create_matrix_1(5,5)
```

This will produce an error since the `create_matrix` function needs the variable `force` set. Lets set it to 0 and try the command

```
force = 0.0; [A,b]=create_matrix_1(5,5)
```

How large is the matrix A. What is its' structure. Here is the code for the `spy` function which is useful to view the structure of the matrix.

```
function spy(A)
   // Mimic the Matlab spy command
   // Draw the non zero components of a matrix

   [i,j] = find(A~=0)

   [N,M] = size(A)

   //  wrect=[x,y,w,h]  frect=[xmin,ymin,xmax,ymax]
   xsetech([0,0,1,1],[1,0,M+1,N])
   xrects([j;N-i+1;ones(i);ones(i)],ones(i));
   //  [x,y,w,h]
   xrect(1,N,M,N);

endfunction
```

Lab Book: Explain what this code is doing, in particular how `index` is being used. Produce a plot of the structure of the matrix A using the `spy` function.

6.5 Solving the Problem

So now we have a way of producing a matrix and a right hand side, so now we only have to solve the equation.

Here is some code to setup and solve the problem. I have added some extra code which times the code and also sets up a wrapper for the solver so that we can change the solver later in the tutorial.

Load in the following code.

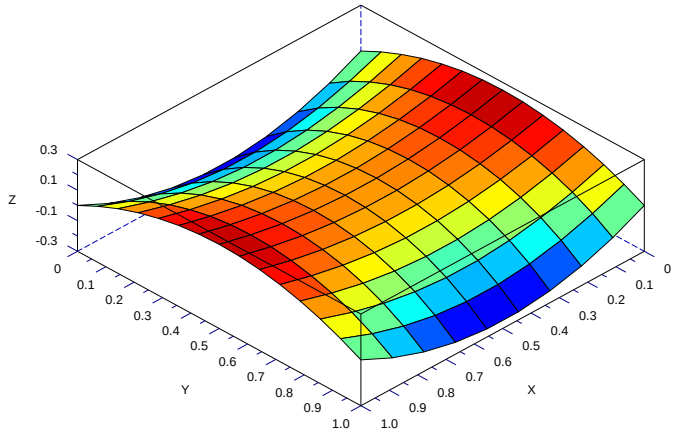
```
function [U] = solve_equation_1(A,b)
   //
   // Wrapper for linear solver
   //
   U = A\b
endfunction

function run_problem(n,m)
   //
   // Setup a problem, create matrix,
   // solve and then plot result.
```

```
  //
  // Input:
  // n,m, number of grid points in x and y direction.
  //
  x = 0:1/(n-1):1;
  y = 0:1/(m-1):1;

  // Setup creation and solver routines
  create_matrix  = create_matrix_1
  solve_equation = solve_equation_1


  printf('Create Matrix:')
  timer()
  [A,b] = create_matrix(n,m);
  printf(' time = %g\n',timer())

  printf('Solve Matrix problem:')
  timer()
  U = solve_equation(A,b)
  printf(' time = %g\n',timer())

  u = matrix(U,n,m);

  // Plot the solution
  xset("colormap",jetcolormap(64))
  xbasc(); xselect();
  plot3d1(x,y,u)

endfunction
```

Now we should run our code. We just have to run the `run_problem` function. For instance to run the code for a 10 by 15 grid use the command

```
run_problem(10,15)
```

When running the previous fragment of code I obtained the following plot.



This is good, as you can now see that the membrane is fixed to the boundary and in the interior the membrane seems to be pulling evenly in each direction.

Lab Book: Produce a plot of a membrane with the same saddle boundary, but with a force of 5 is applied

Lab Book: Produce a plot of a membrane with a zero boundary condition, and force of 5 is applied. You will have to change the `create_matrix` function.

Now lets try some larger grids. Try solving the problem for say a 40 by 40 grid or 50 by 50.

Lab Book: How long does it take to create the matrix associated with a 50 by 50 grid. How long does it take to solve the associated equation. How big is the associated matrix.

6.6 Faster Creation of Matrix

So it is taking longer to create the matrix than to solve the matrix equation. Surely we are doing something wrong! Well individual sparse matrix updates are very time consuming. In the next implementation of the `create_matrix` function we are using the `diag(sparse)` functions to setup the matrix.

Load in the following new implementation of the `create_matrix` function into SCILAB

```
function [A,b] = create_matrix_2(n,m)
   //
   // Create a matrix associated with a grid
   // nodes with x and y coordinates given as
   // input, connected by springs with nodes
   // on the boundary set the function boundary
   //
```

```
dx = 1/(n-1);
dy = 1/(m-1);

dx2 = 1/dx^2;
dy2 = 1/dy^2;

x = 0:dx:1;
y = 0:dy:1;

//-----------------------
// Setup Matrix
//----------------------
A = spzeros(n*m,n*m);


//
// Setup interior problem
//
d0 = -2*(dx2+dy2)*ones(n*m,1);
dmn = dy2*ones((m-1)*n,1)
dpn = dy2*ones((m-1)*n,1)
dm1 = dx2*ones(n*m-1,1)
dp1 = dx2*ones(n*m-1,1)

//
// Apply Boundary condition
//
d0(n:n:n*m) = 1
d0(1:n:n*m) = 1
d0(1:n) = 1;
d0(n*(m-1):n*m) = 1

dmn(1:n:n*(m-1)) = 0
dmn(n:n:n*(m-1)) = 0
dmn(n*(m-2):n*(m-1)) = 0

dpn(1:n:n*(m-1)) = 0
dpn(n:n:n*(m-1)) = 0
dpn(1:n) = 0

dm1(n-1:n:n*m-1) = 0
dm1(n:n:n*m-1) = 0
dm1(1:n-1) = 0
```

```
dm1(n*(m-1):n*m-1) = 0

dp1(n+1:n:n*m-1) = 0
dp1(n:n:n*m-1) = 0
dp1(1:n-1) = 0
dp1(n*(m-1):n*m-1) = 0

A = diag(sparse(d0)) + diag(sparse(dm1),-1) + diag(sparse(dp1),1) + ...
     diag(sparse(dmn),-n) + diag(sparse(dpn),n)


//----------------
// Setup rhs
//----------------
b = force*ones(n*m,1);

//
// Apply boundary condition
//
b(1:n)           = feval(x,y(1),saddle_function);
b(n*(m-1)+1:n*m) = feval(x,y(m),saddle_function);
b(1:n:n*(m-1)+1) = feval(x(1),y,saddle_function)';
b(n:n:n*m)       = feval(x(n),y,saddle_function)';


endfunction
```

Convince yourself that this code is doing the same as the previous implementation, i.e. compare the results from the two `create_matrix` functions.

Lab Book: Run the problem on a 50 by 50 grid. Compare the speed of this new implementation with the previous implementation. Is the code running quicker?

You can see that now the solution of the linear system is now dominating the execution time.

6.7 Conjugate Gradient Method

We introduced iterative methods in the lectures. The conjugate method works very well for matrices such as A. The matrix A is not actually symmetric, but as long as we ensure that the initial guess satisfies the boundary conditions, then the matrix is essentially symmetric with respect to the "interior" grid points.

The conjugate gradient method is not available in Scilab by default, but it is a simple algorithm and we can implement it via the following code (taken from "An Introduction

to the Conjugate Gradient Method Without the Agonizing Pain", by Jonathan Richard Shewchuk)

```
function [x,i] = conjugate_gradient(A,b,x0,imax,tol,iprint)
   //
   // Try to solve linear equation Ax = b using
   // conjugate gradient method
   //
   // Input
   // A: matrix or function which applies a matrix, assumed symmetric
   // b: right hand side
   // x0: inital guess
   // imax: max number of iterations
   // tol: tolerance used for residual
   //
   // Output
   // x: approximate solution
   // i: number of iterations
   //

   //------------------------------
   // Check and set input arguments
   //------------------------------
   nargin = argn(2)
   if nargin<2
      error('need at least 2 input arguments')
   end
   if nargin<3
      x0 = zeros(b);
   end
   if nargin<4
      imax = 2000;
   end
   if nargin<5
      tol = 1e-8
   end
   if nargin<6
     iprint = 0
   end
   function y = apply_A(x)
      if (typeof(A)=='function') then
         y = A(x)
      else
```

```
        y = A*x
     end
  endfunction
  //-------------------------------

  i=0
  x = x0
  r = b - apply_A(x)
  d = r
  rTr = r'*r
  rTr0 = rTr

  while (i<imax & rTr>tol^2*rTr0),
      q = apply_A(d)
      alpha = rTr/(d'*q)
      x = x + alpha*d
      if modulo(i,50)==0 then
          r = b - apply_A(x)
      else
          r = r - alpha*q
      end
      rTrOld = rTr
      rTr = r'*r
      bt = rTr/rTrOld

      d = r + bt*d
      i = i+1
      if modulo(i,iprint)==0 then
         printf('i = %g rTr = %20.15e \n',i,rTr )
      end
  end

  if i==imax then
    warning('max number of iterations attained')
  end

endfunction
```

Load this function into Scilab and try on a small matrix and rhs such as

```
n=21;
B = diag(sparse(ones(n-1,1)),-1) - 2*diag(sparse(ones(n,1))) + ...
   diag(sparse(ones(n-1,1)),1);
c = ones(n,1);
```

by using the command

```
x = conjugate_gradient(B,c)
```

Well that is nice, but we can see a little more detail if we use a few more arguments,

```
x = conjugate_gradient(B,c,zeros(c),50,1e-9,1)
```

The 3rd argument provides an initial guess for the iterative scheme, the 4th gives provides a limit on the number of iterations, the 5th a tolerance on the terminating relative residual, and the 6th argument give the regularity of diagnostic output. The results show that the square of the residual `rTr` decreases (slowly) until the 11th iteration which drops to zero. This is typical of the method.

So finally we can use the conjugate gradient method to try to solve our membrane problem. Our `run_problem` function sets up the `solve_equation` function. To use the conjugate gradient method set up the `run_problem` function to use `solve_equation_2` function defined below.

```
function [U] = solve_equation_2(A,b)

   //
   // Get good initial guess, well at least one that
   // satisfies BC
   //

   x0 = b
   U = conjugate_gradient(A,b,x0,length(x0),1e-9,0)
endfunction
```

Lab Book: Run the membrane problem for a 50 by 50 grid, using the `solve_equation_2` function. Comment on the relative speed of the new implementation of the code. Is it better than the previous implementation. Find approximately how large a problem you can solve in 20 seconds on your computer using the final implementation.

6.8 Faster Matrix Vector Multiplication

This actually is not the end of the story. First we can implement better methods for calculating the matrix vector multiplication (the most expensive part of the conjugate gradient algorithm). Indeed if we are using an iterative method like the conjugate gradient method, we only need to provide a procedure for calculating the matrix vector multiplication. So here is yet another implementation of `create_matrix` procedure.

```
function [A,b] = create_matrix_3(n,m)
   //
   // Create a matrix operator associated with a grid
   // nodes with x and y coordinates given as
   // input, connected by springs with nodes
   // on the boundary set the function boundary
   //
   dx = 1/(n-1);
   dy = 1/(m-1);

   x = 0:dx:1;
   y = 0:dy:1;

   //-----------------------
   // Setup Matrix
   // A function is being created
   // which will be returned
   // instead of a matrix
   //-----------------------
   function y = A(x)
      //
      // Setup finite difference operator for Laplacian
      //
      dx = 1/(n-1);
      dy = 1/(m-1);
      dx2 = 1/dx^2;
      dy2 = 1/dy^2;

      x = matrix(x,n,m)
      y = x

      J = 2:m-1
      I = 2:n-1

      y(I,J) = -2.0*(dx2+dy2)*x(I,J) + dx2*x(I+1,J) ...
               + dx2*x(I-1,J) + dy2*x(I,J+1) + dy2*x(I,J-1)

      y = y(:)
   endfunction


   //----------------
   // Setup rhs
```

```
    //----------------
    b = force*ones(n*m,1);

    //
    // Apply boundary condition
    //
    b(1:n)           = feval(x,y(1),saddle_function);
    b(n*(m-1)+1:n*m) = feval(x,y(m),saddle_function);
    b(1:n:n*(m-1)+1) = feval(x(1),y,saddle_function)';
    b(n:n:n*m)       = feval(x(n),y,saddle_function)';


endfunction
```

In this case the procedure returns not a sparse matrix A but a procedure for apply A to a vector.

Lab Book: Look at the new code for the create matrix procedure. Explain what the procedure is doing. In particular explain the lines

```
J = 2:m-1
I = 2:n-1

y(I,J) = -2.0*(dx2+dy2)*x(I,J) + dx2*x(I+1,J) ...
         + dx2*x(I-1,J) + dy2*x(I,J+1) + dy2*x(I,J-1)
```

Lab Book: Try this new method. How large a problem can you now solve in 20 second. I got approximately 300 by 300, which corresponds to 90,000 unknowns and a 90,000 by 90,000 matrix!

6.9 Conclusion

Even now there is room for improvement. The iterative method can be improved using what is called preconditioning. With these improvements it is possible to solve a 1000 by 1000 grid problem (1,000,000 unknowns) in ten seconds on a normal PC!

By the way, you might question why we need to solve 1000 by 1000 grid problems. In geophysical situations there are often cases where we need to resolve flows over a few metres (through faults) over regions over many kilometres, ie a range of scale over a thousand.

Chapter 7

QR Factorisation and Least Squares

7.1 Introduction

A nice way to understand (and in fact construct) many linear algebra methods is via matrix decompositions or factorisation techniques.

For instance, Gaussian Elimination can be characteristed as an LU factorisation. In lectures we saw how the process of Gaussian elimination can be considered as the application of a sequence of elementary matrices which can be encoded in to a lower triangular matrix L, and leads to an upper diagonal matrix U. SCILAB provides the command `lu` to calculate LU decompositions.

Similarly, the QR factorisation consists of applying a sequence of orthogonal transformations (usually Householder reflections), which eliminate variables, Q being the transpose of the result of these reflections, and the R being the resulting upper triangular matrix. QR factorisation is generally more stable than the Gaussian Elimination, but is generally more expensive (about twice as expensive). SCILAB provides the command `qr` to calculate LU decompositions.

By the way, there are many more factorisations which are useful. You should have already studied eignevalues and eigenvectors. If a matrix has a complete set of eigenvectors, then we obtain the factorisation $A = VDV^{-1}$ where D is a diagonal matrix of the eigenvalues, and V is a matrix of eigenvectors (as columns of V). SCILAB provides the command `spec` to calculate eigenvalue decompositions.

Another very useful decomposition is the Singular Value Decomposition, or SVD. Here for **any** matrix we can find two orthogonal bases, written as matrices U and V, such that $AV = US$, where S is a diagonal matrix of non-negative values called the singular values of the matrix. A can be decomposed, as $A = USV^*$. SCILAB provides the command `svd` to calculate singular value decompositions. The SVD is very stable, but is quite expensive, being on average about 10-15 times more expensive than LU decomposition.

Once we have a decomposition of a matrix, we can use that to solve a matrix equation. Suppose $A = BC$ and we want to solve $A\mathbf{x} = \mathbf{b}$, then we would

1. Solve $B\mathbf{y} = \mathbf{b}$.

2. Solve $C\mathbf{x} = \mathbf{y}$.

Of course to solve each of these steps, we should use any special properties of the B and C matrices to ensure efficiency and accuracy.

7.2 LU method

Lets have a look at the LU decomposition. Here is a method from generating a matrix associated with the minimisation of curvature (elastic band).

```
function A = elastic(n)
  A = diag(ones(n-1,1),-1) -2*diag(ones(n,1)) + diag(ones(n-1,1),1);
endfunction
```

Use this function to produce a 5 by 5 matrix A. Then use the Scilab command `lu` to produce the L and U factors of A. Here is my code

```
[L,U] = lu(A)
```

and here is the matrix I obtained

```
A  =

! - 2.    1.    0.    0.    0. !
!   1.  - 2.    1.    0.    0. !
!   0.    1.  - 2.    1.    0. !
!   0.    0.    1.  - 2.    1. !
!   0.    0.    0.    1.  - 2. !
```

and the L and U factors

```
 U  =

! - 2.    1.     0.           0.      0.  !
!   0.  - 1.5    1.           0.      0.  !
!   0.    0.   - 1.3333333    1.      0.  !
!   0.    0.     0.         - 1.25    1.  !
!   0.    0.     0.           0.    - 1.2 !
 L  =

!   1.     0.           0.      0.     0. !
! - 0.5    1.           0.      0.     0. !
!   0.   - 0.6666667    1.      0.     0. !
!   0.     0.         - 0.75    1.     0. !
!   0.     0.           0.    - 0.8    1. !
```

Notice that both of these matrices maintain the same banded structure as the original matrix A. Notice that the inverse of A loses the banded structure. Here is the inverse

```
->inv(A)
 ans  =

! - 0.8333333  - 0.6666667  - 0.5  - 0.3333333  - 0.1666667 !
! - 0.6666667  - 1.3333333  - 1.   - 0.6666667  - 0.3333333 !
! - 0.5        - 1.         - 1.5  - 1.         - 0.5       !
! - 0.3333333  - 0.6666667  - 1.   - 1.3333333  - 0.6666667 !
! - 0.1666667  - 0.3333333  - 0.5  - 0.6666667  - 0.8333333 !
```

This is typical, and so you should normally avoid calculating inverses in preference to LU factorisations.

Lab Book: Reproduce these results. Record your results in your lab book. Verify that indeed $A = LU$ (at least up to roundoff error).

Now let's use the decomposition to solve a matrix equation. Take $\mathbf{b} = [1\ 1\ ...\ 1]^T$. To solve $A\mathbf{x} = \mathbf{b}$ we can use

```
z = L\b
x = U\z
```

or in one line

```
x = U\(L\b)
```

Actually we should use specially designed forward and backward substitution operators to undertake the L and U solves. Unfortunately SCILAB does not provide such operators, so for this investigation we will just use the backslash operators.

Of course, normally we would use the backslash operator just once

```
x = A\b
```

Lab Book: Use the LU decomposition to solve the equation $A\mathbf{x} = \mathbf{b}$ where $\mathbf{b} = [1\ 2\ 3\ ...\ n]^T$.

7.3 QR Factorisation

Apply the SCILAB method `qr` to the matrix A. The simple command is

```
[Q,R]=qr(A)
```

Here are the factors I obtained.

```
 R  =

!   2.236068  - 1.7888544    0.4472136    0.           0.         !
!   0.          1.6733201  - 1.9123658    0.5976143    0.         !
!   0.          0.           1.4638501  - 1.9518001    0.6831301  !
!   0.          0.           0.           1.3540064  - 1.9694639  !
!   0.          0.           0.           0.         - 0.8090398  !
 Q  =

! - 0.8944272  - 0.3585686  - 0.1951800  - 0.1230915    0.1348400  !
!   0.4472136  - 0.7171372  - 0.3903600  - 0.2461830    0.2696799  !
!   0.           0.5976143  - 0.5855400  - 0.3692745    0.4045199  !
!   0.           0.           0.6831301  - 0.4923660    0.5393599  !
!   0.           0.           0.           0.7385489    0.6741999  !
```

The R matrix is upper triangular, but the Q matrix is near full. You might think it is somewhat hard to work with. But calculate Q^TQ. It is essentially the identity. This is one of the properties of orthogonal matrices. Also calculate the 2-norm of A and R. They should be the same. This is another property of orthogonal matrices. Application of orthogonal matrices leaves the norm invariant.

Let's solve $A\mathbf{x} = \mathbf{b}$, with $\mathbf{b} = [1\ 1\ ...\ 1]^T$ using the QR factorisation. As Q is orthogonal we can solve $Q\mathbf{z} = \mathbf{b}$ by $\mathbf{z} = Q^T\mathbf{b}$. Hence we can solve the equation with the code

```
z = R\(Q'*b)
```

Lab Book: QR factorisations can also be applied to rectangular matrices. Create a random 10 by 5 matrix. Calculate the QR factorisation of this matrix. Once again verify that $A \simeq QR$, $Q^TQ \simeq I$, $QQ^T \simeq I$ and that the norm of A and R are the same. Notice that R contains a lot of zeros. SCILAB (and indeed most other matrix environments) provides a method to produce a QR factorisation which economises on these zeros. Use the command `[Q,R] = qr(A,'e')` and investigate whether $A \simeq QR$, $Q^TQ \simeq I$, $QQ^T \simeq I$ and whether the norm of A and R are the same.

7.4 Conditioning

Random matrices are generally quite well conditioned. Matrices that occur from real problems can often have quite large condition numbers.

Here are some functions to produce matrices somewhat related to real world problems. Copy these into your SCILAB environment.

Here is a simple (not very efficient) function to calculate rectangular vandermonde matrices of varying sizes. Vandermonde matrices are associated with fitting functions to data.

```
function A = vandermonde(n,m)
  t = (0:1/(n-1):1)';
  A = zeros(n,m);
  for j=1:m
      A(:,j) = t.^(j-1);
  end
endfunction
```

Here is the famous Hilbert matrix

```
function H = hilbert(n)
  deff('z = hh(i,j)','z = 1/(i+j-1)')
  H = feval(1:n,1:n,hh)
endfunction
```

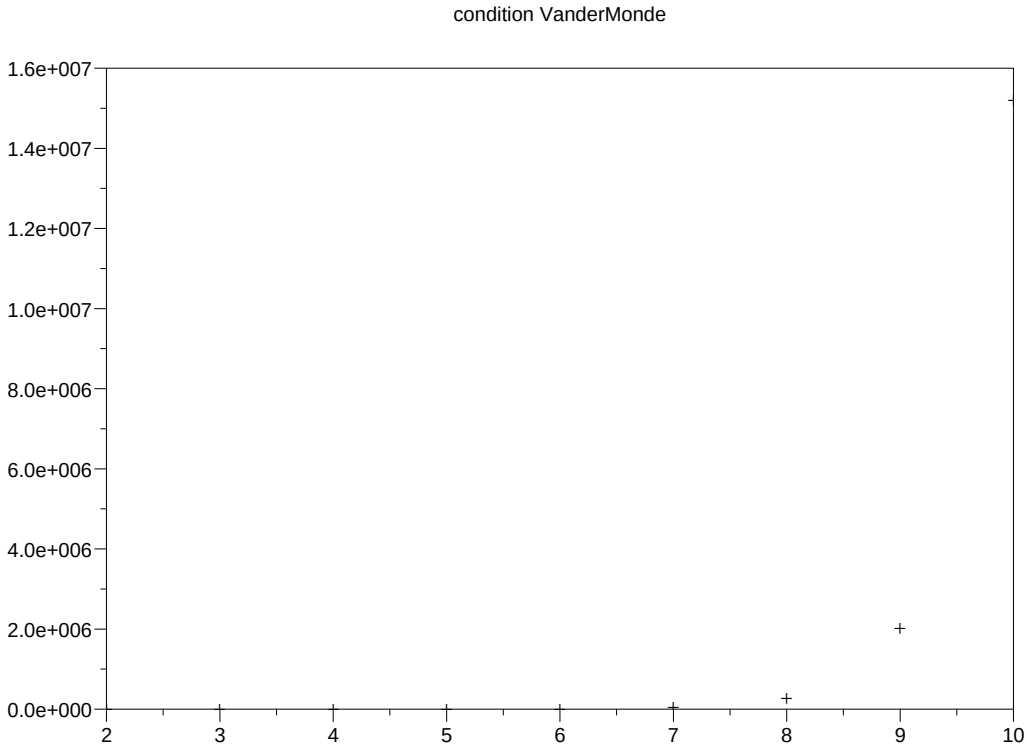
Here is the elastic example used before.

```
function A = elastic(n)
  A = diag(ones(n-1,1),-1) -2*diag(ones(n,1)) + diag(ones(n-1,1),1);
endfunction
```

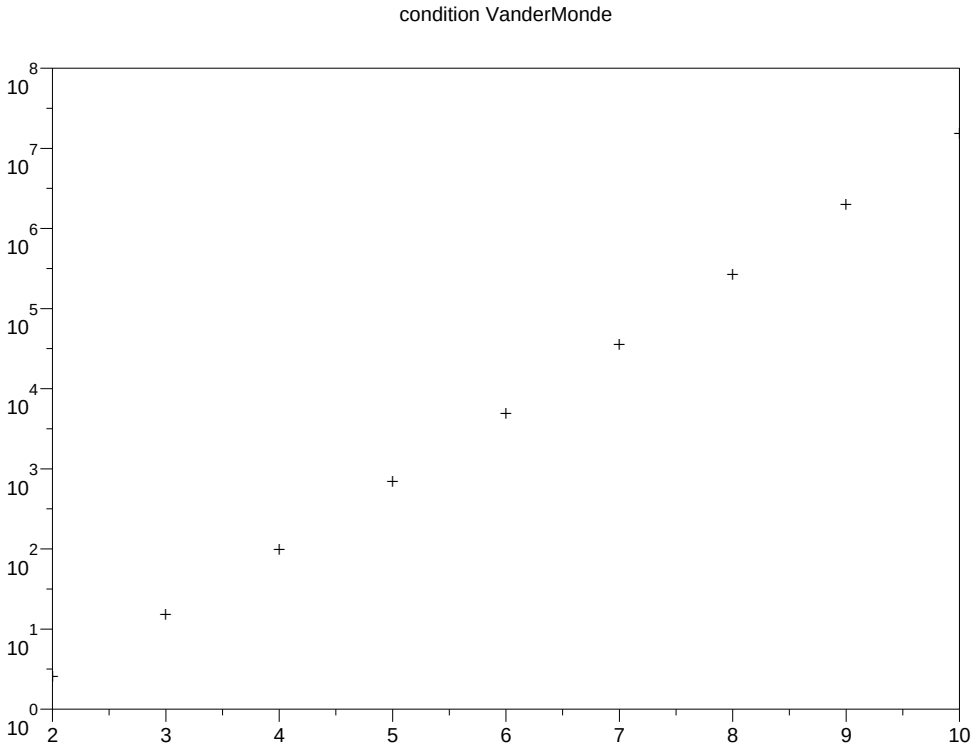
Here is some code to investigate the condition number of the square vandermonde matrix over a range of sizes.

```
function [] = test_vandermonde(m)

data = zeros(1,m-1);

for i=1:m-1
    A = vandermonde(i+1,i+1);
    data(i) = cond(A);
end

scf(0)
xbasc()
xtitle('condition VanderMonde')
plot2d(2:m,data,style=-1)

endfunction
```

Here is the plot produce by the `test_vandermonde` function.



This is actually not so useful. If we change the plot to vertical log scales we obtain:



and we see that the condition number of the square vanderMonde matrix increases exponentially.

Lab Book: Recreate these plots and also produce the analogous plots for the Hilbert and "Elastic" matrices. For the elastic problem, it is useful to use a log log plot. Try to estimate the relationship of condition number versus matrix size for these matrices.

7.5 Solving Equations using Decompositions

To test efficiency of the different methods, we can estimate the number of floating point operations, or more experimentally, we can measure the time it takes to solve a large number of test problems.

To find some test problems, we need to find the exact solution of $A\mathbf{x} = \mathbf{b}$ for some given right hand sides. Note that $A\mathbf{e}_k = \mathbf{a}_k$ where $\mathbf{e}_k$ is the k-th standard basis vector, and $\mathbf{a}_k$ is the k-th column of A. So we have some exact solutions $\mathbf{x}_e$. Then we use our solution technique to obtain an approximate solution $\mathbf{x}_a$ and compare it to the exact solution $\mathbf{x}_e$. Sometimes you get unrealistic accurate results. We could of course specify a solution $\mathbf{x}_e$ and then calculate the rhs $\mathbf{b} = A\mathbf{x}_e$. The trouble with this is that for ill conditioned matrices the simple action of multiplying $A\mathbf{x}_e$ will introduce an error proportional to the

condition number of the matrix. As a compromise, it may be worth using something like $\mathbf{x}_e = 1/3\mathbf{e}_1 + 2/3\mathbf{e}_n$, which mixes up the calculation a little.

Lab Book: Consider 10 by 10 elastic, VanderMonde, Hilbert and random matrices. For each measure the relative size of the residual $\|\mathbf{b} - A\mathbf{x}_a\|/\|\mathbf{b}\|$ and the relative size of the error $\|\mathbf{x}_e - \mathbf{x}_a\|/\|\mathbf{x}_e\|$ for a test problem generated by the exact solution $\mathbf{x}_e = 1/3\mathbf{e}_1 + 2/3\mathbf{e}_n$. Use either the LU decomposition or the QR decomposition (results are similar). Comment on the size of error and residual and the condition number of the matrix. Produce a table of your results.

Chapter 8

Solving Initial Value Differential Equations

8.1 Introduction

This tutorial is designed to familiarise you with solving ordinary differential equations using the SCILAB procedure `ode`.

8.2 Scalar Ordinary Differential Equation (ODE's)

Suppose we want to find the solution $u(t)$ of the ordinary differential equation

$$\frac{du}{dt} = k\,u(t)\,(1-u(t)), \quad t \geq 0. \tag{8.1}$$

This equation is a simple model for the spread of a disease. Consider a herd of P animals, and let $u(t)$ be the proportion of animals infected by the disease after t days. Thus $1-u(t)$ is the proportion of uninfected animals. New infections occur when infected and uninfected animals meet. The number of such meetings is proportional to $u(t)\,(1-u(t))$. Thus a simple model for the spread of the disease is that the new infections per day, du/dt, is proportional $u(t)\,(1-u(t))$. That is we obtain equation (8.1), where the constant k depends on the density of the animals and the infectiousness of the disease. For definiteness, suppose here $P = 10000$ animals and $k = .2$. Initially at time $t = 0$ there is just one infectious animal, and so we add the initial condition

$$u(0) = 1/10000. \tag{8.2}$$

We want to find how many individuals will be infected over the next 100 days, that is to plot $u(t)$ from $t = 0$ to $t = 100$. To solve this problem we need to define a function to calculate the right hand side of (8.1), and then use the SCILAB program `ode`.

Create a SCILAB function which specifies the right hand side of the differential equation. Make sure you get the calling sequence correct. For instance

```
function udot=f(t,u)
udot  =  k * u * ( 1  - u );
endfunction
```

(Although `t` is not used in calculating `udot`, the SCILAB function `ode` requires the function `f` to be passed to `ode` must have two input arguments (in the correct order). Use `help ode` for more information.)

Having set up the problem, the differential equation is solved in just one SCILAB line using `ode`!

```
t=0:100;
k = 0.2;
u = ode( 1/10000, 0.0, t, f );
xbasc(); plot2d( t,u)
```

Note that the value of `k` is available to `ode` and to `f`.

The function `ode` also has parameters which can be adjusted to control the accuracy of the computation. Mostly, these are not necessary for just plotting the solution to ordinary problems. It is always a good idea to check our the original solution by resolving the problem with higher accuracy requirements. For example, we can use `ode` so that the estimated absolute and relative errors are kept less than 1×10^{-9}. This is done with the commands

```
uu = ode( 1/10000, 0.0, t, rtol=1e-9, atol=1e-9, f );
plot2d(t,uu,style=-1)
```

We immediately see that the solution computed in the first call to `ode` lies on top of the more accurate solution curve. That is the two solutions coincide to *graphical accuracy*.

For this simple model it is possible to find an analytic formula for $u(t)$. In fact

$$u(t) = \frac{1}{1 + c\exp(-kt)} \quad \text{where} \quad c = P - 1. \tag{8.3}$$

We can add this formula to the above graph as a final check.

```
t =  0:100 ;
c = 10000-1 ; k = .2 ;

function u=exactu(t)
u = 1 ./ ( 1 + c*exp(-k*t));
endfunction

xbasc(); plot2d(t,u); fplot2d(t,exactu,style=-1 )
```

The exact formula gives a curve that is the same as the numerical solution. In fact the difference between the two solutions is at most 2.5×10^{-7}. So we can be confident in using `ode` to solve the problems below.

Exercise 4.

1. *Check that (8.3) is a solution to (8.1). That is, when we differentiate this formula for $u(t)$ we get the right side of (8.1). And this formula satisfies (8.2)*

2. *From the graph of $u(t)$, estimate how many days are needed for half the population to be infected?*

   Hint: You can inspect the plot to get a solution up to the closest day. The command `xgrid` *will make this easier to answer. You could also create a function using the result from the ode procedure, which returns the ode solution minus 0.5 at any particular time t. Then use* `fsolve` *to solve the problem.*

3. *Suppose now that after the infection is noted at day $t = 0$, an inoculation program is started from day 5, and 200 of the uninfected animals are inoculated per day. After t days, the proportion of animals inoculated per day is $200(t-5)_+/10000$. Here $(t-5)_+$ means $\max(t-5, 0)$. These inoculated animals cannot be infected, an new infections occur when an uninfected, un-inoculated animal meets an infected animal. Thus the rate of increase of u is now modelled by the differential equation*

$$\frac{du}{dt} \quad = \quad k\,u(t) \;\; (1 - u(t) - .02\,(t-5)_+)_+\,;$$

   Now, how many animals are uninfected after 100 days with the inoculation program?

4. *How many animals would be uninfected if only 100 animals could be inoculated each day from day 5? How many animals would have been saved if 100 animals per day were inoculated from day 0?*

This simple model problem also has a formula for the exact solution. However in Scilab it is very easy to get a numerical solution; and this can be done even in complicated problems when there is no simple formulae for the solution. Using Scilab we can simply explore the model by altering the parameters and plotting the solutions!

Chapter 9

Chemical Reaction Example

9.1 Chemical Reaction Systems

Many interesting ODE's can be constructed to describe the evolution of chemical reactions. Suppose we have 4 chemical species A, B, C and D. These might denote O_2, H_2, OH, H_2O etc. We can write one reaction between these species via a chemical equation

$$A + B \xrightarrow{k_1} C + D \tag{9.1}$$

This reaction will not take place instantaneously, but will react at a rate proportional to the concentration of the reactants A and B with a rate constant k_1. Suppose we have a second equation

$$A + A + C \xrightarrow{k_2} B \tag{9.2}$$

and we suppose these two reaction equations fully specify the reactions taking place in our system.

We can generate the evolution equations for the concentrations of chemical species as follows

$$\begin{aligned}
\frac{d[A]}{dt} &= -k_1[A][B] - 2k_2[A][A][C] \\
\frac{d[B]}{dt} &= -k_1[A][B] + k_2[A][A][C] \\
\frac{d[C]}{dt} &= k_1[A][B] - k_2[A][A][C] \\
\frac{d[D]}{dt} &= k_1[A][B]
\end{aligned}$$

where the square brackets denote concentrations.

Lets look at the first equation

$$\frac{d[A]}{dt} = -k_1[A][B] - 2k_2[A][A][C]$$

The rate of reaction (9.1) will depend proportionally on $[A]$ and $[B]$, the constant of proportionality given by k_1. So the rate of reaction 1 will be given by $k_1[A][B]$. Each instance of reaction 1 will use up one unit of A. So the rate of change of concentration of A due to reaction 1 will be equal to $-k_1[A][B]$. The minus signifying one unit being used up. There are similar terms in the other ODE's corresponding to using up the species B and creating species C and D.

The rate of the second reaction will depend proportionally on $[A]$, $[A]$ again, and $[C]$ and so the rate of reaction (9.2) will be $k_2[A][A][C]$. Each reaction (9.2) will use up 2 units of A, so the rate of change of concentration of A due to reaction 2 will be equal to $-2k_1[A][A][C]$, with similar terms in the ODE's denoting using one unit of C and creating one unit of B.

The creation of the differential equations can be automated given the reaction equations and the corresponding rates. We leave it to the Chemists to give up a reasonable set of reaction equations with corresponding rates constants.

Now lets see how we can use SCILAB to approximate the solution to this system of ODE's. We have 4 chemical species, so we will have a system of 4 equations. We will make the relationship $x_1 \leftarrow [A]$, $x_2 \leftarrow [B]$, $x_3 \leftarrow [C]$, $x_4 \leftarrow [D]$.

In SCILAB we would encode the associated ode as

```
function dx = chemical(t,x)

f1 = k1*x(1)*x(2)
f2 = k2*x(1)*x(1)*x(3)

dx(1) = -f1 - 2*f2
dx(2) = -f1 +   f2
dx(3) =  f1 -   f2
dx(4) =  f1
endfunction
```

(Be careful to get the t and x arguments in the correct order!)

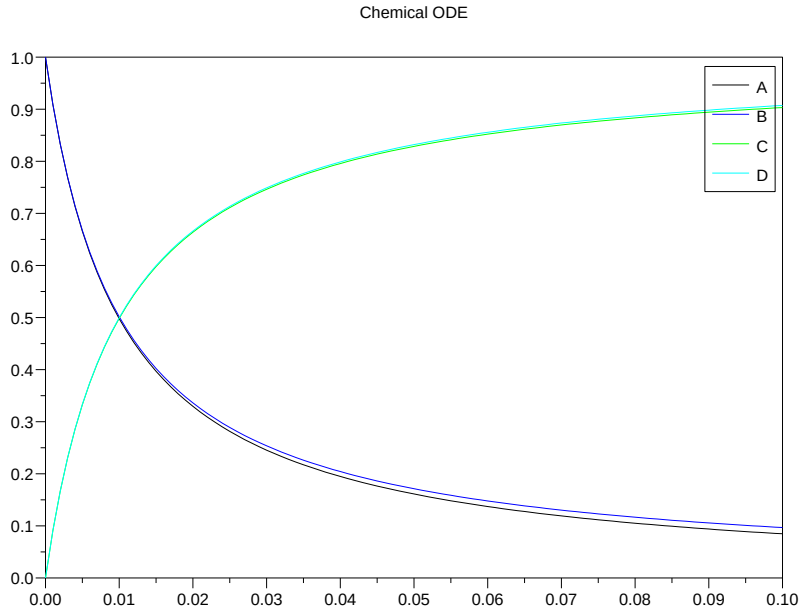
Setting up the rate constants and an initial condition, we can obtain approximations to the solution at the times $t = 0, 0.001, 0.002, ...0.1$ using the code

```
k1 = 1e2
k2 = 1

t = 0:0.001:0.1;
x0 = [1 ;1; 0; 0];

x = ode(x0,0,t,chemical);
```

It is nice to produce some graphical output. I used

```
xbasc()
plot2d(t,x(1,:),style=1)
plot2d(t,x(2,:),style=2)
plot2d(t,x(3,:),style=3)
plot2d(t,x(4,:),style=4)

legends(['A','B','C','D'],[1 2 3 4],"ur")
xtitle('Chemical ODE')
```

to produce the plot



9.2 Robertson's Example

An idealised chemical reaction system which is often used to generate a stiff system of ODE's is given by the reaction equations

$$\begin{aligned} A &\xrightarrow{0.04} B \\ B+B &\xrightarrow{3.0e7} B+C \\ B+C &\xrightarrow{1.0e4} A+C \end{aligned}$$

There are 3 chemical species and so there will be a corresponding system of 3 ODE's modelling the evolution of the concentrations of these chemicals. The choice of rate constants

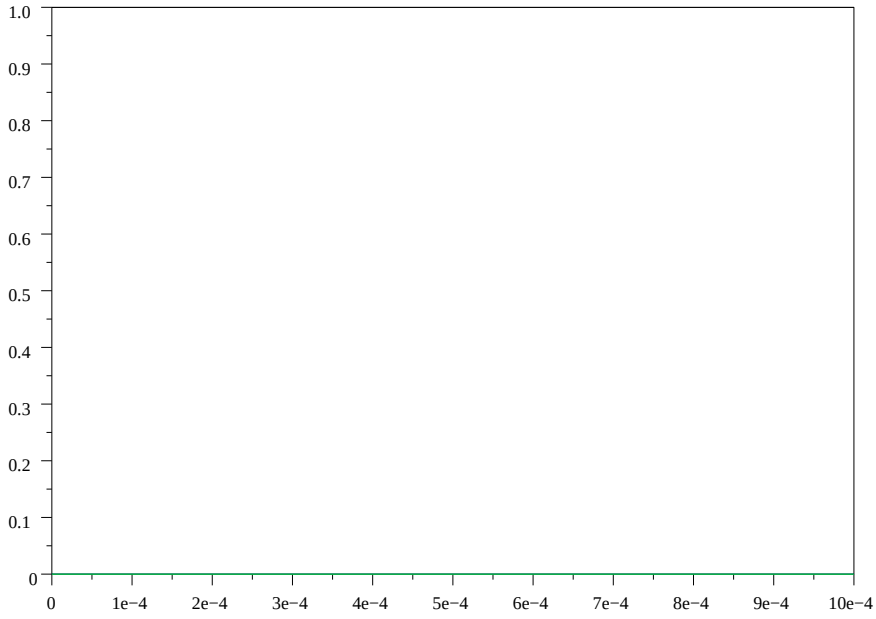
tells us that the first reaction will evolve slowly, the second reaction very fast, and the final reaction fast.

Exercise 5.

1. *Derive the system of ODE's corresponding to Robertson's example.*
2. *Use* Scilab *to solve these equations with initial concentrations* $[A] = 1$, $[B] = [C] = 0$ *for* $t \in [0, 0.001]$. *Plot your results on the same graph using the command of the form*

   ```
   xbasc();plot2d(t',y',leg='A@B@C')
   ```

   I obtained the plot



   The concentrations of B *and* C *are very small compared to* A. *It is sensible to normalise these concentrations by the maximum value of each concentration. We can use the* `max` *function, using the* `'c'` *argument to produce a column vector containing the maximum of each row. I.e*

   ```
   ymax = max(y,'c');
   ```

   The trick is now to scale each row of `y` *by the corresponding entries of* `ymax`. *I came up with*

```
yscaled = diag(1.0./ymax)*y;
```

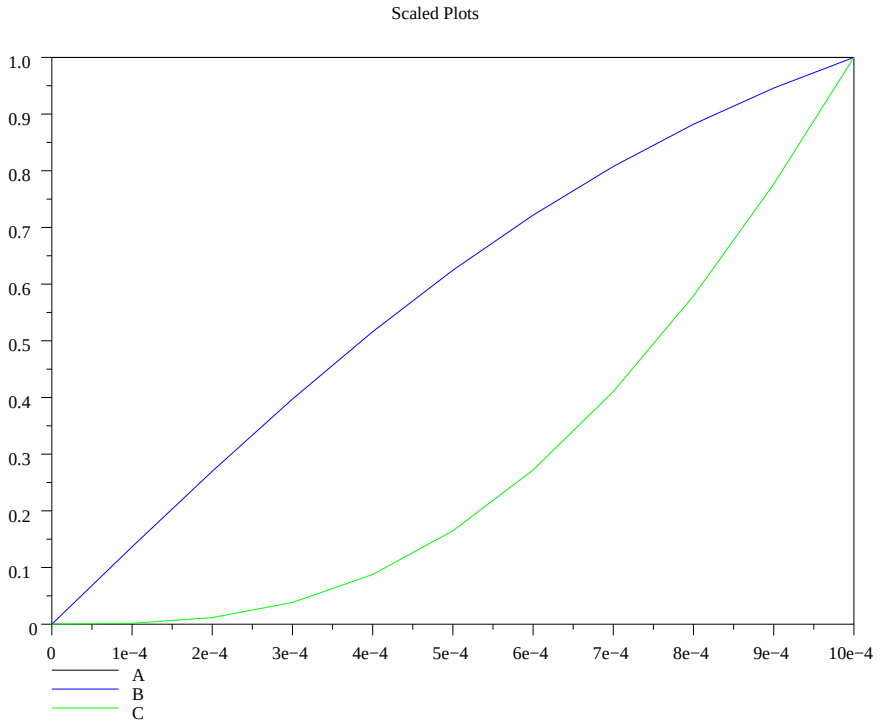
Can you see what this is doing?

Now you can plot `t` *versus* `yscaled` *with*

```
xbasc();
plot2d(t',yscaled')
legends(['A';'B';'C'],[1;2;3])
```

I suggest producing a legend so that you can see which curve corresponds to which variable. The format of the command associates a character string (such as `'A'` *with a line style* `1`*.*

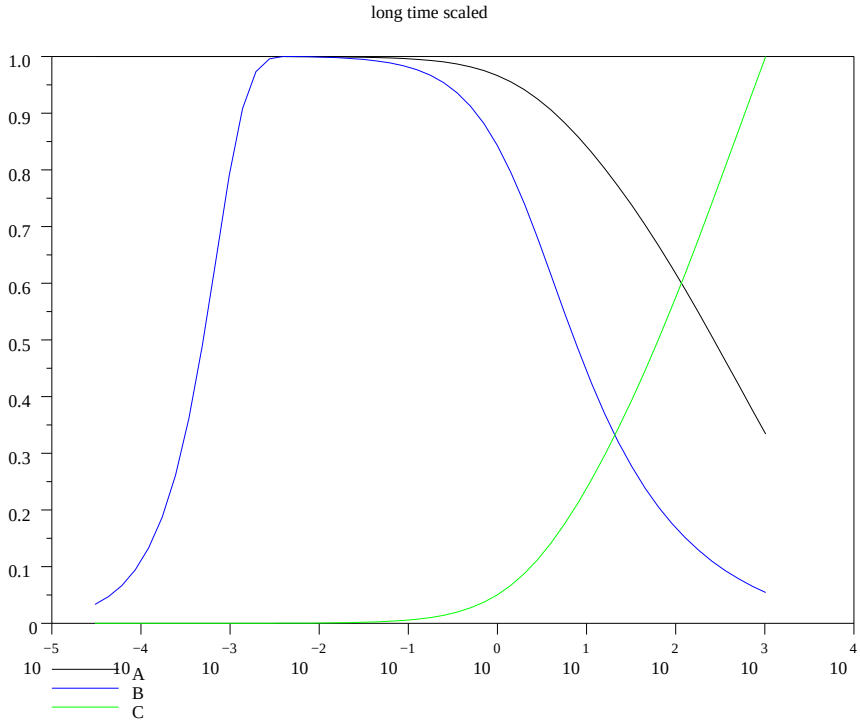
I now obtained the plot



3. *Interesting things happen for very small times as well as for much larger times. To see these things produce approximations of the concentrations at times* 2^i*,* $i = -20, ..., 10$*. Plot the scaled three concentrations on the same plot using the previous scaled method. You should use a log scale for the time axis. Use the* `'ln'` *argument which tells* `plot2d` *to plot the first axis using* ***l****og scale and the second axis using* ***n****ormal scale*

```
xbasc();
plot2d('ln', t',yscaled')
legends(['A';'B';'C'],[1;2;3])
```

Here is the plot now



4. *The concentrations are still in a transient state at time* $t = 1024$. *Run the calculation to larger times to ascertain the long time behaviour of the system. What concentrations do* A B *and* C *limit to?*

Chapter 10

Boundary Value Problems

10.1 Introduction

This tutorial is designed to familiarise you with the shooting method for solving boundary value problems. This is essentially a component of Heath's computer problem 10.1. But first we will have a look at a method for solving general equations

10.2 Plotting

Suppose we wanted to find the value of $\mathbf{x} = (x_1, x_2)$ such that

$$\begin{aligned} k(x_1^2 + x_2^2) - 2 &= 0 \\ k\sin(x_1 x_2) &= 0 \end{aligned} \tag{10.1}$$

for some value of k.

Perhaps the first thing we should do is plot these functions. We need to redefine the function so that we can use the standard SCILAB plotting routines. I suggest defining two functions for the two components of the function

```
function z = eqn1(x,y)
z = k*(x**2+y**2) - 2;
endfunction

function z = eqn2(x,y)
z = k*sin(x*y);
endfunction
```

We can plot the functions using the `fplot3d` or the `fcontour2d` functions. Try

```
k = 1;
x = -4:0.1:4;
```

```
y = x;
xbasc(); fplot3d(x,y,eqn2)
```

Notice that we have set the value of k and it is being passed through to the function `eqn1` implicitly.

Contour plots can sometimes be easier to work with. A contour plot for the first function can be obtained via

```
xbasc();  fcontour2d(x,y,eqn1,[0 5 10 20 25]);
```

Not that we have asked for the contour lines for 0 up to 25 in steps of 5. Make sure you have a look at the plot.

Now let us over plot with the contours of the second function.

```
fcontour2d(x,y,eqn2, [0,1]);
```

The overall plot is quite complicated (at least to me). Perhaps you might like to plot both functions separately to get an idea of the "shape" of both. But with both contour plots on the same graph, use the 2d zoom facility of SCILAB to get an approximate solution (3 sig figs) of equation (10.1) (for $k = 1$).

10.3 Equation Solver

It is useful to automate the solution of equations. We have studied the use of Newton'e method earlier in the course. SCILAB provides a procedure to solve general equations. The procedure is called `fsolve`. This procedure requires an initial guess and the equations written as a SCILAB function. For our problem we would produce a SCILAB function as such

```
function y = f(x)
y(1) = eqn1(x(1),x(2));
y(2) = eqn2(x(1),x(2));
endfunction
```

where the functions `eqn1` and `eqn2` were defined in the last section. Of course you could define the function without using `eqn1` and `eqn2`. Unfortunately the `fsolve` function expects a very particular form, which is different than the calling form for the plotting routines.

We would store this in a file and use `exec` to load the function. We could then solve the equation using `fsolve`, as such

```
k=1;
[x,y] = fsolve([0;0], f)
```

Does the solution provided by `fsolve` match the one you discovered manually?

It might be useful to get some feedback about what `fsolve` is doing. One way is to provide some output from the function. Redefine the function `f` as such:

```
function y = f(x)
y(1) = eqn1(x(1),x(2));
y(2) = eqn2(x(1),x(2));
printf('x(1)=%15e x(2)=%15e y(1)=%15e y(2)=%15e\n',x(1),x(2),y(1),y(2))
endfunction
```

Take note of the `printf` function. This is the formatted print command (like the corresponding C procedure). The formatting is accomplished using the codes like `%15e` which will produce a decimal number using 15 characters.

Lab Book: Produce graphical output from your `f` function, so that the values of `x(1),x(2)` are plotted on each invocation of the function. I.e. add the command `plot2d(x(1),x(2),style = -1,rect = [-2 -2 2 2])`

10.4 Two Point Boundary Problem

Consider the two-point boundary value problem

$$y'' = 10y^3 + 3y + t^2, \quad 0 \le t \le 1,$$

with boundary conditions

$$y(0) = 0, \quad y(1) = 1,$$

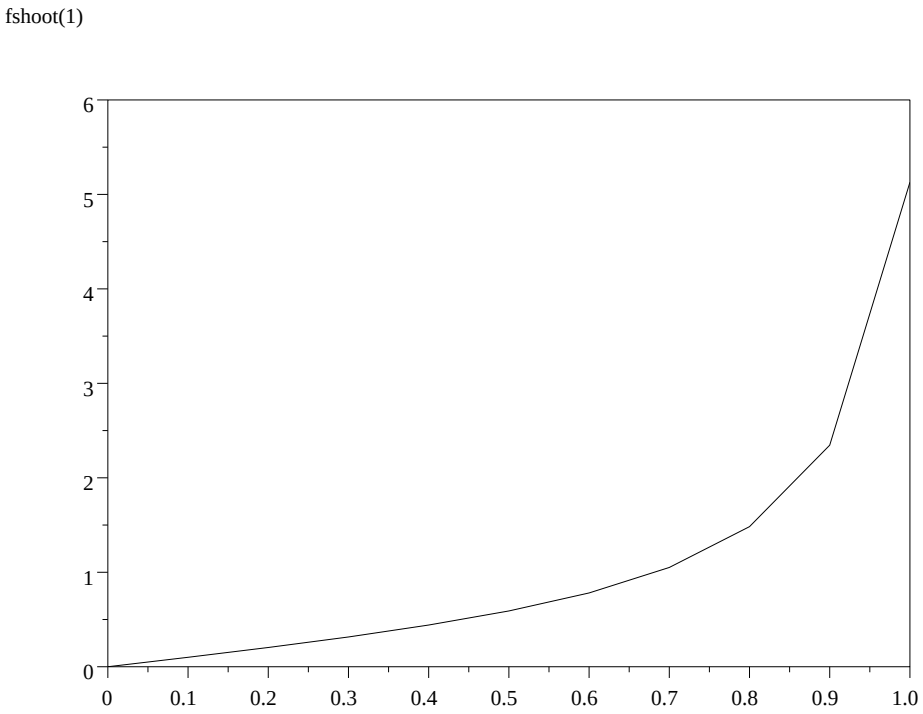
We will try to use the shooting method to solve this problem.

To use the shooting method we must first be able to solve the initial value problem. Then we must be able to test many initial conditions to find our required value.

- First setup a function which defines the differential equation using the calling sequence

  ```
  function udot = shootode(t,u)
  ```

- Check your function by solving the ode with initial conditions $y(0) = 0$, $y'(0) = 1$.
- Plot the solution. Mine looks like the following

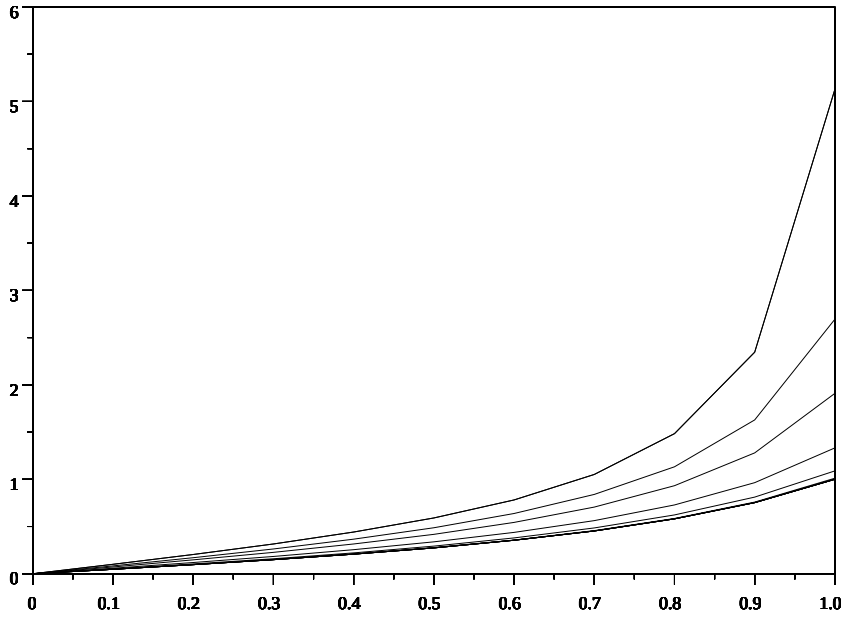


- What is the value of y at 1? (I got about 5.1).
- Take your ode solver (with your plotting routines) and produce a function with calling sequence

  ```
  function y = fshoot(s)
  ```

  which takes an initial value for the slope at 0 and returns the value of $y(1) - 1$. For instance, `fshoot(1.0)` should return approximately 4.1.
- Finally use the `fsolve` function to find an approximation to the boundary value problem.

Here is the graphical result I obtained when running `fsolve(1,fshoot)`.



10.5 Exercises

Solve the two-point boundary value problem

$$y'' = 8y^3 + 3y - 10, \quad 0 \le t \le 1,$$

with boundary conditions

$$y(0) = 0, \quad y(1) = 2,$$

Hint: You might have to hunt for a good starting value for the slope, somewhere between 2 and 3.9.

What is the slope at 0 for the function which solve this boundary value problem?

Chapter 11

Heat and Wave Equations

11.1 Introduction

This tutorial is designed to familiarise you with some methods for solving time dependent partial differential equations. In particular we will implement some of the finite difference methods introduced in lectures to solve the Heat equation and the Wave equation. In this tutorial I will take you through the development of a series of SCILAB functions which will implement the forward and backward Euler method for the Heat equation and the leap frog method for the wave equation.

To save typing you can cut some of the code described in this tutorial into a file on your machine which you can run with SCILAB. I will assume that you use a file named `pde.sce` into which you put all your code.

To recall the Heat equation is

$$u_t = c\,u_{xx} \qquad 0 \le x \le 1, \quad t \ge 0$$

$$u(0,x) = f(x) \qquad u(t,0) = \alpha \qquad u(t,1) = \beta$$

and the Wave equation is

$$u_{tt} = c\,u_{xx} \qquad 0 \le x \le 1, \quad t \ge 0$$

$$u(0,x) = f(x) \qquad u_t(0,x) = g(x)$$
$$u(t,0) = \alpha \qquad u(t,1) = \beta.$$

11.2 Heat Equation

We will first concentrate on the Heat equation. The simplest finite difference method is based on Euler's method and is given as

$$u_j^{k+1} = u_j^k + \frac{c\Delta t}{\Delta x^2}\left(u_{j+1}^k - 2\,u_j^k + u_{j-1}^k\right)$$

$$u_j^0 = f(x_j) \qquad u_0^k = \alpha \qquad u_n^k = \beta$$

where u_j^k represents an approximate of the solution $u(k\Delta t, j\Delta x)$. Here we have discretised the interval $[0, 1]$ into $n + 1$ points $x_j = j\Delta x$ where $\Delta x = 1/n$ and the time interval into discrete times $t_k = k\Delta t$.

To implement this time stepping method we need a vector to contain the u values at any particular time. Indeed it is useful to have another vector for the previous time. As input to our heat function we specify

1. the number of subintervals `n` in the x dimension,
2. the size of the time step `dt`,
3. a function `f` defining the initial conditions,
4. the left boundary value `alpha` and
5. the right boundary value `beta`.

Our SCILAB function will have the following basic structure

```
//-----------------------------------------------------------------
function [] = heat(n,dt,f,alpha,beta)

//-------------------------
// Initialise plotting
//-------------------------
initMyplot()

//-------------------------
// Setup parameters
//-------------------------

//-------------------------
// Setup initial conditions
//-------------------------

//-------------------------
// Setup Boundary conditions
//-------------------------

//-------------------------
// Plot the initial conditions
//-------------------------

//-------------------------
```

```
// The main time loop
//-------------------------

//-------------------------
// Close
//-------------------------
closeMyplot()

endfunction
//-----------------------------------------------------------
```

Cut and paste this into your SCILAB file `heat.sce`.

Now let us expand the code. First, let us look at setting up parameters, as this allows us to define some variables for subsequent use. I chose the following parameters

```
//-------------------------
// Setup parameters
//-------------------------
c = 1.0;
dx = 1/n;
lambda = dt/dx^2*c;

x = (0:dx:1)';
t = 0;
```

Note that I am using the comment lines to help you place the code correctly.

I needed `c` to define the constant in the heat equation, `dx` is obvious, `lambda` is a useful combination of parameters that occurs in the updating of the solution vector (by the way, the `dt` is being passed in so that you can play with changing it to test stability). I have created an `x` vector so that I can plot our result x versus u. A variable for time `t` is useful for the main time loop. Cut and Paste this code into your file `pde.sce` in the initial part of the function.

The initial condition will be given by the value of a function `f` at the x values. It turns out that we also need a second u vector `u1` which we create at this point. We simply use

```
//-------------------------
// Setup initial conditions
//-------------------------
u0 = f(x);
u1 = zeros(u0);
```

We will need to define a function f. Lets do something simple like a sin curve. I used

```
//-----------------------------------------------------------
function u=fsin(x)
```

```
k=1
u = sin(%pi*k*x)

endfunction
//-----------------------------------------------------------------
```

Put this in the file `pde.sce` file.

The boundary conditions can be simply set to α at the left hand end of `u0` and β at the right hand end.

```
//--------------------------
// Setup Boundary conditions
//--------------------------
u0(1) = alpha;
u0($) = beta;
```

In most simulations it is useful to plot out results. This is particularly useful when coding up a method as it often points to errors in the code. So lets plot the initial condition. I suggest something like

```
//--------------------------
// Plot the initial conditions
//--------------------------
myplot(x,u0)
```

The `xselect()` will bring the graphics window to the front and the `myplot` function is just a function to plot x versus u vectors. Copy this code to the correct position within your file.

Now we also have to define our plotting routine. I used

```
//-----------------------------------------------------------------
function []=initMyplot()
//
// Setup pixmap, this is good for animations
//
xselect()
f=gcf();
f.pixmap='on'

endfunction

function []=closeMyplot()
//
// Turn off pixmap mode
```

```
//
f=gcf();
f.pixmap='off'

endfunction

function []=myplot(x,u)

xbasc();
plot2d(x,u,rect=[0,-1,1,1]);
xstring(0.9,0.9,'time:'+string(t));
show_pixmap()


endfunction
//-----------------------------------------------------------------
```

The main part of this function is just the standard `plot2d` command. I have added `show_pixmap()` to make the plots look nicer when we have our time evolution working.

Lets try our code. Save your file and in the Scilab command window load in the functions and run the `heat` function.

```
n=20; heat(n,0.5/n^2,fsin,0,-1)
```

The right boundary condition doesn't match up with the initial conditions. We could run our method with this incompatibility, but let's fix up the boundary conditions. With new boundary conditions try

```
n=20; heat(n,0.5/n^2,fsin,0,0)
```

Nothing happens.

Now we can go back the code and add in the time evolution part. We need a loop, we can use a `for` loop but I like using a `while` loop based on time. I.e. I suggest using

```
//-------------------------
// The main time loop
//-------------------------
while t<1
    t=t+dt
    //--------------------------------
    // Update the solution vector
    //--------------------------------


    myplot(x,u0)
end
```

This loop is a little incorrect, in that it may overshoot the final time by `dt`. Not to worry. The main thing now is how to update the solution vector. The most obvious way is to use a `for` loop to implement the finite difference update

$$u_j^{k+1} = u_j^k + \frac{c\Delta t}{\Delta x^2}\big(u_{j+1}^k - 2\,u_j^k + u_{j-1}^k\big)$$

The u_j^k values will be stored in `u0` and the new values u_j^{k+1} in the vector `u1`. Once the update is completed we will over write `u0` with the new `u1`. The code is

```
//--------------------------------
// Update the solution vector
//--------------------------------
for j=2:n
    u1(j) =  (1-2*lambda)*u0(j) ...
            + lambda*u0(j+1) + lambda*u0(j-1);
end
u0 = u1;
```

It is always important to get the range of `j` correct. We only need to work with the interior points. There are actually $n+1$ entries in `u0` and `u1` and so the range `j=2:n` corresponds to interior points.

An alternative method to the `for` loop is to use the array notation in Scilab

```
//--------------------------------
// Update the solution vector
//--------------------------------
u1(2:$-1) = (1-2*lambda)*u0(2:$-1) ...
          + lambda*u0(3:$) + lambda*u0(1:$-2);
u0 = u1;
```

This second method will be more efficient in Scilab though the `for` method is closer to what you would use in C or Fortran.

Use the first method initially and change over to the second if the evolution seems slow. As we will be plotting the solution for every iteration the overall speed of the method will be slow anyway. The array method becomes more important for higher dimensional problems.

Lets see if our method works. Add in the time loop code, load in the code with `exec` and run the `heat` command again. You might like to increase the number of grid points to say 100.

To stop the evolution you can use the `Control` menu with the pause, resume and abort commands.

Lets try another initial condition. Go back to your code and add the following function to your utility file.

```
//-----------------------------------------------------------------
function u=fline(x)

u = 0.5-x;

endfunction
//-----------------------------------------------------------------
```

This is a solution of the Heat equation with the boundary conditions $\alpha = 0.5$ and $\beta = -0.5$. The exact solution to the heat equation with this set of boundary conditions is the straight line between these two points. Let's check how our solver works.

```
exec heat.sce;
n=100;heat(n,0.5/n^2,fline,0.5,-0.5)
```

Why is the solution changing? Go back to your code and find the bug and fix it so that the boundary condition is set properly.

11.3 Stability of the Method

Our analysis of the method tells us that we need $\Delta t < \frac{\Delta x^2}{2c}$ for stability. Let's see what happens if we go over this constraint. Try

```
n=100; heat(n,0.52/n^2,fsin,0.0,0.0)
```

After some time you should see instabilities arising. You can use cntrĈ to interrupt the code and resume to start it up again.

11.4 Wave Equation

Let's quickly produce an implementation of the leap frog method for the wave equation. We need to implement the algorithm

$$u_j^{k+1} = 2\,u_j^k - u_j^{k-1} + c\frac{\Delta t^2}{\Delta x^2}\left(u_{j+1}^k - u_j^k + u_{j-1}^k\right)$$

We will use three vectors `u2` to hold u_j^{k+1}, `u1` to hold u_j^k and `u0` to hold u_j^{k-1}. The structure is similar to that for the heat equation.

```
//-----------------------------------------------------------------
function [] = wave(n,dt,f,g,alpha,beta)

//-------------------------
```

```
// Initialise plotting
//-------------------------
initMyplot()

//-------------------------
// Setup parameters
//-------------------------
c = 1.0;
dx = 1/n;
lambda = dt^2/dx^2*c;

x = (0:dx:1)';
t=0;

//-------------------------
// Setup initial and
// boundary conditions
//-------------------------
u0 = f(x);
u0(1) = alpha; u0($) = beta;

u1 = zeros(u0);
if (g==1) then
  u1(2:$-1) = u0(1:$-2);
elseif (g==-1) then
  u1(2:$-1) = u0(3:$);
else
  u1(2:$-1) = 0.5*(u0(1:$-2)+u0(3:$));
end

u1(1) = alpha; u1($) = beta;

u2 = zeros(u0);
u2(1) = alpha; u2($) = beta;

//-------------------------
// Plot the initial conditions
//-------------------------
xselect()
myplot(x,u0)
myplot(x,u1)

//-------------------------
```

```
// The main time loop
//-------------------------
while t<20
    t=t+dt;
    //--------------------------------
    // Update the solution vector
    //--------------------------------
    for j=2:n
        u2(j) = (2-2*lambda)*u1(j) - u0(j) ...
                + lambda*u1(j+1) + lambda*u1(j-1);
    end
    u0=u1;
    u1=u2;

    myplot(x,u2)
end

//-------------------------
// Close
//-------------------------
closeMyplot()


endfunction
//---------------------------------------------------------------
```

11.5 Experimentation

We can use the initial conditions for u that we used for the heat equation.

Add in the code in the previous subsection to a `wave.sce` file. Load in the new functions using `exec`. Now play around with the code. For instance try

```
n=100; wave(n,1/n,fsin,0,0,0)
```

Hopefully you will see an "oscillating string".

Lets try a different initial condition, A bump in the middle of the string.

```
//---------------------------------------------------------------
function u = fbump(x)

c = 0.5; w = 0.125;
cl = c-w;
cr = c+w;
```

```
u = bool2s(x>cl & x<=cr)

endfunction
//-----------------------------------------------------------------
```

See how we can use comparisons to create interesting functions.

Now try out this initial condition

```
n=100; wave(n,1/n,fbump,0,0,0)
```

It looks cool as the disturbance reflects off the boundaries. It is amazing, we are actually getting the exact solution.

You can get the bump to move either to the right or left using the g argument. Try

```
n=100; wave(n,1/n,fbump,1,0,0)
```

and

```
n=100; wave(n,1/n,fbump,-1,0,0)
```

11.6 Stability

Try testing the stability of the method for slightly larger timesteps. For even slightly larger timesteps than $1/n$ you should see growing modes in the solution, especially if you use the bump as an initial condition. The effect is not seen so quickly if you use smoother initial conditions.

11.7 Exercises

1. You might like to play with a solver for the heat equation which uses the implicit Euler method. Here is the code

```
//-----------------------------------------------------------------
function [] = iheat(n,dt,f,alpha,beta)

//-------------------------
// Initialise plotting
//-------------------------
initMyplot()

//-------------------------
// Setup parameters
//-------------------------
```

```
c = 1.0;
dx = 1/n;
lambda = dt/dx^2*c;

x = (0:dx:1)';
t=0;

//--------------------------
// Call f to setup initial
// conditions.
//--------------------------
u0 = f(x);


//--------------------------
// Setup the matrix A as a
// sparse matrix
//--------------------------
A = (1+2*lambda)*diag(sparse(ones(n-1,1))) ...
   -lambda*diag(sparse(ones(n-2,1)),-1) ...
   -lambda*diag(sparse(ones(n-2,1)),1);


//--------------------------
// Setup Boundary conditions
//--------------------------
u0(1) = alpha
u0($) = beta
bc = zeros(n-1,1)
bc(1) = lambda
bc($) = lambda

//--------------------------
// Plot the initial conditions
//--------------------------
xselect()
myplot(x,u0)

//--------------------------
// The main time loop
//--------------------------
while t<20
    t = t+dt
```

```
    bc = u0(2:$-1);
    bc(1) = bc(1) + u0(1)*lambda
    bc($) = bc($) + u0($)*lambda
    u0(2:$-1) = A\bc
    myplot(x,u0)
end

//--------------------------
// Close
//--------------------------
closeMyplot()


endfunction
//---------------------------------------------------------------------
```

Try

```
n=100; iheat(n,1/n,fsin,0,0)
```

Try

```
n=100; iheat(n,1/n,fbump,0,0)
```

You might even like to try larger values of n to slow down the evolution!

2. Use the code for `heat` and `iheat` to produce a program that solves the heat equation using the Crank Nicolson method. Try a smooth initial condition like `fsin` and a rough initial condition like `fbump`. Something strange happens with the rough initial condition.

   You might like to experiment with Crank Nicolson using `fsin`, but with progressively high frequencies. Can you make a conjecture about the relative efficiency of Crank Nicolson to damp out errors at different frequencies. How does this explain the observed behaviour of Crank Nicolson method on the `fbump` initial data.