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
```