

# MAT 275 Laboratory 2

## MATLAB solvers for First-Order IVP

In this laboratory session we will learn how to

1. Create and manipulate matrices and vectors.
2. Write simple programs in MATLAB

**NOTE:** Use the provided Live Script template for your lab write-up.

### Matrices and Linear Algebra

★ Matrices can be constructed in MATLAB in different ways. For example the  $3 \times 3$  matrix

$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$  can be entered as

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

or

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

or defined as the concatenation of 3 rows

```
>> row1=[8,1,6]; row2=[3,5,7]; row3=[4,9,2]; A=[row1;row2;row3]
A =
     8     1     6
     3     5     7
     4     9     2
```

or 3 columns

```
>> col1=[8;3;4]; col2=[1;5;9]; col3=[6;7;2]; A=[col1,col2,col3]
A =
     8     1     6
     3     5     7
     4     9     2
```

Note the use of `,` and `;`. Concatenated rows/columns must have the same length. Larger matrices can be created from smaller ones in the same way:

```
>> C=[A,A]    % Same as C=[A A]
C =
     8     1     6     8     1     6
     3     5     7     3     5     7
     4     9     2     4     9     2
```

The matrix  $C$  has dimension  $3 \times 6$  (“3 by 6”). On the other hand smaller matrices (submatrices) can be extracted from any given matrix:

```
>> A(2,3)    % entry of A in the 2nd row, 3rd column
ans =
     7
>> A(1,:)    % 1st row of A
ans =
     8     1     6
>> A(:,3)    % 3rd column of A
ans =
     6
     7
     2
>> A([1,3],[2,3]) % keep rows 1 & 3 and columns 2 & 3
ans =
     1     6
     9     2

>> A(end,end) % entry in the last column and the last row of A
ans =
     2
```

We can also change elements in a matrix  $A$ :

```
>> A(1,2)=9 %change the entry in the 1st row & 2nd column of the matrix A to 9.
A =
     8     9     6
     3     5     7
     4     9     2
```

The entries of a row or column vector can be referenced by a single index. For instance:

```
>> v=[3;-1;2;5]
v =
     3
    -1
     2
     5
>> v(3) % third entry of the vector v
ans =
     2
```

★ Some matrices are already predefined in MATLAB:

```
>> I=eye(3) % the Identity matrix
I =
     1     0     0
     0     1     0
     0     0     1
```

```
>> magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2
```

(what is magical about this matrix?)

★ Matrices can be manipulated very easily in MATLAB (unlike MAPLE). Here are sample commands to exercise with:

```
>> A=magic(3);
>> B=A'      % transpose of A, i.e, rows of B are columns of A
B =
     8     3     4
     1     5     9
     6     7     2
>> A+B      % sum of A and B
ans =
    16     4    10
     4    10    16
    10    16     4
>> A*B      % standard linear algebra matrix multiplication
ans =
   101    71    53
    71    83    71
    53    71   101
>> A.*B     % component-wise multiplication
ans =
    64     3    24
     3    25    63
    24    63     4
```

★ The backslash `\` or “left matrix divide” MATLAB command (type `help mldivide` to learn more about this command) is especially relevant when studying the solution of linear systems of differential equations:  $x=A \backslash b$  determines the solution  $x = A^{-1}b$  of the linear system  $Ax = b$ . Here is an example:

```
>> A=magic(3);
>> z=[1,2,3]' % same as z=[1;2;3]
z =
     1
     2
     3
>> b=A*z
b =
    28
    34
    28

>> x = A\b %solve Ax = b. Compare w/ the exact solution z, found above.
x =
     1
     2
     3
```

```
>> y = inv(A)*b % solve the system using the inverse;
                        % less efficient and accurate than by backslash

ans =
    1.0000
    2.0000
    3.0000
```

Now let's check for accuracy by evaluating the difference  $\mathbf{z} - \mathbf{x}$  and  $\mathbf{z} - \mathbf{y}$ . In exact arithmetic they should both be zero since  $x$ ,  $y$  and  $z$  all represent the solution to the system.

```
>> z - x % error for backslash command
ans =
     0
     0
     0
>> z - y % error for inverse
ans =
 1.0e-015 *
 -0.4441
         0
 -0.8882
```

Note the multiplicative factor  $10^{-15}$  in the last computation. MATLAB performs all operations using standard IEEE double precision.

**Important!:** Because of the finite precision of computer arithmetic and roundoff error, vectors or matrices that are zero (theoretically) may appear in MATLAB in exponential form such as  $1.0\text{e-}15 \mathbf{M}$  where  $\mathbf{M}$  is a vector or matrix with entries between  $-1$  and  $1$ . This means that each component of the answer is less than  $10^{-15}$  in absolute value, so the vector or matrix can be treated as zero (numerically) in comparison to vectors or matrices that are on the order of  $1$  in size.

## EXERCISE 1

Enter the following matrices and vectors in MATLAB

$$A = \begin{bmatrix} -2 & 8 & -9 \\ -5 & -8 & -8 \\ 4 & -4 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} -5 & 3 & 9 \\ -10 & 1 & 0 \\ 20 & 2 & -2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ -2 \\ 14 \end{bmatrix}, \quad \mathbf{c} = [2 \quad 0 \quad 1], \quad \mathbf{d} = \begin{bmatrix} 1 \\ 4 \\ 1 \end{bmatrix}.$$

- Perform the following operations:  $AB$ ,  $BA$ ,  $\mathbf{c}A$  and  $B\mathbf{d}$  (use standard linear algebra multiplication).
- Construct a  $6 \times 3$  matrix  $C = \begin{bmatrix} A \\ B \end{bmatrix}$  and a  $3 \times 4$  matrix  $D = [B \quad \mathbf{d}]$ .
- Use the “backslash” command to solve the system  $A\mathbf{x} = \mathbf{b}$ .
- Replace (1,1)-entry of the matrix  $A$  by 0.
- Extract the 2nd row of the matrix  $A$  and store it as a vector  $\mathbf{a}$ .
- A row or a column of a matrix can be deleted by assigning the empty vector  $[]$  to the row or the column; (for instance  $A(2,:) = []$  deletes the second row of the matrix  $A$ ).  
Delete the first column of the matrix  $B$ .

# MATLAB Programming

It is often advantageous to be able to execute a segment of a code a number of times. A segment of a code that is executed repeatedly is called a *loop*.

To understand how loops work, it is important to recognize the difference between an algebraic equality and a MATLAB assignment. Consider the following commands:

```
>> counter = 2
counter =
     2
>> counter = counter + 1
counter =
     3
```

The last statement does **not** say that `counter` is one more than itself. When MATLAB encounters the second statement, it looks up the present value of `counter` (2), evaluates the expression `counter + 1` (3), and stores the result of the computation in the variable on the left, here `counter`. The effect of the statement is to increment the variable `counter` by 1, from 2 to 3.

Now, consider the commands:

```
>> v=[1,2,3]
v =
     1     2     3
>> v=[v,4]
v =
     1     2     3     4
```

When MATLAB encounters the second statement, it looks up the present value of `v`, adds the number 4 as entry of the vector, and stores the result in the variable on the left, here `v`. The effect of the statement is to augment the vector `v` with the entry 4.

There are two types of loops in MATLAB: **for** loops and **while** loops

## for loops

When we know exactly how many times to execute the loop, the **for** loop is often a good implementation choice. One form of the command is as follows:

```
for k=kmin:kmax
    <list of commands>
end
```

The loop index or loop variable is `k`, and `k` takes on integer values from the loop's initial value, `kmin`, through its terminal value, `kmax`. For each value of `k`, MATLAB executes the body of the loop, which is the list of commands.

Here are a few examples:

- Determine the sum of the squares of integers from 1 to 10:  $1^2 + 2^2 + 3^2 + \dots + 10^2$ .

```
S = 0; % initialize running sum
for k = 1:10
    S = S+k^2;
end
S
```

Because we are not printing intermediate values of **S**, we display the final value of **S** after the loop by typing **S** on a line by itself. Try removing the “;” inside the loop to see how **S** is incremented every time we go through the loop.

- Determine the product of the integers from 1 to 10:  $1 \cdot 2 \cdot 3 \cdot \dots \cdot 10$ .

```
p = 1; % initialize running product
for k = 2:10
    p = p*k;
end
p
```

The counter in a **for** loop can be given explicit increments: **for k = kmin:kstep:kmax** to advance the counter **k** by **kstep** each time. For instance the following code evaluates the sum of the squares of the even integers  $2^2 + 4^2 + \dots + 12^2$

```
SE = 0; % initialize running sum
for k = 2:2:12 % the counter starts at 2, ends at 12 and has increments of 2
    SE = SE+k^2;
end
SE
```

Note that **k = kmin:kstep:kmax** is also a vector. A **for** loop essentially iterates through each element of the vector. Also, the counter “**k**” is a variable itself. In fact, the counter doesn’t have to be called “**k**” - it can be called anything.

## The sum and prod functions

Whenever possible all these constructs should be avoided and built in MATLAB functions used instead to improve efficiency. In particular lengthy loops introduce a substantial overhead.

Two examples of MATLAB built in functions are **sum** and **prod**.

If **x** is a vector, then **sum(x)** sums all the elements of **x**. Therefore the **sum** function takes a vector as input and sums its elements. Hence, we can create a vector containing all the terms of the sum we wish to compute and pass that vector as input to the sum function.

The value of **S** in the example above can be evaluated using the built-in function **sum**. We first create the vector 1 2 3 4 5 6 7 8 9 10 using the colon operator, **1:10**, we then square the entries of this vector using **(1:10).^2** (note the dot before the caret, which is necessary to perform componentwise operations.) This yields the vector 1 4 9 16 25 36 49 64 81 100. Then we compute the **sum** of the entries of this vector. All this can be accomplished with a single MATLAB statement:

```
>> S = sum((1:10).^2)
```

Similarly, we can compute the value of **SE** with a single MATLAB command:

```
>> SE = sum((2:2:12).^2)
```

Type **help sum** to see how the built in **sum** function works.

The built in **prod** function takes a vector as input and multiplies all its elements. So, for instance, the product of the integers from 1 to 10:  $1 \cdot 2 \cdot 3 \cdot \dots \cdot 10$  can be evaluated as

```
>> p = prod(1:10)
```

Type **help prod** to see how the built in **prod** function works.

## EXERCISE 2

Recall that a geometric sum is a sum of the form  $a + ar + ar^2 + ar^3 + \dots$

- (a) Write a *function* file called `geomsum1` which accepts the values of  $r$ ,  $a$  and  $n$  (in that order) as input arguments and uses a `for` loop to return the sum of the first  $n$  terms of the geometric series. Test your function for  $r = -7/9$ ,  $a = 3$ ,  $n = 12$ .
- (b) Write a *function* file called `geomsum2` which accepts the values of  $r$ ,  $a$  and  $n$  (in that order) as input arguments and uses the built in command `sum` to find the sum of the first  $n$  terms of the geometric series. Test your function for  $r = -7/9$ ,  $a = 3$ ,  $n = 12$ .

**Hint:** Start by defining the vector  $\mathbf{e} = [0:n-1]$  and then evaluate the vector  $\mathbf{R} = \mathbf{r}.\wedge\mathbf{e}$ . It should be easy to figure out how to find the sum from there.

## EXERCISE 3

The counter in a `for` or `while` loop can have an explicit increment: `for i=m:k:n`. This advances the counter  $i$  by increment  $k$  each time. In this problem we will evaluate the product of the first 9 odd numbers  $1 \cdot 3 \cdot 5 \cdot \dots \cdot 17$  in two ways:

- (a) Write a *script* file that evaluates the product of the first 9 odd numbers using a `for` loop.
- (b) Evaluate the product of the first 9 odd numbers using a single MATLAB command. Use the MATLAB command `prod`.

## while loop

The `while` loop repeats a sequence of commands as long as some condition is met. The basic structure of a `while` loop is the following:

```
while <condition>
    <list of commands>
end
```

Here are some examples:

- Determine the sum of the inverses of squares of integers from 1 until the inverse of the integer square becomes less than  $10^{-10}$ :  $\frac{1}{1^2} + \frac{1}{2^2} + \dots + \frac{1}{k^2}$  while  $\frac{1}{k^2} \geq 10^{-10}$ .

```
S = 0; % initialize running sum
k = 1; % initialize current integer
incr = 1; % initialize test value
while incr >= 1e-10
    S = S + incr;
    k = k + 1;
    incr = 1/k^2;
end
```

What is the value of  $S$  returned by this script? Compare to  $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$ .

- Create a row vector  $\mathbf{y}$  that contains all the factorials below 2000:  $\mathbf{y} = [1!, 2!, 3!, \dots, k!]$  while  $k! < 2000$ .

```
y = []; % initialize the vector y to the empty vector
k = 1; % initialize the counter
value = 1; % initialize the test value to be added to the vector y
```

```

while value < 2000
    y(k) = value; % assign k-th component of the vector y
    k = k+1;      % update the counter
    value = factorial(k); % evaluate the next test value
end
y

```

#### EXERCISE 4

Write a *script* file that creates a row vector  $v$  containing all the powers of 3 that are (strictly) less than  $10^5$ . The output vector should have the form:  $v = [3, 9, 27, 81 \dots]$ . Use the **while** loop.

#### if statement

The basic structure of an if statement is the following:

```

if condition
    <list of commands>
elseif condition
    <list of commands>
    :
else
    <list of commands>
end

```

Here is an example:

- Evaluate

$$y = \begin{cases} x^3 + 2, & x \leq 1 \\ \frac{1}{x-2}, & x > 1 \end{cases}$$

for a given (but unknown) scalar  $x$  and, if  $x = 2$ , display “y is undefined at  $x = 2$ ”.

```

function y=f(x)
if x==2
    disp('y is undefined at x = 2')
elseif x <= 1
    y=x^3+2;
else
    y=1/(x-2);
end
end

```

We can test the file by evaluating it at different values of  $x$ . Below we evaluate the function at  $x = -1$ ,  $x = 2$  and  $x = 4$ .

```

>> f(-1)
ans =
    1
>> f(2)
y is undefined at x = 2
>> f(4)
ans =
    0.5000

```



### EXERCISE 5

Write a *function* file that creates the following piecewise function:

$$f(x) = \begin{cases} 2 - x, & x \leq 5 \\ e^{x-13}, & 5 < x \leq 10 \\ \frac{x}{x-13}, & 10 < x \neq 13 \end{cases}$$

Assume  $x$  is a scalar.

The function file should contain an **if** statement to distinguish between the different cases. Also, if the input is  $x = 13$ , the function should display “**the function is undefined at x = 13**”. Test your function by evaluating  $f(4)$ ,  $f(5)$ ,  $f(5.5)$ ,  $f(10)$ ,  $f(13)$  and  $f(14)$