AARMS-CRM Workshop on NA of SPDEs, July 2016

MATLAB primer

Niall Madden (Niall.Madden@NUIGalway.ie)

(If you are familiar with MATLAB, you can skip this)

MATLAB is the standard tool for numerical computing in industry and research. It specialises in matrix computations (Matrix Laboratory), but includes functions for graphics, numerical integration and differentiation, solving differential equations, image and signal analysis, and much more.

GNU Octave is a free, open source, implementation of MATLAB. Its GUI, IDE and graphics capabilities are not quite as well-developed as MATLAB's, but all of the examples given in this session will work in Octave.

MATLAB is an *interpretive* environment – you type a command and it will execute it immediately. Nonetheless, one can group a set of commands together into a script or function file.

The details given below cover just enough of the fundamentals to get started. For further reading I suggest the following books. In particular, the first is freely available.

- Cleve B. Moler, *Numerical Computing with MATLAB* ([Moler, 2004]). Written by the creator of MATLAB, it mixes MATLAB programming with theory and algorithms of numerical methods. Also freely available from the MathWorks site.
- Tobin A. Driscoll, *Learning MATLAB* ([Driscoll, 2009]). An excellent primer if you are just starting to learn MATLAB.
- Desmond Higham and Nicholas Higham, *MATLAB Guide* ([Higham and Higham, 2005]), is detailed and well-written. If you know a little MATLAB, this is a great book to help you develop your skills and deepen you knowledge.

1 The Basics

1.1. In MATLAB, everything is a *matrix*. A scalar variable is just a 1×1 matrix. To check this set, say, `t = 10`, and use the `size()` command to find the numbers of rows and columns of `t`.

1.2. To declare a row-vector array, try:

```
>> x=[-4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Or, more simply,

```
>>  x=-4:4
```

To access, say, the 3rd entry

```
>>  x(3)
```

1.3. We usually like to think of vectors as *column* vectors. To define one, try

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

Or you can take the (Hermitian) transpose of a row vector: `>>  x = [1,2,3]';`

Verify that is the Hermitian transpose by defining a complex-valued vector, and looking at its transpose:

```
>>  i = sqrt(-1); x=[i, 1+i, 2]
>>  x'
```

1.4. If you put a semicolon at the end of a line of MATLAB, the line is executed, but the output is not shown. (This is useful if you are dealing with large vectors). If no semicolon is used, the output is shown in the command window.

1.5. We'll often want to run a collection of commands repeatedly. So, rather than type them individually, create a file containing the following code

```
x=-4:4
for i=1:9
  y(i) = cos( x(i) );
end
plot(x,y);
```

Save this as, say `class1.m`. To execute it, just type `>> class1` in the MATLAB command window.

Your file is an example of a MATLAB *script file*.

1.6. If the picture isn't particularly impressive, then this might be because MATLAB is actually only printing the 9 points that you defined. To make this more clear, use

```
plot(x, y, '-o')
```

This means to plot the vector `y` as a function of the vector `x`, placing a circle at each point, and joining adjacent points with a straight line.

1.7. The plot generated is not particularly good. The points plotted are a unit apart. To get a better picture, try "easy plot" `>> ezplot(@cos,[-4,4])`

1.8. A row vector may be declared as follows:

```
>>  x = a:h:b;
```

This sets $x_1 = a$, $x_2 = a+h$, $x_3 = x+2h$, ..., $x_n = b$. If `h` is omitted, it is assumed to be 1.

1.9. The script file from Part (5) is a little redundant. In MATLAB, most functions can take a vector or matrix as an argument. So, in fact, we can just use

```
>>  y = cos( x )
```

which sets `y` to be a vector such that $y_i = \cos(x_i)$.

1.10. The `*` operator performs matrix-matrix multiplication. So, to compute the inner product of the (column) vector $\mathbf{x}$, try `>> IP = x'*x;`

For element-by-element multiplication use `.*` For example, `y = x.*x` sets $y_i = (x_i)^2$.
So does `y = x.^2`.

2 Matrices

2.1. Declare a Matrix as
`>> A = [3 -1 ; -2 3]`

2.2. The entry in row `i` and column `j` of a matrix is given by `A(i,j)`
The i^{th} row of matrix A can be addressed as `A(i,:)`, and the jth column as `A(:, j)`.

2.3. To compute the inverse of a matrix (where possible)

```
>> inv(A)
ans =
   4.2857e-01   1.4286e-01
   2.8571e-01   4.2857e-01
```

Other common linear algebra functions are also available, e.g., `det`, `trace`, `rank` .

2.4. To compute the Singular Value Decomposition of a matrix, use `svd`. It can be used in two ways:
`>> sigma = svd(A)` returns the singular values of A and stores them in the vector `sigma`.
`>> [U,Sigma,V] = svd(A)`
computes that factorisation (in MATLAB notation) `A=U*Sigma*V'`

2.5. Other useful functions include

- `>> A = rand(m,n)` – creates a matrix with (uniformly distributed) random entries. Use `randn` to get normally distributed entries. The functions `zeros(m,n)` and `ones(m,n)` return decidedly nonrandom matrices.
- `>> I = eye(n)` – identity matrix
- `>> Z = complex(A,B)` – where A and B are real matrices of the same size, sets $Z = A + iB$.
- `>> E = eig(A)` – (tries) to return the eigenvalues of A.
- `>> norm(x)` computes the 2-norm of the vector (or Matrix) x. `norm(x,p)` computes the p-norm, and `norm(x, inf)` returns $\|x\|_\infty$.
  `>> norm(A, 'fro')` computes the Frobenius norm of the matrix A.

2.6. It A is an $n \times n$ matrix, and b is a vector with n entries, we can solve $Ax = b$ using $x = A\backslash b$.

2.7. Many of the matrices one encounters, e.g., when solving boundary values problems are *sparse*, meaning that they have relatively few non-zero entries. In that case, by defining a matrix to be sparse, great efficiencies can be achieved. More about this later...

3 Functions

In MATLAB, you can write your own functions in several ways, including,

Anonymous functions: Used for simple functions (one line of code), e.g.,
`>> f = @(x)sin(pi*x)`
Try `>> ezplot(f,[-4,4])`
For functions of two (or more) variables, the syntax is `>> z = @(x,y)(exp(-x).*y.*(1-y))` Try:
`>> [X,Y]=meshgrid(linspace(0,1));`
`>> mesh(z(X,Y))` *or* `>> surf(z(X,Y))`

Function files: Create a file called say, `MyFunction.m`. Its first line should have the keyword `function`, followed by the return values, the function name (same as the file), and the argument list:

```
1 function [OutputArgs] = FileName(InputArgs)
```

For example, the following function takes a vector as its argument, and if it is not a column vector, returns its transpose.

```
1 function v = tocolumn(x)
2 if ( min(size(x)) ~= 1 || size(x,1)==2)
3     v=x;
4 else
5     v=x';
6 end
```

4 Initial value problems

4.1. **Solving ODEs.** MATLAB/Octave has a set of numerical ODE solvers. Some are specialised; the work-horse is `ode45`. The general form on an initial value problem (IVP) is:

$$y'(t) = f(t, y) \quad t > t_0$$

$$y(t_0) = y_0.$$

We'll try and solve a particular example: $y(0) = 1$, and

$$y'(t) = y \sin(t) \quad t > 0,$$

on the interval $[0, 4]$. The exact solution is $y(t) = e^{1-\cos(t)}$.

First we define the RHS: `>> f = @(t,y)(y.*sin(t));`
Then solve the ODE: `>> [T,Y] = ode45(f,[0,4],1);`
Now define the true solution (for comparison):
`>> y_true = @(t)exp(1-cos(t));`
And the plot the true and approximate solutions:
`>> plot(T, Y, T, y_true(T),'--o')`
Or their difference:
`>> plot(T, Y-y_true(T),'-o')`

4.2. **Solving Coupled IVPs** The approach for solving single equations easily extends to systems, such as

$$y_1'(t) = y_2(t) \sin(t), \qquad y_1(0) = 1$$
$$y_2'(t) = -10 y_1(t), \qquad y_2(0) = 0.$$

The trick is to define the function, f, so that it returns a vector:
`>> f = @(t,y)([y(2).*sin(t);-10*y(1)])`
Then solve on (for example) $[0, 3]$:
`>> [t,y] = ode45(f,[0,3],[1,0]);` Notice that we had to provide the initial value as a vector too.
To plot the solutions:
`>> plot(t,y(:,1), t,y(:,2),'--')`

5 Boundary value problems

The general form a second-order, two point, linear BVP is

$$-u''(x) + b(x)u(x) = f(x) \quad 0 < x < 1$$
$$u(0) = \alpha, \quad u(1) = \beta.$$

There are built-in functions for solving these, but we'll look at how to solve them using our own *finite difference method*.

5.1 The algorithm

- Choose N, the number of *mesh intervals*
- Set up a set of $N + 1$ equally spaced points:

$$0 = x_0 < x_1 < x_2 < x_3 \cdots < x_{n-1} < x_n = 1.$$

- Construct A, a $(N + 1) \times (N + 1)$ matrix of zeros, except for
  - $A_{1,1} = 1$
  - For $i = 2, 3, \ldots, N$

$$A_{i,j} = \begin{cases} -1/h^2 & j = i - 1 \\ 2/h^2 + b_k & j = i \\ -1/h^2 & j = i + 1 \\ 0 & \text{otherwise.} \end{cases}$$

  - $A_{N,N+1} = 1$

  In MATLAB this could be implemented as

```
A(1,1) = 1;
for i=2:N
    A(i,i-1) = -1/h^2;
    A(i,i)   = 2/h^2 + r(x(i));
    A(i,i+1) = -1/h^2;
end
A(N+1,N+1)=1;
```

  (We would **not** do this in practice: it is very slow).

- Solve the linear system: `u = A \ B` where `B(i)=f(x(i))`.

Download the script FiniteDifference.m from `http://www.maths.nuigalway.ie/~niall/NASPDEs2016/` and try it out.

Consider the problem:

$$-u''(x) + u(x) = 1 + x \text{ on } (0, 1), \qquad u(0) = u(1) = 0.$$

The solution to this is

$$u(x) = 1 + x - \big(e^{-x}(e^2 - 2e) + e^x(2e - 1)\big)/(e^2 - 1).$$

Use this to test the code. In particular, does the error tend to zero as $N \to \infty$? If so, how rapidly? (These two questions can also be rephrased as "Does the method converge? If so, how quickly?)

5.2 The Profiler

This is not a good way to construct a linear system. Whenever you write a MATLAB program, particularly for solving differential equations, you should use the **profiler** to find any bottle-necks in the code.
If most of the time is **not** spent solving the linear system, then there is a problem.
Another simple method for code-timing are the `tic` and `toc` functions.

5.3 Some Optimisations

To improve, and speed up this code, initialise the matrix `A` and vector `b`:
`A = zeros(N+1, N+1);  b = zeros(N+1,1)`
However, the real improvement is to avoid using loops to initialise matrices or vectors.
For vectors, this is easy:
`b = [alpha; r(x(2:N)); beta];`
For Matrices, we need *sparse matrices*. To initialise:
`A = sparse(N+1, N+1);`
However, the best way to use it is as:
`S = sparse(i,j,s)`
which sets `S(i(k),j(k)) = s(k)`. This can be used as follows:

```
A = sparse(2:N, 1:N-1, -1/h^2) + ...
    sparse(2:N, 2:N, 2/h^2+r(x(2:N))) + ...
     sparse(2:N, 3:N+1, -1/h^2);
```

6 References

References

[Driscoll, 2009] Driscoll, T. A. (2009). *Learning MATLAB*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

[Higham and Higham, 2005] Higham, D. J. and Higham, N. J. (2005). *MATLAB Guide*. Second edition.

[Moler, 2004] Moler, C. B. (2004). *Numerical computing with MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA.