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
A-C
```

You can also multiply any matrix by a scalar:

```
2*A
```

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 2**A** to **C**:

```
2*A+C
```

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
```

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
```

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

1

```
D*C
```

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** =/= **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
B*A
```

Note that **AB** =/= **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 mxn matrix **A** = **LU**, where L is an m x m lower triangular matrix and **U** is a m x n echolon form of **A**. If **A** is square (n x n), then **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
```

Now you are ready to try it.

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

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

```
A-L*U
```

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

Since **L** and **U** are lower triangular and of row echolon 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
```

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
```

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
```

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

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

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

```
invA*A
A*invA
```

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

```
invA*b
```

You can see that you got the same solution 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 **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 **A**

```matlab
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
```

Now try LU factorization:

```matlab
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
```

Now try the backslash:

```matlab
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
```

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

4

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., **x** is the n-dimensional production vector, listing the output of each sector. The n-dimensional vector **d** describes the demand, i.e., how much of the various products are needed. The Leontief model is to state that the demand **d** equals the amount produced, **x**, minus an intermediate demand which is a *linear* function of the amounts produced:

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

where **C** is the consumption matrix. The goal is to find the production vector **x**, given a demand **d** and a consumption matrix **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 x n identidy matrix. To find **x**, we thus solve a linear equation with **A** = **I**-**C** and **b** = **d**.

Consider an ecomony 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 identiy 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
```

You can use eye to create the 3x3 identify matrix:

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

Now you are ready to find the production vector:

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

Using the inverse of (**I**-**C**), you can write the production vector as

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

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

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

With this approximation we can find an approximation to **x**:

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

The larger m is, the better the approximation gets, i.e., x^m is a very good approximation of **x**.

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

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

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
```

Note that the formula for x^m defines a recursive formula:

$$x^1 = d$$

$$x^2 = d + Cd = d + Cx^1$$

$$x^3 = d + Cd + C^2d = d + C(d + Cd) = d + Cx^2$$

$$x^4 = d + Cd + C^2d + C^3d = d + C(d + Cd + C^2d) = d + Cx^3$$

Thus, given an x^n, we can compute

$$x^{n+1} = d + Cx^n$$

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
```

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?

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];
```

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