LAB RECORD
2021-2022

Linear Algebra Using Python
MAT551

Name: Pranav Gopalkrishna | Register number: 1940223

TABLE OF CONTENTS

1. BASICS

a) Loops & if-else statements

November 27, 2021

1 AIM

Before moving on to the linear algebraic concepts, we will first recap some basics in Python programming. This includes loops and if-else statements, which is the focus of this record.

2 Grade calculator

Write a python program to generate the grade of a student based on the percentage obtained.

[12]:
```
# Inputting percentage and validating input
while True:
    try:
        percentage = int(input("Enter student's percentage: "))
        if percentage < 0 or percentage > 100:
            percentage = 1/0
        else:
            break
    except:
        print("Invalid input!")

# Percentage to grade conversion
if percentage > 90:
    grade = "A+"
elif percentage > 80:
    grade = "A"
elif percentage > 75:
    grade = "B+"
elif percentage > 70:
    grade = "B"
elif percentage > 65:
    grade = "C+"
elif percentage > 60:
    grade = "C"
elif percentage > 55:
    grade = "D+"
elif percentage > 50:
    grade = "D"
```

```
elif percentage > 40:
    grade = "E"
else:
    grade = "F"

print("Received grade: " + grade)
```

```
Enter student's percentage: 78
Received grade: B+
```

3 Finding the factors of a number

Write a python program to accept a number from the user and predict whether it is a prime or not. If it is not a prime, then display all the factors of the number.

[8]:
```
# Inputting integer
while True:
    try:
        n = int(input("Enter an integer: "))
        break
    except:
        print("Not an integer!")

# Checking if prime, and finding factors if not
i = 2
isPrime = 1
factors = []
# This uses the minimum exit condition to check if n is prime
while i * i <= n:
    if n % i == 0:
        factors.append(i)
        isPrime = 0
    i = i + 1
# Finding the remaining factors if number is non-prime
if isPrime == 0:
    while i <= n/2:
        if n % i == 0:
            factors.append(i)
        i = i + 1
if isPrime:
    print(n, "is prime.")
else:
    print("The factors of", n, "are", factors)
```

```
Enter an integer: 677
677 is prime.
```

4 Identifying the triangle type

Write a python program to check if a triangle is equilateral, isosceles or scalene.

[11]:
```
print("Program to check if your triangle is equilateral, isosceles or scalene")
print("------------------------")
print("Enter x to input lengths of edges")
print("Enter anything else to input interior angles")
option = input()

if option == "x":
    while True:
        try:
            print("Enter lengths of edges...")
            a = float(input("a = "))
            b = float(input("b = "))
            c = float(input("c = "))
            break
        except:
            print("Non-numeric inputs!")
else:
    while True:
        try:
            print("Enter two interior angles in degrees...")
            a = float(input("a = "))
            b = float(input("b = "))
            c = 180 - a - c
            print("c =", c)
            break
        except:
            print("Non-numeric inputs!")
print("Triangle type:", end = " ")
if a == b and b == c:
    print("Equilateral")
elif a == b or b == c:
    print("Isosceles")
else:
    print("Scalene")
```

```
Program to check if your triangle is equilateral, isosceles or scalene
------------------------
Enter x to input lengths of edges
Enter anything else to input interior angles
r
Enter two interior angles in degrees…
a = 45
b = 87
c = 54.0
```

```
Triangle type: Scalene
```

5 Sum of first n natural numbers

Write a python program to find the sum of squares of first n natural numbers.

[17]:
```
while True:
    try:
        n = int(input("Enter upper limit: "))
        if n < 0:
            n = 1/0
        else:
            break
    except:
        print("Input must be a natural number!")
mySum = 0
for i in range(1, n + 1):
    mySum = mySum + i * i
print("The sum of the squares of the first", n, "natural numbers is", mySum)
```

```
Enter upper limit: 87
The sum of the squares of the first 87 natural numbers is 223300
```

6 CONCLUSION

Loops enable us to go through numerous values within a range or a list, and perform the same operations on each one until a condition is reached. This is seen in finding the factors of a number, or in adding the first n natural numbers, where the same set of operations and conditions are applied to values within a range one by one, until the desired conclusion is reached. We have also used loops for ensuring valid inputs, where the loop only breaks if the input is valid.

If-else conditions are the basic way in which we can check if certain conditions are met for one or more entities. This is seen clearly in grade calculation and identifying the triangle type, where we check if the input value or values fulfil the given conditions.

1. BASICS

b) Plotting basics

November 27, 2021

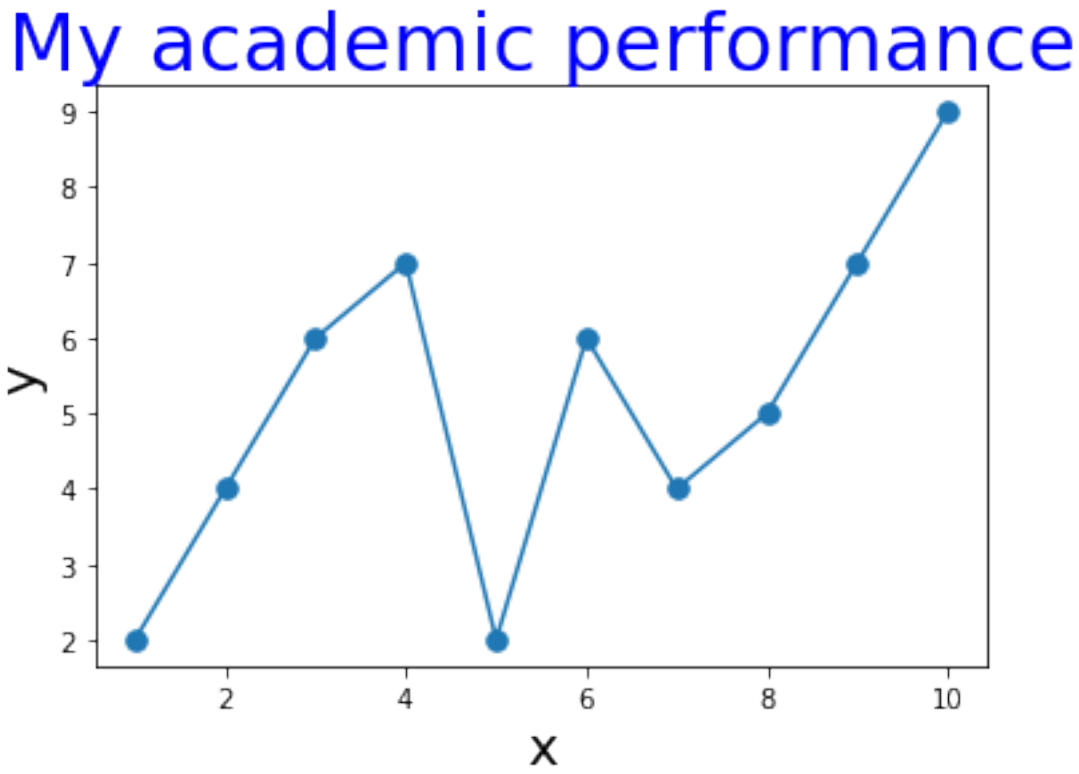
1 AIM

Plotting is one of the most essential tools in visualising data and results of our computations. Here, I will go through the basic forms of plotting, i.e. simple line graphs, scatterplots, bar graphs and histograms.

2 Simple line graph

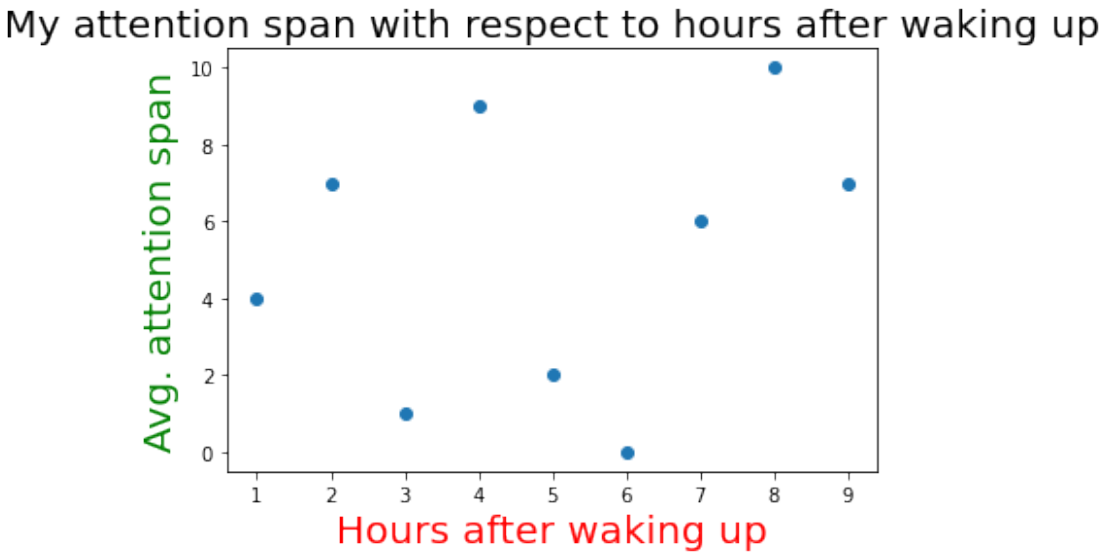
[51]:
```
from matplotlib.pyplot import scatter, plot, hist, xlabel, ylabel, title
from numpy import random
```

[55]:
```
x = range(1, 11)
y = [2, 4, 6, 7, 2, 6, 4, 5, 7, 9]
plot(x, y, marker = ".", markersize = 15)
xlabel("x", size = 20)
ylabel("y", size = 20)
title("My academic performance", size = 30, color = "blue")
None
```



3 Scatterplot

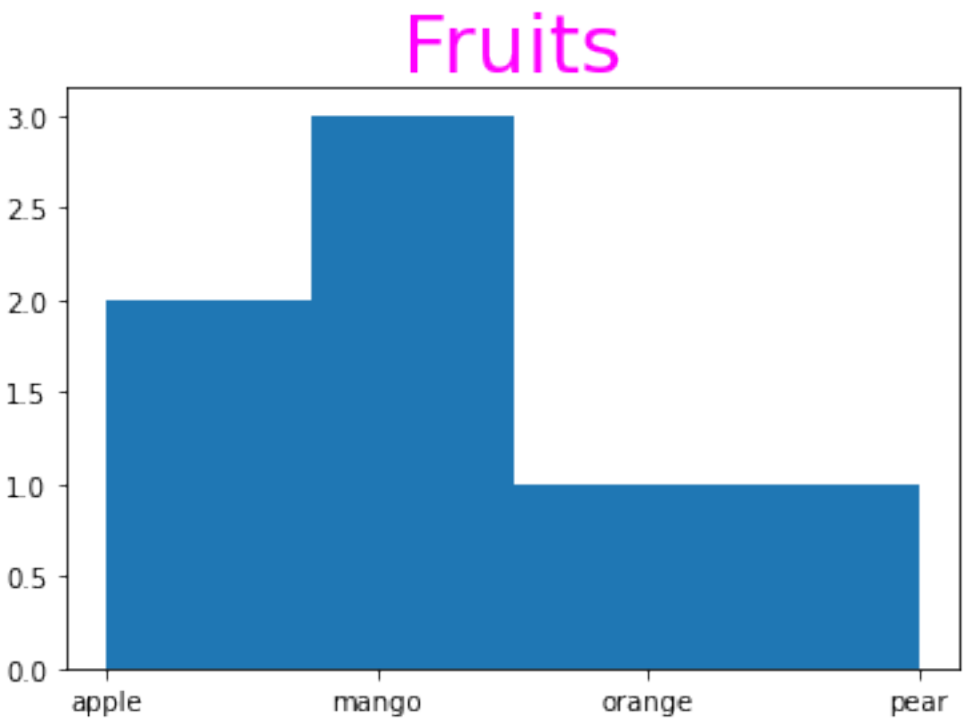
[68]:
```
x = range(1, 10)
y = [4, 7, 1, 9, 2, 0, 6, 10, 7]
scatter(x, y)
xlabel("Hours after waking up", size = 20, color = "red")
ylabel("Avg. attention span", size = 20, color = "green")
title("My attention span with respect to hours after waking up", size = "20")
None
```



4 Bar graph

While this is technically a histogram, its usage here is closer to that of a bar graph...

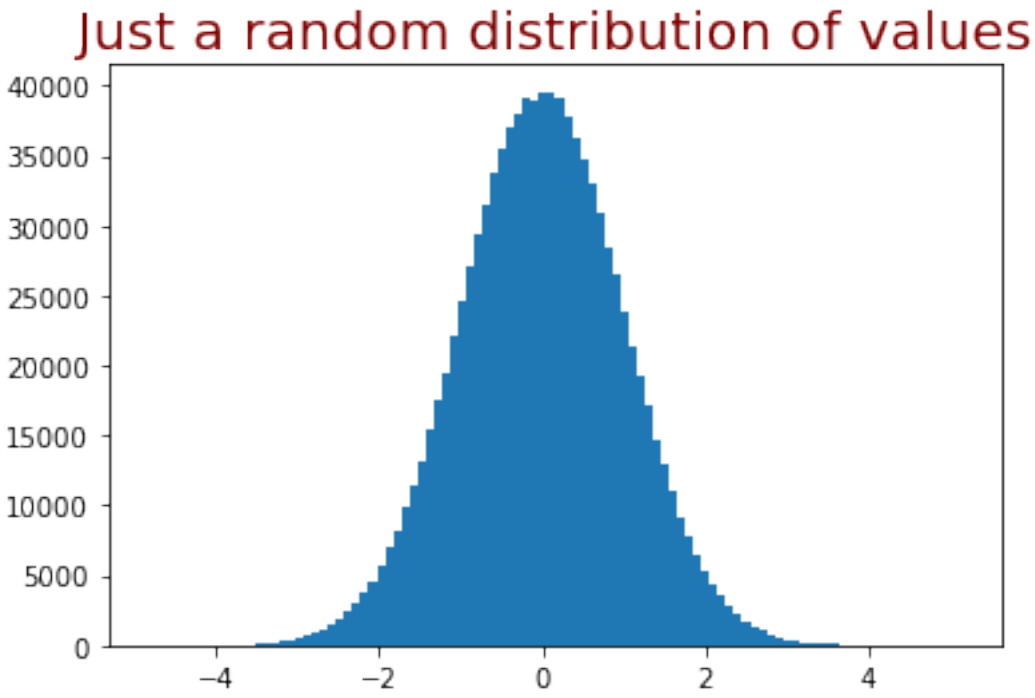
[69]:
```
# Histogram gives you the frequency of each value in the array
values = ["apple", "apple", "mango", "orange", "mango", "mango", "pear"]
hist(values, bins = 4)
# "bins" is the option for the number of different values you want to allow for
# Note that values are attached to frequencies
title("Fruits", size = "30", color = "magenta")
None
```



5 Histogram

Visualising a normally distributed random sample of numbers...

[66]:
```
x = range(1, 51)
# Creating a random normal distribution of some number of points
y = random.randn(1000000)
# As the number of points increases, the distribution becomes closer to the␣
 ↪normal curve
hist(y, bins = 100)
title("Just a random distribution of values", size = 20, color = "maroon")
None
```



6 CONCLUSION

Python provides a wide variety of plots and plotting options, and the demonstration here only showcases a limited number of these. Regardless, the above concepts will form the basis of our more advanced plotting assignments.

1. BASICS

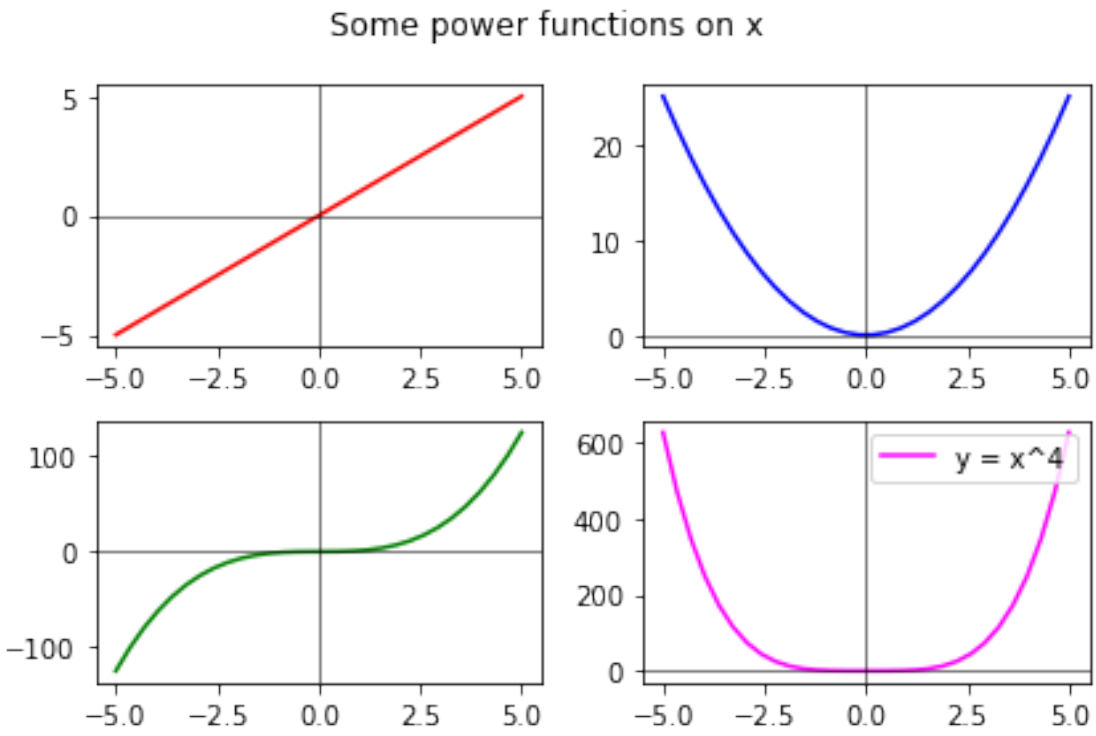
c) Subplots

November 27, 2021

1 AIM

This is a part of plotting recap. Here, we will be plotting the first four power functions i.e. x, x^2, x^3 and x^4. We will plot them as separate graphs, but also as graphs within a single grid i.e. each plot will be a subplot in a grid of plots.

[14]:
```
import matplotlib.pyplot as plt
from numpy import linspace
plt.suptitle("Some power functions on x")

x = linspace(-5, 5, 30)

plt.subplot(2, 2, 1)
plt.plot(x, x, color = "red", label = "y = x")
plt.axhline(lw=0.5, color='black')
plt.axvline(lw=0.5, color='black')

plt.subplot(2, 2, 2)
plt.plot(x, x**2, color = "blue", label = "y = x^2")
plt.axhline(lw=0.5, color='black')
plt.axvline(lw=0.5, color='black')

plt.subplot(2, 2, 3)
plt.plot(x, x**3, color = "green", label = "y = x^3")
plt.axhline(lw=0.5, color='black')
plt.axvline(lw=0.5, color='black')

plt.subplot(2, 2, 4)
plt.plot(x, x**4, color = "magenta", label = "y = x^4")
plt.axhline(lw=0.5, color='black')
plt.axvline(lw=0.5, color='black')


plt.legend()

plt.tight_layout()
None
```



2 CONCLUSION

Subplots allow us to view multiple graphs in an organised arrangement.

1. BASICS

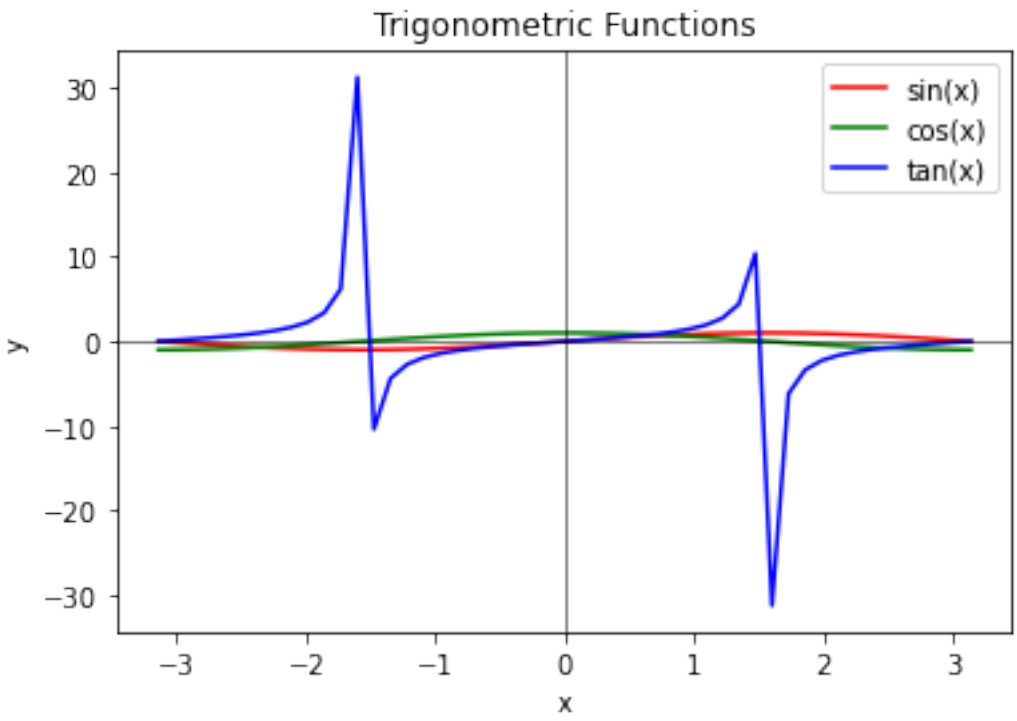
d) Multiple plots in a single graph

November 27, 2021

1 AIM

This is another addition to plotting recap. Here, instead of producing multiple plots as separate graphs, we will produce these plots in a single graph. In this record, we will produce multiple trigonometric functions in a single graph, range and domain.

[29]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi, np.pi, 50)
plt.plot(x, np.sin(x), color = "red", label = "sin(x)")
plt.plot(x, np.cos(x), color = "green", label = "cos(x)")
plt.plot(x, np.tan(x), color = "blue", label = "tan(x)")

plt.title('Trigonometric Functions')
plt.xlabel('x')
plt.ylabel('y')

plt.axhline(lw = 0.5, color = "black")
plt.axvline(lw = 0.5, color = "black")
plt.legend()
None
```



2 CONCLUSION

Multiple plots in a single graph are an effective way to compare the plots more closely and in a corresponding manner.

2. MATRIX RELATED

a) Matrix basics

November 27, 2021

1 AIM

Matrices are a vital tool in linear algebra. Here, we will look at their basic properties and functionalities.

2 Operations on matrices

A matrix is a 2D array where elements are divided into rows and columns.

2.1 Defining matrices using lists (not a good idea)

Defining matrix using list...

[1]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(A)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Accessing particular rows, columns and elements...

[7]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print("First row: {0}".format(A[0]))
print("First element: {0}".format(A[0][0]))
```

```
First row: [1, 2, 3]
First element: 1
```

Columns cannot be selected easily using this definition of matrices.

Matrices cannot be operated on using this definition of matrices (list of lists)...

[12]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
B = [[-1, -2, -3], [-4, -5, -6], [-7, -8, -9]]
print(A + B)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [-1, -2, -3], [-4, -5, -6], [-7, -8, -9]]
```

You cannot use -, * and / for lists.

The above issues that arise from defining a matrix as a list compel us to use numpy to define matrices instead.

2.2 Defining matrices using the numpy library (numerical python)

[3]:
```
import numpy as np
```

2.2.1 As an array

[16]:
```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A)
```

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

[21]:
```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
B = np.array([[-1, -2, -3], [-4, -5, -6], [-7, -8, -9]])
print(A + B)
print("---")
print(A - B)
print("---")
print(A * B)
print("---")
print(A / B)
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
---
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
---
[[ -1  -4  -9]
 [-16 -25 -36]
 [-49 -64 -81]]
---
[[-1. -1. -1.]
 [-1. -1. -1.]
 [-1. -1. -1.]]
```

2.2.2 As a matrix

[18]:
```
B = np.matrix([[-1, -2, -3], [-4, -5, -6], [-7, -8, -9]])
print(A)
```

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

[22]:
```
A = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
B = np.matrix([[-1, -2, -3], [-4, -5, -6], [-7, -8, -9]])
print(A + B)
print("---")
print(A - B)
print("---")
print(A * B)
print("---")
print(A / B)
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
---
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
---
[[ -30  -36  -42]
 [ -66  -81  -96]
 [-102 -126 -150]]
---
[[-1. -1. -1.]
 [-1. -1. -1.]
 [-1. -1. -1.]]
```

Array allows any number of dimensions, while matrix is strictly 2D. Also, the methods available for the latter also differ, and are more specialised for matrix operations.

2.2.3 Accessing rows, columns and elements

[56]:
```
A = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("The matrix:\n{0}".format(A))
print("3rd row:\n{0}".format(A[2]))
print("1st column:\n{0}".format(A[:,0]))
print("Element at 2nd row, 3rd column:\n{0}".format(A[1, 2]))
```

```
The matrix:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
3rd row:
[[7 8 9]]
1st column:
[[1]
 [4]
 [7]]
Element at 2nd row, 3rd column:
```

6

2.2.4 Methods

[4]:
```
A = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A.size) # Gives size of matrix
print(A.shape) # Gives order of matrix i.e. (no. of rows, no. of columns)
# NOTE: These work for arrays also
```

```
9
(3, 3)
```

[6]:
```
N = np.zeros((3,2)) # Makes a matrix of the given order consisting only of zeros
print(N)
M = np.ones((3,6)) # Makes a matrix of the given order consisting only of ones
print(M)
# NOTE: These can also create arrays, ex. np.zeros(4) i.e. an array of 4 zeros
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]]
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

[43]:
```
A = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("The matrix: {0}".format(A))
print("The determinant: {0}".format(np.linalg.det(A)))
```

```
The matrix: [[1 2 3]
 [4 5 6]
 [7 8 9]]
The determinant: -9.51619735392994e-16
```

Note that to find the determinant, the matrix must be a square matrix.

[53]:
```
A = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("The matrix:\n{0}".format(A))
print("The inverse:\n{0}".format(np.linalg.inv(A)))
print("The inverse:\n{0}".format(A.getH() / np.linalg.det(A))) # A.getH() gets␣
↪the adjoint of A
```

```
The matrix:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
The inverse:
[[ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]
 [-6.30503948e+15  1.26100790e+16 -6.30503948e+15]
```

```
 [ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]]
The inverse:
[[-1.05083991e+15 -4.20335965e+15 -7.35587939e+15]
 [-2.10167983e+15 -5.25419957e+15 -8.40671930e+15]
 [-3.15251974e+15 -6.30503948e+15 -9.45755922e+15]]
```

2.2.5 Matrix operations

[29]:
```
A = np.matrix([[3, 4, 5], [5, 7, 4], [6, 3, 2]])
B = np.matrix([[-2, 4, 2], [-6, 3, -4], [-3, 5, 2]])
```

[30]:
```
print(A + B)
print(np.add(A, B))
print(A - B)
print(np.add(A, -B))
```

```
[[ 1  8  7]
 [-1 10  0]
 [ 3  8  4]]
[[ 1  8  7]
 [-1 10  0]
 [ 3  8  4]]
[[ 5  0  3]
 [11  4  8]
 [ 9 -2  0]]
[[ 5  0  3]
 [11  4  8]
 [ 9 -2  0]]
```

[31]:
```
print("To do matrix multiplication")
print(A * B) # Apparently only works for square matrices
print(np.dot(A, B))
print("To multiply element-wise")
print(np.multiply(A, B))
```

```
To do matrix multiplication
[[-45  49   0]
 [-64  61 -10]
 [-36  43   4]]
[[-45  49   0]
 [-64  61 -10]
 [-36  43   4]]
To multiply element-wise
[[ -6  16  10]
 [-30  21 -16]
 [-18  15   4]]
```

[34]:
```
print(A / B)
print(np.divide(A, B))
print(np.multiply(A, 1/B))
```

```
[[-1.5         1.          2.5       ]
 [-0.83333333  2.33333333 -1.        ]
 [-2.          0.6         1.        ]]
[[-1.5         1.          2.5       ]
 [-0.83333333  2.33333333 -1.        ]
 [-2.          0.6         1.        ]]
[[-1.5         1.          2.5       ]
 [-0.83333333  2.33333333 -1.        ]
 [-2.          0.6         1.        ]]
```

Note that for matrix multiplication, the matrices must be such that the no. of columns of one matrix must equal the no. of rows in the second matrix.

[25]:
```
print(A**2)
# Repeated multiplication of matrix. Only works with square matrix because of␣
 ↪the condition for matrix multiplication.
```

```
[[59 55 41]
 [74 81 61]
 [45 51 46]]
```

[46]:
```
print("Original")
print(A)
print("Transposed")
print(np.transpose(A))
print(A.transpose())
print(A.T)
```

```
Original
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Transposed
[[1 4 7]
 [2 5 8]
 [3 6 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

3 CONCLUSION

Matrices are far more powerful than lists, when it comes to numerical collections. The wide availablility of functions makes it easy to use for linear algebra.

2. MATRIX RELATED

b) More on matrices

November 27, 2021

1 AIM

We will be looking at more properties and features of Python matrices that will be useful for linear algebra.

2 Symmetric or skew symmetric

[123]:
```
from numpy import matrix
def checkSymmetry(M, name):
    print("\n{0} =\n{1}".format(name, M))
    # Transpose = Original...?
    print("Is symmetric? {0}".format((False not in (M == M.T))))
    # Negative of transpose = Original...?
    print("is skew symmetric? {0}".format(False not in (M == -M.T)))

# Matrices to be tested
A = matrix([[1, 5, 4], [4, 6, 6], [-5, 1, 5]])
B = matrix([[-1, 3, 9], [-8, 2, 1], [-5, 1, -7]])

# Function call
checkSymmetry(A, "A")
checkSymmetry(B, "B")
```

```
A =
[[ 1  5  4]
 [ 4  6  6]
 [-5  1  5]]
Is symmetric? False
is skew symmetric? False

B =
[[-1  3  9]
 [-8  2  1]
 [-5  1 -7]]
Is symmetric? False
is skew symmetric? False
```

3 Row & column operations and reshaping

[4]:
```
from numpy import matrix, zeros, reshape, sqrt, random
# Random matrix
M = zeros([6, 5])
for i in range(0, 6):
    for j in range(0, 5):
        M[i, j] = random.randint(1, 25)
print(M)

# Square root of every element of 5th row
tmp = sqrt(M[4])
print("\nSquare root of 5th row's elements...")
for i, e in enumerate(tmp): print(i, ":", e)

# Adding corresponding elements of the 2nd and 4th row
tmp = M[1] + M[3]
print("\nSum of corresponding elements of 2nd and 4th rows...")
for i, e in enumerate(tmp): print(i, ":", e)

# Extracting the second column of the matrix.
# Reshaping this column into a 2 x 3 matrix.
tmp = M[:, 1]
# NOTE:
# All rows, column 2 => 2nd column
# You can given a specified number of rows or columns as well.
# Example: M[2:3, 1:4] => rows 3 to 4, columns 2 to 5.
# This enables you to extract submatrices of various dimensions from M.
print("\nColumn extracted...")
for i in tmp: print(i)
print("Reshaped into a 2 x 3 matrix...")
print(matrix(reshape(tmp, (2, 3))))
# NOTES
# The reshape function returns an array of the given order.
# I have converted this array to a matrix using the matrix class constructor.
# Reshaping of the 1D array above can be also done as tmp.reshape(2,3).
```

```
[[18. 21. 22.  2.  1.]
 [ 7. 24. 14. 21. 15.]
 [14.  6.  9.  9. 10.]
 [20.  3. 10.  6.  7.]
 [ 9.  1. 11.  6. 21.]
 [23.  4.  5. 12. 10.]]

Square root of 5th row's elements…
0 : 3.0
1 : 1.0
```

```
2 : 3.3166247903554
3 : 2.449489742783178
4 : 4.58257569495584

Sum of corresponding elements of 2nd and 4th rows…
0 : 27.0
1 : 27.0
2 : 24.0
3 : 27.0
4 : 22.0

Column extracted…
21.0
24.0
6.0
3.0
1.0
4.0
Reshaped into a 2 x 3 matrix…
[[21. 24.  6.]
 [ 3.  1.  4.]]
```

For more on reshaping a matrix… https://numpy.org/doc/stable/reference/generated/numpy.reshape.html

4 Aggregation & sorting operations for matrices

[3]:
```
from numpy import matrix, zeros, sum, product, trace, min, max, sort
# NOTE:
# For a multi-dimensional array or matrix...
# - sum from numpy can add all the elements
# - product from numpy can multiply all the elements
# - min and max from numpy can find the minimum and maximum values

# Random matrix
M = zeros([5, 5])
for i in range(0, 5):
    for j in range(0, 5):
        M[i, j] = random.randint(1, 5)
print("The matrix...\n{0}\n".format(M))
print("Sum =", sum(M))
print("Product =",product(M))
print("Trace =", trace(M))
print("Minimum =", min(M))
print("Maximum =", max(M))
print("\nThe matrix with every row sorted...\n{0}".format(sort(M)))
# NOTE
```

```
# Even though we pass the matrix, the sort function only sorts each row, since␣
↪it only sorts 1D arrays.
```

```
The matrix…
[[3. 2. 1. 2. 2.]
 [4. 3. 1. 4. 3.]
 [1. 4. 2. 1. 1.]
 [2. 2. 2. 4. 4.]
 [1. 3. 3. 4. 3.]]

Sum = 62.0
Product = 382205952.0
Trace = 15.0
Minimum = 1.0
Maximum = 4.0

The matrix with every row sorted…
[[1. 2. 2. 2. 3.]
 [1. 3. 3. 4. 4.]
 [1. 1. 1. 2. 4.]
 [2. 2. 2. 4. 4.]
 [1. 3. 3. 3. 4.]]
```

2. MATRIX RELATED

c) SymPy matrices

November 27, 2021

1 AIM

Matrices are available in both NumPy and SymPy. In SymPy, we have functionalities that are not offered in NumPy matrices. SymPy matrices can be considered as extensions of NumPy matrices, since they add on to the already wide range of features available for matrix computation, such as row echelon forms, rank, nullspaces, etc.

2 Row echelon form

Getting the row echelon form of matrices…

[1]:
```
from sympy import Matrix, pprint
B = Matrix([[1, 1, 1], [3, 1, -2], [2, 4, 7]])
C = Matrix([[7, 5, 3], [9, 8, -1], [8, 4, 3]])
print("------------\nMatrix B is...")
pprint(B)
print("Row echelon form of B...")
pprint(B.rref()[0])
# Matrix.rref(M) returns a tuple.
# The first element is the row reduced matrix.
# The second element is a tuple of non-zero i.e. pivot column numbers.
print("------------\nMatrix C is...")
pprint(C)
print("Row echelon form of C...")
pprint(C.rref()[0])
```

```
------------
Matrix B is…
⎡1  1  1 ⎤
⎢        ⎥
⎢3  1  -2⎥
⎢        ⎥
⎣2  4  7 ⎦
Row echelon form of B…
⎡1  0  -3/2⎤
⎢          ⎥
⎢0  1  5/2 ⎥
⎢          ⎥
⎣0  0  0   ⎦
```

```
 0  0   0
------------
Matrix C is…
 7  5  3

 9  8  -1

 8  4  3
Row echelon form of C…
 1  0  0

 0  1  0

 0  0  1
```

3 Matrix as sum of symmetric and skew-symmetric matrices

Expressing a matrix as a sum of symmetric and skew-symmetric matrices…

A square matrix M can be expressed as

$M = \frac{1}{2}(M + M^T) + \frac{1}{2}(M - M^T)$

Now, a transpose of any matrix M is such that row i of M becomes column i of M's transpose. Hence, row i in M's transpose is column i in M. Hence, by adding M and its transpose, we are essentially adding row i of M to column i of M, and adding every column i of M to row i. Hence, every row becomes row i + column i, and every column becomes column i + row i, meaning the rows and columns become equal i.e. interchangeable. Hence, by definition, $M + M^T$ is a symmetric matrix, meaning $\frac{1}{2}(M + M^T)$ is also symmetric.

Now, if we subtract M's transpose from M, we get that every row i of M becomes row i - column i, and the left-most element becomes zero. However, every column i of M becomes column i - row i by the same operation, and the top element becomes zero. Hence, we have the 1st row and 1st column as zero, and every other row and column such that row i = negative of column i i.e. every row becomes interchangeable with the corresponding column's negative. Hence, by definition, $M - M^T$ is a skew-symmetric matrix, meaning $\frac{1}{2}(M + M^T)$ is also skew-symmetric.

This, way, M can be expressed as a sum of symmetric and skew-symmetric matrices.

[20]:
```
from sympy import Matrix, pprint
M = Matrix([[1, 8, 0], [3, 5, 2], [-8, 9, -2]])
M_t = M.transpose()
M1 = (M + M_t)/2
M2 = (M - M_t)/2
# Transposes of M1 and M2...
M1_t = M1.transpose()
M2_t = M2.transpose()
print("We have that...")
pprint(M)
```

```
print("can be expressed as the sum of the following...")
print("------------")
pprint(M1)
print("Is symmetric? {0}".format(M1 == M1_t))
print("------------")
pprint(M2)
print("Is skew-symmetric? {0}".format(M2 == -M2_t))
print("------------")
print("To confirm...")
pprint(M1 + M2)
```

```
We have that…
1   8  0

3   5  2

-8  9  -2
can be expressed as the sum of the following…
------------
 1    11/2   -4

11/2   5    11/2

 -4   11/2   -2
Is symmetric? True
------------
 0    5/2   4

-5/2   0   -7/2

 -4   7/2   0
Is skew-symmetric? True
------------
To confirm…
1   8  0

3   5  2

-8  9  -2
```

4 Measures available for SymPy matrices

Various measures of a user-defined sympy matrix…

[9]:
```
# Support functions...
from sympy import Matrix, zeros, pprint
def inputPositiveInteger(prompt):
```

```
    while True:
        try:
            i = input(prompt)
            if i == "x": return 0
            i = int(i)
            if i <= 0: i = 1/0
            return i
        except:
            print("Invalid integer, please re-enter.")
def floatInput(prompt):
    while True:
        try:
            i = float(input(prompt))
            return i
        except:
            print("Invalid number, please re-enter.")

def matrixInput(nRow, nCol):
    print("\nEnter row by row, each element in the row separated by comma...")
    A, i = zeros(nRow, nCol), 0
    while i < nRow:
        row = input("R{0}: ".format(i + 1)).split(",")
        if "x" in row: break # To stop inputting anymore
        if len(row) != nCol:
            print("ERROR: You must only enter", nCol, "per row")
            continue
        for j in range(0, nCol):
            try:
                A[i, j] = float(row[j])
            except:
                print("ERROR: Non-numeric inputs.")
                j = -1
                break
        if j != -1: i = i + 1
    return A
```

[34]:
```
# Main program
singular = False
print("MATRIX 1")
nRow = inputPositiveInteger("Rows\t: ")
nCol = inputPositiveInteger("Columns\t: ")
A = matrixInput(nRow, nCol)
print("Your matrix:")
pprint(A)
print("------------\nResults of certain functions on this matrix...")
# Determinant
try:
```

```
    det = A.det()
    print("Determinant:", det)
    if det == 0: singular = True
except:
    print("Determinant: Unobtainable for non-square matrix!")
# Row echelon form
print("Row echelon form:")
pprint(A.rref()[0])
# Is singular?
print("Is singular:", singular)
# Trace
try:
    print("Trace:", A.trace())
except:
    print("Trace: Unobtainable for non-square matrix!")
# Rank
print("Rank:", A.rank())
# Nullity
nullspace = A.nullspace()
nullity = len(nullspace)
print("Nullity:", nullity)
# Nullspace
if nullity == 0:
    print("Nullspace: Empty")
else:
    print("Nullspace:")
    for M in nullspace:
        pprint(M)
```

```
MATRIX 1
Rows    : 3
Columns : 3

Enter row by row, each element in the row separated by comma…
R1: 32,-2, 0
R2: 21, 0, 231
R3: 2, 3, -3
Your matrix:
 32.0  -2.0   0.0

 21.0  0.0   231.0

 2.0   3.0   -3.0
------------
Results of certain functions on this matrix…
Determinant: -23226.0000000000
Row echelon form:
```

```
1  0  0

0  1  0

0  0  1
Is singular: False
Trace: 29.0000000000000
Rank: 3
Nullity: 0
Nullspace: Empty
```

4.1 NOTES

4.1.1 Nullspace

The null space of any matrix A consists of all the vectors B such that AB = 0 and B is not zero. It can also be thought as the solution obtained from AB = 0 where A is known matrix of size m x n and B is matrix to be found of size n x 1. The nullity of A is the number of vectors in its nullspace.

The nullspace function for sympy matrices returns a list of vectors that are in the nullspace of A, as defined above. By getting its length, we can figure out the nullity of A.

5 CONCLUSION

As we can see, SymPy matrices offer feaures and functions such as rank, nullspace and row echelon forms, that are widely used and very important in linear algebra. On top of this, pretty printing function as well a the default rendering of SymPy matrices makes them much more appealing to present and study.

2. MATRIX RELATED

d) Matrices (more properties + orthogonal matrix)

November 27, 2021

1 AIM

Here, we will explore more properties of inputted matrices. We will also discuss orthogonal matrices.

2

We will input a matrix and find some of its properties.

[37]:
```
# Support functions...
from numpy import matrix, zeros, sum, product, trace, min, max, sort
def inputPositiveInteger(prompt):
    while True:
        try:
            i = input(prompt)
            if i == "x": return 0
            i = int(i)
            if i <= 0: i = 1/0
            return i
        except:
            print("Invalid integer, please re-enter.")
def floatInput(prompt):
    while True:
        try:
            i = float(input(prompt))
            return i
        except:
            print("Invalid number, please re-enter.")

def matrixInput(nRow, nCol):
    print("\nEnter row by row, each element in the row separated by comma...")
    A, i = zeros((nRow, nCol)), 0
    while i < nRow:
        row = input("R{0}: ".format(i + 1)).split(",")
        if "x" in row: break # To stop inputting anymore
        if len(row) != nCol:
            print("ERROR: You must only enter", nCol, "per row")
            continue
```

```
        for j in range(0, nCol):
            try:
                A[i][j] = float(row[j])
            except:
                print("ERROR: Non-numeric inputs.")
                j = -1
                break
        if j != -1: i = i + 1
    return A
```

[13]:
```
# NOTE:
# For a multi-dimensional array or matrix...
# - sum from numpy can add all the elements
# - product from numpy can multiply all the elements
# - min and max from numpy can find the minimum and maximum values

# Matrix input
nRow = inputPositiveInteger("Rows\t: ")
nCol = inputPositiveInteger("Columns\t: ")
M = matrixInput(nRow, nCol)
print("The matrix...\n{0}\n".format(M))
print("Sum =", sum(M))
print("Product =",product(M))
print("Trace =", trace(M))
print("Minimum =", min(M))
print("Maximum =", max(M))
print("\nThe matrix with every row sorted...\n{0}".format(sort(M)))
# NOTE
# Even though we pass the matrix, the sort function only sorts each row, since␣
 ↪it only sorts 1D arrays.
```

```
Rows    : 2
Columns : 3

Enter row by row, each element in the row separated by comma…
R1: 1,2,3
R2: 1,5,2
The matrix…
[[1. 2. 3.]
 [1. 5. 2.]]

Sum = 14.0
Product = 60.0
Trace = 6.0
Minimum = 1.0
Maximum = 5.0
```

```
The matrix with every row sorted…
[[1. 2. 3.]
 [1. 2. 5.]]
```

3

We will generate one random matrix, and one random skew-matrix. We will find various properties of these matrices.

[96]:
```
# Some functions...
def randomMatrix(n, m):
    A = zeros((n, m))
    for i in range(0, n):
        for j in range(0, m):
            A[i][j] = random.randint(1, 25)
    return A
def randomSkewMatrix(n, m):
    A = zeros((n, m))
    # Creating upper triangle...
    for i in range(0, n):
        A[i][i] = 0
        for j in range(i + 1, m):
            A[i][j] = random.randint(1, 25)
    for i in range(0, n):
        for j in range(0, i):
            A[i][j] = -A[j][i]
    return A
```

[130]:
```
from numpy import linalg, matrix, identity, tril, random, round
# tril returns the lower triangular matrix for an array or matrix.
# The 1st argument is the array or matrix.
# The 2nd arguemnt specifies whether zeros should be at (k = -1) or above (k =␣
 ↪0) the diagonal.

# Matrix input
# n = inputPositiveInteger("n (rows or columns in the square matrix): ")
# M = matrixInput(n, n)
M = matrix([[4, 3, 2], [1, 4, 1], [3, 10, 4]])
print("Matrix:\n{0}".format(M))
# Proving properties of eigenvalues and eigenvectors...
# 1. For a nxn matrix, the number of eigen values is n.
print("------------------------")
print("PROPERTY 1:")
print("Eigen values:")
eigenValues = linalg.eigvals(M)
for e in eigenValues: print(e)
print("\nNumber of eigen values:", len(eigenValues))
```

```
# 2. The sum of eigen values is equal to the sum of the diagonal elements of␣
 ↪matrix.
print("------------------------")
print("PROPERTY 2:")
print("Sum of eigen values:", round(sum(eigenValues), 3))
print("Sum of diagonal values:", trace(M))
# 3. The product of eigenvalues is equal to the determinant of the matrix.
print("------------------------")
print("PROPERTY 3:")
print("Product of eigen values:", round(product(eigenValues), 3))
print("Determinant of matrix of:", (linalg.det(M)))
# 4. The eigen value for an identity matrix is 1.
print("------------------------")
print("PROPERTY 4:")
I = identity(3)
print("Identity matrix:\n{}".format(I))
print("Eigen values:")
tmp = linalg.eigvals(I)
for e in tmp: print(e)
# 5. The eigen value of a triangular matrix is same as the diagonal elements of␣
 ↪a matrix.
print("------------------------")
print("PROPERTY 5:")
# Random matrix...
A = matrix(randomMatrix(5, 5))
# Lower triangular matrix...
T = tril(A, 0)
print("Lower triangular matrix:\n{}".format(T))
print("Eigen values:")
tmp = linalg.eigvals(T)
for e in tmp: print(e)
# 6. For a skew symmetric matrix, the eigen values are imaginary.
print("------------------------")
print("PROPERTY 6:")
A = matrix(randomSkewMatrix(3, 3))
print("Matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
for e in tmp: print(e)
# 7. For orthogonal matrix the values of eigen values are 1 or -1.
print("------------------------")
print("PROPERTY 7:")
A = matrix([[1, 2, 2], [2, 1, -2], [-2, 2, -1]]) / 3
print("Orthogonal matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
for e in tmp: print(round(e, 3))
```

```
# 8. For idempotent matrix the eigenvalues are 0 and 1.
print("------------------------")
print("PROPERTY 8:")
A = matrix([[2, -2, -4], [-1, 3, 4], [1, -2, -3]])
print("Idempotent matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
for e in tmp: print(round(e, 3))
```

```
Matrix:
[[ 4  3  2]
 [ 1  4  1]
 [ 3 10  4]]
------------------------
PROPERTY 1:
Eigen values:
8.982056720677654
2.1289177050273396
0.8890255742950103

Number of eigen values: 3
------------------------
PROPERTY 2:
Sum of eigen values: 12.0
Sum of diagonal values: 12
------------------------
PROPERTY 3:
Product of eigen values: 17.0
Determinant of matrix of: 17.0
------------------------
PROPERTY 4:
Identity matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Eigen values:
1.0
1.0
1.0
------------------------
PROPERTY 5:
Lower triangular matrix:
[[ 5.  0.  0.  0.  0.]
 [21. 23.  0.  0.  0.]
 [ 9. 12.  2.  0.  0.]
 [24. 20. 19. 14.  0.]
 [13.  6. 13.  6.  9.]]
```

```
Eigen values:
9.0
14.0
2.0
23.0
5.0
------------------------
PROPERTY 6:
Matrix:
[[  0.   8.  22.]
 [ -8.   0.  10.]
 [-22. -10.   0.]]
Eigen values:
(-6.661338147750939e-16+25.455844122715714j)
(-6.661338147750939e-16-25.455844122715714j)
(6.676925621055143e-16+0j)
------------------------
PROPERTY 7:
Orthogonal matrix:
[[ 0.33333333  0.66666667  0.66666667]
 [ 0.66666667  0.33333333 -0.66666667]
 [-0.66666667  0.66666667 -0.33333333]]
Eigen values:
(1+0j)
(-0.333+0.943j)
(-0.333-0.943j)
------------------------
PROPERTY 8:
Idempotent matrix:
[[ 2 -2 -4]
 [-1  3  4]
 [ 1 -2 -3]]
Eigen values:
0.0
1.0
1.0
```

3.1 NOTES

3.1.1 Orthogonal matrices

When we say two vectors are orthogonal, we mean that they are perpendicular or form a right angle.

In linear algebra, an orthogonal matrix, or orthonormal matrix, is a real square matrix whose columns and rows are orthonormal vectors.

A square matrix with real numbers or elements is said to be an orthogonal matrix, if its transpose is equal to its inverse matrix. Or we can say, when the product of a square matrix and its transpose

gives an identity matrix, then the square matrix is known as an orthogonal matrix.

2. MATRIX RELATED

e) Exercise on row echelon form, rank & nullity

November 27, 2021

1 Finding rank of 2 given matrices

[26]:
```
from sympy import Matrix, zeros, pprint
G = Matrix([[1, 2, 1], [3, 1, -2], [2, 5, 7]])
S = Matrix([[7, 5, 1], [9, 8, -1], [8, 4, 3]])
print("G =")
pprint(G)
print("S =")
pprint(S)
```

```
G =
⎡1  2  1 ⎤
⎢         ⎥
⎢3  1  -2⎥
⎢         ⎥
⎣2  5  7 ⎦
S =
⎡7  5  1 ⎤
⎢         ⎥
⎢9  8  -1⎥
⎢         ⎥
⎣8  4  3 ⎦
```

[28]:
```
print("Ranks of G and S")
print("G rank =", G.rank())
print("S rank =", S.rank())
```

```
Ranks of G and S
G rank = 3
S rank = 3
```

2 Working on 2 user-inputted matrices

[1]:
```
# Support functions...
from sympy import Matrix, zeros, pprint
def inputPositiveInteger(prompt):
```

```
    while True:
        try:
            i = input(prompt)
            if i == "x": return 0
            i = int(i)
            if i <= 0: i = 1/0
            return i
        except:
            print("Invalid integer, please re-enter.")
def floatInput(prompt):
    while True:
        try:
            i = float(input(prompt))
            return i
        except:
            print("Invalid number, please re-enter.")

def matrixInput(nRow, nCol):
    print("\nEnter row by row, each element in the row separated by comma...")
    A, i = zeros(nRow, nCol), 0
    while i < nRow:
        row = input("R{0}: ".format(i + 1)).split(",")
        if "x" in row: break # To stop inputting anymore
        if len(row) != nCol:
            print("ERROR: You must only enter", nCol, "per row")
            continue
        for j in range(0, nCol):
            try:
                A[i, j] = float(row[j])
            except:
                print("ERROR: Non-numeric inputs.")
                j = -1
                break
        if j != -1: i = i + 1
    return A
```

[3]:
```
print("MATRIX 1")
A = matrixInput(3, 3)
print("\nMATRIX 2")
B = matrixInput(3, 3)
```

```
MATRIX 1

Enter row by row, each element in the row separated by comma…
R1: 1, 4, 5
R2: 2, 7, 4
R3: 9, 9, 0
```

```
MATRIX 2

Enter row by row, each element in the row separated by comma…
R1: 0, 3, 1
R2: 9, 6, 4
R3: 0, 8, 2
```

[9]:
```
print("A =")
pprint(A)
print("B =")
pprint(B)
```

```
A =
 1.0  4.0  5.0

 2.0  7.0  4.0

 9.0  9.0  0.0
B =
 0.0  3.0  1.0

 9.0  6.0  4.0

 0.0  8.0  2.0
```

[12]:
```
print("Row echelon forms...")
print("For A:")
pprint(A.rref()[0])
print("For B:")
pprint(B.rref()[0])
```

```
Row echelon forms…
For A:
 1  0  0

 0  1  0

 0  0  1
For B:
 1  0  0

 0  1  0

 0  0  1
```

[17]:
```
print("Checking if singular or not...")
def isSingular(M, name):
    try:
        M.det()
        print(name + " is non-singular!")
    except: print(name + " is singular!")

isSingular(A, 'A')
isSingular(B, 'B')
```

```
Checking if singular or not…
A is non-singular!
B is non-singular!
```

[19]:
```
print("Rank and nullity...")
def rankAndNullity(M, name):
    print("------------\nFor " + name + ":")
    # Rank
    print("Rank:", M.rank())
    # Nullity
    nullspace = M.nullspace()
    nullity = len(nullspace)
    print("Nullity:", nullity)

rankAndNullity(A, 'A')
rankAndNullity(B, 'B')
```

```
Rank and nullity…
------------
For A:
Rank: 3
Nullity: 0
------------
For B:
Rank: 3
Nullity: 0
```

NULLITY: The null space of any matrix A consists of all the vectors B such that AB = 0 and B is not zero. It can also be thought as the solution obtained from AB = 0 where A is known matrix of size m x n and B is matrix to be found of size n x 1. The nullity of A is the number of vectors in its nullspace. The nullspace function for sympy matrices returns a list of vectors that are in the nullspace of A, as defined above. By getting its length, we can figure out the nullity of A.

[21]:
```
print("Ranks of A+B, A-B and A•B...")
print("Rank:", (A + B).rank())
print("Rank:", (A - B).rank())
print("Rank:", (A * B).rank())
```

```
Ranks of A+B, A-B and A•B…
Rank: 3
Rank: 3
Rank: 3
```

As we can see, the ranks of all the above matrices are 3.

2. MATRIX RELATED

f) Exercise on matrices, subplots & differential equations

November 27, 2021

1 Solving differential equations

1.1 $\frac{dy}{dx} = -6xy$: $y(0) = 3$

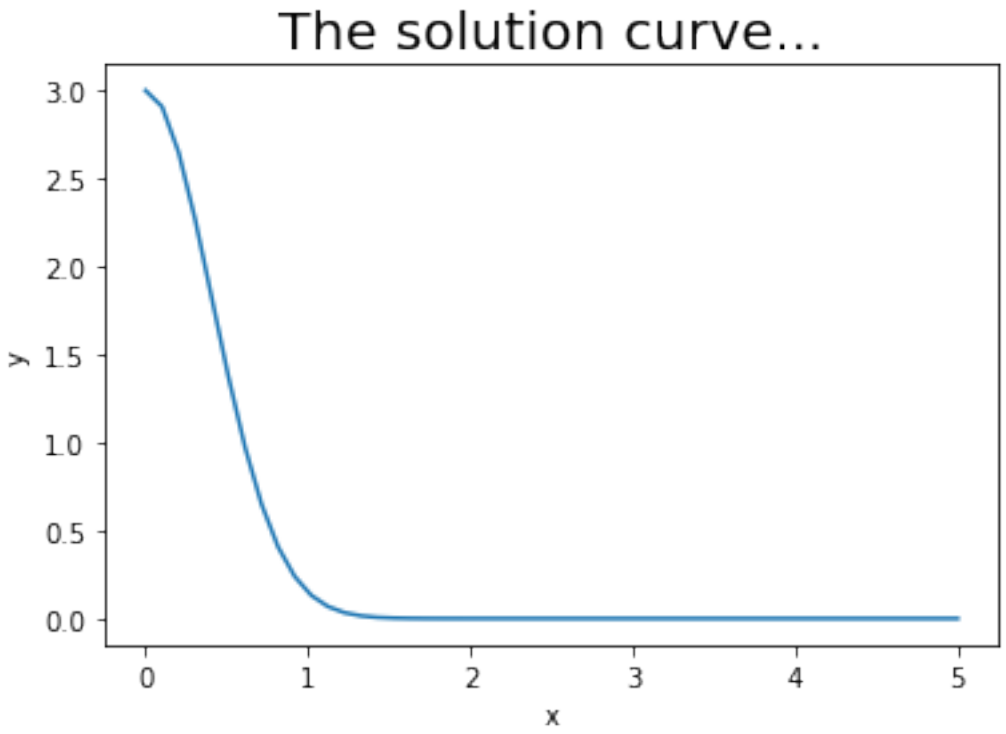
[3]:
```
from sympy import Eq, Symbol, Function, dsolve, solve

x = Symbol("x")
y = Function("y")(x)
ds = Eq(y.diff(x), -6*x*y)
gs = dsolve(ds, y)

C1 = Symbol("C1")
# NOTES
# subs accepts a dictionary as argument.
# In solve, no need to give 2nd argument if there is only 1 variable.
ps = gs.subs({C1 : solve(gs.subs({y : 3, x : 0}), C1)[0]})
# The index 0 is referred to for the solution of the GS when y = 3 and x = 0.
# This is because the return value of solve is a list, and the solution(s) are␣
 ↪the elements.
# This equation has only one solution for C1, hence only one element at index 0.
print("The solution...")
ps
```

```
The solution...
```

[3]: $y(x) = 3e^{-3x^2}$

[5]:
```
from numpy import linspace
from matplotlib.pyplot import plot, title, xlabel, ylabel, legend
from scipy.integrate import odeint

# Function that returns dy/dt
def dy_dx(y, x):
    return -6*y*x

# The initial condition
y0 = 3
```

```
# Time points
x = linspace(0, 5)

# Solve ODE
y = odeint(dy_dx, y0, x) # y-values

# Plot result
plot(x, y)
title("The solution curve...", size = 20)
xlabel("x")
ylabel("y")
None
```



1.2 $\frac{dy}{dx} = k + \frac{y}{2}$: $y(0) = 1$

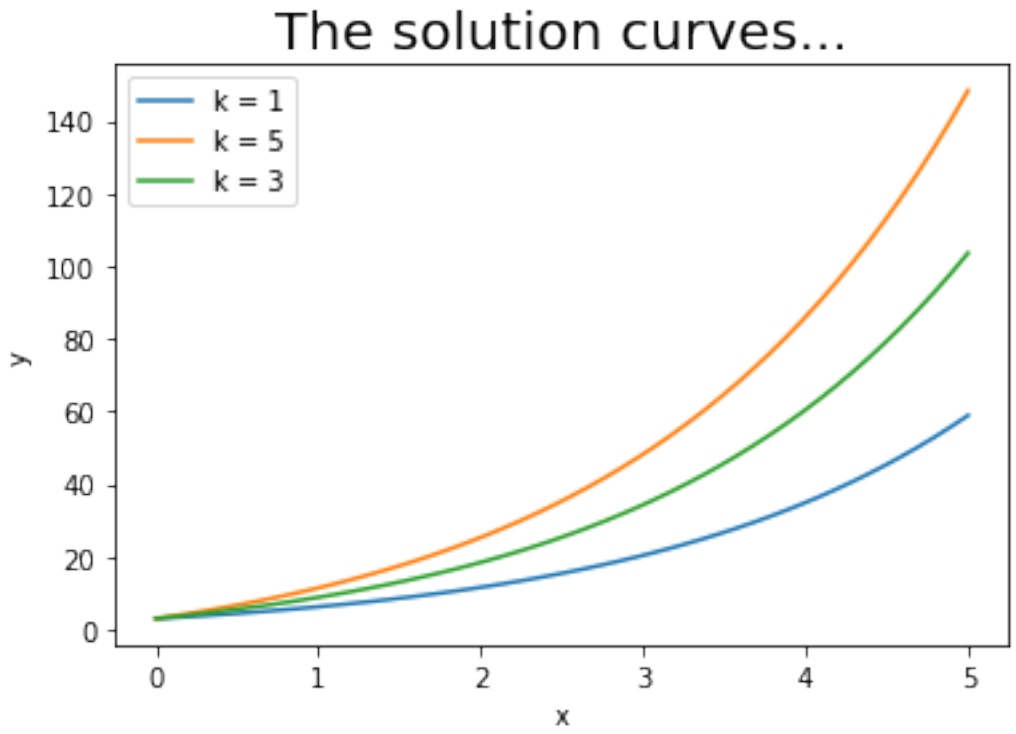
[6]:
```
from sympy import Eq, Symbol, Function, dsolve, solve

x = Symbol("x")
y = Function("y")(x)
k = Symbol("k")
ds = Eq(y.diff(x), k + y/2)
gs = dsolve(ds, y)
```

```
C1 = Symbol("C1")
# NOTES
# subs accepts a dictionary as argument.
# In solve, no need to give 2nd argument if there is only 1 variable.
ps = gs.subs({C1 : solve(gs.subs({y : 1, x : 0}), C1)[0]})
# The index 0 is referred to for the solution of the GS when y = 3 and x = 0.
# This is because the return value of solve is a list, and the solution(s) are␣
 ↪the elements.
# This equation has only one solution for C1, hence only one element at index 0.
print("The solution...")
ps
```

```
The solution...
```

[6]:
$$y(x) = -2k + (2k+1)\,e^{\frac{x}{2}}$$

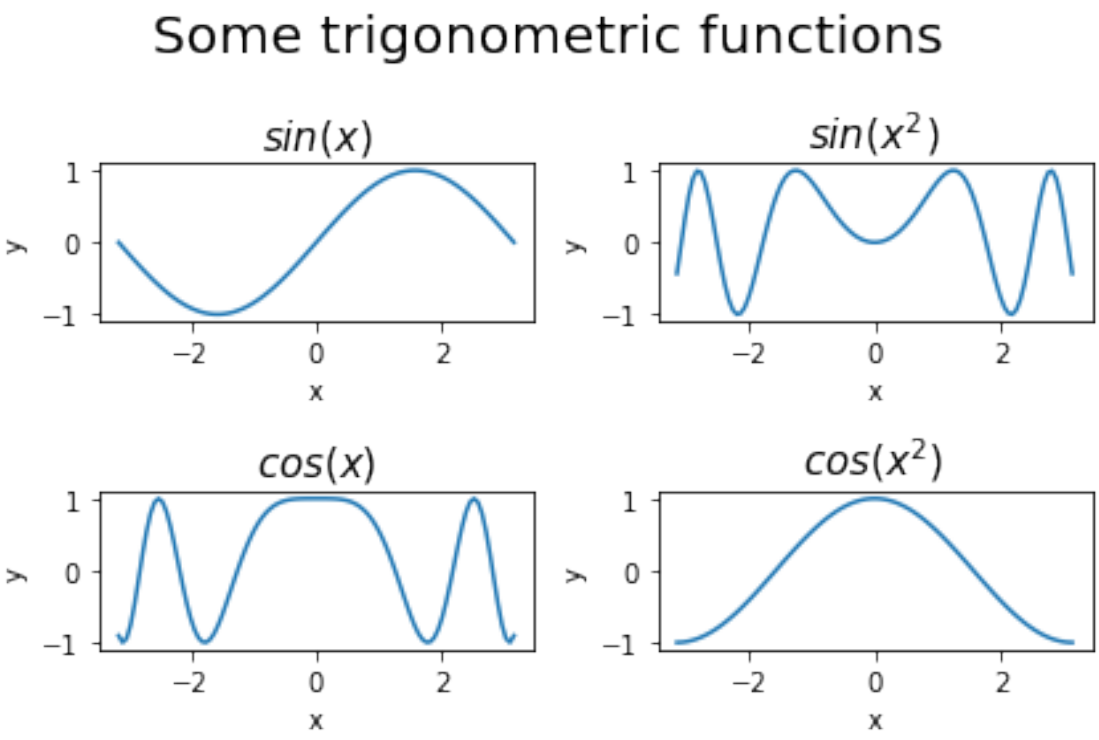
[7]:
```
from numpy import linspace
from matplotlib.pyplot import plot, title, xlabel, ylabel, legend
from scipy.integrate import odeint

# Function that returns dy/dt
def dy_dx(y, x):
    return k + y/2

# The initial condition
y0 = 3

# Time points
x = linspace(0, 5)

# Solve ODE
ys = list() # To append a whole list of y-values in each iteration
legends = list() # To store the various legend strings
ks = (1, 5, 3) # Stores different values for k
for k in ks:
    ys.append(odeint(dy_dx, y0, x)) # y-values
    legends.append("k = {0}".format(k)) # Legends

# Plot result
plot(x, ys[0], x, ys[1], x, ys[2])
legend([legends[0], legends[1], legends[2]])
title("The solution curves...", size = 20)
xlabel("x")
ylabel("y")
None
```



2 Subplots for certain trigonometric functions

[8]:
```
from numpy import sin, cos, linspace, pi
from matplotlib.pyplot import plot, title, xlabel, ylabel, subplot, suptitle,␣
 ↪tight_layout

# Since all plots apply the same labels, I have made a function to assign␣
 ↪labels to the plot.
def labels():
    xlabel("x", size = 10)
    ylabel("y", size = 10)

x = linspace(-pi, pi, 100)

suptitle("Some trigonometric functions", size = 20)
subplot(2, 2, 1)
y = sin(x)
plot(x, y)
labels()
title("$sin(x)$", size = 15)

subplot(2, 2, 2)
```

```
y = sin(x**2)
plot(x, y)
labels()
title("$sin(x^2)$", size = 15)

subplot(2, 2, 3)
y = cos(x**2)
plot(x, y)
labels()
title("$cos(x)$", size = 15)

subplot(2, 2, 4)
y = cos(x)
plot(x, y)
labels()
title("$cos(x^2)$", size = 15)

tight_layout()
# Must be given after the subplots.
# It ensures that there is sufficient space between the subplots

None
```



3 Miscellaneous problem

The sum of the digits of a three digit number is 16. The unit digit is two more than the sum of the other two digits. And the tens digit is five more than the hundreds digit. What is the number?

We can use algebra, but here, we will apply the brute force method using loops.

[97]:
```
for hundreds in range(0, 10):
    tens = hundreds + 5
    units = tens + hundreds + 2
    if units + tens + hundreds == 16:
        number = units + 10*tens + 100*hundreds
        print("The number required is", number)
```

```
The number required is 169
```

3. EIGENVALUES & EIGENVECTORS

a) Eigenvalues & eigenvectors

November 27, 2021

1 INTRODUCTION

Eigen vector of a matrix A is a vector represented by a matrix X such that when X is multiplied with matrix A, then the direction of the resultant matrix remains same as vector X. Mathematically, above statement can be represented as:

$AX = X$ where A is any arbitrary matrix, are eigen values and X is an eigen vector corresponding to each eigen value.

(i) The eigen values and corresponding eigen vectors are given by the characteristic equation, $|A–I| = 0$

(ii) To find the eigen vectors, we use the equation (A – I) X = 0 and solve it by Gaussian elimination, that is, convert the augmented matrix (A – I) = 0 to row echelon form and solve the linear system of equations thus obtained.

2 PYTHON CODE

SYNTAX: np.linalg.eigvals(A) (Returns eigen values)

SYNTAX: np.linalg.eig(A) (Returns eigen vectors)

[7]:
```
from numpy import linalg, matrix
M = matrix([[4, 3, 2], [1, 4, 1], [3, 10, 4]])
eigenValues = linalg.eigvals(M)
eigenVectors = linalg.eig(M)
print("\nEigenvalues...")
for i in eigenValues: print(i)
print("\nEigenvectors...")
for i in eigenVectors: print(i)
```

```
Eigenvalues…
8.982056720677654
2.1289177050273396
0.8890255742950103

Eigenvectors…
[8.98205672 2.12891771 0.88902557]
```

```
[[-0.49247712 -0.82039552 -0.42973429]
 [-0.26523242  0.14250681 -0.14817858]
 [-0.82892584  0.55375355  0.89071407]]
```

3. EIGENVALUES & EIGENVECTORS

b) Properties of eigenvalues

November 27, 2021

1 AIM

We will input one matrix, generate one random matrix, and one random skew-matrix. For these matrices, we will present various properties regarding eigenvalues.

Matrix input functions...

[11]:
```
from numpy import linalg, matrix, identity, tril, random, round, zeros, trace,␣
↪product
def inputPositiveInteger(prompt):
    while True:
        try:
            i = input(prompt)
            if i == "x": return 0
            i = int(i)
            if i <= 0: i = 1/0
            return i
        except:
            print("Invalid integer, please re-enter.")
def floatInput(prompt):
    while True:
        try:
            i = float(input(prompt))
            return i
        except:
            print("Invalid number, please re-enter.")

def matrixInput(nRow, nCol):
    print("\nEnter row by row, each element in the row separated by comma...")
    A, i = zeros((nRow, nCol)), 0
    while i < nRow:
        row = input("R{0}: ".format(i + 1)).split(",")
        if "x" in row: break # To stop inputting anymore
        if len(row) != nCol:
            print("ERROR: You must only enter", nCol, "per row")
            continue
        for j in range(0, nCol):
            try:
```

```
                A[i][j] = float(row[j])
            except:
                print("ERROR: Non-numeric inputs.")
                j = -1
                break
        if j != -1: i = i + 1
    return A
```

Matrix generation functions...

[12]:
```
from numpy import linalg, matrix, identity, tril, random, round
# tril returns the lower triangular matrix for an array or matrix.
# The 1st argument is the array or matrix.
# The 2nd arguemnt specifies whether zeros should be at (k = -1) or above (k =
↪0) the diagonal.
# Some functions...

def randomMatrix(n, m):
    A = zeros((n, m))
    for i in range(0, n):
        for j in range(0, m):
            A[i][j] = random.randint(1, 25)
    return A
def randomSkewMatrix(n, m):
    A = zeros((n, m))
    # Creating upper triangle...
    for i in range(0, n):
        A[i][i] = 0
        for j in range(i + 1, m):
            A[i][j] = random.randint(1, 25)
    for i in range(0, n):
        for j in range(0, i):
            A[i][j] = -A[j][i]
    return A
```

Main...

[13]:
```
# Matrix input
n = inputPositiveInteger("n (rows or columns in the square matrix): ")
M = matrixInput(n, n)
print("Matrix:\n{0}".format(M))
# Proving properties of eigenvalues and eigenvectors...
# 1. For a nxn matrix, the number of eigen values is n.
print("------------------------")
print("PROPERTY 1:")
print("Eigen values:")
eigenValues = linalg.eigvals(M)
for e in eigenValues: print(e)
```

```
print("\nNumber of eigen values:", len(eigenValues))
# 2. The sum of eigen values is equal to the sum of the diagonal elements of␣
 ↪matrix.
print("------------------------")
print("PROPERTY 2:")
print("Sum of eigen values:", round(sum(eigenValues), 3))
print("Sum of diagonal values:", trace(M))
# 3. The product of eigenvalues is equal to the determinant of the matrix.
print("------------------------")
print("PROPERTY 3:")
print("Product of eigen values:", round(product(eigenValues), 3))
print("Determinant of matrix of:", (linalg.det(M)))
# 4. The eigen value for an identity matrix is 1.
print("------------------------")
print("PROPERTY 4:")
I = identity(3)
print("Identity matrix:\n{}".format(I))
print("Eigen values:")
tmp = linalg.eigvals(I)
for e in tmp: print(e)
# 5. The eigen value of a triangular matrix is same as the diagonal elements of␣
 ↪a matrix.
print("------------------------")
print("PROPERTY 5:")
# Random matrix...
A = matrix(randomMatrix(5, 5))
# Lower triangular matrix...
T = tril(A, 0)
print("Lower triangular matrix:\n{}".format(T))
print("Eigen values:")
tmp = linalg.eigvals(T)
for e in tmp: print(e)
# 6. For a skew symmetric matrix, the eigenvalues are imaginary.
print("------------------------")
print("PROPERTY 6:")
A = matrix(randomSkewMatrix(3, 3))
print("Matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
for e in tmp: print(e)
# 7. For orthogonal matrix the values of eigenvalues are 1 or -1.
print("------------------------")
print("PROPERTY 7:")
A = matrix([[1, 2, 2], [2, 1, -2], [-2, 2, -1]]) / 3
print("Orthogonal matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
```

```
for e in tmp: print(round(e, 3))
# 8. For idempotent matrix the eigenvalues are 0 and 1.
print("------------------------")
print("PROPERTY 8:")
A = matrix([[2, -2, -4], [-1, 3, 4], [1, -2, -3]])
print("Idempotent matrix:\n{0}".format(A))
print("Eigen values:")
tmp = linalg.eigvals(A)
for e in tmp: print(round(e, 3))
```

```
n (rows or columns in the square matrix): 3

Enter row by row, each element in the row separated by comma…
R1: 2,3,4
R2: 2,6,3
R3: 0,7,4
Matrix:
[[2. 3. 4.]
 [2. 6. 3.]
 [0. 7. 4.]]
------------------------
PROPERTY 1:
Eigen values:
(10.747189497992093+0j)
(0.6264052510039564+1.7729705556948898j)
(0.6264052510039564-1.7729705556948898j)

Number of eigen values: 3
------------------------
PROPERTY 2:
Sum of eigen values: (12+0j)
Sum of diagonal values: 12.0
------------------------
PROPERTY 3:
Product of eigen values: (38+0j)
Determinant of matrix of: 37.99999999999999
------------------------
PROPERTY 4:
Identity matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Eigen values:
1.0
1.0
1.0
------------------------
```

```
PROPERTY 5:
Lower triangular matrix:
[[11.  0.  0.  0.  0.]
 [11.  4.  0.  0.  0.]
 [11. 21. 18.  0.  0.]
 [13. 18. 21. 17.  0.]
 [12.  9. 23.  8. 18.]]
Eigen values:
18.0
17.0
18.0
4.0
11.0
------------------------
PROPERTY 6:
Matrix:
[[  0.   2.   8.]
 [ -2.   0.  13.]
 [ -8. -13.   0.]]
Eigen values:
(3.8556595020785216e-17+0j)
(4.440892098500626e-16+15.394804318340654j)
(4.440892098500626e-16-15.394804318340654j)
------------------------
PROPERTY 7:
Orthogonal matrix:
[[ 0.33333333  0.66666667  0.66666667]
 [ 0.66666667  0.33333333 -0.66666667]
 [-0.66666667  0.66666667 -0.33333333]]
Eigen values:
(1+0j)
(-0.333+0.943j)
(-0.333-0.943j)
------------------------
PROPERTY 8:
Idempotent matrix:
[[ 2 -2 -4]
 [-1  3  4]
 [ 1 -2 -3]]
Eigen values:
0.0
1.0
1.0
```

4. EQUATION SOLVING

a) System of linear equations

November 27, 2021

1 INTRODUCTION

A linear equation for n variables is an equation wherein each variable appears only as a simple linear function of itself i.e. it is simply multiplied by a constant coefficient (which can also be 0 and 1). Apart from these variables, constants will also be present (at least 0 will be present, maybe after some remodelling).

1.1 Matrix method

Consider the system of equations $a_1x - b_1y + c_1z = d_1$, $a_2x + b_2y - b_3z = d_2$ and $a_3x + b_3y + c_3z = d_3$

$$A = \text{Coefficient matrix} = \begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix}$$

$$B = \text{Constant matrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$X = \text{Variable matrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

We have $AX = B$, which means $X = A^1B$. Note that $X[0] = x, X[1] = y, X[2] = z$ Hence, we need to find the inverse of A, and perform matrix multiplication between it and B.

[1]:
```
import numpy as np
```

Solve the system of equations $2x - 3y + 5z = 11$, $5x + 2y - 7z = -12$ and $-4x + 3y + z = 5$

[20]:
```
# Defining the coefficent matrix
A = np.matrix([[2, -3, 5], [5, 2, -7], [-4, 3, 1]])
B = np.matrix([[11], [-12], [5]])
# Find the inverse of A
A_inverse = np.linalg.inv(A)
# Applying the formula
X = np.dot(A_inverse, B)
print("x = {0}, y = {1}, z = {2}".format(X[0][0], X[1][0], X[2][0]))
```

```
x = [[1.]], y = [[2.]], z = [[3.]]
```

1.2 Direct method (using inbuilt function)

[2]:
```
import numpy as np
```

Solve the system of equations $2x - 3y + 5z = 11$, $5x + 2y - 7z = -12$ and $-4x + 3y + z = 5$

[3]:
```
# Defining the coefficent matrix
A = np.matrix([[2, -3, 5], [5, 2, -7], [-4, 3, 1]])
B = np.matrix([[11], [-12], [5]])
# Applying the function
X = np.linalg.solve(A, B)
print("x = {0}, y = {1}, z = {2}".format(X[0][0], X[1][0], X[2][0]))
```

```
x = [[1.]], y = [[2.]], z = [[3.]]
```

1.3 NOTES

Note that A is a 3 x 3 matrix, while B and X are 3 x 1 matrices i.e. 3 rows, 1 column

4. EQUATION SOLVING

b) Simple system solver

November 27, 2021

1 AIM

Creating a basic function that inputs coefficients and constants of a system of linear equations and solves this system.

[1]:
```
import numpy as np
def solveEqs(order):
    try:
        n = int(order)
    except:
        print("Not a positive integer!")
    if n < 1:
        print("Number of equations is too small")
        return 0
    A = np.zeros((n, n))
    B = np.zeros((n, 1))
    for i in range(0, n):
        print("Equation #{0}:".format(i + 1))
        for j in range(0, n):
            try:
                A[i][j] = float(input(("Coefficient #{0}: ").format(j + 1)))
            except:
                print("Invalid input")
                return 0
        try:
            B[i][0] = float(input(("Constant sum: ")))
        except:
            print("Invalid input")
            return 0
        print()
    X = np.linalg.solve(A, B)
    print(X)
    print("x = {0}".format(float(X[0, 0])))
    print("y = {0}".format(float(X[1, 0])))
    print("z = {0}".format(float(X[2, 0])))
```

[2]:
```
solveEqs(3)
```

```
Equation #1:
Coefficient #1: 2
Coefficient #2: 3
Coefficient #3: 4
Constant sum: -3

Equation #2:
Coefficient #1: 3
Coefficient #2: 4
Coefficient #3: 2
Constant sum: -3

Equation #3:
Coefficient #1: -4
Coefficient #2: 0
Coefficient #3: 23
Constant sum: -1

[[-3.47058824]
 [ 2.17647059]
 [-0.64705882]]
x = -3.4705882352941178
y = 2.1764705882352944
z = -0.6470588235294118
```

4. EQUATION SOLVING

c) Advanced system solver

November 27, 2021

Support functions...

[10]:
```
import numpy as np
# Reads the equation strings and converts them into lists of various values...
def getLists(equationList):
    var, coef, const, varCount, eqCount = {}, [], [], 0, 0
    # NOTES:
    # 'coef' is a list of lists.
    # Each sublist is for a variable, each sublist element corresponds to the␣
↪equation number.
    # 'var' is the dictionary, with key as variable name and value as the␣
↪variable index.
    # Variable indices are simply to help associate the coefficient lists to␣
↪the variables.
    for e in equationList:
        e, i, varsFound, sign = e + "\\", 0, [], 1
        # If coefficient of a variable is to the right of "=", we will take it␣
↪to the LHS, reversing its sign.
        # If constant term is to the left of "=", we will take it to the RHS,␣
↪reversing its sign.

        # Going through the equation...
        while e[i] != "\\":
            coefValue = ""
            while e[i].isspace(): i = i + 1 # To traverse possible spaces␣
↪before '-'.
            if e[i] == "-": coefValue, i = coefValue + "-", i + 1 # Negative␣
↪sign detection.
            while e[i].isspace(): i = i + 1 # To traverse possible spaces after␣
↪'-'.

            # Number encountered...
            if e[i].isnumeric():
                coefValue, i = coefValue + e[i], i + 1
                while e[i].isnumeric() and e[i] != "\\":
                    coefValue, i = coefValue + e[i], i + 1
```

```
            # Alphabet encountered (potential variable)...
            if e[i].isalpha():
                varName, i = e[i], i + 1
                while e[i].isalnum() and e[i] != "\\":
                    varName, i = varName + e[i], i + 1
                # (This stores the entire unspaced string as a variable, if␣
↪encountered)

                # If variable already encountered in equation...
                if varName in varsFound:
                    coef[var[varName]][-1] = coef[var[varName]][-1] +␣
↪float(coefValue) * sign
                    # Coefficients get added.

                # If variable is newly encountered in the equation...
                else:
                    varsFound.append(varName)
                    # If the variable is newly encountered in the system...
                    if(varName not in var):
                        var[varName], varCount = varCount, varCount + 1
                        coef.append([])
                    # If no numerical coefficient specified...
                    if coefValue == "" or coefValue == "-": coefValue =␣
↪coefValue + "1"
                    # Making sure zero constants are put where required...
                    l = len(coef[var[varName]])
                    while l < eqCount:
                        coef[var[varName]].append(0)
                        l = l + 1
                    coef[var[varName]].append(float(coefValue) * sign)

            # If a constant is identified...
            elif coefValue != "":
                # If a constant already exists in the equation...
                if "c" in varsFound:
                    const[-1] = const[-1] + float(coefValue) * -sign
                # If a constant hasn't been encountered before...
                else:
                    varsFound.append("c")
                    const.append(float(coefValue) * -sign)

            # If equal-to sign encountered, invert the sign variable...
            else:
                if e[i] == "=": sign = -1
                i = i + 1
        eqCount = eqCount + 1
```

```
        # Making sure zero constant sums are put where required...
        if len(const) < eqCount: const.append(0)
    return (coef, const, var)

# Uses the lists of values from "getLists" and creates the necessary matrices...
def getMatrices(equationList):
    (coef, const, var) = getLists(equationList)
    nVar, nEq = len(coef), len(const)
    A = np.zeros((nEq, nVar))
    B = np.zeros((nEq, 1))
    for i in range(0, nEq):
        for j in range(0, nVar):
            try: A[i][j] = coef[j][i]
            except: A[i][j] = 0
    for i in range(0, nEq):
        B[i][0] = const[i]
    return (A, B, var)
```

Main function...

[15]:
```
# Uses the matrices from "getMatrices" and finds the solutions if they exists...

def solveSystem(equationList):
    (A, B, var) = getMatrices(equationList)
    A = np.matrix(A)
    B = np.matrix(B)
    print("Coefficient matrix:")
    print(A)
    print("Constant sum matrix:")
    print(B)
    try: np.linalg.inv(A)
    except:
        print("Inverse of coefficient matrix does not exist.")
        print("No solutions.")
    else: print("Inverse of coefficient matrix exists.")
    X = np.linalg.solve(A, B)
    for i in var:
        print("{0} = {1}".format(i, X[var[i], 0]))
```

1 EXAMPLES

1.1

[22]:
```
eqs = [ "8x + 5y = 9z",
        "3x + 2z = 9",
        "4y + 3z = 11x"]
solveSystem(eqs)
```

```
Coefficient matrix:
[[  8.   5.  -9.]
 [  3.   0.   2.]
 [-11.   4.   3.]]
Constant sum matrix:
[[0.]
 [9.]
 [0.]]
Inverse of coefficient matrix exists.
x = 1.4036697247706422
y = 2.064220183486239
z = 2.3944954128440368
```

1.2

[23]:
```
eqs = [ "x1 + 2x2 + x3 - 2x3 = -1",
        "2x1 + x2 + 4x3 = 2",
        "3x1 + 3x2 + 4x3 = 1"]
solveSystem(eqs)
```

```
Coefficient matrix:
[[ 1.  2. -1.]
 [ 2.  1.  4.]
 [ 3.  3.  4.]]
Constant sum matrix:
[[-1.]
 [ 2.]
 [ 1.]]
Inverse of coefficient matrix exists.
x1 = 1.6666666666666679
x2 = -1.3333333333333341
x3 = -3.7007434154171906e-16
```

2 CONCLUSION

This function allows for any arrangement of variables, any spacing, any variable names and any number of equations and variables, since its data structures are designed to dynamically generate their structure and data based on the inputs.

4. EQUATION SOLVING

d) Solving systems using row echelon form

November 27, 2021

1 Question

Obtain the row reduced echelon form from the following system of equations

$2x + 8y + 4z = 2$

$2x + 5y + z = 5$

$4x + 10y - z = 1$

Mention the values of x , y and z obtained.

2 Solution

[96]:
```
from sympy import Matrix, pprint
A = Matrix([[2, 8, 4], [2, 5, 1], [4, 10, -1]])
B = Matrix([[2], [5], [1]])
A_B = A.col_insert(1, B)
```

[84]:
```
A
```

[84]:
$$\begin{bmatrix} 2 & 8 & 4 \\ 2 & 5 & 1 \\ 4 & 10 & -1 \end{bmatrix}$$

[85]:
```
B
```

[85]:
$$\begin{bmatrix} 2 \\ 5 \\ 1 \end{bmatrix}$$

[98]:
```
print("Row echelon form of A:B is")
pprint(A_B.rref()[0])
```

```
Row echelon form of A:B is
1  0  0  -11/3

0  1  0   1/3

0  0  1   4/3
```

x is -11/3, y is 1/3 and z is 4/3.

4. EQUATION SOLVING

e) Solving systems using Cramer's rule

November 27, 2021

1 INTRODUCTION

Cramer's rule is an explicit formula for solving systems of linear equations.

Consider the system of equations $a_1x-b_1y+c_1z=d_1$, $a_2x+b_2y-b_3z=d_2$ and $a_3x+b_3y+c_3z=d_3$

$A=$ Coefficient matrix $=\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix}$

$B=$ Constant matrix $=\begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$

$X=$ Variable matrix $=\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$

Cramer's rule states that $x_i=\frac{|A_i|}{|A|}$, where $i=1,2,3$, and A_i is the resultant matrix when you replace the ith column of A with the column vector B.

2 PYTHON CODE

Solving equation systems using Cramer's rule. Hence, consider the following in the context of simultaneous equations.

[1]:
```
import numpy as np
import copy as cp
def cramersRule(A, B):
# A is the matrix of coefficients.
# B is the column matrix of constant sums.
    X = []
    solutions = [] # Intended list of solutions.
    nVars = len(A[0]) # Length of a row => Number of variables.
    for i in range(0, nVars + 1):
        X.append(cp.deepcopy(A))
        X[i][:, i] = B
        solutions.append(np.linalg.det(X[i])/np.linalg.det(A))
    return solutions
```

3 EXAMPLE

$4x + y = 6200$

$3x + 3y = 5600$

[2]:
```
A = np.matrix([[4, 1], [3, 3]]) # Matrix of coefficients.
B = np.matrix([[6200], [5600]]) # Matrix of constant sums.
cramersRule(A, B)
```

[2]:
```
[1444.4444444444437, 422.22222222222246]
```

4. EQUATION SOLVING

f) Different approaches to solving systems of linear equations

November 27, 2021

1 AIM

We will try solving systems of linear differential equations using matrices, and using different approaches...

Support functions...

[2]:
```
# Imports and support functions for all the below operations...
from numpy import matrix, zeros, linalg
import numpy as np
# Reads the equation strings and converts them into lists of various values...
def getLists(equationList):
    var, coef, const, varCount, eqCount = {}, [], [], 0, 0
    # NOTES:
    # 'coef' is a list of lists.
    # Each sublist is for a variable, each sublist element corresponds to the␣
↪equation number.
    # 'var' is the dictionary, with key as variable name and value as the␣
↪variable index.
    # Variable indices are simply to help associate the coefficient lists to␣
↪the variables.
    for e in equationList:
        e, i, varsFound, sign = e + "\\", 0, [], 1
        # If coefficient of a variable is to the right of "=", we will take it␣
↪to the LHS, reversing its sign.
        # If constant term is to the left of "=", we will take it to the RHS,␣
↪reversing its sign.

        # Going through the equation...
        while e[i] != "\\":
            coefValue = ""
            while e[i].isspace(): i = i + 1 # To traverse possible spaces␣
↪before '-'.
            if e[i] == "-": coefValue, i = coefValue + "-", i + 1 # Negative␣
↪sign detection.
            while e[i].isspace(): i = i + 1 # To traverse possible spaces after␣
↪'-'.
```

```
            # Number encountered...
            if e[i].isnumeric():
                coefValue, i = coefValue + e[i], i + 1
                while e[i].isnumeric() and e[i] != "\\":
                    coefValue, i = coefValue + e[i], i + 1

            # Alphabet encountered (potential variable)...
            if e[i].isalpha():
                varName, i = e[i], i + 1
                while e[i].isalnum() and e[i] != "\\":
                    varName, i = varName + e[i], i + 1
                # (This stores the entire unspaced string as a variable, if␣
↪encountered)

                # If variable already encountered in equation...
                if varName in varsFound:
                    coef[var[varName]][-1] = coef[var[varName]][-1] +␣
↪float(coefValue) * sign
                    # Coefficients get added.

                # If variable is newly encountered in the equation...
                else:
                    varsFound.append(varName)
                    # If the variable is newly encountered in the system...
                    if(varName not in var):
                        var[varName], varCount = varCount, varCount + 1
                        coef.append([])
                    # If no numerical coefficient specified...
                    if coefValue == "" or coefValue == "-": coefValue =␣
↪coefValue + "1"
                    # Making sure zero constants are put where required...
                    l = len(coef[var[varName]])
                    while l < eqCount:
                        coef[var[varName]].append(0)
                        l = l + 1
                    coef[var[varName]].append(float(coefValue) * sign)

            # If a constant is identified...
            elif coefValue != "":
                # If a constant already exists in the equation...
                if "c" in varsFound:
                    const[-1] = const[-1] + float(coefValue) * -sign
                # If a constant hasn't been encountered before...
                else:
                    varsFound.append("c")
                    const.append(float(coefValue) * -sign)
```

```
            # If equal-to sign encountered, invert the sign variable...
            else:
                if e[i] == "=": sign = -1
                i = i + 1
        eqCount = eqCount + 1

        # Making sure zero constant sums are put where required...
        if len(const) < eqCount: const.append(0)
    return (coef, const, var)

# Uses the lists of values from "getLists" and creates the necessary matrices...
def getMatrices(equationList):
    (coef, const, var) = getLists(equationList)
    nVar, nEq = len(coef), len(const)
    A = np.zeros((nEq, nVar))
    B = np.zeros((nEq, 1))
    for i in range(0, nEq):
        for j in range(0, nVar):
            try: A[i][j] = coef[j][i]
            except: A[i][j] = 0
    for i in range(0, nEq):
        B[i][0] = const[i]
    return (A, B, var)
```

Main...

[9]:
```
eq = ["3x + 6y - 4z = -3",
      "5 - 5z + y = x",
      "7x + 9y - 16z = 0"]
(A, B, var) = getMatrices(eq)
A = matrix(A)
B = matrix(B)
```

[10]:
```
# Method 1
print("\nMethod 1 solutions:")
X = linalg.inv(A)*B
for i in var:
    print("{0} = {1}".format(i, X[var[i], 0]))
# Method 2
print("\nMethod 2 solutions:")
X = linalg.solve(A, B)
for i in var:
    print("{0} = {1}".format(i, X[var[i], 0]))
# Method 2
print("\nMethod 2 solutions:")
X = A**(-1)*B
```

```
for i in var:
    print("{0} = {1}".format(i, X[var[i], 0]))
```

```
Method 1 solutions:
x = 2.4967741935483874
y = -1.6322580645161289
z = 0.17419354838709689

Method 2 solutions:
x = 2.496774193548387
y = -1.6322580645161289
z = 0.17419354838709686

Method 2 solutions:
x = 2.4967741935483874
y = -1.6322580645161289
z = 0.17419354838709689
```

5. SETS OF VECTORS FROM VECTOR SPACE

a) Linear span

November 27, 2021

1 AIM

Span of a set of vectors S is the set of all vectors that can be expressed as a linear combination of the elements of S. Our aim in this record is to use linear equation system solving methods to find out whether a vector belongs to the linear span of a set of vectors, or equivalently, if a vector can be expressed as a linear combination of a set of vectors.

2 Does a given vector belong to the span of a set of vectors?

To see if a vector belongs in the span of a set of vectors, we must see if any linear combination of the set of vectors equals the given vector.

[25]:
```
import numpy as np
from sympy import *
print("\nGiven vector u:")
u = np.matrix([4, 7, 1])
print(u)

print("\nMatrix of vectors {v1, v2, v3} (vectors are the columns):")
v1 = np.matrix([[5], [-2], [7]])
v2 = np.matrix([[6], [-8], [3]])
v3 = np.matrix([[4], [7], [-1]])
V = np.hstack([v1, v2, v3])
print(V)

print("\nCoefficients of v1, v2 and v3 so that their linear combination is u:")
print(np.linalg.solve(V, u.T))
```

```
Given vector u:
[[4 7 1]]

Matrix of vectors {v1, v2, v3} (vectors are the columns):
[[ 5  6  4]
 [-2 -8  7]
 [ 7  3 -1]]
```

```
Coefficients of v1, v2 and v3 so that their linear combination is u:
[[ 0.35491607]
 [-0.20623501]
 [ 0.86570743]]
```

The presence solutions means that there is some linear combination of the set of vectors V = $\{v_1, v_2, v_3\}$ that results in the given vector i.e. $(4, 7, 1)$. Hence, the given vector is in the linear span of V.

3 Is a given vector a linear combination of a set of vectors?

[26]:
```
print("\nGiven vector y")
y = np.matrix([6, 4, 3])
print(y)

x1 = np.matrix([[1], [2], [1]])
x2 = np.matrix([[3], [1], [2]])
x3 = np.matrix([[3], [2], [1]])

print("\nMatrix of vectors {x1, x2, x3}:")
X = np.hstack([x1, x2, x3])
print(X)

print("\nCoefficients of x1, x2 and x3 so that their linear combination is y:")
print(np.linalg.solve(X, y.T))
```

```
Given vector y
[[6 4 3]]

Matrix of vectors {x1, x2, x3}:
[[1 3 3]
 [2 1 2]
 [1 2 1]]

Coefficients of x1, x2 and x3 so that their linear combination is y:
[[0.5       ]
 [0.66666667]
 [1.16666667]]
```

3.0.1 Expressing the above fact

Hence, we have that $(6, 4, 3) = \frac{1}{2}(1, 2, 1) + \frac{2}{3}(3, 1, 2) + \frac{7}{6}(3, 2, 1)$

(NOTE: 0.666666... = 2/3 and 1.1666666... = 7/6)

4 CONCLUSION

A linear combination of a set of vectors can be expressed as a system of equations, where the nth equation represents the sum of the scalar multiples of the elements in the nth position in each vector. So, for a linear combination $c_1a + c_2b = c_1(a_1, a_2) + c_2(b_1, b_2)$, equation 1's left hand side would be $c_1a_1 + c_2b_1$, and equation 2's left hand side would be $c_1a_2 + c_2b_2$. The right hand side is the nth element of the given vector, for which you should check if it lies in the linear span or not.

5. SETS OF VECTORS FROM VECTOR SPACE

b) Linear independence (method 1)

November 27, 2021

1 INTRODUCTION

If a set of vectors is linearly independent, then the only way their linear combination will equal zero is if all their scalar coefficients are zero. There are other ways to check if a set of vectors is linearly independent, based on different results. Here,I will look at two of them, and demonstrate the insufficiency of the first method...

2 Using system of linear equations

We cannot use the methods for solving equations for finding the independence of a set of vectors, since the 'solve' function in the 'linalg' module in the NumPy library will only return the trivial solution (i.e. all scalars are zero) if we use the linear equation system solving methods as follows...

[2]:
```
import numpy as np
v1 = np.matrix([[5], [-2], [7]])
v2 = np.matrix([[6], [-8], [3]])
v3 = np.matrix([[4], [7], [-1]])

print("\nMatrix of vectors {v1, v2, v3}:")
V = np.hstack([v1, v2, v3])
print(V)

print("\nVector of zeros:")
c = np.zeros((3, 1))
print(c)

print("\nCoefficients for v1, v2 and v3 if their linear combination is zero:")
print(np.linalg.solve(V, c))
```

```
Matrix of vectors {v1, v2, v3}:
[[ 5  6  4]
 [-2 -8  7]
 [ 7  3 -1]]

Vector of zeros:
[[0.]
```

```
 [0.]
 [0.]]

Coefficients for v1, v2 and v3 if their linear combination is zero:
[[ 0.]
 [-0.]
 [ 0.]]
```

The given coefficients i.e. scalars multiplied to the vectors in the given set of vectors will always be given as 0, since this is the simplest solution, and this is all the solve function looks for.

3 Checking the determinant of the vector matrix

So, to check for linear independence, we use a result. Note that this result is applicable only to vector spaces that are defined over the real number field. The result is as follows...

The set of vectors S from the vector space V defined over the field of real numbers is linearly independent if and only if $|S|$ 0, where S is the matrix constructed from the set of vectors S.

[18]:
```
import numpy as np
v1 = np.matrix([1, -2, 7])
v2 = np.matrix([6, -2, 3])
v3 = np.matrix([3, 1, -2])

print("\nMatrix of vectors V = {v1, v2, v3}:")
V = np.vstack([v1, v2, v3])
print(V)

print("\nDeterminant of V = 0?:", np.linalg.det(V) == 0)
```

```
Matrix of vectors V = {v1, v2, v3}:
[[ 1 -2  7]
 [ 6 -2  3]
 [ 3  1 -2]]

Determinant of V = 0?: False
```

Hence, we have that the set of vectors $\{(1, -2, 7), (6, -2, 3), (3, 1, -2)\}$ is linearly independent.

5. SETS OF VECTORS FROM VECTOR SPACE

c) Linear independence (method 2)

November 27, 2021

1 AIM

One result states that a set of vectors is linearly dependent if and only if a vector in the set can be expressed as a linear combination of the other vectors. Using this result, we can identify whether the set of vectors in linearly independent or not.

Support functions...

[1]:
```
from sympy import Matrix, zeros, pprint
def inputPositiveInteger(prompt):
    while True:
        try:
            i = input(prompt)
            if i == "x": return 0
            i = int(i)
            if i <= 0: i = 1/0
            return i
        except:
            print("Invalid integer, please re-enter.")
def floatInput(prompt):
    while True:
        try:
            i = float(input(prompt))
            return i
        except:
            print("Invalid number, please re-enter.")

def matrixInput(nRow, nCol):
    print("\nEnter row by row, each element in the row separated by comma...")
    A, i = zeros(nRow, nCol), 0
    while i < nRow:
        row = input("R{0}: ".format(i + 1)).split(",")
        if "x" in row: break # To stop inputting anymore
        if len(row) != nCol:
            print("ERROR: You must only enter", nCol, "per row")
            continue
        for j in range(0, nCol):
            try:
```

```
                A[i, j] = float(row[j])
            except:
                print("ERROR: Non-numeric inputs.")
                j = -1
                break
        if j != -1: i = i + 1
    return A
```

Main...

[8]:
```
M = []
print("\nINPUT VECTORS (1 ROW, 3 COLUMNS)")
for i in range(0, 4):
    M.append(matrixInput(1, 3))
print("\nYOUR VECTORS")
for i in range(0, 4):
    print(M[i])
N = Matrix(M)
print("\nMATRIX OF YOUR VECTORS")
pprint(N)
r = N.rank()
print("Rank: ", r)
if(r < 4): print("At least one vector is a linear combination of the others.")
else: print("None of these vectors are linear combinations each other.")
```

```
INPUT VECTORS (1 ROW, 3 COLUMNS)

Enter row by row, each element in the row separated by comma…
R1: 1,2,3

Enter row by row, each element in the row separated by comma…
R1: 2,5,3

Enter row by row, each element in the row separated by comma…
R1: 2,7,4

Enter row by row, each element in the row separated by comma…
R1: 3,8,9

YOUR VECTORS
Matrix([[1.00000000000000, 2.00000000000000, 3.00000000000000]])
Matrix([[2.00000000000000, 5.00000000000000, 3.00000000000000]])
Matrix([[2.00000000000000, 7.00000000000000, 4.00000000000000]])
Matrix([[3.00000000000000, 8.00000000000000, 9.00000000000000]])

MATRIX OF YOUR VECTORS
 1.0  2.0  3.0
```

```
 2.0  5.0  3.0

 2.0  7.0  4.0

 3.0  8.0  9.0
Rank:  3
At least one vector is a linear combination of the others.
```

2 CONCLUSION

To get the rank of a matrix, you need to reduce to row echelon form. To reduce to row echelon form, you need to do row operations, and row operations involve linear combination of vectors. Now, if one of the rows becomes 0 in the process, that means the vector that was in that row was equal to some linear combination of the other vectors (which is why it was cancelled out). So, since there are 4 rows, if the rank is less than 4, we know that one or more rows have been cancelled out, hence at least one vector is a linear combination of the others. Since we have that if a vector in a set of vectors can be expressed as a linear combination of the other vectors, then the set of vectors is linearly dependent, we have that the above set of vectors is linearly dependent.

5. SETS OF VECTORS FROM VECTOR SPACE

d) Plotting linear transformations

November 27, 2021

1 AIM

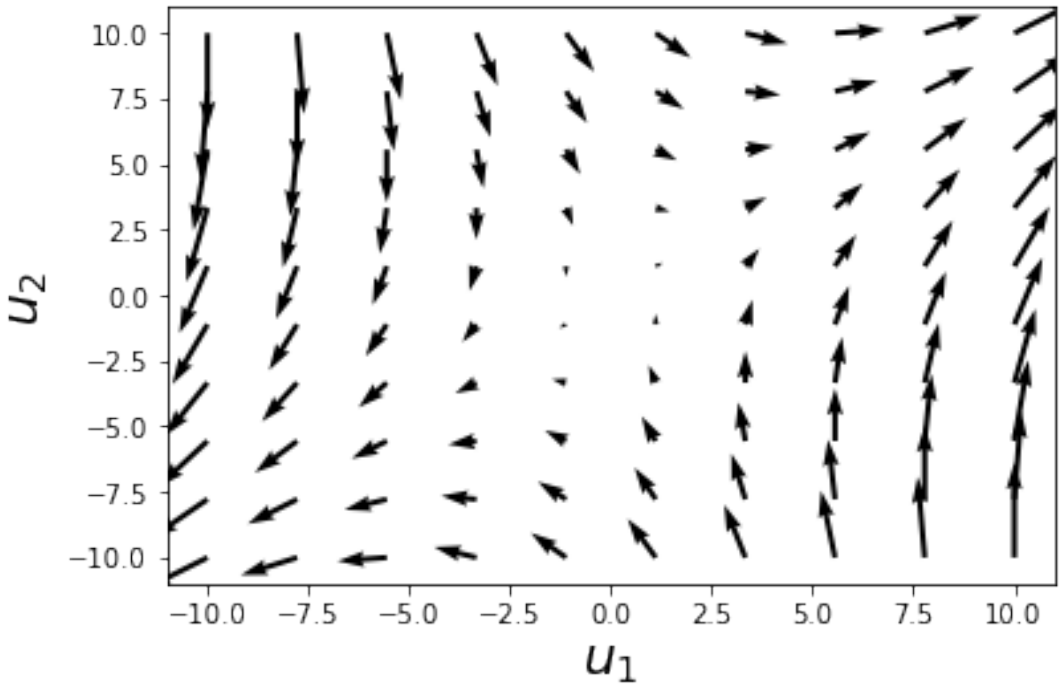
Linear transformation is a homomrphism between two vector spaces. It is a one-to-one function from a domain to a range, and hence, can be plotted. This is the focus of this record.

2 EXAMPLE

Let U and V be two vector spaces of two dimensional vectors defined over the field of real numbers. Consider the transformation $T : U - - > V$ such that $T(u) = T((u_1, u_2)) = (u_1 + u_2, 2u_1 - u_2)$

[47]:
```
import matplotlib.pyplot as plt
import numpy as np
u1, u2 = np.meshgrid(np.linspace(-10, 10, 10), np.linspace(-10, 10, 10))
v1, v2 = u1 + u2, 2*u1 - u2
plt.quiver(u1, u2, v1, v2, alpha = 1)
plt.title("\n$T(u)=T((u_1,u_2))=(u_1+u_2,2u_1-u_2)$\n", size = 20)
plt.xlabel('$u_1$', size = 20)
plt.ylabel('$u_2$', size = 20)
None
```

$$T(u) = T((u_1, u_2)) = (u_1 + u_2, 2u_1 - u_2)$$



3 CONCLUSION

Here, each set of coordinates (u_1, u_2) is a vector from the vector space U, and the vector originating from this point on the graph is the transformed vector.