diary on

format compacts

%Exercise 1

%(1)

%creates a matrix 5x4 with each entry being 1 or 0

A = randi([0 1], 5,4)

A =

1 0 0 0

1 0 1 0

0 1 1 1

1 1 0 1

1 1 1 1

A = randi([0 1], 5,4)

A =

1 1 1 1

0 1 0 1

1 0 0 0

1 1 0 1

1 0 0 0

A = randi([0 1], 5,4)

A =

0 0 0 0

0 0 1 1

1 1 1 0

1 1 0 1

0 1 0 0

%(2)

%randomly generates a column of values

B = randi([0 9],4,1)

B =

9

5

1

1

%reference the column values to create a alternant matrix

C =

[B(1,1)^0,B(1,1)^1,B(1,1)^2,B(1,1)^3,B(1,1)^4,B(1,1)^5;B(2,1)^0,B(2,1)^1,B(2,

1)^2,B(2,1)^3,B(2,1)^4,B(2,1)^5;B(3,1)^0,B(3,1)^1,B(3,1)^2,B(3,1)^3,B(3,1)^4,

B(3,1)^5;B(4,1)^0,B(4,1)^1,B(4,1)^2,B(4,1)^3,B(4,1)^4,B(4,1)^5]

C =

1 9 81 729 6561 59049

1 5 25 125 625 3125

1 1 1 1 1 1

1 1 1 1 1 1

%(3)

%create a diagonal matrix and use the flip command to create the

%desired matrix

v = randi([0 9],1,5)

v =

3 1 2 6 4

flip(diag(v))

ans =

0 0 0 0 4

0 0 0 6 0

0 0 2 0 0

0 1 0 0 0

3 0 0 0 0

%4

%create random diagonal

c = diag(randi([10 100],1,6))

c =

18 0 0 0 0 0

0 77 0 0 0 0

0 0 11 0 0 0

0 0 0 46 0 0

0 0 0 0 26 0

0 0 0 0 0 32

%replace the values of row 1 with random values 10-100 rounding

%to nearest whole number

c(1,:) = floor(100\*rand([1,6]))

c =

57 13 61 76 58 99

0 77 0 0 0 0

0 0 11 0 0 0

0 0 0 46 0 0

0 0 0 0 26 0

0 0 0 0 0 32

%replace the values of column 1 with random values 10-100

%rounding to nearest whole number

c(:,1) = floor(100\*rand([6,1]))

c =

41 13 61 76 58 99

24 77 0 0 0 0

48 0 11 0 0 0

76 0 0 46 0 0

84 0 0 0 26 0

55 0 0 0 0 32

%Exercise2

type multi

function [C,CRows,CColumns]= multi(A,B)

[m,p] = size(A);

[q,n] = size(B);

C=zeros(m, n);

CRows=zeros(m, n);

CColumns=zeros(m, n);

if p==q

%row by row

for i=1:m

for j=1:n

for k=1:p

CRows(i,j) = CRows(i,j) + (A(i,k)\*B(k,j));

end

end

end

%column by column

for j=1:n

for i=1:m

for k=1:p

CColumns(i,j) = CColumns(i,j) + (A(i,k)\*B(k,j));

end

end

end

%by definition

for i=1:n

C(:,i)= A\*B(:,i);

end

else

disp('The dimensions of A and B disagree')

C=[];

CRows=[];

CColumns=[];

end

end

%(a)

A=randi(10,2,3)

A =

3 10 2

6 10 10

B=magic(2)

B =

1 3

4 2

[C, CRows, CColumns] = multi(A,B)

The dimensions of A and B disagree

C =

[]

CRows =

[]

CColumns =

[]

%(b)

A= magic(5)

A =

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= ones(4,6)

B =

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

[C, CRows, CColumns] = multi(A,B)

The dimensions of A and B disagree

C =

[]

CRows =

[]

CColumns =

[]

%(c)

A = magic(4)

A =

16 2 3 13

5 11 10 8

9 7 6 12

4 14 15 1

B = ones(4,3)

B =

1 1 1

1 1 1

1 1 1

1 1 1

[C, CRows, CColumns] = multi(A,B)

C =

34 34 34

34 34 34

34 34 34

34 34 34

CRows =

34 34 34

34 34 34

34 34 34

34 34 34

CColumns =

34 34 34

34 34 34

34 34 34

34 34 34

%(d)

A = ones(4)

A =

1 1 1 1

1 1 1 1

1 1 1 1

1 1 1 1

B = diag([2,3,4,5])

B =

2 0 0 0

0 3 0 0

0 0 4 0

0 0 0 5

[C, CRows, CColumns] = multi(A,B)

C =

2 3 4 5

2 3 4 5

2 3 4 5

2 3 4 5

CRows =

2 3 4 5

2 3 4 5

2 3 4 5

2 3 4 5

CColumns =

2 3 4 5

2 3 4 5

2 3 4 5

2 3 4 5

%verifying (a)

A=randi(10,2,3)

A =

10 9 5

5 2 10

B=magic(2)

B =

1 3

4 2

C = A\*B

{\_Error using <a href="matlab:matlab.internal.language.introspective.errorDocCallback('mtimes')" style="font-weight:bold"> \* </a>

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 perform elementwise multiplication, use '.\*'.

}\_

%verifying (b)

A= magic(5)

A =

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= ones(4,6)

B =

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

C=A\*B

{\_Error using <a href="matlab:matlab.internal.language.introspective.errorDocCallback('mtimes')" style="font-weight:bold"> \* </a>

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 perform elementwise multiplication, use '.\*'.

}\_

%verifying (c)

A = magic(4)

A =

16 2 3 13

5 11 10 8

9 7 6 12

4 14 15 1

B = ones(4,3)

B =

1 1 1

1 1 1

1 1 1

1 1 1

C=A\*B

C =

34 34 34

34 34 34

34 34 34

34 34 34

%verifying (d)

A = ones(4)

A =

1 1 1 1

1 1 1 1

1 1 1 1

1 1 1 1

B = diag([2,3,4,5])

B =

2 0 0 0

0 3 0 0

0 0 4 0

0 0 0 5

C=A\*B

C =

2 3 4 5

2 3 4 5

2 3 4 5

2 3 4 5

%all outputs and expected error outputs matched ours.

%Exercise3

type givensrot

function [outputArg1] = givensrot(n,i,j,theta)

if n >= 2 && i >= 1 && j > i && n >= j,

G=eye(n);

G(i,i)=cos(theta);

G(j,j)=cos(theta);

G(i,j)=-sin(theta);

G(j,i)=sin(theta);

else,

G=[];

end

outputArg1 = G;

end

%(a)

%(1)

G=givensrot(4,3,2,pi/2)

G =

[]

%(2)

G=givensrot(5,2,4,pi/4)

G =

1.0000 0 0 0 0

0 0.7071 0 -0.7071 0

0 0 1.0000 0 0

0 0.7071 0 0.7071 0

0 0 0 0 1.0000

%(3)

G=givensrot(3,1,2,pi)

G =

-1.0000 -0.0000 0

0.0000 -1.0000 0

0 0 1.0000

%(b)

G\*[1; 0; 0]

ans =

-1.0000

0.0000

0

G\*[0; 1; 0]

ans =

-0.0000

-1.0000

0

G\*[0; 0; 1]

ans =

0

0

1

G\*[1; 1; 1]

ans =

-1.0000

-1.0000

1.0000

%This is the exact rotation I was expecting. Since theta = pi,

%it should be a 180 degree rotation on the ij plane.

%the result is that all ij values in the vector were

%flipped and the k values were left unchanged.

%This fits the Geometrical Meaning described in exerxise 3.

%Exercise#4

type toeplitz

function A = toeplitz(m, n, a)

for i=1:m

for j=1:n

A(i,j) = a(n + i - j, 1);

end

end

%(1)

%(a)

a = transpose([1:6]);

A = toeplitz(4,3,a)

A =

3 2 1

4 3 2

5 4 3

6 5 4

%(b)

a = randi(10,6,1);

A = toeplitz(3,4,a)

A =

10 5 2 9

8 10 5 2

10 8 10 5

%(c)

a = [zeros(3,1);[1:4]'];

A = toeplitz(4,4,a)

A =

1 0 0 0

2 1 0 0

3 2 1 0

4 3 2 1

%(2)

a = [randi([10 100],5,1); zeros(5,1)]

a =

69

13

87

94

71

0

0

0

0

0

A = toeplitz(5,5,a)

A =

71 94 87 13 69

0 71 94 87 13

0 0 71 94 87

0 0 0 71 94

0 0 0 0 71

%(3)

a = [0;0;0;1;0;0;0]

a =

0

0

0

1

0

0

0

A = toeplitz(4,4,a)

A =

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

%Exercise 5

type stochastic

function P = stochastic(A)

%This takes a matrix and checks if the matrix fits the

%conditions to be a left, right, or double stochastic matrix.

%It also checks if the matrix can be scaled to become a stochastic matrix.

S1 = sum(A,1); %sum of columns

S2 = (sum(A,2))'; %sum of rows

Right = 0; %These are used to keep track if it is a stochastic

RScale = 0; %matrix or if it can be scaled to one

Left = 0;

LScale = 0;

[r,c] = size(A); %Used to see if the matrix is a square, also used to %traverse the matrix when needed

if any(S1==0)==1 && any(S2==0)==1 %This checks if there is a row AND %and column that are all 0's

disp('Error: A is not stochastic and cannot be scaled to stochastic')

S1 = []

S2 = []

P = []

return

end

if any(A(:)<0) == 1 %checks for negative values

disp('Error: matrix cannot contain a negative value. A is not stochastic and cannot be scaled to stochastic')

S1 = []

S2 = []

P = []

return

end

if r ~= c %checks if the matrix is square

disp('Error: not a square matrix')

P = []

return

end

if all(S1==1) %If all entries are 1 it is left stochastic

Left = 1;

end

if all(S1~=0) %If all are not 0 it can be scaled to left

LScale = 1;

end

if all(S2==1) %If all are 1 it is right stochastic

Right = 1;

end

if all(S2~=0) %If all are not 0 it can be scaled to right

RScale = 1;

end

if Left==1 && Right==1 %If right and left, it is both

disp('This is a double stochastic matrix')

P=A

return

end

if Left==1 && Right==0

disp('This is a left stochastic matrix')

P=A

return

end

if Left==0 && Right==1

disp('This is a right stochastic matrix')

P=A

return

end

%If it was none of the options above, it checks if it is scale-able

if LScale==1

disp('This is neither left nor right stochastic but can be scaled to left stochastic')

disp('S1 = ')

disp(S1)

disp('S2 = ')

disp(S2)

for i=1:c

A(:,i)=A(:,i)\*(1/S1(1,i));

end

Z1=sum(A,1); %I used these to check to see if the matrix %is double stochastic after the scaling

Z2=(sum(A,2))';

if all(Z1==1) && all(Z2==1) %If all entries are 1 for both than it has %now become a double stochastic Matrix

disp('After scaling to become left stochastic, P has become a double stochastic matrix ')

P=A

else

disp('P after scaling to left stochastic ' )

P=A

end

elseif RScale==1

disp('This is neither left nor right stochastic but can be scaled to right stochastic')

disp('S1 = ')

disp(S1)

disp('S2 = ')

disp(S2)

for i=1:r

A(i,:)=A(i,:)\*(1/S2(1,i));

end

Z1=sum(A,1); %I used these to check to see if the matrix %is double stochastic after the scaling

Z2=(sum(A,2))';

if all(Z1==1) && all(Z2==1) %If all entries are 1 for both than it has %now become a double stochastic Matrix

disp('After scaling to become right stochastic, P has become a double stochastic matrix ')

P=A

else

disp('P after scaling to right stochastic ' )

P=A

end

else %This should never occur, but I put it in case I missed something %and it is possible

disp('It is neither left nor right stochastic and it is not scale-able')

end

end

%(a)

A=[0.5, 0, 0.5; 0, 0, 1; 0.5, 0, 0.5]

A =

0.5000 0 0.5000

0 0 1.0000

0.5000 0 0.5000

stochastic(A);

This is a right stochastic matrix

P =

0.5000 0 0.5000

0 0 1.0000

0.5000 0 0.5000

%(b)

A=A'

A =

0.5000 0 0.5000

0 0 0

0.5000 1.0000 0.5000

stochastic(A);

This is a left stochastic matrix

P =

0.5000 0 0.5000

0 0 0

0.5000 1.0000 0.5000

%(c)

A=[0.5, 0, 0.5; 0, 0, 1; 0, 0, 0.5]

A =

0.5000 0 0.5000

0 0 1.0000

0 0 0.5000

stochastic(A);

This is neither left nor right stochastic but can be scaled to right stochastic

S1 =

0.5000 0 2.0000

S2 =

1.0000 1.0000 0.5000

P after scaling to right stochastic

P =

0.5000 0 0.5000

0 0 1.0000

0 0 1.0000

%(d)

A=A'

A =

0.5000 0 0

0 0 0

0.5000 1.0000 0.5000

stochastic(A);

This is neither left nor right stochastic but can be scaled to left stochastic

S1 =

1.0000 1.0000 0.5000

S2 =

0.5000 0 2.0000

P after scaling to left stochastic

P =

0.5000 0 0

0 0 0

0.5000 1.0000 1.0000

%(e)

A=[0.5, 0, 0.5; 0, 0.5, 0.5; 0.5, 0.5, 0]

A =

0.5000 0 0.5000

0 0.5000 0.5000

0.5000 0.5000 0

stochastic(A);

This is a double stochastic matrix

P =

0.5000 0 0.5000

0 0.5000 0.5000

0.5000 0.5000 0

%(f)

A=magic(3)

A =

8 1 6

3 5 7

4 9 2

stochastic(A);

This is neither left nor right stochastic but can be scaled to left stochastic

S1 =

15 15 15

S2 =

15 15 15

After scaling to become left stochastic, P has become a double stochastic matrix

P =

0.5333 0.0667 0.4000

0.2000 0.3333 0.4667

0.2667 0.6000 0.1333

%(g)

A= diag([1,2,3])

A =

1 0 0

0 2 0

0 0 3

stochastic(A);

This is neither left nor right stochastic but can be scaled to left stochastic

S1 =

1 2 3

S2 =

1 2 3

After scaling to become left stochastic, P has become a double stochastic matrix

P =

1 0 0

0 1 0

0 0 1

%(h)

A=[0, 0, 0; 0, 0.5, 0.5; 0, 0.5, 0.5]

A =

0 0 0

0 0.5000 0.5000

0 0.5000 0.5000

stochastic(A);

Error: A is not stochastic and cannot be scaled to stochastic

S1 =

[]

S2 =

[]

P =

[]

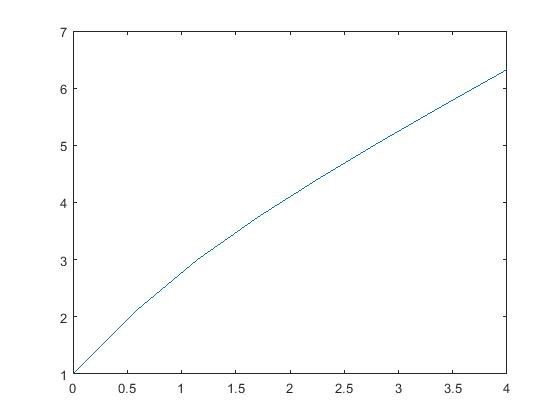
%Exercise 6

%(a)

x=linspace(0,4,8);

y=atan(x)+x+1;

plot(x,y)



%Initial Approx x = -2

syms x

f=atan(x)+x+1

f =

x + atan(x) + 1

g=diff(f)

g =

1/(x^2 + 1) + 1

N=5;

x=-2;

root = newtons(N,x)

x =

-0.244042735171591

x =

-0.509808605510183

x =

-0.520249378464267

x =

-0.520268992650220

x =

-0.520268992719590

root =

-0.520268992719590

%It seems as if the output oscillated between the calculated root. Since I inputted in -2, it jumped higher than the root, then came down at a lower increment from the root

%Approximate Root: -0.52026899

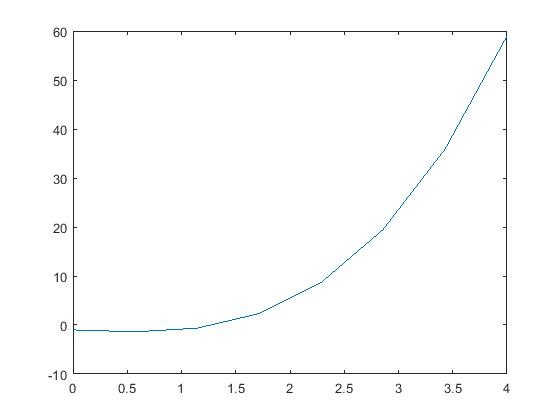
%The fourth and fifth values match on the first 8 decimal points, thus the approximate root must be very close to actual root.

%(b)

x = linspace(0,4,8);

y = x.^3-x-1;

plot(x,y)



syms x

f = x^3 - x - 1

f =

x^3 - x - 1

g = diff(f)

g =

3\*x^2 - 1

N=5;

%(1)

x = 1.5;

root = newtons\_1(N,x)

x =

1.347826086956522

x =

1.325200398950907

x =

1.324718173999054

x =

1.324717957244790

x =

1.324717957244746

root =

1.324717957244746

%This approximate measurement seems to be very close to the root because the fourth and fifth values are very close to one another, implying that x is close to converging.

%(2)

x = 1;

root = newtons\_1(N,x)

x =

1.500000000000000

x =

1.347826086956522

x =

1.325200398950907

x =

1.324718173999054

x =

1.324717957244790

root =

1.324717957244790

%The root of this approximation is the fourth value of x of the previous approximation

%(3)

x = 0.6;

root = newtons\_1(N,x)

x =

17.899999999999984

x =

11.946802328608761

x =

7.985520351936207

x =

5.356909314795458

x =

3.624996032946096

root =

3.624996032946096

%It seems as if going below one throws off the approximation. However, if you increase N, then the root matches the previous values!

%(4)

x = 0.57;

root = newtons\_1(N,x)

x =

-54.165454545454324

x =

-36.114292524925531

x =

-24.082094252098212

x =

-16.063387407817846

x =

-10.721483416797254

root =

-10.721483416797254

%This one was even more off than #3 despite only being a little different.

%Approximate Root Value: 1.32471796

%(c)

N = 10;

%(3)

x = 0.6;

root = newtons\_1(N,x)

x =

17.899999999999984

x =

11.946802328608761

x =

7.985520351936207

x =

5.356909314795458

x =

3.624996032946096

x =

2.505589190106631

x =

1.820129422319469

x =

1.461044109887682

x =

1.339323224262526

x =

1.324912867718656

root =

1.324912867718656

%(4)

x = 0.57;

root = newtons\_1(N,x)

x =

-54.165454545454324

x =

-36.114292524925531

x =

-24.082094252098212

x =

-16.063387407817846

x =

-10.721483416797254

x =

-7.165534466881882

x =

-4.801703812712865

x =

-3.233425234527273

x =

-2.193674204844573

x =

-1.496866569237556

root =

-1.496866569237556

N = 100;

x = 0.6;

root = newtons\_1(N,x)

root =

1.324717957244746

x = 0.57;

root = newtons\_1(N,x)

root =

1.324717957244746

%The process coverges for the initial values of 3 and 4 but will do so slower than 1 and 2.

%Newton's method uses subtraction of values of certain x divided by their derivative values. As the function goes through iterations, it gets closer to the root value. However, the further away the initial x, the more iterations it will take to get to the root. This is because some iterations are needed to reach a closer approximation of the root. Hence the closer the initial x, the less number of iterations necessary.

%This is evident in the program we just ran. Within the first 10 iterations, a and b were already close to the root. However, it took more iterations to get c and d just as close. Thus, a and b must converge quicker than c and d.

diary off