Project 0

Exercise 1

Part 1

```
A = [0 3 6 9; 1 4 7 10; 2 5 8 11]
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
```

```
B = [2 -2; 3 -3; 4 -4]
```

```
B = 3×2
     2    -2
     3    -3
     4    -4
```

```
X = [2; 4; 6]
```

```
X = 1×3
     2     4     6
```

```
x = [2 3 4 5]
```

```
x = 1×4
     2     3     4     5
```

```
y = [1; 3; 5; 7]
```

```
y = 4×1
     1
     3
     5
     7
Calculating matrix sizes
A:
```

```
size(A)
```

```
ans = 1×2
     3     4
B:
```

```
size(B)
```

```
ans = 1×2
     3     2
X:
```

```
size(X)
```

```
ans = 1×2
     1     3
x:
```

```
size(x)
```

```
ans = 1×2
     1     4
y:
```

```
size(y)
```

```
ans = 1×2
     4     1
```

Matrices **x** and **y** are not the same size at all! Matrix **x** is 1 row x 4 columns and **y** is 4 rows and 1 column.

```
size(A, 1)
```

```
ans = 3
```

```
size(A, 2)
```

```
ans = 4
```

Second Parameter: The former of these results is the number of rows, the latter is the number of columns. These results are obtained by specifying which result is desired by giving the command a 1 or a 2 as the second parameter.

Part 2

```
A, A(1,3), A(:,3), A(2,:), A(:, 1:3)
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 6
ans = 3×1
     6
     7
     8
ans = 1×4
     1     4     7    10
ans = 3×3
     0     3     6
     1     4     7
     2     5     8
```

Printed the matrix out in different styles, cutting and shortening it for specific indexes and results.

```
A, A([1  2], [2  4])
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 2×2
     3     9
     4    10
```

Prints out values based on the matrix range inside the brackets for row and column

```
F(:,4)=[-1  1  -4  3], F([1  3], [2  3]) = [1 -3; 2 -5]
```

```
F = 4×4
     1     1    -3    -1
     7     3     6     1
     5     2     8    -4
     3     0     6     3
F = 4×4
     1     1    -3    -1
     7     3     6     1
     5     2    -5    -4
     3     0     6     3
```

Creates a new matrix and assigns the 4th column values, additionally, it adds some specific values to a narrow matrix of rows and columns

```
A, F, F([2  3], :) = A([1  3], :)
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
F = 4×4
     1     1    -3    -1
     7     3     6     1
     5     2    -5    -4
     3     0     6     3
F = 4×4
     1     1    -3    -1
     0     3     6     9
     2     5     8    11
     3     0     6     3
```

Copies some values from matrix A to matrix F using the matrix index selection and range operators.

```
F, F(:, [1 2])=F(:, [2 1])
```

```
F = 4×4
     1     1    -3    -1
     0     3     6     9
     2     5     8    11
     3     0     6     3
F = 4×4
     1     1    -3    -1
     3     0     6     9
     5     2     8    11
     0     3     6     3
```

Manipulates matrix F using the full scope operator and the matrix index selection operator to move values around inside the matrix.

```
F, F(:, 2:4)
```

```
F = 4×4
     1     1    -3    -1
     3     0     6     9
     5     2     8    11
     0     3     6     3
ans = 4×3
     1    -3    -1
     0     6     9
     2     8    11
     3     6     3
```

Prints out full range of rows and colums 2 through 4.

(I)

```
y, F
```

```
y = 4×1
     1
     3
     5
     7
F = 4×4
     1     1    -3    -1
     3     0     6     9
     5     2     8    11
     0     3     6     3
```

```
F(:,1) = y
```

```
F = 4×4
     1     1    -3    -1
     3     0     6     9
     5     2     8    11
     7     3     6     3
```

(II)

```
F
```

```
F = 4×4
     1     1    -3    -1
     3     0     6     9
     5     2     8    11
     7     3     6     3
```

```
tmp = F(2,:);
F(2,:) = F(4,:);
F(4,:) = tmp;
```

(III)

```
F
```

```
F = 4×4
     1     1    -3    -1
     7     3     6     3
     5     2     8    11
     3     0     6     9
```

```
F1 = F([1 2], :)
```

```
F1 = 2×4
     1     1    -3    -1
     7     3     6     3
```

Part 3

```
A, B, [A B], [B A]
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
B = 3×2
     2    -2
     3    -3
     4    -4
ans = 3×6
     0     3     6     9     2    -2
     1     4     7    10     3    -3
     2     5     8    11     4    -4
ans = 3×6
     2    -2     0     3     6     9
     3    -3     1     4     7    10
     4    -4     2     5     8    11
```

```
X(1, 4) = 0;
A, X, [A; X]
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
X = 1×4
     2     4     6     0
ans = 4×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
     2     4     6     0
```

```
A, x, [A; x]
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
x = 1×4
     2     3     4     5
ans = 4×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
     2     3     4     5
```

Part 4

```
eye(5), zeros(3,4), zeros(2), ones(3,2), ones(5)
```

```
ans = 5×5
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
ans = 3×4
     0     0     0     0
     0     0     0     0
     0     0     0     0
ans = 2×2
     0     0
```

```
     0     0
ans = 3×2
     1     1
     1     1
     1     1
ans = 5×5
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

- **eye(x)** - outputs a diagonal row of 1's starting from position (1, 1) down to (x, x). The remaining values are 0's.
- **zeros(a, b)** - Creates an a x b matrix of 0's in every poisition, if b is left empty, creates an a x a matrix.
- **ones(a, b) -** Creates an a x b matrix of 1's in every poisition, if b is left empty, creates an a x a matrix.

```
diag([1 2 5 6 7]), diag([1 2 5 6 7],1), diag([1 2 5 6 7],-2)
```

```
ans = 5×5
     1     0     0     0     0
     0     2     0     0     0
     0     0     5     0     0
     0     0     0     6     0
     0     0     0     0     7
ans = 6×6
     0     1     0     0     0     0
     0     0     2     0     0     0
     0     0     0     5     0     0
     0     0     0     0     6     0
     0     0     0     0     0     7
     0     0     0     0     0     0
ans = 7×7
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     1     0     0     0     0     0     0
     0     2     0     0     0     0     0
     0     0     5     0     0     0     0
     0     0     0     6     0     0     0
     0     0     0     0     7     0     0
```

- **diag(v, f) -** Returns a matrix with values in vector v spanning diagonally across the matrix. Starting position is offset by f columns if positive or f rows if negative, no offset if left blank.

```
B, diag(B), diag(diag(B))
```

```
B = 3×2
     2    -2
     3    -3
     4    -4
ans = 2×1
     2
    -3
ans = 2×2
     2     0
     0    -3
```

- **diag(m) -** Returns a vector of the values spanning diagonally across matrix m.

- **diag(diag(v)) -** Returns a diagonal matrix of vector v

```
A, triu(A), tril(A)
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 3×4
     0     3     6     9
     0     4     7    10
     0     0     8    11
ans = 3×4
     0     0     0     0
     1     4     0     0
     2     5     8     0
```

- **triu(X) -** Returns the upper of the diagonal part of matrix X, replacing the lower values with 0
- **tril(X) -** Returns lower of the diagonal of matrix X replacing the uppers with 0

```
F, triu(F), triu(F,1), triu(F,-1)
```

```
F = 4×4
     1     1    -3    -1
     7     3     6     3
     5     2     8    11
     3     0     6     9
ans = 4×4
     1     1    -3    -1
     0     3     6     3
     0     0     8    11
     0     0     0     9
ans = 4×4
     0     1    -3    -1
     0     0     6     3
     0     0     0    11
     0     0     0     0
ans = 4×4
     1     1    -3    -1
     7     3     6     3
     0     2     8    11
     0     0     6     9
```

- **triu(X, f) -** Returns upper of the diagonal shifted f columns (if f +, else shift f rows)

```
F, tril(F), tril(F,-1),tril(F,1)
```

```
F = 4×4
     1     1    -3    -1
     7     3     6     3
     5     2     8    11
     3     0     6     9
ans = 4×4
     1     0     0     0
     7     3     0     0
     5     2     8     0
     3     0     6     9
ans = 4×4
     0     0     0     0
     7     0     0     0
```

```
     5     2     0     0
     3     0     6     0
ans = 4×4
     1     1     0     0
     7     3     6     0
     5     2     8    11
     3     0     6     9
```

- **tril(X, f) -** Returns upper of the diagonal shifted f columns (if f +, else shift f rows)

```
C = diag([1 1 1])
```

```
C = 3×3
     1     0     0
     0     1     0
     0     0     1
```

```
D = diag([1 2 3], 1)
```

```
D = 4×4
     0     1     0     0
     0     0     2     0
     0     0     0     3
     0     0     0     0
```

```
E = ones(2, 3)
```

```
E = 2×3
     1     1     1
     1     1     1
```

Part 5

```
V1=1:7,  V2=2:0.5:6.5,   V3=3:-1:-5
```

```
V1 = 1×7
     1     2     3     4     5     6     7
V2 = 1×10
    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000    5.5000 ···
V3 = 1×9
     3     2     1     0    -1    -2    -3    -4    -5
```

```
V4 = -5:1:1,    V5 = 10:-3:-2,  V6 = 5:-0.5:2,  V7 = 0:0.4:4
```

```
V4 = 1×7
    -5    -4    -3    -2    -1     0     1
V5 = 1×5
    10     7     4     1    -2
V6 = 1×7
    5.0000    4.5000    4.0000    3.5000    3.0000    2.5000    2.0000
V7 = 1×11
         0    0.4000    0.8000    1.2000    1.6000    2.0000    2.4000    2.8000 ···
```

Part 6

```
R=2.46721652
```

```
R = 2.4672
```

Shortened value to 5 sig-figs

```
format long
R, round(R,5)
```

```
R =
   2.467216520000000
ans =
   2.467220000000000
```

Formats value to 15 places behind the decimal point

- **round(v, n) -** Rounds the value of v to the 10^n'th place after the decimal

```
format rat
R
```

```
R =
    5983/2425
```

Formats value into a fraction

```
format short
R, round(R,3)
```

```
R = 2.4672
ans = 2.4670
```

Formats value to 4 places behind the decimal point

Part 7

```
A, A+A
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 3×4
     0     6    12    18
     2     8    14    20
     4    10    16    22
```

```
A, 3*A
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 3×4
     0     9    18    27
     3    12    21    30
     6    15    24    33
```

```
F, magic(4), U=F+magic(4)
```

```
F = 4×4
     1     1    -3    -1
```

```
     7     3     6     3
     5     2     8    11
     3     0     6     9
ans = 4×4
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
U = 4×4
    17     3     0    12
    12    14    16    11
    14     9    14    23
     7    14    21    10
```

```
x, y, x+y
```

```
x = 1×4
     2     3     4     5
y = 4×1
     1
     3
     5
     7
ans = 4×4
     3     4     5     6
     5     6     7     8
     7     8     9    10
     9    10    11    12
```

x + y returns a matrix which every column of x is added to every row of y and placed in the position (xindex, yindex)

```
A,A',transpose(A)
```

```
A = 3×4
     0     3     6     9
     1     4     7    10
     2     5     8    11
ans = 4×3
     0     1     2
     3     4     5
     6     7     8
     9    10    11
ans = 4×3
     0     1     2
     3     4     5
     6     7     8
     9    10    11
```

The apostrophe after the matrix "transposes" it so the rows become columns and the columns become rows.

- **transpose(m) -** Inverts the matrix m, making the rows into columns and vice versa.

Part 8

```
P=[1 2 3 4; 1 1 1 1], Q=transpose(P)
```

```
P = 2×4
     1     2     3     4
     1     1     1     1
Q = 4×2
```

```
     1     1
     2     1
     3     1
     4     1
```

```
P * Q
```

```
ans = 2×2
    30    10
    10     4
```

Calculated by multiplying [PxQ1 PxQ2 ...]

```
 P .* P
```

```
ans = 2×4
     1     4     9    16
     1     1     1     1
```

Multiplies matrix by itself using dot product, simply: P1xP1, P2xP2, P3xP3...

```
 P .^ 2
```

```
ans = 2×4
     1     4     9    16
     1     1     1     1
```

Essentially squares every value in the matrix

```
G = [1 2 3; 4 5 6; 7 8 9]
```

```
G = 3×3
     1     2     3
     4     5     6
     7     8     9
```

```
G^2, G * G
```

```
ans = 3×3
    30    36    42
    66    81    96
   102   126   150
ans = 3×3
    30    36    42
    66    81    96
   102   126   150
```

```
G*G == G^2, isequal(G*G,G^2)
```

```
ans = 3×3 logical array
   1   1   1
   1   1   1
   1   1   1
ans = logical

   1
```

Part 9

```
 rand(4),   rand(3,4),  randi(100,2),  randi(10,2,4),  randi([10 40],2,4)
```

```
ans = 4×4
    0.3897    0.1320    0.0598    0.0154
    0.2417    0.9421    0.2348    0.0430
    0.4039    0.9561    0.3532    0.1690
    0.0965    0.5752    0.8212    0.6491
ans = 3×4
    0.7317    0.5470    0.1890    0.3685
    0.6477    0.2963    0.6868    0.6256
    0.4509    0.7447    0.1835    0.7802
ans = 2×2
     9    78
    93    49
ans = 2×4
     5     4     6     8
     5     6     9     7
ans = 2×4
    21    26    39    27
    35    20    37    29
```

- **rand(a, b) -** Creates a matrix of size a x b with random values between 0 and 1. If b is blank, matrix size is a x a
- **randi(r, a, b) -** Creates a matrix of size a x b with random whole values between 0 and r, if r is a vector with 2 numbers, function returns values between those two numbers.

```
5*rand(3), -3+5*rand(3)
```

```
ans = 3×3
    2.9352    2.3546    0.9738
    1.0387    1.1524    1.1296
    1.5062    4.2215    0.8535
ans = 3×3
   -1.8617    1.6169    1.5244
   -0.8215   -0.8490    1.8987
   -1.4445   -2.0759   -0.8057
```

```
6*rand(3, 2) + 2
```

```
ans = 3×2
    2.6667    5.5694
    3.5484    3.5733
    4.4523    5.6171
```

```
randi([40 70], 3, 2)
```

```
ans = 3×2
    62    49
    46    49
    43    53
```

Exercise 2

Part 1

```
M=3*ones(2), N=[0 3; 3 3]
```

```
M = 2×2
     3     3
     3     3
N = 2×2
```

```
     0     3
     3     3
```

```
M == N, M ~= N
```

```
ans = 2×2 logical array
   0   1
   1   1
ans = 2×2 logical array
   1   0
   0   0
```

```
isequal(M, N), ~isequal(M, N)
```

```
ans = logical

   0
ans = logical

   1
```

The difference between the operator **==** and the **isequal()** function is the output format. For the operator, it checks individual indexes and displays the matrix to the screen as 0 (if false) and 1 (if true). For the function, it either returns as a single 0 (if false) or 1 (if true)

Part 2

```
x = [2 3 4 5], y = [1;3;5;7], y', size(x), size(y), size(y')
```

```
x = 1×4
     2     3     4     5
y = 4×1
     1
     3
     5
     7
ans = 1×4
     1     3     5     7
ans = 1×2
     1     4
ans = 1×2
     4     1
ans = 1×2
     1     4
```

```
if isequal(size(x),size(y))
    disp('matrices x and y are of the same size')
else
    disp('the sizes of x and y are different')
end
```

```
the sizes of x and y are different
```

```
if isequal(x, transpose(y))
    disp('matrixes y and transpose y are equal')
elseif isequal(size(x), size(transpose(y)))
    disp('matrices x and transpose y are of the same size')
else
    disp('matrices x and transpose y are neither equal or of the same size')
```

```
end
```

```
matrices x and transpose y are of the same size
```

#1

```
if ~isequal(M, N)
    disp('M and N are different')
else
    disp('M and N are the same')
end
```

```
M and N are different
```

#2

```
if M ~= N
    disp('M and N are different')
else
    disp('M and N are the same')
end
```

```
M and N are the same
```

#3

```
if M==N
    disp('M and N are the same')
else
    disp('M and N are different')
end
```

```
M and N are different
```

- #2 returns bad output, most likely because the first value in the matrix is a 1, at that is the value that the if statement reads, thus returning that M~=N is false.

Part 3

```
L=zeros(5);
for i=1:5
    for j=1:5
        L(i,j)=i+j;
    end
end
L
```

```
L = 5×5
     2     3     4     5     6
     3     4     5     6     7
     4     5     6     7     8
     5     6     7     8     9
     6     7     8     9    10
```

The matrix entries were calculated using the sum of the index position of the row and column. Ex.(Row) 1 + (Col) 1 = 2

```
z=transpose(1:5);
ZF=zeros(5,3);
n=size(ZF,2);
for i=1:n
    ZF(:,i)=z.^i;
end
ZF
```

```
ZF = 5×3
     1     1     1
     2     4     8
     3     9    27
     4    16    64
     5    25   125
```

```
z=transpose(1:5);
ZV=zeros(5,3);
n=size(ZF,2);
i=1:n;
ZV(:,i)=z.^i
```

```
ZV = 5×3
     1     1     1
     2     4     8
     3     9    27
     4    16    64
     5    25   125
```

1. The loop version sets i equal to one column at a time and performs the computation of setting each value in the active column to the respective value in z raised by the column number i.
2. The vectorized statement does nearly exactly what the for loop does, but in a elegant and more sophisticated manner. It sets the indexer i to the range of the columns and the vectorized statement computes each value in ZV the same way the loop does, but in one swift, incremental, and behind the scenes opearation

(a)

```
H = hilb(5)
```

```
H = 5×5
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

(b)

```
HF = zeros(5);
m = size(H,1);  n = size(H,2);
for i = 1:m
    for j = 1:n
```

```
        HF(i, j) = 1/(i+j-1);
    end
end
HF
```

```
HF = 5×5
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

```
i = 1:5; j=transpose(1:5);
HV = 1./(i+j-1)
```

```
HV = 5×5
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

Exercise 3

```
format compact
p = 1;
H = hilb(8)
```

```
H = 8×8
    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250
    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111
    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000
    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000    0.0909
    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000    0.0909    0.0833
    0.1667    0.1429    0.1250    0.1111    0.1000    0.0909    0.0833    0.0769
    0.1429    0.1250    0.1111    0.1000    0.0909    0.0833    0.0769    0.0714
    0.1250    0.1111    0.1000    0.0909    0.0833    0.0769    0.0714    0.0667
```

```
closetozeroroundoff(H, p)
```

```
ans = 8×8
    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250
    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111
    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000
    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000         0
    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000         0         0
    0.1667    0.1429    0.1250    0.1111    0.1000         0         0         0
    0.1429    0.1250    0.1111    0.1000         0         0         0         0
    0.1250    0.1111    0.1000         0         0         0         0         0
```

The values under .10 were shortened to 0. This makes sense because the function made anything less than 10^-1 equal to 0.

```
type closetozeroroundoff
```

```
function B = closetozeroroundoff(A,p)
A(abs(A)<10^-p)=0;
B=A;
end
```

```
type produc
```

```
%The function calculates the product of two matrices A and B, when it is
%defined. Each column of the matrix AB is calculated as a product of the
%matrix A and the corresponding column of the matrix B. The output AB is
%compared with the output of the MATLAB built-in function A*B. Then, we
%demonstrate that the property of invertible matrices holds: the inverse
%of the product of two matrices inv(A*B) is equal to the product of the
%inverses but in the reverse order: inv(B)*inv(A.
function [AB,InvAB,InvBInvA] = produc(A,B,p)
%calculating the sizes:
[m,n]=size(A);
[k,q]=size(B);
%making assignments:
AB=[];
InvAB=[];
InvBInvA=[];
%checking if the matrices inner dimensions agree for multiplication
if isequal(n,k)
    disp('the matrices dimensions agree for matrix multiplication')
    %calculating the product of A and B:
    i=1:q;
    AB(:,i)=A*B(:,i);
    fprintf('the product of two matrices A and B is\n')
    AB
    %verifying if the code is correct by running matlab function A*B
    C=zeros(m,q);
    fprintf('the output for matlab function A*B is\n')
    C=A*B;
    disp(C)
    if isequal(AB,C)
        fprintf('the output AB is the same as the output for A*B\n')
    else
        disp('check the code!')
        return
    end
    %The next task applies to square matrices A and B of the same size
    %which are also invertible. We demonstrate that the following
    %property holds: inv(A*B)=inv(B)*inv(A)
    if m==n && k==q && rank(A)==m && rank(B)==k
        disp('A and B are invertible matrices of the same size')
        fprintf('the inverse of A*B is\n')
        InvAB=inv(AB)
        fprintf('product of inverses in reverse order inv(B)*inv(A) is\n')
        InvBInvA=inv(B)*inv(A)
        if closetozeroroundoff(InvAB-InvBInvA,p)==0
            fprintf('inv(A*B)=inv(B)*inv(A) holds for the given A and B\n')
            fprintf('and within the given precision 10^(-%i)\n',p)
        else
            fprintf('inv(A*B)=inv(B)*inv(A) does not hold for the given\n')
            fprintf('A and B within the given precision 10^(-%i)\n',p)
        end
    end
else
    fprintf('the matrices dimensions disagree for matrix multiplication\n')
end
end
```

```
p = 6;
```

(a).

```
A = magic(3), B = hilb(3)
```

```
A = 3×3
     8     1     6
     3     5     7
     4     9     2
B = 3×3
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

```
[AB, InvAB, InvBInvA] = produc(A, B, p);
```

```
the matrices dimensions agree for matrix multiplication
the product of two matrices A and B is
AB = 3×3
   10.5000    5.8333    4.1167
    7.8333    4.9167    3.6500
    9.1667    5.5000    3.9833
the output for matlab function A*B is
   10.5000    5.8333    4.1167
    7.8333    4.9167    3.6500
    9.1667    5.5000    3.9833
the output AB is the same as the output for A*B
A and B are invertible matrices of the same size
the inverse of A*B is
InvAB = 3×3
    2.9417    3.5667   -6.3083
  -13.5333  -24.5333   36.4667
   11.9167   25.6667  -35.5833
product of inverses in reverse order inv(B)*inv(A) is
InvBInvA = 3×3
    2.9417    3.5667   -6.3083
  -13.5333  -24.5333   36.4667
   11.9167   25.6667  -35.5833
inv(A*B)=inv(B)*inv(A) holds for the given A and B
and within the given precision 10^(-6)
```

```
InvAB = InvBInvA
```

```
InvAB = 3×3
    2.9417    3.5667   -6.3083
  -13.5333  -24.5333   36.4667
   11.9167   25.6667  -35.5833
```

The reason we need to use the function closetozeroroundoff is to ensure the matrices are similar enough to be called identical, to take advantage of the roundoff function, we say that if we found the difference between two matrices to be so small, will be deemed insignificant, and rounded off to 0.

(b)

```
A=magic(4), B=ones(4)
```

```
A = 4×4
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
B = 4×4
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
[AB,InvAB,InvBInvA]=produc(A,B,p);
```

```
the matrices dimensions agree for matrix multiplication
the product of two matrices A and B is
AB = 4×4
    34    34    34    34
    34    34    34    34
    34    34    34    34
    34    34    34    34
the output for matlab function A*B is
    34    34    34    34
    34    34    34    34
    34    34    34    34
    34    34    34    34
the output AB is the same as the output for A*B
```

(c)

```
A=magic(5), B=ones(5,4)
```

```
A = 5×5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
B = 5×4
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
[AB,InvAB,InvBInvA]=produc(A,B,p);
```

```
the matrices dimensions agree for matrix multiplication
the product of two matrices A and B is
AB = 5×4
    65    65    65    65
    65    65    65    65
    65    65    65    65
    65    65    65    65
    65    65    65    65
the output for matlab function A*B is
    65    65    65    65
    65    65    65    65
    65    65    65    65
    65    65    65    65
    65    65    65    65
the output AB is the same as the output for A*B
```

(d)

```
A=magic(5), B=ones(4,5)
```

```
A = 5×5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
B = 4×5
```

```
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

```
[AB, InvAB, InvBInvA]=produc(A,B,p);
```

```
the matrices dimensions disagree for matrix multiplication
```

(e)

```
A=magic(5), B=hilb(5)
```

```
A = 5×5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
B = 5×5
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

```
[AB,InvAB,InvBInvA]=produc(A,B,p);
```

```
the matrices dimensions agree for matrix multiplication
the product of two matrices A and B is
AB = 5×5
   34.3333   20.8500   15.3429   12.2345   10.2095
   34.5333   20.3833   14.9357   11.9167    9.9611
   20.7333   14.9167   11.9095    9.9738    8.6016
   28.1833   18.4500   14.0619   11.4417    9.6726
   30.6333   19.6500   14.7857   11.9274   10.0214
the output for matlab function A*B is
   34.3333   20.8500   15.3429   12.2345   10.2095
   34.5333   20.3833   14.9357   11.9167    9.9611
   20.7333   14.9167   11.9095    9.9738    8.6016
   28.1833   18.4500   14.0619   11.4417    9.6726
   30.6333   19.6500   14.7857   11.9274   10.0214
the output AB is the same as the output for A*B
A and B are invertible matrices of the same size
the inverse of A*B is
InvAB = 5×5
   -0.0050    0.0028    0.0015   -0.0090    0.0097
    0.0871   -0.0491   -0.0304    0.1738   -0.1816
   -0.3616    0.2055    0.1383   -0.7657    0.7844
    0.5320   -0.3049   -0.2170    1.1735   -1.1853
   -0.2553    0.1474    0.1092   -0.5803    0.5799
product of inverses in reverse order inv(B)*inv(A) is
InvBInvA = 5×5
   -0.0050    0.0028    0.0015   -0.0090    0.0097
    0.0871   -0.0491   -0.0304    0.1738   -0.1816
   -0.3616    0.2055    0.1383   -0.7657    0.7844
    0.5320   -0.3049   -0.2170    1.1735   -1.1853
   -0.2553    0.1474    0.1092   -0.5803    0.5799
inv(A*B)=inv(B)*inv(A) holds for the given A and B
and within the given precision 10^(-6)
```

I cannot pin point the exact reason why **(e)** does not hold when inversed over the two functions. However, I believe it has something to do with the round off value. It differs from **(a)** by stating that the inverse set equal to the computed inverse does not hold.

Exercise 4

```
type dotangle
```

```
function [d1,d2,d] = dotangle(u,v)
m = length(u);
n = length(v);
if (m == n)
    fprintf('both vectors have %i entries\n',n)
else
    disp('the dot product is not defined')
    d1 = [];
    d2 = [];
    d = [];
    return
end

d1 = sum(diag(transpose(u) * v));
d2 = 0;
for i = 1:m
    d2 = d2 + (u(i) * v(i));
end
d = dot(u,v);

if (d1 == d2 && d1 == d && d2 == d)
    fprintf('the code is correct\n')
else
    disp('check the code!')
end
theta = acosd(d /(norm(u) * norm(v)));

if (closetozeroroundoff(abs(theta) - 90, 5) == 0)
    disp('the angle between the vectors is 90 degrees.')
elseif (closetozeroroundoff(abs(theta), 5) == 0)
    disp('the angle between the vectors is zero.')
elseif (closetozeroroundoff(abs(theta) - 180, 5) == 0)
    disp('the angle between the vectors is 180 degrees.')
elseif (theta < 90)
    X = ['angle between vectors is acute and its value theta= ', theta];
    disp(X)
elseif (theta > 90)
    X = ['angle between vectors is obtuse and its value theta= ', theta];
    disp(X)
end
end
```

(a)

```
u = randi(15, 15, 1), v = randi(15,4, 1)
```

```
u = 15×1
     7
    13
     2
     2
     3
```

```
     6
    13
    13
     1
     6
     ⋮
v = 4×1
     7
     1
    15
     3
```

```
[d1, d2, d] = dotangle(u, v)
```

```
the dot product is not defined
d1 =
     []
d2 =
     []
d =
     []
```

(b)

```
u=randi(15,5,1), v=randi(15,5,1)
```

```
u = 5×1
     2
     6
     3
     8
     6
v = 5×1
    15
    14
     1
    12
     5
```

```
[d1,d2,d]=dotangle(u,v);
```

```
both vectors have 5 entries
the code is correct
angle between vectors is acute and its value theta= #
```

(c)

```
u=u, v=-v
```

```
u = 5×1
     2
     6
     3
     8
     6
v = 5×1
   -15
   -14
    -1
   -12
    -5
```

```
[d1,d2,d]=dotangle(u,v);
```

```
both vectors have 5 entries
the code is correct
angle between vectors is obtuse and its value theta=
```

(d)

```
u=u, v=2*u
```

```
u = 5×1
     2
     6
     3
     8
     6
v = 5×1
     4
    12
     6
    16
    12
```

```
[d1,d2,d]=dotangle(u,v);
```

```
both vectors have 5 entries
the code is correct
the angle between the vectors is zero.
```

(e)

```
u=u, v=-3*u
```

```
u = 5×1
     2
     6
     3
     8
     6
v = 5×1
    -6
   -18
    -9
   -24
   -18
```

```
[d1,d2,d]=dotangle(u,v);
```

```
both vectors have 5 entries
the code is correct
the angle between the vectors is 180 degrees.
```

(f)

```
u=[1;3],v=[-3;1]
```

```
u = 2×1
     1
     3
v = 2×1
    -3
```

1

```
[d1,d2,d]=dotangle(u,v);
```

```
both vectors have 2 entries
the code is correct
the angle between the vectors is 90 degrees.
```