## Table of Contents



# Actual:Let's first define an identity matrix of size n

```
n = 4;
I = eye(n);
J = ones(n);
Z = zeros(n);
m = 3;

delta = eye(n,m) %(rows, columns)

C = [1 2;3 4;5 6]
B = 3.*J

%One way to initialize a matrix is to start out with one of these and
 use a
%loop to populate it iteratively.
A = ones(n);
for i=1:n
    A(i,:) = i+1; %this performs the same action on all of the
 elements of the i-th row
    A(:,i) = i^2; %this performs the same action on all of the
 elements of the i-th column
    A(i,i) = A(1,3); %we can also self-reference
end
A
x = A(:,2)

D = rand(n); %Generate an nxn matrix with random entries between 0 and
 1
E = randi(n); %Generates a random integer between 1 and n
F = randi(100,n) %Generates an nxn matrix with integer entries between
 1 and 100
G = magic(n)
```

```
delta =

     1     0     0
```

```
     0     1     0
     0     0     1
     0     0     0


C =

     1     2
     3     4
     5     6


B =

     3     3     3     3
     3     3     3     3
     3     3     3     3
     3     3     3     3


A =

     2     4     9    16
     3     2     9    16
     4     4     9    16
     5     5     5     9


x =

     4
     2
     4
     5


F =

    68    60    85    71
    25    81    36    75
    48    11    44    76
    40    83    58    39


G =

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

# New Section

```
A = ones(n);
for i=1:n
    A(i,:) = i+1; %this performs the same action on all of the
 elements of the i-th row
    A(:,i) = i^2; %this performs the same action on all of the
 elements of the i-th column
    A(i,i) = A(1,3); %we can also self-reference
end
[evectors,evalues] = eig(A)
[Q,R] = qr(A)
[U,S,V] = svd(A) %A = USV^T
```

```
evectors =

    0.5192    0.4773   -0.6109    0.0085
    0.5020    0.6577    0.7568    0.0043
    0.5561    0.2718    0.1757    0.8706
    0.4112   -0.5155   -0.1523   -0.4918


evalues =

   28.1783         0         0         0
         0   -4.6446         0         0
         0         0   -1.5533         0
         0         0         0    0.0197


Q =

   -0.2722    0.8795    0.3255    0.2156
   -0.4082   -0.4729    0.6510    0.4312
   -0.5443   -0.0332    0.1772   -0.8193
   -0.6804   -0.0415   -0.6625    0.3105


R =

   -7.3485   -7.4846  -14.4248  -25.7196
         0    2.2319    3.1529    5.6006
         0         0    7.0700   12.4953
         0         0         0    0.0345


U =

   -0.5412   -0.2433    0.7757    0.2149
   -0.5348   -0.3895   -0.6144    0.4298
   -0.5521    0.0689   -0.1365   -0.8196
   -0.3410    0.8856   -0.0465    0.3118
```

```
S =

   34.7868         0         0         0
         0    4.4194         0         0
         0         0    1.5317         0
         0         0         0    0.0170


V =

   -0.1897    0.6898   -0.6986   -0.0088
   -0.2055    0.6678    0.7153   -0.0062
   -0.4702   -0.1464   -0.0059   -0.8703
   -0.8371   -0.2380   -0.0139    0.4924
```

# Notes: Some Copies of stuff above but also some other stuff I didn't get to.

```
n = 4; %If I don't want something to print, put a semi-colon to end
 the line
I = eye(n)

%We can also generate a matrix of zeros or a matrix of ones
Z = zeros(n)
O = ones(n)

%Both of these take second input arguments if you need non-square
 matrices
m=3;
delta = eye(n,m)
z = zeros(n,m)
o = ones(n,m)

%One way to initialize a matrix is to start out with one of these and
 use a
%loop to populate it iteratively.
A = ones(n)
for i=1:n
    A(i,:) = i+1; %this performs the same action on all of the
 elements of the i-th row
    A(:,i) = i^2; %this performs the same action on all of the
 elements of the i-th column
    A(i,i) = A(1,3); %we can also self-reference
end
A  %Simply typing a variable will print it out
B = rand(n) % will generate a nxn matrix with random entries between 0
 and 1
C = randi(n) %generates a random integer between 1 and n
```

```
D = randi(m,n) %generates an nxn matrix with random integers entries
 between 1 and m
E = [1 2;3 4;5 6] %lastly we could simply type it out.Use spaces to
 separate entries, semi-colons rows.
```

```
I =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1


Z =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0


O =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


delta =

     1     0     0
     0     1     0
     0     0     1
     0     0     0


z =

     0     0     0
     0     0     0
     0     0     0
     0     0     0


o =

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

```
A =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


A =

     2     4     9    16
     3     2     9    16
     4     4     9    16
     5     5     5     9


B =

    0.4293    0.2763    0.0859    0.9047
    0.9563    0.6223    0.5005    0.8844
    0.5730    0.5884    0.5216    0.4390
    0.8497    0.9635    0.0902    0.7817


C =

     1


D =

     2     1     3     3
     1     1     3     2
     2     2     1     3
     3     2     1     3


E =

     1     2
     3     4
     5     6
```

## Say we had a two matrices and we wanted to compare them

```
A = ones(n)
B = rand(n)
A == B %Because none of the entries match, the logical array contains
 only zeros.
```

```
%If they did match, it would contain 1's in those entries. Sizes do
 need to match.

%Definitely want basic matrix operations:
sum = A+B
prod = A*B
power = A^m

%If we want to perform entry-wise operations we do .(operation):
(0.5).*A
A.^m
A.*B

%If you want conjugate transpose, it's ' but if you only want
 transpose, it's .'
A'

%If you want the inverse of a matrix
A_inv = inv(A)
%BUT if you're trying to solve Ax=b, better to use x = b\A
b = [1;2;3;4]
x = b\A


A =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


B =

    0.6437    0.9852    0.4840    0.3954
    0.8601    0.5595    0.6390    0.9922
    0.4019    0.9336    0.8876    0.4024
    0.6319    0.7203    0.1987    0.6589


ans =

  4×4 logical array

   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0


sum =

    1.6437    1.9852    1.4840    1.3954
    1.8601    1.5595    1.6390    1.9922
```

```
    1.4019    1.9336    1.8876    1.4024
    1.6319    1.7203    1.1987    1.6589


prod =

    2.5376    3.1986    2.2094    2.4487
    2.5376    3.1986    2.2094    2.4487
    2.5376    3.1986    2.2094    2.4487
    2.5376    3.1986    2.2094    2.4487


power =

    16    16    16    16
    16    16    16    16
    16    16    16    16
    16    16    16    16


ans =

    0.5000    0.5000    0.5000    0.5000
    0.5000    0.5000    0.5000    0.5000
    0.5000    0.5000    0.5000    0.5000
    0.5000    0.5000    0.5000    0.5000


ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


ans =

    0.6437    0.9852    0.4840    0.3954
    0.8601    0.5595    0.6390    0.9922
    0.4019    0.9336    0.8876    0.4024
    0.6319    0.7203    0.1987    0.6589


ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

Warning: Matrix is singular to working precision.

A_inv =
```

```
   Inf   Inf   Inf   Inf
   Inf   Inf   Inf   Inf
   Inf   Inf   Inf   Inf
   Inf   Inf   Inf   Inf


b =

     1
     2
     3
     4


x =

    0.3333    0.3333    0.3333    0.3333
```

# So you wanna talk about eigen-things

```
A = [1 2;3 4] %Use A = [1 2;3 4] to show that errors kick in.
e = eig(A) %Just gives a vector containing the eigenvalues
[evectors,evalues] = eig(A) % produces both, with eigenvalues in a
 diagonal matrix
%Check
A*evectors(:,1) == evectors(:,1)*evalues(1,1) %Errors abound, so be
 careful

%To actually check, check norm of difference:
norm(A*evectors(:,1)-evectors(:,1)*evalues(1,1))%Pretty small,
 basically at machine precision

% Characteristic polynomial
syms x;
p = charpoly(A,x)
factor(p) %can't be factored over the rationals

rank(A)%full rank
```

```
A =

     1     2
     3     4


e =

   -0.3723
    5.3723
```

```
evectors =

   -0.8246   -0.4160
    0.5658   -0.9094


evalues =

   -0.3723         0
         0    5.3723


ans =

  2×1 logical array

   1
   0


ans =

   5.5511e-17


p =

x^2 - 5*x - 2


ans =

x^2 - 5*x - 2


ans =

     2
```

# Singular Value Decomposition

```
A = ones(n);
for i=1:n
    A(i,:) = i+1; %this performs the same action on all of the
 elements of the i-th row
    A(:,i) = i^2; %this performs the same action on all of the
 elements of the i-th column
    A(i,i) = A(1,3); %we can also self-reference
end
A
s = svd(A)%Just grabs the singular values
```

```
[U,S,V] = svd(A) %S will be diagonal, U and V will be unitary
norm(A - U*S*V') %darn small
norm(A-U*S*V)%not small, require the transpose of V

%When trying to solve a linear system, it's important to know if the
 matrix
%A is ill-conditioned or not. You can call
%The coefficient matrix is called ill-conditioned if a small change in
 the
%constant coefficients results in a large change in the solution.
dA = decomposition(A)
tf = isIllConditioned(dA) %not ill conditioned, could solve Ax=b
 without issue

%Also works on rectangular matrices
E = [1 2;3 4;5 6]
[U1,S1,V1] = svd(E)
```

```
A =

     2     4     9    16
     3     2     9    16
     4     4     9    16
     5     5     5     9


s =

   34.7868
    4.4194
    1.5317
    0.0170


U =

   -0.5412   -0.2433    0.7757    0.2149
   -0.5348   -0.3895   -0.6144    0.4298
   -0.5521    0.0689   -0.1365   -0.8196
   -0.3410    0.8856   -0.0465    0.3118


S =

   34.7868         0         0         0
         0    4.4194         0         0
         0         0    1.5317         0
         0         0         0    0.0170


V =

   -0.1897    0.6898   -0.6986   -0.0088
```

```
   -0.2055    0.6678    0.7153   -0.0062
   -0.4702   -0.1464   -0.0059   -0.8703
   -0.8371   -0.2380   -0.0139    0.4924


ans =

   2.9889e-14


ans =

   43.2047


dA =

  decomposition with properties:

    MatrixSize: [4 4]
          Type: 'lu'

  Show <a href="matlab:if
 exist('dA','var'),displayAllProperties(dA),else,disp('Unable
 to display properties for variable dA because it no longer
 exists.');end">all properties</a>


tf =

  logical

   0


E =

     1     2
     3     4
     5     6


U1 =

   -0.2298    0.8835    0.4082
   -0.5247    0.2408   -0.8165
   -0.8196   -0.4019    0.4082


S1 =

    9.5255         0
         0    0.5143
         0         0
```

```
V1 =

   -0.6196   -0.7849
   -0.7849    0.6196
```

# QR Decomposition

```
A = ones(n);
for i=1:n
    A(i,:) = i+1;
    A(:,i) = i^2;
    A(i,i) = A(1,3);
end
A
[Q,R] = qr(A) %Q is orthogonal, R will be upper triangular
norm(Q*R - A) %Pretty darn small, good enough

%Also works on rectangular matrices
E = [1 2;3 4;5 6]
[Q1,R1] = qr(E)
```

```
A =

     2     4     9    16
     3     2     9    16
     4     4     9    16
     5     5     5     9


Q =

   -0.2722    0.8795    0.3255    0.2156
   -0.4082   -0.4729    0.6510    0.4312
   -0.5443   -0.0332    0.1772   -0.8193
   -0.6804   -0.0415   -0.6625    0.3105


R =

   -7.3485   -7.4846  -14.4248  -25.7196
         0    2.2319    3.1529    5.6006
         0         0    7.0700   12.4953
         0         0         0    0.0345


ans =

   2.0256e-14
```

```
E =

     1     2
     3     4
     5     6


Q1 =

   -0.1690    0.8971    0.4082
   -0.5071    0.2760   -0.8165
   -0.8452   -0.3450    0.4082


R1 =

   -5.9161   -7.4374
         0    0.8281
         0         0
```

# Galleries for testing

```
%Sometimes we desire to test something on a bunch of matrices of a
 certain
%type. the gallery() function can do this

A1 = gallery('jordbloc', 3, 2)%So I can generate Jordan Blocks and
 then insert them if needed.
A2 = gallery('minij',n)%I can also generate a symmetric positive
 definite matrix
```

```
A1 =

     2     1     0
     0     2     1
     0     0     2


A2 =

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

*Published with MATLAB® R2018a*