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

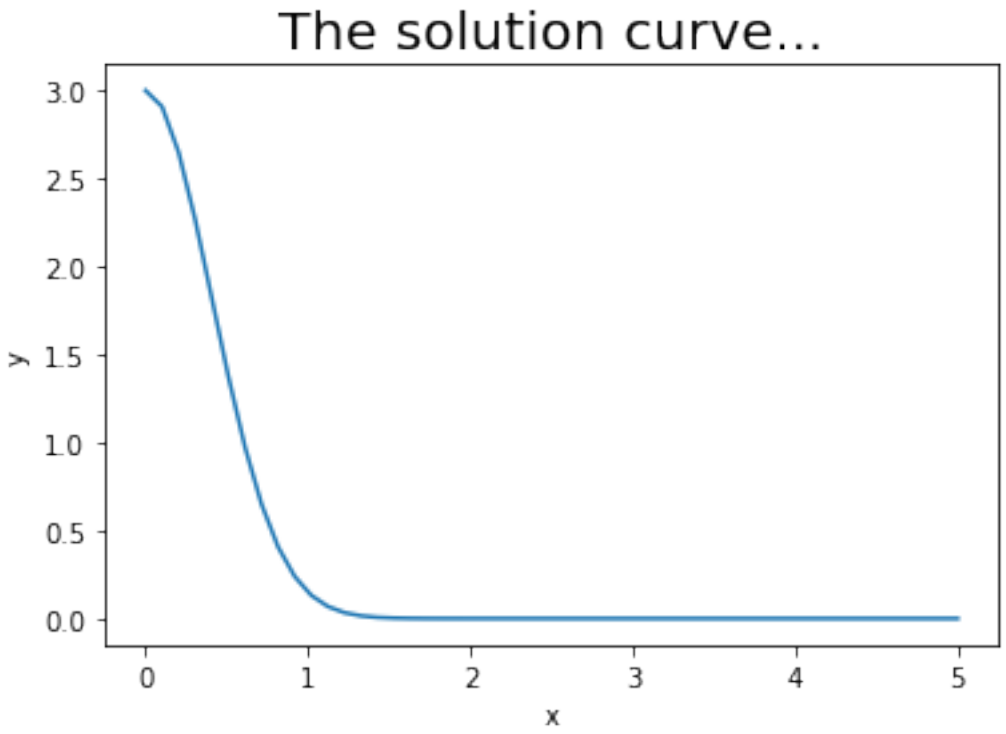
[3]:
```
from sympy import Eq, Symbol, Function, dsolve, solve

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

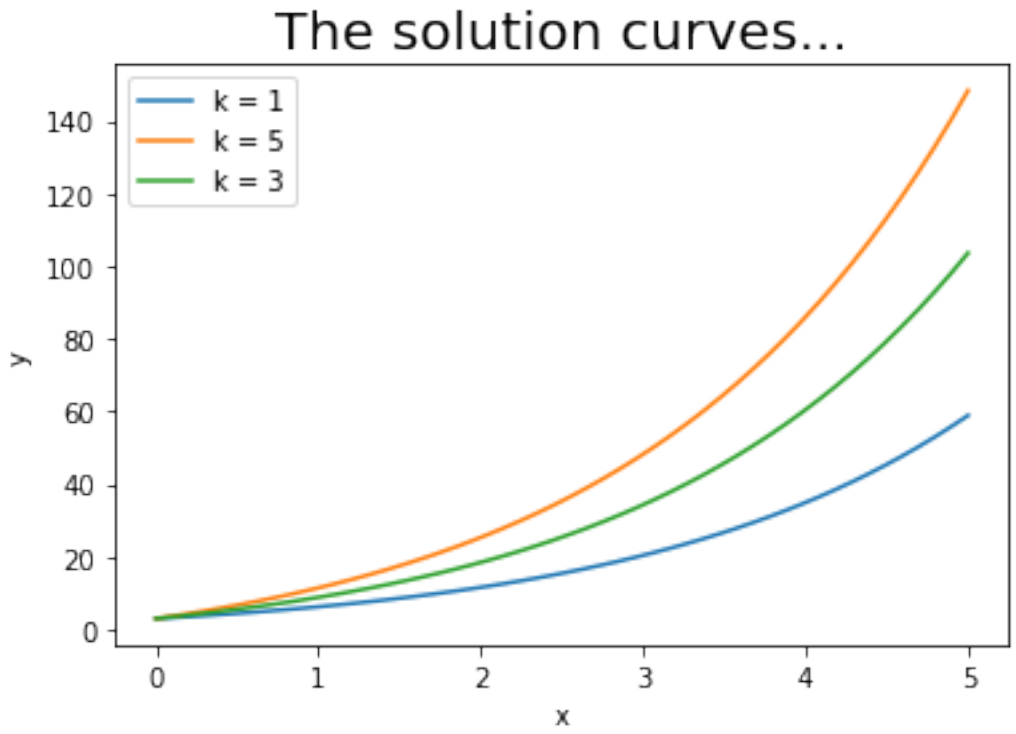
[6]:
```
from sympy import Eq, Symbol, Function, dsolve, solve

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

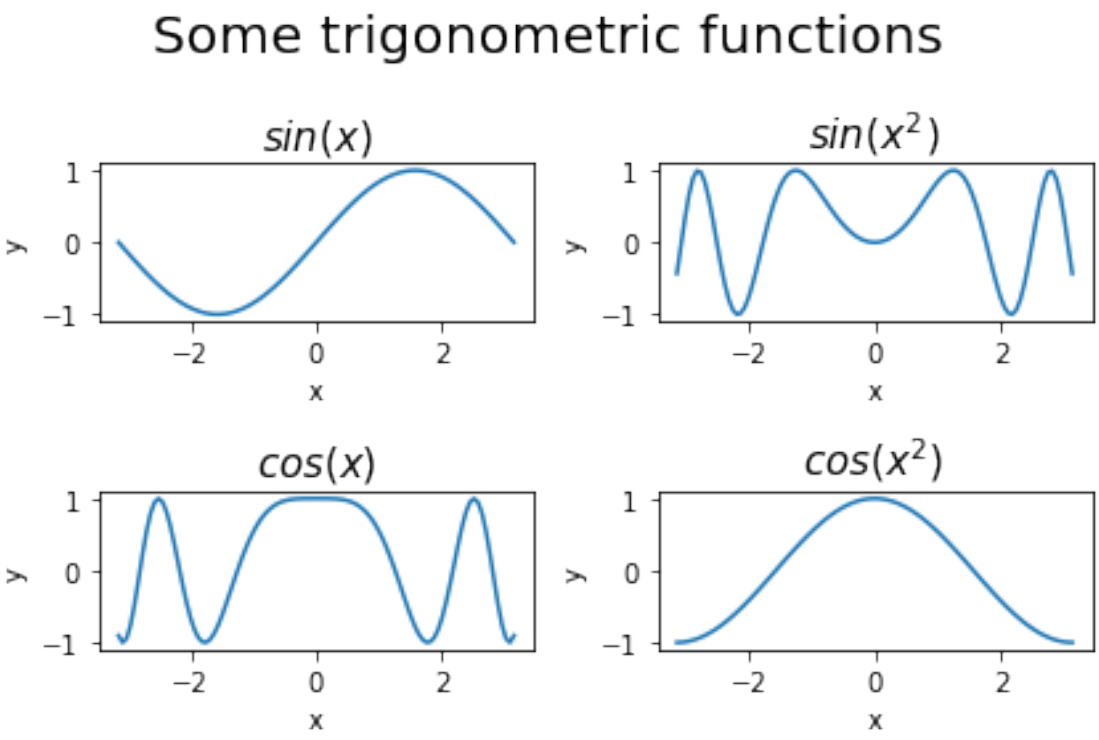
[7]:
```
from numpy import linspace
from matplotlib.pyplot import plot, title, xlabel, ylabel, legend
from scipy.integrate import odeint

# 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
```