

Tutorial 2: matrix algebra, LU factorization, the inverse of a matrix, some computations, and the Leontief model.

In this Tutorial you will learn about how to do matrix algebra (multiplication, addition) in Matlab. You will also learn how to do an LU factorization, how to compute the inverse of a matrix and you will learn about some computational issues when solving linear equations. You will apply what you know about matrices and linear equations to the Leontief model.

Matrix Algebra

You can multiply and add/subtract two matrices in Matlab, provided that their sizes are compatible. Define some matrices:

```
A = [1 2 3
     4 5 6];
B = [7 10
     8 11
     9 12];
C = [2 4 6
     8 10 12];
D = [1
     2
     3]; % sometimes an nx1 matrix is called a vector...
```

You can only add or subtract two matrices when they have the same size (the same number of rows and columns). You can add or subtract **A** and **C** (both are 2x3 matrices):

A+C

```
ans = 2x3
     3     6     9
    12    15    18
```

A-C

```
ans = 2x3
    -1    -2    -3
    -4    -5    -6
```

You can also multiply any matrix by a scalar:

2*A

```
ans = 2x3
     2     4     6
     8    10    12
```

Note that the size of the matrix does not change when you multiply by a scalar (do you know why?). Thus, you can also add **2A** to **C**:

2*A+C

```
ans = 2x3
     4     8    12
    16    20    24
```

You cannot add **A** and **B**, because the number of row and columns of **A** (a 2x3 matrix) is different from the number of row and columns of **B** (a 3x2 matrix). If you try adding **A** and **B** in Matlab, it will give you an error:

```
A+B
```

```
Arrays have incompatible sizes for this operation.
```

```
Related documentation
```

You can multiply **D** by **C** from the left, because the number of columns of **C** is equal to the number of rows in **D**

```
C*D
```

```
ans = 2x1
    28
    64
```

You cannot multiply **C** by **D** from the left:

```
D*C
```

```
Error using *
Incorrect dimensions for matrix multiplication. Check that the number of columns in the first
matrix matches the number of rows in the second matrix. To operate on each element of the matrix
individually, use TIMES (.* ) for elementwise multiplication.
```

```
Related documentation
```

Note that Matlab's error message tells you why: the number of columns in **D** is not equal to the number of rows in **C**. This means that matrices do not commute, i.e., **CD** \neq **DC**. In this example, **DC** does not even exist.

For any two matrices **A** (mxn), **B** (nxm), **AB** is not equal to **BA**. You can see this from the **A** and **B** defined above:

```
A*B
```

```
ans = 2x2
    50    68
   122   167
```

```
B*A
```

```
ans = 3x3
    47    64    81
    52    71    90
    57    78    99
```

Note that **AB** \neq **BA** and that **AB** is of different size than **BA**. In general, if **A** is mxn and **B** is nxm, then **AB** is mxm and **BA** is nxn.

By now you have experienced enough matrix algebra to appreciate that multiplying or adding matrices is more complicated than multiplying or adding scalars. The reason is that the number of columns and rows is important for multiplication and addition. For scalars, the number of columns and rows is one, which makes things easy. In the matrix case, the number of rows and columns adds a layer of difficulty and complexity.

LU Factorization

In class you learned that an $m \times n$ matrix $\mathbf{A} = \mathbf{LU}$, where \mathbf{L} is an $m \times m$ lower triangular matrix and \mathbf{U} is a $m \times n$ echolon form of \mathbf{A} . If \mathbf{A} is square ($n \times n$), then \mathbf{U} is upper triangular.

You can use Matlab to compute the LU factorization using the LU command. Use the help command to find out about the lu command:

```
help lu
```

```
lu      lu factorization.
[L,U] = lu(A) returns an upper triangular matrix in U and a permuted
lower triangular matrix in L, such that A = L*U. The input matrix A can
be full or sparse.
```

```
[L,U,P] = lu(A) returns unit lower triangular matrix L, upper
triangular matrix U, and permutation matrix P such that P*A = L*U.
```

```
[L,U,P] = lu(A, outputForm) returns P in the form specified by
outputForm:
```

```
    'matrix' - (default) P is a matrix such that P*A = L*U.
    'vector' - P is a vector such that A(P,:) = L*U.
```

The following syntaxes **only** apply for a sparse input matrix A:

```
[L,U,P,Q] = lu(A) returns unit lower triangular matrix L, upper
triangular matrix U, and row/column permutation matrices P and Q such
that P*A*Q = L*U. For sparse matrices this is significantly more time
and memory efficient than the three-output syntax.
```

```
[L,U,P,Q,D] = lu(A) also returns a diagonal scaling matrix D such that
P*(D\A)*Q = L*U for sparse A. Typically, the row-scaling leads to a
sparser and more stable factorization. Note that this is the
factorization used by sparse MLDDIVIDE.
```

```
[___] = lu(A,THRESH) specifies the pivoting strategy employed by lu.
THRESH is a scalar or two-element vector with values in [0 1].
Increasing the value of THRESH tends to lead to higher accuracy, but
typically at a greater cost in time and memory. Depending on the number
of output arguments specified, the default value and requirements for
the thresh input are different:
```

- * If 3 or fewer outputs are specified, then THRESH is a scalar. The default value is 1.0.
- * If 4 or more outputs are specified, then THRESH is a two-element vector. The default value is [0.1 0.001].

```
[___] = lu(____, outputForm) specifies the output form of P and Q. Use
this option to return P and Q as vectors instead of matrices. P and Q
satisfy different identities depending on the number of outputs
specified and whether they are matrices or vectors:
```

With 4 outputs:

```
    'matrix' - (default) P is a matrix such that P*A*Q = L*U.
    'vector' - P is a vector such that A(P,Q) = L*U.
```

With 5 outputs:

'matrix' - (default) P is a matrix such that $P(D \setminus A)Q = L*U$.

'vector' - P is a vector such that $D(:,P) \setminus A(:,Q) = L*U$.

See also `qr`, `chol`, `ilu`, `mldivide`, `decomposition`, `inv`, `cond`.

Documentation for `lu`

Other uses of `lu`

Now you are ready to try it.

```
A = [2 -1 1
      1 2 3
      3 0 -1];
[L U] = lu(A);
A
```

```
A = 3x3
     2     -1      1
     1      2      3
     3      0     -1
```

$L*U$

```
ans = 3x3
     2.0000    -1.0000     1.0000
     1.0000     2.0000     3.0000
     3.0000         0    -1.0000
```

You can also check the output of the `lu` function by subtracting LU from A :

$A-L*U$

```
ans = 3x3
10-15 ×
         0         0    0.4441
         0         0         0
         0         0         0
```

Note the numerical error familiar from Tutorial 1: The above matrix is indistinguishable from zero for your computer (and for most practical purposes).

Since L and U are lower triangular and of row echelon form, you can solve $Ly = b$ and $Ux = y$ by substitution. Since $Ax = LUx = Ly = b$ you can use the LU factorization for solving linear systems $Ax = b$. You first compute the LU factorization of A , then solve $Ly = b$ by substitution and then solve $Ux = y$ by substitution.

```
b = [1
      2
      3]; % define a right hand side
y = L\b;
x = U\y;
x
```

```
x = 3x1
```

```
0.9500
0.7500
-0.1500
```

You can check that the solution you got by LU factorization is the same as the solution you get by using the backslash (Matlab's all purpose linear solver):

```
A\b
```

```
ans = 3x1
    0.9500
    0.7500
   -0.1500
```

The inverse of a matrix

You can compute the inverse of a matrix by using the `inv` command. You can learn about the `inv` command using `help`:

```
help inv
```

```
inv    Matrix inverse.
      inv(X) is the inverse of the square matrix X.
      A warning message is printed if X is badly scaled or
      nearly singular.

      See also pinv, cond, condest, lsqminnorm, lsqnonneg, lscov.

      Documentation for inv
      Other uses of inv
```

You are now ready to use it. Compute the inverse of **A**:

```
invA = inv(A)
```

```
invA = 3x3
    0.1000    0.0500    0.2500
   -0.5000    0.2500    0.2500
    0.3000    0.1500   -0.2500
```

You can check that $A^{-1}A = I$ and that $AA^{-1} = I$

```
invA*A
```

```
ans = 3x3
    1.0000         0         0
         0    1.0000   -0.0000
    0.0000         0    1.0000
```

```
A*invA
```

```
ans = 3x3
    1.0000    0.0000   -0.0000
   -0.0000    1.0000    0.0000
         0         0    1.0000
```

You can also use the inverse to solve linear equations since $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$

```
invA*b
```

```
ans = 3x1
    0.9500
    0.7500
   -0.1500
```

You can see that you got the same solution \mathbf{x} as with LU or the backslash.

What is the fastest way to solve a linear equations?

You now know three ways of solving a linear equation:

1. Compute the inverse of \mathbf{A}
2. Use LU factorization and substitution
3. Use the backslash

One wonders: which one is the fastest?

We find out by letting the three methods run a race: which method is the fastest at solving 1000 randomly generated linear equations? To make things interesting, we use large matrices of size 1000 x 1000.

We start with stoping how long it takes to solve a linear system using the inverse of \mathbf{A}

```
clear % delete old variables
times = zeros(100,1); % pre-allocate an array to save computing times
for kk=1:100 % count to 100
    A = randn(1000,1000); % generate a 1000x1000 matrix
    b = randn(1000,1); % generate a 1000x1 right hand side
    tic % start the clock
        x = inv(A)*b; % solve the linear system
    times(kk)=toc; % stop the clock
end
fprintf('Time to solve 100 linear equations (inv): %g',sum(times)) % print
out results
```

```
Time to solve 100 linear equations (inv): 1.53783
```

Now try LU factorization:

```
clear % delete old variables
times = zeros(100,1); % pre-allocate an array to save computing times
for kk=1:100 % count to 100
    A = randn(1000,1000); % generate a 1000x1000 matrix
    b = randn(1000,1); % generate a 1000x1 right hand side
    tic % start the clock
```

```

[L U] = lu(A); % LU factorization
y = L\b; % substitution
x = U\y; % substitution
times(kk)=toc; % stop the clock
end
fprintf('Time to solve 100 linear equations (LU): %g',sum(times)) % print
out results

```

Time to solve 100 linear equations (LU): 1.45853

Now try the backslash:

```

clear % delete old variables
times = zeros(100,1); % pre-allocate an array to save computing times
for kk=1:100 % count to 100
    A = randn(1000,1000); % generate a 1000x1000 matrix
    b = randn(1000,1); % generate a 1000x1 right hand side
    tic % start the clock
    x = A\b; % substitution
    times(kk)=toc; % stop the clock
end
fprintf('Time to solve 100 linear equations (backslash): %g',sum(times)) %
print out results

```

Time to solve 100 linear equations (backslash): 0.986432

Note that LU is faster than the inverse. Not surprisingly, the inverse of a matrix is rarely used for computations when solving linear equations. You also note that the backslash is faster than LU. This is due to the fact that here we are sloppy when implementing the substitution (we use the backslash). Moreover, the backslash is a highly optimized command and many people spend lots of time on making it as fast as it is. At its heart, however, you will find the LU factorization and Gaussian elimination, just coded in a smart way.

Leontief model

Recall from class that the Leontief model describes the production and demand in an economy via a linear system of equations. Specifically, for an economy with n sectors, such as manufacturing, agriculture, services etc., \mathbf{x} is the n -dimensional production vector, listing the output of each sector. The n -dimensional vector \mathbf{d} describes the demand, i.e., how much of the various products are needed. The Leontief model is to state that the demand \mathbf{d} equals the amount produced, \mathbf{x} , minus an intermediate demand which is a *linear* function of the amounts produced:

$$\mathbf{x} - \mathbf{C}\mathbf{x} = \mathbf{d},$$

where \mathbf{C} is the consumption matrix. The goal is to find the production vector \mathbf{x} , given a demand \mathbf{d} and a consumption matrix \mathbf{C} . Using matrix algebra, we can re-write the Leontief model as

$$(\mathbf{I} - \mathbf{C}) \mathbf{x} = \mathbf{d},$$

where I is the $n \times n$ identity matrix. To find x , we thus solve a linear equation with $A = I - C$ and $b = d$.

Consider an economy with demand and consumption matrix

```
clear % delete old variables
d = [40
     60
     80];
C = [0.2 0.2 0.0
     0.1 0.1 0.3
     0.1 0.0 0.2];
```

To find x , you also need the identity matrix I . You can simply type it

```
I = [1 0 0
     0 1 0
     0 0 0];
```

Or you can use Matlab's `eye` command. Use help to find out about `eye`:

```
help eye
```

```
eye Identity matrix.
eye(N) is the N-by-N identity matrix.

eye(M,N) or eye([M,N]) is an M-by-N matrix with 1's on
the diagonal and zeros elsewhere.

eye(SIZE(A)) is the same size as A.

eye with no arguments is the scalar 1.

eye(..., CLASSNAME) is a matrix with ones of class specified by
CLASSNAME on the diagonal and zeros elsewhere.

eye(..., 'like', Y) is an identity matrix with the same data type, sparsity,
and complexity (real or complex) as the numeric variable Y.

Note: The size inputs M and N should be nonnegative integers.
Negative integers are treated as 0.

Example:
    x = eye(2,3,'int8');

See also speye, ones, zeros, rand, randn.

Documentation for eye
Other uses of eye
```

You can use `eye` to create the 3×3 identity matrix:

```
I = eye(3);
```

Now you are ready to find the production vector:

```
x = (I-C)\d
```



```
x = 3x1
    77.9783
    111.9134
    109.7473
```

Using the inverse of $(\mathbf{I} - \mathbf{C})$, you can write the production vector as

$$\mathbf{x} = (\mathbf{I} - \mathbf{C})^{-1} \mathbf{d}.$$

We know that using the inverse for computations is not a good idea, but we can approximate the inverse of the matrix $(\mathbf{I} - \mathbf{C})^{-1}$ by

$$(\mathbf{I} - \mathbf{C})^{-1} \approx \mathbf{I} + \mathbf{C} + \mathbf{C}^2 + \mathbf{C}^3 + \dots + \mathbf{C}^m,$$

With this approximation we can find an approximation to \mathbf{x} :

$$\mathbf{x}^m = (\mathbf{I} + \mathbf{C} + \mathbf{C}^2 + \mathbf{C}^3 + \dots + \mathbf{C}^m) \mathbf{d}.$$

The larger m is, the better the approximation gets, i.e., \mathbf{x}^m is a very good approximation of \mathbf{x} .

You can try this easily in Matlab. For example, with $m = 3$ the approximation is

```
xa = (I+C+C^2+C^3)*d
```

```
xa = 3x1
    75.2400
    109.4200
    108.2800
```

Not bad, but if you want the first three digits correctly, you will have to go up to $m = 7$:

```
xa = (I+C+C^2+C^3+C^4+C^5+C^6+C^7)*d
```

```
xa = 3x1
    77.9189
    111.8585
    109.7151
```

Note that the formula for \mathbf{x}^m defines a recursive formula:

$$\mathbf{x}^1 = \mathbf{d}$$

$$\mathbf{x}^2 = \mathbf{d} + \mathbf{C}\mathbf{d} = \mathbf{d} + \mathbf{C}\mathbf{x}^1$$

$$\mathbf{x}^3 = \mathbf{d} + \mathbf{C}\mathbf{d} + \mathbf{C}^2\mathbf{d} = \mathbf{d} + \mathbf{C}(\mathbf{d} + \mathbf{C}\mathbf{d}) = \mathbf{d} + \mathbf{C}\mathbf{x}^2$$

$$\mathbf{x}^4 = \mathbf{d} + \mathbf{C}\mathbf{d} + \mathbf{C}^2\mathbf{d} + \mathbf{C}^3\mathbf{d} = \mathbf{d} + \mathbf{C}(\mathbf{d} + \mathbf{C}\mathbf{d} + \mathbf{C}^2\mathbf{d}) = \mathbf{d} + \mathbf{C}\mathbf{x}^3$$

Thus, given an \mathbf{x}^n , we can compute

$$\mathbf{x}^{n+1} = \mathbf{d} + \mathbf{C}\mathbf{x}^n$$

In Matlab, you can implement recursive formulas via a for-loop. To get to x^7 , you iterate seven times:

```
x = d; % this is how the iteration starts
for kk=1:7 % count from 1 to 7
    x = d+C*x;
end
x
```

```
x = 3x1
    77.9189
   111.8585
   109.7151
```

Exercises

1. How long did it take your computer to solve the 100 linear systems using (a) the inverse of **A**; (b) the LU factorization; (c) the backslash? Find a classmate (or a different computer) and compare the times. Are they the same?

Answer:

a) It took me 1.53783 seconds for the inverse of A method.

b) It took me 1.45853 seconds for the LU factorization method.

c) It took me 0.986432 seconds for the backlash method.

I have compared my answer to Paulina and she got:

a) 2.25992 seconds for inverse A

b) 1.46586 seconds for LU factorization

c) 1.5346 seconds for backlash.

Her result is a little bit slower than mine and I think it is weird that she has the LU method faster than the backlash method. She ran her code again the the LU is still faster by 0.02 seconds so probably it is a computer problem.

2. Given are the consmption matrix **C** and demand **d**

```
clear
C=[0.1588 0.0064 0.0025 0.0304 0.0014 0.0083 0.1594
    0.0057 0.2645 0.0436 0.0099 0.0083 0.0201 0.3413
    0.0264 0.1506 0.3557 0.0139 0.0142 0.0070 0.0236
    0.3299 0.0565 0.0495 0.3636 0.0204 0.0483 0.0649
    0.0089 0.0081 0.0333 0.0295 0.3412 0.0237 0.0020
    0.1190 0.0901 0.0996 0.1260 0.1722 0.2368 0.3369
```

```
0.0063 0.0126 0.0196 0.0098 0.0064 0.0132 0.0012];
```

```
d =[74000  
56000  
10500  
25000  
17500  
196000  
5000];
```

```
I = eye(7) %identity matrix
```

```
I = 7x7  
1 0 0 0 0 0 0  
0 1 0 0 0 0 0  
0 0 1 0 0 0 0  
0 0 0 1 0 0 0  
0 0 0 0 1 0 0  
0 0 0 0 0 1 0  
0 0 0 0 0 0 1
```

```
x = (I-C)\d %production vector x
```

```
x = 7x1  
105 ×  
0.9958  
0.9770  
0.5123  
1.3157  
0.4949  
3.2955  
0.1384
```

```
xa = (I+C+C^2+C^3+C^4+C^5+C^6+C^7+C^8+C^9+C^10+C^11+C^12+C^13+C^14)*d
```

```
xa = 7x1  
105 ×  
0.9957  
0.9770  
0.5122  
1.3156  
0.4948  
3.2954  
0.1383
```

```
%So m = 14 to get the first 3 digits right
```

What is the production vector \mathbf{x} ? How large an m is required to get the first three digits right when using the approximation $(\mathbf{I} - \mathbf{C})^{-1} \approx \mathbf{I} + \mathbf{C} + \mathbf{C}^2 + \mathbf{C}^3 + \dots + \mathbf{C}^m$?