# Matrices

Olagoke Oladokun  
Covenant University, Ota, Nigeria

©2021

!pip install - q numpy  
!pip install - q scipy  
!pip install - q sympy

ERROR: Invalid requirement: '-'  
ERROR: Invalid requirement: '-'  
ERROR: Invalid requirement: '-'

# GEC220

# Engineering Mathematics II

Matrices and Determinants: Solution of system of linear equations by determinants. Linear dependence and independence, rank of a matrix. General system of linear equations, existence and properties of solution, Gaussian elimination. Matrix inverse by elementary matrices, adjoint, and partitioning methods. Characteristic polynomial, characteristic equation, eigenvalues and eigenvectors.

GEC220 Engineering Mathematics II 3 Units LH 45 Lecturers Prof. Vincent Dr. Adeeyo Dr. Olagoke

Partial Differentiation: Functions of several variables, continuity and partial derivatives. Total differentials, approximate calculations using differentials. Chain rule. Implicit differentiation. Series representation of functions (Maclaurin & Taylor’s), Taylor’s Theorem. Extremum problems, (analytic method) without and with constraints, Lagrange multipliers, global extremum. Ordinary Differential Equations: Definition, degree, order, linear, non-linear, solution. First order equations, separable variables, equations reducible to separable form, exact equations, integrating factors, homogenous differential equations. Modeling of engineering systems leading to first order differential equations- electric circuit, mixing/dilution, radioactive decay, bacterial culture. 2nd order differential equations with constant coefficients, homogeneous, non-homogeneous, complementary functions, particular integrals, D-operator method. General linear second-order differential equations (without using matrices). Power series solution, Legendre’s differential equation. Modeling of engineering systems leading to 2nd order differential equations- electric circuit, mechanical oscillations-free and forced, resonance. Pre-requisites: MAT 121 and MAT 122. Matrices and Determinants: Solution of system of linear equations by determinants. Linear dependence and independence, rank of a matrix. General system of linear equations, existence and properties of solution, Gaussian elimination. Matrix inverse by elementary matrices, adjoint, and partitioning methods. Characteristic polynomial, characteristic equation, eigenvalues and eigenvectors.

Contents 1 Matrices 6 1.1 Matrix notation 7 1.2 Addition and Subtraction of Matrices 7 1.3 Multiplication of Matrices 8 1.3.1 Scalar Multiplication 8 1.3.2 Two matrices multiplication 8 1.4 Transpose of matrix 10 1.5 Square matrix 10 1.5.1 Symmetry matrix 10 1.6 Triangular matrix 11 1.7 Diagonal matrix 11 1.8 Unit matrix 11 1.9 Null matrix 12 2 Determinants 13 2.1 Determinants of the second order 13 2.2 Determinants of the third order 14 3 Solution of system of linear equations by determinants 17 3.1 Determinant Equation Method 18 3.1.1 Secondary Degree 18 3.1.2 Third Degree 19 3.2 Cramers Rule 22 3.2.1 For second degree 22 3.2.2 Third degree 22 4 Linear dependence and independence 25 5 Rank of a matrix 27 6 General system of linear equations, 28 7 Existence and properties of solution 31 7.1 Existence of solution 31 7.2 Other properties of the solution 31 8 Gaussian elimination. 33 8.1 Algorithm of Gaussian Elimination 33 8.2 Determinant from Gaussian Elimination 38 9 Matrix inverse by elementary matrices, adjoint, and partitioning methods. 39 9.1 Inverse of a square matrix 39 9.2 Cofactors 39 9.3 Adjoint 41 10 Eigenvalues and Eigenvectors. 43 10.1 Eigenvalues 43 10.1.1 Example 43 10.2 Eigenvectors 44 10.2.1 Example 44

# Matrices

A matrix is a set of real or complex numbers (or elements) arranged in rows and columns to form a rectangular array. A matrix can be identified by its order denoted as matrix.

What is the order of matrix above?. Matrix has . The number of rows of , and number of column,

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
print('===============================')  
print('matrix A =')  
print(A)  
print()  
print('mxn =', np.shape(A))  
print('===============================')

===============================  
matrix A =  
[[ 3 -3 -4]  
 [ 1 -1 -1]  
 [ 2 -1 4]]  
  
mxn = (3, 3)  
===============================

Use symbolic code

from sympy import \*  
  
A = Matrix([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
A

Matrix([  
[3, -3, -4],  
[1, -1, -1],  
[2, -1, 4]])

print('mxn =', A.shape)

mxn = (3, 3)

Similarly, the matrix below is of

import numpy as np  
  
C = np.array([  
 [1, 2], [4, 3], [-1, -5], [3, 3],  
])  
  
print('===============================')  
print('matrix C =\n', C)  
print()  
print('mxn =', np.shape(C))  
print('===============================')

===============================  
matrix C =  
 [[ 1 2]  
 [ 4 3]  
 [-1 -5]  
 [ 3 3]]  
  
mxn = (4, 2)  
===============================

from sympy import \*  
  
C = Matrix([  
 [1, 2], [4, 3], [-1, -5], [3, 3],  
])  
  
C

Matrix([  
[ 1, 2],  
[ 4, 3],  
[-1, -5],  
[ 3, 3]])

print('mxn =', C.shape)

mxn = (4, 2)

# Quiz

What is the of the matrices below:

(A)

(B)

(C)

(D)

## Matrix notation

Matrix is of order with elements where and represents the index of the row and column it belongs in the matrix.

## Addition and Subtraction of Matrices

Addition and subtraction of two matrices can only be achieved if they are the same order.

Only possible if

and

Example

Then Check the order of and compare to order ? Since they are both equal the addition and subtraction is possible.

import numpy as np  
  
A = np.array([[1, -3], [4, -2]])  
B = np.array([[2, -3], [4.1, -4.5]])  
  
sum = A + B  
dif = A - B  
  
print('============================')  
  
print('sum = \n', sum)  
print()  
print('diference = \n', dif)  
  
print('============================')

============================  
sum =   
 [[ 3. -6. ]  
 [ 8.1 -6.5]]  
  
diference =   
 [[-1. 0. ]  
 [-0.1 2.5]]  
============================

from sympy import \*  
  
A = Matrix([[1, -3], [4, -2]])  
B = Matrix([[2, -3], [4.1, -4.5]])  
  
sum = A + B  
dif = A - B

sum

Matrix([  
[ 3, -6],  
[8.1, -6.5]])

dif

Matrix([  
[ -1, 0],  
[-0.0999999999999996, 2.5]])

## Multiplication of Matrices

### Scalar Multiplication

import numpy as np  
  
A = np.array(  
 [  
 [1, -3],  
 [4, -2]  
 ])  
  
prod = 4\*A  
  
print('============================')  
print('A= ')  
print(A)  
print('4\*A =\n', prod)  
print('============================')

============================  
A=   
[[ 1 -3]  
 [ 4 -2]]  
4\*A =  
 [[ 4 -12]  
 [ 16 -8]]  
============================

from sympy import \*  
  
A = Matrix(  
 [  
 [1, -3],  
 [4, -2]  
 ])  
  
prod = 4\*A  
  
print('4\*A =')  
  
prod

4\*A =

Matrix([  
[ 4, -12],  
[16, -8]])

## Two matrices multiplication

To multiply any two matrices the number of columns in the first matrix must be equal to the number of rows of the second matrix. The product matrix will have number of rows of the first matrix and number of columns of the second matrix

Is possible if and only if

# Example

$$
A \times C \\
(2 \times 2)\ \times (4 \times 2)
$$

$$ 2\times(2\neq4)\times2

$$

$$

(2\neq4)∶ A\times C \textrm{ is not possible.} $$

import numpy as np  
  
C = np.array(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = np.array(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
P = A @ C

---------------------------------------------------------------------------  
ValueError Traceback (most recent call last)  
~\AppData\Local\Temp/ipykernel\_3984/1568524905.py in <module>  
 17 )  
 18   
---> 19 P = A @ C  
  
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 4 is different from 2)

from sympy import \*  
  
C = Matrix(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = Matrix(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
P = A \* C  
  
P

---------------------------------------------------------------------------  
ShapeError Traceback (most recent call last)  
~\AppData\Local\Temp/ipykernel\_9168/922994730.py in <module>  
 17 )  
 18   
---> 19 P = A \* C  
 20   
 21 P  
  
C:\sdk\Python310\lib\site-packages\sympy\core\decorators.py in binary\_op\_wrapper(self, other)  
 104 if f is not None:  
 105 return f(self)  
--> 106 return func(self, other)  
 107 return binary\_op\_wrapper  
 108 return priority\_decorator  
  
C:\sdk\Python310\lib\site-packages\sympy\matrices\common.py in \_\_mul\_\_(self, other)  
 2772 """  
 2773   
-> 2774 return self.multiply(other)  
 2775   
 2776 def multiply(self, other, dotprodsimp=None):  
  
C:\sdk\Python310\lib\site-packages\sympy\matrices\common.py in multiply(self, other, dotprodsimp)  
 2794 getattr(other, 'is\_MatrixLike', True))):  
 2795 if self.shape[1] != other.shape[0]:  
-> 2796 raise ShapeError("Matrix size mismatch: %s \* %s." % (  
 2797 self.shape, other.shape))  
 2798   
  
ShapeError: Matrix size mismatch: (2, 2) \* (4, 2).

Interchanging

$$
C\times A \\
(4\times2)\ \times(2\times2) \\
4\times(2=2)\times2 \\
$$

$$
(2=4)∶\ C\times A \textrm{ is possible.} \\
C\times A\equiv4\times2
$$

Let’s carryout the multiplication.

import numpy as np  
  
C = np.array(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = np.array(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
P = C @ A  
  
print('=========================')  
print('P=\n', P)  
print('=========================')

=========================  
P=  
 [[ 9 -7]  
 [ 16 -18]  
 [-21 13]  
 [ 15 -15]]  
=========================

from sympy import \*  
  
C = Matrix(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = Matrix(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
P = C \* A  
  
print('P=')  
  
P

P=

Matrix([  
[ 9, -7],  
[ 16, -18],  
[-21, 13],  
[ 15, -15]])

## Transpose of matrix

Interchanging the rows and columns of a matrix is called transpose.

import numpy as np  
  
C = np.array(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = np.array(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
Atrp = np.transpose(A)  
  
print('=========================')  
print('A=\n', A)  
print('A^T=\n', Atrp)  
print()  
print('C=\n', C)  
print('C^T=\n', np.transpose(C))  
print('=========================')

=========================  
A=  
 [[ 1 -3]  
 [ 4 -2]]  
A^T=  
 [[ 1 4]  
 [-3 -2]]  
  
C=  
 [[ 1 2]  
 [ 4 3]  
 [-1 -5]  
 [ 3 3]]  
C^T=  
 [[ 1 4 -1 3]  
 [ 2 3 -5 3]]  
=========================

from sympy import \*  
  
C = Matrix(  
 [  
 [1, 2],  
 [4, 3],  
 [-1, -5],  
 [3, 3],  
 ]  
)  
  
A = Matrix(  
 [  
 [1, -3],  
 [4, -2]  
 ]  
)  
  
A\_T = transpose(A)  
C\_T = transpose(C)

print('A=\n')  
A

A=

Matrix([  
[1, -3],  
[4, -2]])

print('A\_T=\n')  
A\_T

A\_T=

Matrix([  
[ 1, 4],  
[-3, -2]])

print('C=\n')  
C

C=

Matrix([  
[ 1, 2],  
[ 4, 3],  
[-1, -5],  
[ 3, 3]])

print('C\_T=\n')  
C\_T

C\_T=

Matrix([  
[1, 4, -1, 3],  
[2, 3, -5, 3]])

# Exercise

import numpy as np  
  
A = np.array(  
 [  
 [1, -1, -1],  
 [0, -2, 3],  
 [2, 1, 5],  
 ]  
)  
  
B = np.array(  
 [  
 [1, 1, 1],  
 [1, 1, 1],  
 ]  
)  
  
try:  
 print('A\*B=\n', A \* B)  
except ValueError:  
 print('Error multiplying A\*B')  
  
  
try:  
 print('B\*A=\n', B @ A)  
except ValueError:  
 print('Error multiplying B\*A')  
  
try:  
 print('A\*B^T=\n', A @ np.transpose(B))  
except ValueError:  
 print('Error multiplying A\*B^T')  
  
try:  
 print('B\*A^T=\n', B @ np.transpose(A))  
except ValueError:  
 print('Error multiplying B\*A^T=')

Error multiplying A\*B  
B\*A=  
 [[ 3 -2 7]  
 [ 3 -2 7]]  
A\*B^T=  
 [[-1 -1]  
 [ 1 1]  
 [ 8 8]]  
B\*A^T=  
 [[-1 1 8]  
 [-1 1 8]]

from sympy import \*  
  
A = Matrix(  
 [  
 [1, -1, -1],  
 [0, -2, 3],  
 [2, 1, 5],  
 ]  
)  
  
B = Matrix(  
 [  
 [1, 1, 1],  
 [1, 1, 1],  
 ]  
)

try:  
 print('A\*B=\n')  
 P = None  
 P = A \* B  
except Exception as ex:  
 print('Error multiplying A\*B')  
 print('error: ', ex.args[0])  
  
P

A\*B=  
  
Error multiplying A\*B  
error: Matrix size mismatch: (3, 3) \* (2, 3).

print('B\*A=\n')  
  
try:  
 P = None  
 P = B \* A  
except Exception as ex:  
 print('error: ', ex.args[0])  
  
P

B\*A=

Matrix([  
[3, -2, 7],  
[3, -2, 7]])

print('A\*B^T=\n')  
  
try:  
 P = None  
 P = A \* transpose(B)  
except Exception as ex:  
 print('error: ', ex.args[0])  
  
P

A\*B^T=

Matrix([  
[-1, -1],  
[ 1, 1],  
[ 8, 8]])

print('B\*A^T=\n')  
  
try:  
 P = None  
 P = B \* transpose(A)  
except Exception as ex:  
 print('error: ', ex.args[0])  
  
P

B\*A^T=

Matrix([  
[-1, 1, 8],  
[-1, 1, 8]])

## Square matrix

Square matrix is a matrix of order or . It contains same number of rows and columns.

import numpy as np  
  
A = np.array(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
Atrp = np.transpose(A)  
  
print('=========================')  
print('A=\n', A)  
print('\nmxn= ', np.shape(A))  
print()  
print('A^T=\n', Atrp)  
print('\nmxn= ', np.shape(Atrp))  
print('=========================')

=========================  
A=  
 [[ 3 -3 -4]  
 [ 1 -1 -2]  
 [ 2 -1 4]]  
  
mxn= (3, 3)  
  
A^T=  
 [[ 3 1 2]  
 [-3 -1 -1]  
 [-4 -2 4]]  
  
mxn= (3, 3)  
=========================

from sympy import \*  
  
A = Matrix(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
A\_T = transpose(A)  
  
print('A=\n')  
  
A

A=

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

print('mxn= ')  
A.shape

mxn=

(3, 3)

print('A^T=')  
A\_T

A^T=

Matrix([  
[ 3, 1, 2],  
[-3, -1, -1],  
[-4, -2, 4]])

print('mxn= ')  
A\_T.shape

mxn=

(3, 3)

### Symmetry matrix

A square matrix is symmetrical if , meaning .

import numpy as np  
  
S = np.array(  
 [  
 [1, 2, 3],  
 [2, 4, 5],  
 [3, 5, 6],  
 ]  
)  
  
S\_T = np.transpose(S)  
print('=========================')  
print('S=\n', S)  
print('\nmxn = ', np.shape(S))  
print()  
print('S^T=\n', S\_T)  
print('\nmxn = ', np.shape(S\_T))  
print()  
print('S == S^T')  
print(S == S\_T)  
print('=========================')

=========================  
S=  
 [[1 2 3]  
 [2 4 5]  
 [3 5 6]]  
  
mxn = (3, 3)  
  
S^T=  
 [[1 2 3]  
 [2 4 5]  
 [3 5 6]]  
  
mxn = (3, 3)  
  
S == S^T  
[[ True True True]  
 [ True True True]  
 [ True True True]]  
=========================

A = np.array(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
A\_T = np.transpose(A)  
  
print('=========================')  
  
print('A=\n', A)  
print('\nmxn = ', np.shape(A))  
print()  
print('A^T=\n', A\_T)  
print('\nmxn = ', np.shape(A\_T))  
print()  
print('A == A^T')  
print(A == A\_T)  
print('=========================')

=========================  
A=  
 [[ 3 -3 -4]  
 [ 1 -1 -2]  
 [ 2 -1 4]]  
  
mxn = (3, 3)  
  
A^T=  
 [[ 3 1 2]  
 [-3 -1 -1]  
 [-4 -2 4]]  
  
mxn = (3, 3)  
  
A == A^T  
[[ True False False]  
 [False True False]  
 [False False True]]  
=========================

from sympy import \*  
  
S = Matrix(  
 [  
 [1, 2, 3],  
 [2, 4, 5],  
 [3, 5, 6],  
 ]  
)  
  
S\_T = transpose(S)  
  
print('S=')  
  
S

S=

Matrix([  
[1, 2, 3],  
[2, 4, 5],  
[3, 5, 6]])

print('mxn = ', S.shape)

mxn = (3, 3)

print('S^T=')  
S\_T

S^T=

Matrix([  
[1, 2, 3],  
[2, 4, 5],  
[3, 5, 6]])

print('mxn =', S\_T.shape)

mxn = (3, 3)

print('(S == S^T) =', S == S\_T)

(S == S^T) = True

# Example

A = Matrix(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
A\_T = transpose(A)  
  
print('A=')  
A

A=

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

print('mxn = ', A.shape)

mxn = (3, 3)

print('A^T=')  
A\_T

A^T=

Matrix([  
[ 3, 1, 2],  
[-3, -1, -1],  
[-4, -2, 4]])

print('mxn = ', A\_T.shape)

mxn = (3, 3)

print('(A == A^T) =', A == A\_T)

(A == A^T) = False

### Triangular matrix

A triangular matrix is a square matrix with all elements below the leading diagonal equal to zero.

import numpy as np  
  
A = np.array(  
 [  
 [1, 2, 3],  
 [0, 4, 5],  
 [0, 0, 6],  
 ]  
)  
  
print('A=\n', A)

A=  
 [[1 2 3]  
 [0 4 5]  
 [0 0 6]]

from sympy import \*  
  
A = Matrix(  
 [  
 [1, 2, 3],  
 [0, 4, 5],  
 [0, 0, 6],  
 ]  
)  
  
print('A=')  
A

A=

Matrix([  
[1, 2, 3],  
[0, 4, 5],  
[0, 0, 6]])

### Diagonal matrix

A diagonal matrix is a square matrix with all elements zero except the leading diagonal.

import numpy as np  
  
A = np.array(  
 [  
 [1, 0, 0],  
 [0, 4, 0],  
 [0, 0, 6],  
 ]  
)  
  
print('A=\n', A)

A=  
 [[1 0 0]  
 [0 4 0]  
 [0 0 6]]

from sympy import \*  
  
A = Matrix(  
 [  
 [1, 0, 0],  
 [0, 4, 0],  
 [0, 0, 6],  
 ]  
)  
  
print('A=')  
A

A=

Matrix([  
[1, 0, 0],  
[0, 4, 0],  
[0, 0, 6]])

### Unit matrix

A Unit matrix is a special diagonal matrix with the leading diagonal element values all unity or ones. The unit matrix is denoted by I.

import numpy as np  
  
I = np.array(  
 [  
 [1, 0, 0],  
 [0, 1, 0],  
 [0, 0, 1],  
 ]  
)  
  
print('I=\n', I)

I=  
 [[1 0 0]  
 [0 1 0]  
 [0 0 1]]

from sympy import \*  
  
I = Matrix(  
 [  
 [1, 0, 0],  
 [0, 1, 0],  
 [0, 0, 1],  
 ]  
)  
  
print('I=')  
I

I=

Matrix([  
[1, 0, 0],  
[0, 1, 0],  
[0, 0, 1]])

import numpy as np  
  
I = np.eye(3, dtype=int)  
  
print('I=\n', I)

I=  
 [[1 0 0]  
 [0 1 0]  
 [0 0 1]]

#### Useful property

A useful property

import numpy as np  
  
A = np.array(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
I = np.identity(3)  
  
print('=========================')  
print('A =\n', A)  
print()  
print('I =\n', I)  
print()  
print('A \* I=\n', A @ I)  
print()  
print('I \* A=\n', I @ A)  
print('=========================')

=========================  
A =  
 [[ 3 -3 -4]  
 [ 1 -1 -2]  
 [ 2 -1 4]]  
  
I =  
 [[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]  
  
A \* I=  
 [[ 3. -3. -4.]  
 [ 1. -1. -2.]  
 [ 2. -1. 4.]]  
  
I \* A=  
 [[ 3. -3. -4.]  
 [ 1. -1. -2.]  
 [ 2. -1. 4.]]  
=========================

from sympy import \*  
  
A = Matrix(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
print('A =')  
A

A =

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

I = Identity(3)  
print('I =')  
I

I =

I

print('A \* I=')  
A \* I

A \* I=

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

print('I \* A=')  
I \* A

I \* A=

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

## Null matrix

A null matrix is one whose elements are all zero.

import numpy as np  
  
N = np.array(  
 [  
 [0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0],  
 ]  
)  
  
print('=========================')  
print('N =\n', N)  
print('=========================')

=========================  
N =  
 [[0 0 0]  
 [0 0 0]  
 [0 0 0]]  
=========================

import numpy as np  
  
N = np.zeros((3, 3))  
  
print('=========================')  
print('N =\n', N)  
print('=========================')

=========================  
N =  
 [[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]  
=========================

from sympy import \*  
  
N = zeros(3)  
  
print('N =')  
N

N =

Matrix([  
[0, 0, 0],  
[0, 0, 0],  
[0, 0, 0]])

#### Useful property

A useful property

we cannot say or

import numpy as np  
  
A = np.array(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
N = np.zeros((3, 3))  
  
print('=========================')  
print('A =\n', A)  
print()  
print('N =\n', N)  
print()  
print('A \* N=\n', A @ N)  
print()  
print('N \* A=\n', N @ A)  
print('=========================')

=========================  
A =  
 [[ 3 -3 -4]  
 [ 1 -1 -2]  
 [ 2 -1 4]]  
  
N =  
 [[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]  
  
A \* N=  
 [[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]  
  
N \* A=  
 [[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]  
=========================

from sympy import \*  
  
A = Matrix(  
 [  
 [3, -3, -4],  
 [1, -1, -2],  
 [2, -1, 4],  
 ]  
)  
  
N = zeros(3)  
  
print('=========================')  
print('A =\n', A)  
print()  
print('N =\n', N)  
print()  
print('A \* N=\n', A @ N)  
print()  
print('N \* A=\n', N @ A)  
print('=========================')

=========================  
A =  
 Matrix([[3, -3, -4], [1, -1, -2], [2, -1, 4]])  
  
N =  
 Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]])  
  
A \* N=  
 Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]])  
  
N \* A=  
 Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]])  
=========================

print('A =')  
A

A =

Matrix([  
[3, -3, -4],  
[1, -1, -2],  
[2, -1, 4]])

print('N =')  
N

N =

Matrix([  
[0, 0, 0],  
[0, 0, 0],  
[0, 0, 0]])

print('A \* N=')  
A \* N

A \* N=

Matrix([  
[0, 0, 0],  
[0, 0, 0],  
[0, 0, 0]])

print('N \* A=')  
N \* A

N \* A=

Matrix([  
[0, 0, 0],  
[0, 0, 0],  
[0, 0, 0]])

# Determinants

The arrangement of numbers in an equal rows and columns bounded by a straight bar is called determinant.

## Determinants of the second order

The symbol above is determinant of a (two by two matrix) with element on row 1 and column 1 and is element on row 1 and column 2. It means each element is identified by row and column number subscript .

By definition,

To find determinant of

we must multiply the elements diagonally to form the product terms in the expansion:

$$ \textrm{we multiply } \left| \begin{matrix} a\_{11} & \ & a\_{22}\ \end{matrix} \right|

\textrm{ then subtract the product }

\left| \begin{matrix} & a\_{12}\ a\_{21} & \ \end{matrix} \right| $$

Note:  
 before i.e column 1 before 2.

Or

import numpy as np  
  
A = np.array(  
 [  
 [1, -3],  
 [4, -2],  
 ]  
)  
  
print("A = \n", A)  
  
det\_A = np.linalg.det(A)  
  
print("|A| = ", det\_A)

A =   
 [[ 1 -3]  
 [ 4 -2]]  
|A| = 10.000000000000002

from sympy import \*  
x, y = symbols('x y')  
  
A = Matrix(  
 [  
 [1.0, -3],  
 [4, -2]  
 ]  
)  
print("A=")  
A

A=

Matrix([  
[1.0, -3],  
[ 4, -2]])

print('|A| =', A.det())

|A| = 10.0000000000000

# More example in class

Generate question from the students/participants

## Determinants of the third order

A determinant of the third order contains 3 rows and 3 columns.

In order to find the determinant of a higher order the matrix minor must be developed until second order is reached. Developing each element in the first row minor:

For

For

For

Summing all together:

# Example

Try

Answer

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4],  
])  
  
detA = np.linalg.det(A)  
  
print("A = \n", A)  
  
print("|A| = ", detA)

A =   
 [[ 3 -3 -4]  
 [ 1 -1 -1]  
 [ 2 -1 4]]  
|A| = -0.9999999999999998

from sympy import \*  
  
A = Matrix([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4],  
])  
  
detA = A.det()  
  
print("A = \n")  
A

A =

Matrix([  
[3, -3, -4],  
[1, -1, -1],  
[2, -1, 4]])

print("|A| = ", detA)

|A| = -1

from sympy import \*  
  
A = Matrix([  
 [3.0, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4],  
])  
  
detA = A.det()  
  
print("A = \n")  
A

A =

Matrix([  
[3.0, -3, -4],  
[ 1, -1, -1],  
[ 2, -1, 4]])

print("|A| = ", detA)

|A| = -1.00000000000000

# Solution of system of linear equations by determinants

A system of linear equations of the second degree of two unknowns can be written below:

$$
a\_{11}x+a\_{12}y=b\_1 \\
a\_{21}x+a\_{22}y=b\_2
$$

and can further be written in matrices form as

$$

A . x=b

Solving Simultaneous Equation

$$
2x + 3y = 10\\
4x + y = 1
$$

By elimination and substitution $$

x=-\frac{7}{10},\quad y=\frac{38}{10}

## Determinant Equation Method

### Secondary Degree

$$
2x + 3y = 10\\
4x + y = 1
$$

could be written as: $$

2x+3y-10=0 \quad \left(a\right) \ 4x+y-1=0 \quad \left(b\right)

a*{11}x+a*{12}y+b*1=0 \quad \left(a\right) \ a*{21}x+a\_{22}y+b\_2=0 \quad \left(b\right) $$

The solution

where

Solving Equation

$$
2x+3y-10=0 \quad \left(a\right) \\
4x+y-1=0 \quad \left(b\right)
$$

$$
x=\frac{∆\_1}{∆\_0}=\frac{7}{-10} =\left. -\frac{7}{10} \right. \\
y=\frac{-∆\_2}{∆\_0}=\left.-\frac{38}{-10}\right.=\frac{38}{10}
$$

### The solution for higher degree system of linear equations are:

### For third degree:

Note: The positive and negative signs alternate has the degree increases.

### Third Degree

Solve

$$
3x -3y - 4z = 1 \\
x - y - z = 2 \\
2x-y+ 4z =3
$$

Rearrange

$$
3x -3y -4z -1= 0 \\
x - y - z -2 =0 \\
2x - y + 4z -3 = 0
$$

Extract the matrix of coefficients:

$$

∆\_1 = \left|\begin{matrix}

-3 & -4 & -1 \ -1 & -1 & -2 \ -1 & 4 & -3 \end{matrix} \right| =

$$

From Equation 2.13

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
b = np.array([1, 2, 3])  
  
M = np.zeros((3,4))  
  
M[:,:3] = A[:]  
M[:,3] = -np.transpose(b)  
  
print('M =\n', M)  
  
M0 = M[:,:3]  
print('M0 =\n', M0)  
  
M1 = M[:,1:]  
print('M1 =\n', M1)  
  
M2 = M[:,[0,2,3]]  
print('M2 =\n', M2)  
  
M3 = M[:,[0,1,3]]  
print('M3 =\n', M3)  
  
D0 = np.linalg.det(M0)  
D1 = np.linalg.det(M1)  
D2 = np.linalg.det(M2)  
D3 = np.linalg.det(M3)  
  
print('D0 = ', D0)  
print('D1 = ', D1)  
print('D2 = ', D2)  
print('D3 = ', D3)  
  
x = -D1/D0  
y = D2/D0  
z = -D3/D0  
  
print('x = ',x)  
print('y = ',y)  
print('z = ',z)

M =  
 [[ 3. -3. -4. -1.]  
 [ 1. -1. -1. -2.]  
 [ 2. -1. 4. -3.]]  
M0 =  
 [[ 3. -3. -4.]  
 [ 1. -1. -1.]  
 [ 2. -1. 4.]]  
M1 =  
 [[-3. -4. -1.]  
 [-1. -1. -2.]  
 [-1. 4. -3.]]  
M2 =  
 [[ 3. -4. -1.]  
 [ 1. -1. -2.]  
 [ 2. 4. -3.]]  
M3 =  
 [[ 3. -3. -1.]  
 [ 1. -1. -2.]  
 [ 2. -1. -3.]]  
D0 = -0.9999999999999998  
D1 = -24.000000000000014  
D2 = 31.000000000000014  
D3 = 5.000000000000001  
x = -24.00000000000002  
y = -31.00000000000002  
z = 5.000000000000002

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
b = np.array([1, 2, 3])  
  
[x,y,z] = np.linalg.solve(A, b)  
  
x,y,z

(-24.00000000000001, -31.00000000000001, 5.000000000000002)

from sympy import \*   
  
M = Matrix([  
 [3, -3, -4, 1],  
 [1, -1, -1, 2],  
 [2, -1, 4, 3]  
])  
  
x = linsolve(M)  
  
x

{(-24, -31, 5)}

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
b = np.array([1, 2, 3])  
  
M = np.zeros((3,4))  
  
M[:,:3] = A[:]  
M[:,3] = -np.transpose(b)  
  
print('M =\n', M)  
  
m,n = M.shape  
  
mk=[]  
k = []  
for i in range(n):  
 k = []  
 for j in range(n):  
 if(i == j):  
 continue  
 k.append(j)  
 mk.append(k)  
  
  
MD = []  
MD.append(M[:,mk[m]])  
  
for i in range(1,n):  
 MD.insert(i, M[:,mk[i-1]])  
   
  
D = []  
X = []  
for i in range(n):  
 print('M[',i,'] =\n', MD[i])  
 D.append(np.linalg.det(MD[i]))  
 print('D[',i,'] =\n', D[i])  
 x = ((-1)\*\*i)\*(D[i]/D[0])  
 X.append(x)  
   
print('X=', X[1:])

M =  
 [[ 3. -3. -4. -1.]  
 [ 1. -1. -1. -2.]  
 [ 2. -1. 4. -3.]]  
M[ 0 ] =  
 [[ 3. -3. -4.]  
 [ 1. -1. -1.]  
 [ 2. -1. 4.]]  
D[ 0 ] =  
 -0.9999999999999998  
M[ 1 ] =  
 [[-3. -4. -1.]  
 [-1. -1. -2.]  
 [-1. 4. -3.]]  
D[ 1 ] =  
 -24.000000000000014  
M[ 2 ] =  
 [[ 3. -4. -1.]  
 [ 1. -1. -2.]  
 [ 2. 4. -3.]]  
D[ 2 ] =  
 31.000000000000014  
M[ 3 ] =  
 [[ 3. -3. -1.]  
 [ 1. -1. -2.]  
 [ 2. -1. -3.]]  
D[ 3 ] =  
 5.000000000000001  
X= [-24.00000000000002, -31.00000000000002, 5.000000000000002]

## Cramers Rule

### For second degree

For a system of two linear equations in two unknowns (x, y) given below:

$$
a\_{11}x + a\_{12}y = b\_1 \\
a\_{21}x + a\_{22}y = b\_2
$$

Cramer’s rule states that:

The denominator is the determinant of the coefficients of x and y. The numerator for x is the determinant of coefficient matrix when coefficient of x is replaced by b. Similarly, the numerator for y is the determinant of coefficient matrix when coefficient of y is replaced by b.

### Third degree

$$
a\_{11}x+a\_{12}y+a\_{13}z=b\_1 \\
a\_{21}x+a\_{22}y+a\_{23}z=b\_2 \\
a\_{31}x+a\_{32}y+a\_{33}z=b\_3
$$

The Cramer’s rule can be generalized:

Example 3.3

Solve

$$
3x\ -3y-4z=1 \\
x-y- z=2 \\
2x-y+ 4y=3
$$

Note: For both the determinant Equation Method and Cramer’s rule determinant, $ ∆\_0 or D $ , the coefficient of the unknown must not equal zero.

import numpy as np  
  
A = np.array([  
 [3, -3, -4],  
 [1, -1, -1],  
 [2, -1, 4]  
])  
  
b = np.array([1, 2, 3])  
  
M = A[:]  
print('M0 =\n', M)  
  
M1 = M.copy()  
M1[:,0] = np.transpose(b)  
print('M1 =\n', M1)  
  
M2 = M.copy()  
M2[:,1] = np.transpose(b)  
print('M2 =\n', M2)  
  
M3 = M.copy()  
M3[:,2] = np.transpose(b)  
print('M3 =\n', M3)  
  
D0 = np.linalg.det(M0)  
D1 = np.linalg.det(M1)  
D2 = np.linalg.det(M2)  
D3 = np.linalg.det(M3)  
  
print('D0 = ', D0)  
print('D1 = ', D1)  
print('D2 = ', D2)  
print('D3 = ', D3)  
  
x = D1/D0  
y = D2/D0  
z = D3/D0  
  
print('x = ',x)  
print('y = ',y)  
print('z = ',z)

M0 =  
 [[ 3 -3 -4]  
 [ 1 -1 -1]  
 [ 2 -1 4]]  
M1 =  
 [[ 1 -3 -4]  
 [ 2 -1 -1]  
 [ 3 -1 4]]  
M2 =  
 [[ 3 1 -4]  
 [ 1 2 -1]  
 [ 2 3 4]]  
M3 =  
 [[ 3 -3 1]  
 [ 1 -1 2]  
 [ 2 -1 3]]  
D0 = -0.9999999999999998  
D1 = 24.000000000000014  
D2 = 31.0  
D3 = -5.000000000000001  
x = -24.00000000000002  
y = -31.000000000000007  
z = 5.000000000000002

# Linear dependence and independence

A matrix is said to be linearly dependent if the determinant is zero and linearly independent if the determinant is not zero.

Linear dependence

And Linear independence

Check if the matrices below are linear dependence or independence:

What is your answer?

import numpy as np  
  
A = np.array([  
 [2,3,4],  
 [6,1,0],  
 [2,3,4],  
])  
  
np.linalg.det(A)

0.0

import numpy as np  
  
B = np.array([  
 [3,-3,-4],  
 [1,-1,-1],  
 [2,-1,4],  
])  
  
np.linalg.det(B)

-0.9999999999999998

# Rank of a matrix

The rank of an $ m x n $ matrix $ A $ is the order of the largest square, linear independence sub-matrix. In order to get a matrix rank you must get a non-zero determinant matrix of the original m x n matrix or a sub-matrix.

Check the rank of matrix below:

$$ B=\left|\begin{matrix}3&-3&-4\1&-1&-1\2&-1&4\\end{matrix}\right|=-1

$$

Since the determinant is non-zero the $ rank = 3 $.

import numpy as np  
  
B = np.array([  
 [3,-3,-4],  
 [1,-1,-1],  
 [2,-1,4],  
])  
  
print('|B| = ', np.linalg.det(B))  
  
r = np.linalg.matrix\_rank(B)  
print('rank = ', r)

|B| = -0.9999999999999998  
rank = 3

Similarly, check for matrix .

The determinant is zero, we need to check for the determinant of its sub-matrix:

Since the determinant of the sub-matrix A1 is non-zero, we can conclude that matrix is of .

import numpy as np  
  
A = np.array([  
 [2,3,4],  
 [6,1,0],  
 [2,3,4],  
])  
  
print('|A| = ', np.linalg.det(A))  
  
r = np.linalg.matrix\_rank(A)  
print('rank = ', r)

|A| = 0.0  
rank = 2

# General system of linear equations

A system linear of equations below with three unknown :

$$
a\_{11}x\_1+a\_{12}x\_2+a\_{13}x\_3=b\_1 \\
a\_{21}x\_1+a\_{22}x\_2+a\_{23}x\_3=b\_2 \\
a\_{31}x\_1+a\_{32}x\_2+a\_{33}x\_3=b\_3
$$

can be represented using matrix notation:

Similarly,

$$
a\_{11}x\_1+a\_{12}x\_2+a\_{13}x\_3=b\_1 \\
a\_{21}x\_1+a\_{22}x\_2+a\_{23}x\_3=b\_2
$$

Generally,

where

If we multiply by the inverse of .

and

where is the transpose of the co-factor of matrix

# Existence and properties of solution

## Existence of solution

Forming an augmented matrix .

For a solution to exist for a set of linear equations the following must be true:

1. Determinant of A must not be equal to zero, i.e .
2. A unique solution exists:
3. An infinite number of solutions exist:
4. No solution exists: rank A<rank A\_b

## Other properties of the solution

# Gaussian elimination.

## Algorithm of Gaussian Elimination

Gaussian elimination method converts the combine matrix of A and b, [A|b] to triangular matrix. Then proceed with backward substitution to obtain x.

For a system of linear equations

$$
a\_{11}x\_1+a\_{12}x\_2+a\_{13}x\_3=b\_1 \\
a\_{21}x\_1+a\_{22}x\_2+a\_{23}x\_3=b\_2 \\
a\_{31}x\_1+a\_{32}x\_2+a\_{33}x\_3=b\_3
$$

Step 1: Write in matrix form.

Step 2: Arrange matrix together to form augmented matrix .

Step 3: Using matrix row equivalent operation reduce to triangular matrix .

Step 4: Back substitution

# Example

Solve the set of equations.

Step 1: Write in matrix form.

Step 2: Arrange matrix together augmented matrix .

Step 3: Using matrix row equivalent operation reduce to triangular matrix .

Note:

If interchange row with a none-zero pivot column element row.

$$ R\_i =R\_i-\left(R\_p\times \frac{a\_{ip}}{a\_{pp}}\right)

$$

Reduce

when

$$ R\_2 =R\_2-\left(R\_1\times \frac{a\_{21}}{a\_{11}}\right)

$$

$$ R\_3 =R\_3-\left(R\_1\times \frac{a\_{31}}{a\_{11}}\right)

$$

$$

R\_3=\left[\begin{matrix}2&-1&\begin{matrix}4&|&3\\end{matrix}\\end{matrix}\right]-\left(\left[\begin{matrix}3&-3&\begin{matrix}-4&|&1\\end{matrix}\\end{matrix}\right]\times\frac{2}{3}\right) $$

Reduce

Interchange row 2 and 3

Step 4: Back substitution

## Determinant from Gaussian Elimination

where is the number of times rows were interchanged.

From the example above and

# Matrix inverse by elementary matrices, adjoint, and partitioning methods.

## Inverse of a square matrix

Recall the solution to system of linear equation:

To solve Eq (9.3) the inverse of matrix , must be evaluated.

where is the adjoint of matrix , and Adjoint is the transpose of the co-factor of matrix .

## Cofactors

Cofactor of a square matrix is obtained from the determinant of its elements.

where each element of the cofactor is formed from the determinant of its sub-matrix multiplied by element position sign. The element positional sign alternate between plus + and minus -.

Evaluate the cofactor of the matrix below:

For element ;

For element ;

For element ;

For element ;

For element ;

For element ;

For element ;

For element ;

For element ;

The cofactor form is:

## Adjoint

Adjoint of matrix is the transpose of the cofactor of .

From

Therefore,

To solve the system linear of equation

$$

\left[\begin{matrix}x\y\z\\end{matrix}\right]=\left[\begin{matrix}5&-16&1\6&-20&1\-1&3&0\\end{matrix}\right]\left[\begin{matrix}1\2\3\\end{matrix}\right] $$

# Eigenvalues and Eigenvectors.

## Eigenvalues

Eigenvalues are characteristic scalar value of a square matrix that satisfy the .

In other to find the eigenvalues, we develop the characteristic equation and solve the characteristic polynomial of matrix .

To prevent the trivial solution . Therefore we have the characteristic equation .

The solution of gives the eigenvalues .

Example 10.1

Find the eigenvalues of the matrix.

The solutions to the characteristic polynomial in

$$

\lambda=-2 or 5 $$

## Eigenvectors

Each eigenvalues obtained has a corresponding eigenvector or eigenline associated with it.

Example 10.2

When matrix is:

For

The eigenvector for

For

$$

\begin{matrix}\begin{matrix}2x\_1&+\\end{matrix}&\begin{matrix}3x\_2&=\\end{matrix}  \\begin{matrix}4x\_1&+\\end{matrix}&\begin{matrix}x\_2&=\\end{matrix}\\end{matrix}\begin{matrix}{5x}\_1\{5x}\_2\\end{matrix} $$

The eigenvector for

# Exercises

### Exercise 3

Find the determinant of A:

import numpy as np  
  
A = np.array(  
 [  
 [1, -1, -1, 0],  
 [0, -2, 3, 2],  
 [2, 1, 5, 1],  
 [-2, 1, -2, 9],  
 ]  
)  
  
np.linalg.det(A)

-227.99999999999986

from sympy import \*  
  
A = Matrix(  
 [  
 [1, -1, -1, 0],  
 [0, -2, 3, 2],  
 [2, 1, 5, 1],  
 [-2, 1, -2, 9],  
 ]  
)  
  
A.det()

-228

from sympy import \*  
  
A = Matrix(  
 [  
 [1.0, -1, -1, 0],  
 [0, -2, 3, 2],  
 [2, 1, 5, 1],  
 [-2, 1, -2, 9],  
 ]  
)  
  
A.det()

-228.000000000000

### Exercise 4

$$
A . x = b \\
$$

$$
A= \left(
\begin{matrix}
1 & -1 & -1 & 0\\
0 & -2 & 3 & 2\\
2 & 1 & 5 & 1\\
-2 & 1 & -2 & 9\
\end{matrix}
\right) \\
b= \left(
\begin{matrix}
4\\
4\\
4\\
4\\
\end{matrix}
\right)
$$

import numpy as np  
  
A = np.array(  
 [  
 [1, -1, -1, 0],  
 [0, -2, 3, 2],  
 [2, 1, 5, 1],  
 [-2, 1, -2, 9],  
 ]  
)  
  
b = np.array(  
 [  
 [4],  
 [4],  
 [4],  
 [4],  
 ]  
)  
  
x = np.linalg.solve(A, b)  
  
print('x = \n', x)

x =   
 [[ 2.57894737]  
 [-1.21052632]  
 [-0.21052632]  
 [ 1.10526316]]

from sympy import \*  
  
a, b, c, d = symbols('a b c d')  
  
linsolve(Matrix((  
 [1, -1, -1, 0, 4],  
 [0, -2, 3, 2, 4],  
 [2, 1, 5, 1, 4],  
 [-2, 1, -2, 9, 4])), (a, b, c, d))

{(49/19, -23/19, -4/19, 21/19)}

a = 49/19  
b = -23/19  
c = -4/19  
d = 21/19  
  
a, b, c, d

(2.5789473684210527,  
 -1.2105263157894737,  
 -0.21052631578947367,  
 1.105263157894737)

### Exercise 5

Solve for in the augment matrix .

import numpy as np  
  
A = np.array(  
 [  
 [8, 4, -1],  
 [1, 6, 2],  
 [4, 0, 2],  
 ]  
)  
  
b = np.array(  
 [  
 [3],  
 [3],  
 [3],  
 ]  
)  
  
x = np.linalg.solve(A, b)  
  
print('x = \n', x)

x =   
 [[0.375 ]  
 [0.1875]  
 [0.75 ]]

from sympy import \*  
  
x, y, z = symbols('x y z')  
  
linsolve(Matrix((  
 [8, 4, -1, 3],  
 [1, 6, 2, 3],  
 [4, 0, 2, 3])), (x, y, z))

FiniteSet((3/8, 3/16, 3/4))

x = 3/8  
y = 3/16  
z = 3/4  
  
x, y, z

(0.375, 0.1875, 0.75)

### Exercise 6

Solve for in the augment matrix .

import numpy as np  
from fractions import Fraction  
  
M = np.array(  
 [  
 [3, -1, -1, 4],  
 [4, -2, 3, 2],  
 [2, 1, -5, -1],  
 ], dtype=float  
)  
  
print('M =\n', M)  
(m, n) = M.shape  
print('mxn =\n', m, 'x', n)  
print('|A| = \n', M[:3, :3], '\n =', np.linalg.det(M[:3, :3]))  
  
k = 0  
p = 0  
  
for p in range(0, m-1):  
 for i in range(p+1, m):  
 if M[p, p] == 0:  
 # change row  
 for j in range(i, m):  
 if M[j, p] != 0:  
 k += 1  
 Mp = np.copy(M[p])  
 M[p] = M[j]  
 M[j] = Mp  
  
 print('Mx =\n', M)  
 break  
  
 M[i] = M[i, :] - M[p, :]\*M[i, p]/M[p, p]  
 print('M =\n', M)  
  
detA = 1  
for p in range(m):  
 detA \*= M[p, p]  
  
print('|A| =', detA)

M =  
 [[ 3. -1. -1. 4.]  
 [ 4. -2. 3. 2.]  
 [ 2. 1. -5. -1.]]  
mxn =  
 3 x 4  
|A| =   
 [[ 3. -1. -1.]  
 [ 4. -2. 3.]  
 [ 2. 1. -5.]]   
 = -13.0  
M =  
 [[ 3. -1. -1. 4. ]  
 [ 0. -0.66666667 4.33333333 -3.33333333]  
 [ 2. 1. -5. -1. ]]  
M =  
 [[ 3. -1. -1. 4. ]  
 [ 0. -0.66666667 4.33333333 -3.33333333]  
 [ 0. 1.66666667 -4.33333333 -3.66666667]]  
M =  
 [[ 3. -1. -1. 4. ]  
 [ 0. -0.66666667 4.33333333 -3.33333333]  
 [ 0. 0. 6.5 -12. ]]  
|A| = -12.999999999999995

from sympy import \*  
  
x, y, z = symbols('x y z')  
  
linsolve(  
 Matrix(  
 (  
 [3, -1, -1, 4],  
 [4, -2, 3, 2],  
 [2, 1, -5, -1]  
 )  
 ),  
 (x, y, z)  
)

{(-21/13, -7, -24/13)}

from sympy import \*  
x, y, z = symbols('x y z')  
  
linsolve(Matrix((  
 [1, 1, 1, 1],  
 [1, 1, 2, 3],  
 [1, 0, 2, 3])), (x, y, z))

{(-1, 0, 2)}

# Test 2 Make Up

Consider the system of linear equation below:

$$ \begin{matrix} 3x & - & y & - & 2 & - & z &= & 4 \ 4z & - & 2y & + & 3x & + & y & = & 2 \ 2x & + & y & - & 5y & - & 1 & + & z & + & x & = & -1 \ \end{matrix}

$$

Q1.  
Write the matrices form.   
What is the order of matrix   
What is the order of matrix .  
Determinant of matrix   
Rank of

Q2. Write the augment matrix ,   
What is the order of matrix .  
Solve the system of linear equation using Gaussian Elimination.

import numpy as np  
  
A = np.array(  
 [  
 [3.0, -1., -1.],  
 [3., -1., 4.],  
 [2., -4., 1.],  
 ]  
)  
  
b = np.array(  
 [  
 [6.0],  
 [2.0],  
 [0.0],  
 ]  
)  
  
x = np.linalg.solve(A, b)  
print('x = \n', x)  
  
d = np.linalg.det(A)  
print('det(A) = ', d)

x =   
 [[ 2. ]  
 [ 0.8]  
 [-0.8]]  
det(A) = 50.000000000000014

from sympy import \*  
x, y, z = symbols('x y z')  
  
linsolve(Matrix((  
 [3.0, -1., -1., 6.0],  
 [3., -1., 4., 2.0],  
 [2., -4., 1., 0],  
)), (x, y, z))

FiniteSet((2.0, 0.8, -0.8))