

Matlab Basics 2

© 2009, Yonatan Katznelson

Matrices 2: *Reduced echelon form; L-U factorization; eigenvalues and eigenvectors.*

MATLAB is designed to work with matrices, and so it is not surprising that some of the more sophisticated operations that we do with matrices are programmed in MATLAB as basic functions.

- (i) *Gauss-Jordan elimination:* The MATLAB command `rref(A)` produces the *reduced row echelon form* of the matrix **A**. E.g., to find the reduced echelon form of

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ -1 & 3 & 0 & 5 \\ 6 & 0 & -1 & 2 \end{bmatrix},$$

you enter the matrix and then apply `rref` to the matrix.

```
>> A=[1 2 3 4; -1 3 0 5; 6 0 -1 2];
>> rref(A)
ans =
    1.0000         0         0    0.3390
         0    1.0000         0    1.7797
         0         0    1.0000    0.0339
```

Note: MATLAB automatically rounds to 4 decimal places which produces round-off error in many cases, even when the numbers in question are rational numbers. The MATLAB command `rats()` converts the rounded decimal entries of its argument into ratios of (relatively) small integers. For example, applying `rats()` to the reduced row echelon form above yields

```
>> rats(rref(A))
ans =
     1         0         0    20/59
     0         1         0   105/59
     0         0         1     2/59
```

The rational numbers produced by `rats` are usually approximations as well, albeit very good ones. If the numbers that they are approximating are rational numbers with small denominators, then `rats()` generally produces the precise values, as in the example above.

- (ii) *L-U factorization*: In a variety of contexts, it is often convenient to express a given matrix A as a product of 2 or 3 matrices with special properties. One of the most commonly used factorizations is the *L-U* factorization described in section 2.5 of our textbook.[†]

To produce the (permuted) lower triangular matrix L and the upper triangular matrix U satisfying $A=L \cdot U$, you use the command `[L U]=lu(A)`. As indicated by the parenthetical word ‘*permuted*’ above, the matrix L that MATLAB produces is often a *permuted lower triangular* matrix — the rows of L need to be permuted (rearranged) to give a proper lower triangular matrix.[‡] This is the case, for example, when the `lu()` command is applied to the matrix A from the previous examples.

```
>> A=[1 2 3 4; -1 3 0 5; 6 0 -1 2];
>> [L U]=lu(A)
L =
    0.1667    0.6667    1.0000
   -0.1667    1.0000         0
    1.0000         0         0
U =
    6.0000         0   -1.0000    2.0000
         0    3.0000   -0.1667    5.3333
         0         0    3.2778    0.1111
```

Notice that if you swap the first and third rows of L above, the result is a proper lower triangular matrix.

The format `[X Y] = func()` is used in MATLAB when the function `func()` returns two distinct objects and you want to store both.[§] If you invoke `func()` by itself, then MATLAB returns some combination of the objects and/or omits some of the available output. E.g, if you invoke the command `lu(A)` without specifying the names of the upper and lower triangular factors, then MATLAB produces a single matrix with the entries of both the upper and the lower factors — the entries of U on and above the main diagonal, and the (non-diagonal) entries of L below the main diagonal — but without the information of how the rows of L should be permuted.

```
>> lu(A)
ans =
    6.0000         0   -1.0000    2.0000
   -0.1667    3.0000   -0.1667    5.3333
    0.1667    0.6667    3.2778    0.1111
```

[†]MATLAB has functions that produce other commonly used factorizations as well.

[‡]See the comments in the middle of page 146.

[§]If `func()` returns n objects, then you should enter something like `[X1 ... Xn] = func()`.

- (iii) *Eigenvalues and eigenvectors:* To produce the *eigenvalues* and corresponding *eigenvectors* of a (square) matrix A ,[¶] we use the command `[E D] = eig(A)`. The matrix D is a diagonal matrix with the eigenvalues of A as its diagonal entries, and the columns of the matrix E are eigenvectors of A , where the j^{th} column of E is an eigenvector corresponding to the eigenvalue that appears in the j^{th} position of the diagonal of D .

Note that even if the entries of the matrix A are real, the eigenvalues of A may be complex, and corresponding eigenvectors may have complex entries.

```
>> A = [1 2 3; 1 1 1; 3 0 2]
A =
     1     2     3
     1     1     1
     3     0     2
>> [E D] = eig(A)
E =
   -0.6667   -0.7688   -0.2573
   -0.3333    0.0502   -0.7886
   -0.6667    0.6375    0.5585
D =
   5.0000         0         0
         0   -1.6180         0
         0         0    0.6180
>> B = [0.4 0.3; -0.3 0.4]
B =
    0.4000    0.3000
   -0.3000    0.4000
>> [F G] = eig(B)
F =
    0.7071         0.7071
0 + 0.7071i    0 - 0.7071i
G =
0.4000 + 0.3000i         0
         0    0.4000 - 0.3000i
```

- (iv) *Basis for the null space:* Given a matrix A , the commands `null(A)` and `null(A, 'r')` return a set of columns that form a basis for the null space of A . The command `null(A)` returns an **orthonormal** basis for *null* A ,^{||} and adding the optional argument `'r'` causes MATLAB to return a *'rational'* basis for *null* A — this is the basis that you

[¶]See chapter 5 in Lay's book.

^{||}An *orthonormal* set is a set of vectors that (a) all have length 1, and (b) are all perpendicular to each other ('orthogonal' means 'right-angle'). See Chapter 6.

would construct by finding the parametric description of the solution set of $A\bar{x} = \bar{0}$, using the free columns of `rref(A)`.

```
>> A=[1 1 2 3 5; 2 0 1 0 2; 1 2 1 3 1];
>> null(A)
ans =
    0.2105    0.2065
   -0.7938    0.1071
   -0.0481   -0.9333
    0.5372    0.0841
   -0.1864    0.2601
>> null(A,'r')
ans =
    0.6000    0.6000
   -1.2000    0.8000
   -1.2000   -3.2000
    1.0000         0
         0    1.0000
```

Simple programs 1.

While we have been using MATLAB primarily as a fancy (graphing) calculator so far, the software goes well beyond that, offering among other things, the possibility of writing programs ('scripts') and functions to handle the repetitive tasks associated with computation and problem solving.

M-files and scripts. Any sequence of commands that you enter on the command line in MATLAB can be entered in a file, called an 'm-file', which you save with an (appropriate) name and the suffix '.m'. After the file is saved, typing its name (without the '.m') on the command line tells MATLAB to perform the sequence of commands listed in the file. Such a sequence of commands is often called a 'script'.

To create an m-file, you type `edit` on the command line, which opens an editing window where you can enter the commands. When you are done, save the file. If you want to edit an existing m-file (called 'file-name'), you type `edit file-name`.

Functions. Often, the sequence of operations that you want to do are being done to a variable of one kind or another. In this case, you want to be able to tell MATLAB something like 'do these operations to the matrix **A**'. One way to handle this, is to write a script in which the commands make reference to a variable with a given name. Consider, for example, the sequence of commands below, saved in an m-file called 'lastcol.m'.

```
x=size(A);
A(1:x(1),x(2))
```

If you enter a matrix **A** and then enter the command `lastcol`, MATLAB returns the last column of the matrix **A**. On the other hand, if you did not save a matrix with the variable name **A** before you enter the command `lastcol`, then MATLAB returns an error message.

A somewhat cleaner approach is to create a *function*. A function is special kind of script that returns one or more variables of output, and typically (though not always) has one or more variables of input. An m-file that defines a function must begin with a line that tells MATLAB that this m-file is a function. The syntax of this first line is

```
function [out1...outk] = name(in1,...,inm)
```

where **out₁** through **out_k** (separated by spaces in square brackets) are the output variables, and **in₁** through **in_m** (separated by commas in parentheses) are the input variables on which the function acts. The function should be saved as **name.m**. For example, the function below, saved as ‘lastcolfunc.m’, does the same thing as the script above, but has the advantage that it can be invoked more easily, for example as a command in another script or function, and applied to any matrix. The function assigns the input to a temporary variable called **A**, which it uses to produce the output.

```
function lastcol = lastcolfunc(A)

x=size(A);

lastcol=A(1:x(1),x(2));
```

Entering the command `lastcolfunc(X)`, where **X** is any matrix, produces the the last column of **X**. The command `c = lastcolfunc(X)`, stores the output in the vector **c**.

Comments: When there is only one output variable, the square brackets may be omitted in the first line of the function. Note also that each of the declared output variables must be assigned a value in the body of the function. Finally, other variables declared in the body of the function, e.g., the **size** variable **x**, above, are *not* visible outside the function. E.g., if after entering `c=lastcolfunc(X)`, you enter **x**, MATLAB will either return an error message (‘variable x undefined’) or, if the variable **x** had been previously defined before invoking the function, MATLAB will display that value. This applies to the named output variables too. In other words, if you want to save the function’s output to specific variables, then you need to enter a command like `[y1 ... yk] = name(x1,...,xm)`.

More to the point: if a function has more than one output and you *don’t* save each one to a designated variable — you just type **function-name** on the command line — then MATLAB will typically just display the first listed output variable. I.e., MATLAB will display **out₁**, if the first line of the function is `function [out1...outk] = name(in1,...,inm)`.**

Loops. Frequently, a certain operation needs to be repeated many times. For example, if you want to create a vector whose entries are the powers of 1.1 from 0 to 3, you can simply type `vec=[1 1.1 1.21 1.331]`. On the other hand, if you want a vector whose entries are the powers of 1.1 from 0 to 20, this approach would be time-consuming to say the least. Using a *loop*, you can have MATLAB do the work for you. There are several commands sequences that produce loops, and the simplest one is the **for loop**. The syntax for a **for-loop** is

**You can define more sophisticated functions that tell MATLAB what to display in case the function is called by itself, as for example in the case of the function `lu`.

```

for j=1:n
    command 1
    command 2
    :
    command k
end

```

As the counter j runs through the designated values (1 through n in the example above), MATLAB loops through the commands that appear between the line ‘`for j=1:n`’ and the line ‘`end`’. Usually, some or all of the commands depend on the value of the counter. Thus, for example, the simple loop

```

for j=0:20
    x(j+1)=1.1^j;
end

```

produces the vector of powers of 1.1 described above. Note that vectors in MATLAB have indices that must be positive integers, so the 0th power of 1.1 is stored in `x(1)`, the first power of 1.1 is stored in `x(2)`, etc. For loops may be entered directly on the command line, or used in the definition of scripts and functions.

The function in the example below takes as inputs two vectors `c` and `x` and returns a vector `y` defined by the property

$$y(j) = c(m)x(j)^{(m-1)} + c(m)x(j)^{(m-2)} + \cdots + c(2)x(j) + c(1)$$

for each $1 \leq j \leq n$, where n is the length of the vector `x`. Your job is to understand how it works (and see if you can guess what it’s good for).

Comment: In an m-file, anything in a line that follows a percent sign (%) is ignored by MATLAB. This allows us to insert *documentation* that explains what’s going on.

```

function y = poly(x,c)

n=length(c); % determine the number of coefficients in the vector c.

m=length(x); % determine the size of the vector x.

y=zeros(1,m); % initiate the vector y as a row of 0's.

for j=1:n

    y=y+c(j)*x.^(n-j);

end

```

Plotting 2: *Three-dimensional plots.*

MATLAB has a range of plotting functions for three-dimensional plots: `plot3()`, `surf()`, `mesh()` and their variants. To plot lines and curves in three dimensions we use `plot3` and to plot surfaces we use `mesh` or `surf`. The command `mesh` produces a *mesh*, or grid, while `surf` fills in the grid to give the appearance of a continuous surface. All of these commands need to be invoked with matrix or vector arguments, as described below, and all of them have a variety of optional arguments. I will only describe the most basic versions of the commands — please see the MATLAB help files or documentation files for more information.

- (i) `plot3()`: This command is used to plot lines and curves in three dimensions. If \mathbf{x} , \mathbf{y} and \mathbf{z} are vectors with the same number of entries, then `plot3(x,y,z)` plots the line (or curve) that passes through the points $(\mathbf{x}(j), \mathbf{y}(j), \mathbf{z}(j))$ for every j between 1 and n , where n is the number of entries in the three vectors.

The sequence of commands below are used to plot the graph of the *helix* in Figure 1.

```

>> t=0:0.05:6*pi; x=sin(t); y=cos(t); z=t;
>> plot3(x,y,z)

```

The plotting options described for two-dimensional plots carry over to the three-dimensional setting. E.g., you can choose different markers for the points in the plot, different line styles, whether or not to include a grid, etc.

- (ii) `surf(x,y,Z)`: This command is used to plot *surfaces* in three dimensions. The arguments \mathbf{x} and \mathbf{y} are vectors of length n and m respectively, and \mathbf{Z} is an $m \times n$ matrix. The coordinates of the surface patches (as they are called) are given by $(\mathbf{x}(j), \mathbf{y}(i), \mathbf{Z}(i,j))$. The surface is colored according to the height, $\mathbf{Z}(i,j)$.

The sequence of commands below is used to plot the graph of $z = x^2 - y^2$ that appears in Figure 2. Notice how I used a *nested* pair of for-loops to fill in the matrix \mathbf{Z} .

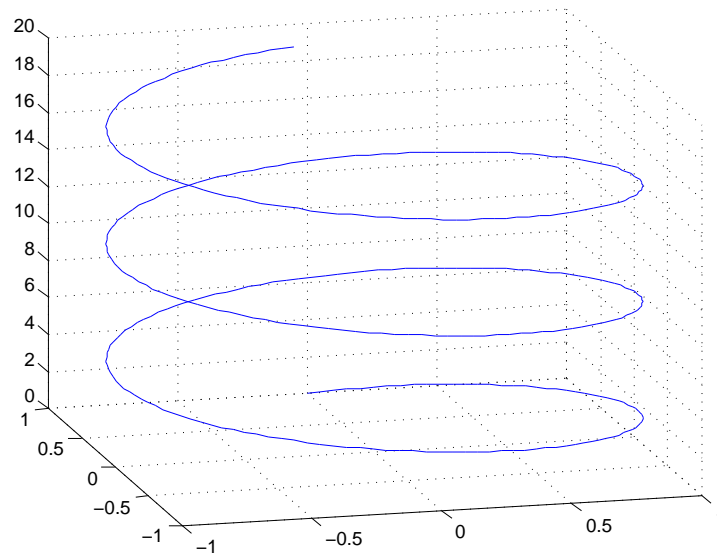


Figure 1: The graph of the parametric curve $F(t) = (\sin(t), \cos(t), t)$.

```
>> x=-2:0.1:2; y=-2:0.1:2;
>> for j=1:41
    for i=1:41
        Z(i,j)=x(j)^2-y(i)^2;
    end
end;
>> surf(x,y,Z)
```

- (iii) **mesh(x,y,Z)**: This command works just like **surf()**, except that MATLAB produces a *mesh* instead of a continuous surface. I.e., the patches between the points that are plotted are *not* filled in. Replacing the command **surf(x,y,Z)** by **mesh(x,y,Z)** in the sequence of commands above produces the graph in Figure 3.

Other options: Both **surf** and **mesh** have other variations. In particular, both commands can take arguments (**X,Y,Z**) that are all matrices (rather than two vectors and a matrix). This form can be used to plot parametric surfaces. See the MATLAB help index for more information.

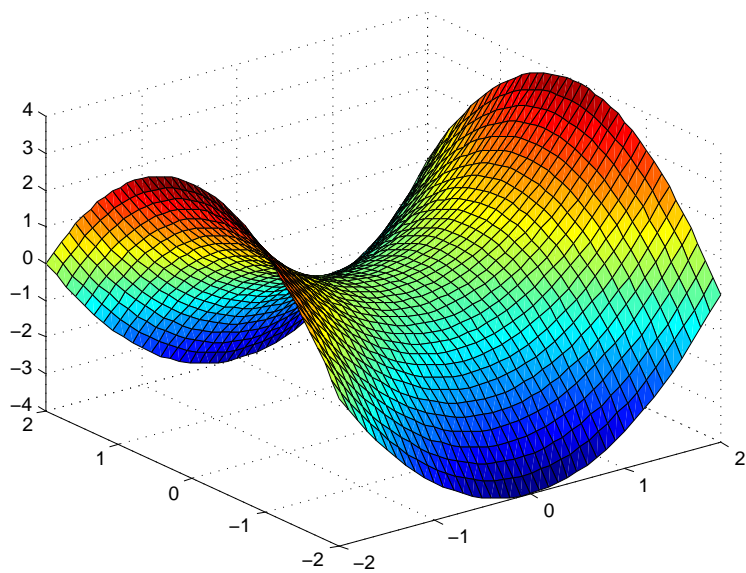


Figure 2: The graph of $z = x^2 - y^2$.

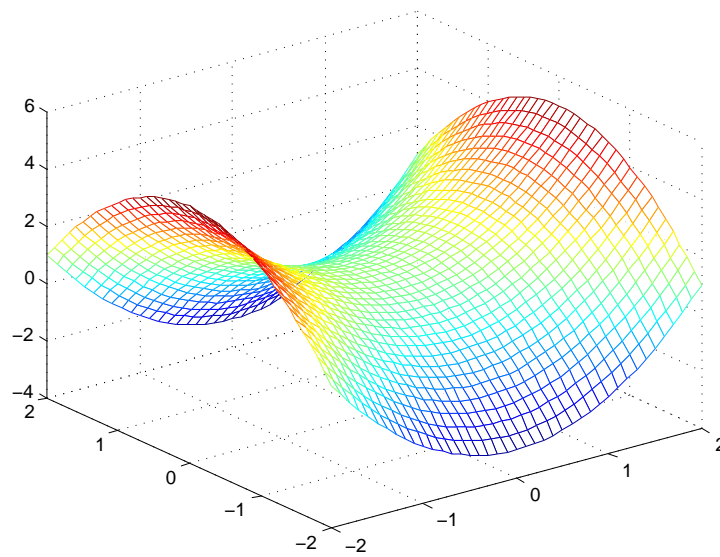


Figure 3: The graph of $z = x^2 - y^2$, in *mesh* form.