Week2Prob

GVV Praneeth Reddy <EE21B048>

February 10, 2023

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

Imported the libraries needed to run all the cells in this notebook

1 Factorial

[2]:
```
def factorial_1(N):
    if N==0:
        return 1
    if N>0:
        return N*factorial_1(N-1)
x = int(input("Enter a number"))
print(factorial_1(x))
%timeit factorial_1(x)
```

```
Enter a number 5

120
558 ns ± 13.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

- The factorial is calculated using recursion i.e. the function factorial_1() calls itself inside its definition. So here factorial_1(0) will be 1 and factorial_1 of any number(N) greater than zero is N multiplied by factorial_1(N-1). This iterates till factorial_1(0) occurs.
- %timeit is used to find the average time taken to implement that function. Here factorial_1(5) takes 558 ns.

[3]:
```
def factorial_2(N):
    for i in range(1,N):
        N=N*i
    return N
x = int(input("Enter a number"))
print(factorial_2(x))
%timeit factorial_2(x)
```

```
Enter a number 5

120
234 ns ± 4.44 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

- Here is an other way of writing a function for factorial. In this a for loop is used to multiply the given number N by all numbers from 1 to N-1 to find the factorial.
- factorial_2(5) takes 234 ns i.e. less than half of the time taken by factorial_1(5).

2 Linear equation Solver

[2]:
```
def lin_eqn_sol(A, b):
    try:
        aug=A
        r = len(A)
        c = len(A[0])
        for i in range(r):
            aug[i].append(b[i])
        for i in range(r):
            if aug[i][i] ==0:
                a=0
                for j in range(i+1,r):
                    if a==1:
                        break
                    if aug[j][j]!= 0:
                        for k in range(r+1):
                            t=aug[i][k]
                            aug[i][k]=aug[j][k]
                            aug[j][k]= t
                            a=1
                            break
        for i in range(r):
            aug[i] = [aug[i][j]/aug[i][i] for j in range(c+1)]
            for k in range(i+1,r):
                aug[k] = [aug[k][p] - aug[i][p]*(aug[k][i]/aug[i][i]) for p in␣
 ↪range(c+1)]

        x = [0 for q in range(c)]
        for z in range(r-1,-1,-1):
            u = aug[z][c] - sum([aug[z][j]*x[j] for j in range(z+1,r)])
            if u==0 and aug[z][z]==0:
                return "The given system of equations has infinite solutions"
            elif u!=0 and aug[z][z]==0:
                return "The given system of linear equations is inconsistent"
            else:
                x[z] = u/aug[z][z]
        return x
    except ValueError:
        print("The input matrix contains elements other than numbers")
```

- This is a function to solve linear equations using gaussian elimination.
- The input parameters are a coefficient matrix A and a constant vector b.'r' is the no.of rows

and ‘c’ is the no.of columns of the coefficient matrix.

- The function first appends the constant vector to the coefficient matrix to create an augmented matrix.
- Then moves all the rows having the diagonal element as zero to the bottom. Later it does some row operations to get the row-echelon form.
- Now we go through the matrix from the bottom row to find the values of each variable and stores them to a list x.
- If condition for no solutions or infinte solutions it returns a statement saying the same otherwise it stores the solution in the list x and returns it.
- The try except is used to throw a ValueError if the input matrix contains something other than complex numbers.

[6]:
```
r = int(input("Enter the number of rows of the coefficient matrix: "))
c = int(input("Enter the number of columns of the coefficient matrix: "))

if r!=c:
    print("The coefficient matrix should be a square matrix")
else:
    print("Enter each row of the coefficient matrix with a single space between␣
 ↪the elements")
    A=[]
    for i in range(r):
        m=list(map(complex, input().split()))
        A.append(m)
    print("Enter the constant matrix with a single space between the elements")
    b=list(map(complex, input().split()))
    A_1=np.array(A)
    b_1=np.array(b)
    print(A,b)
print(lin_eqn_sol(A, b))
%timeit lin_eqn_sol(A, b)
```

```
Enter the number of rows of the coefficient matrix:  10
Enter the number of columns of the coefficient matrix:  10

Enter each row of the coefficient matrix with a single space between the
elements

 1 2 3 4 5 6 7 8 9 10
 11 10 12 36 25 98 65 32 14 12
 2 21 3 36 39 56 42 58 51 53
 10 11 23 15 14 18 19 16 32 25
 74 75 96 58 42 16 35 25 69 96
 38 39 36 54 52 50 15 14 17 18
 1 2 6 55 42 32 95 47 49 2
 88 75 94 86 53 50 12 18 19 20
 15 75 82 83 81 10 12 19 17 57
 24 26 51 57 58 49 48 20 23 24
```

```
Enter the constant matrix with a single space between the elements

 100 101 589 54 632 759 666 21 230 25

[[(1+0j), (2+0j), (3+0j), (4+0j), (5+0j), (6+0j), (7+0j), (8+0j), (9+0j),
(10+0j)], [(11+0j), (10+0j), (12+0j), (36+0j), (25+0j), (98+0j), (65+0j),
(32+0j), (14+0j), (12+0j)], [(2+0j), (21+0j), (3+0j), (36+0j), (39+0j), (56+0j),
(42+0j), (58+0j), (51+0j), (53+0j)], [(10+0j), (11+0j), (23+0j), (15+0j),
(14+0j), (18+0j), (19+0j), (16+0j), (32+0j), (25+0j)], [(74+0j), (75+0j),
(96+0j), (58+0j), (42+0j), (16+0j), (35+0j), (25+0j), (69+0j), (96+0j)],
[(38+0j), (39+0j), (36+0j), (54+0j), (52+0j), (50+0j), (15+0j), (14+0j),
(17+0j), (18+0j)], [(1+0j), (2+0j), (6+0j), (55+0j), (42+0j), (32+0j), (95+0j),
(47+0j), (49+0j), (2+0j)], [(88+0j), (75+0j), (94+0j), (86+0j), (53+0j),
(50+0j), (12+0j), (18+0j), (19+0j), (20+0j)], [(15+0j), (75+0j), (82+0j),
(83+0j), (81+0j), (10+0j), (12+0j), (19+0j), (17+0j), (57+0j)], [(24+0j),
(26+0j), (51+0j), (57+0j), (58+0j), (49+0j), (48+0j), (20+0j), (23+0j),
(24+0j)]] [(100+0j), (101+0j), (589+0j), (54+0j), (632+0j), (759+0j), (666+0j),
(21+0j), (230+0j), (25+0j)]
[(-12.186839273516398+0j), (-277.9486378443721-0j), (-99.559692892257-0j),
(496.10199269871254+0j), (-231.01607360104833+0j), (19.479560717572447+0j),
(-101.230989325415+0j), (-186.88277737895635+0j), (33.49044217633475+0j),
(192.2823391266532+0j)]
The slowest run took 4.56 times longer than the fastest. This could mean that an
intermediate result is being cached.
43.4 ms ± 18.3 ms per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- This cell is used to take input matrices for the lin_eqn_sol() function and to print the solution.
- First it asks the user for the no.of rows and columns of the coefficient matrix they are going to give if they are unequal it will say the coefficient matrix should be a square matrix.
- It will take each row of the coefficient matrix as a string with space between the elements and the splits it and converts it to a list and appends it to the the coefficient matrix. Similarly it takes the constant matrix too.
- The lin_eqn_sol() function takes 43.4 ms to solve the given 10x10 matrix.

[8]:
```
print(np.linalg.solve(A_1,b_1))
%timeit np.linalg.solve(A_1,b_1)
```

```
[ -12.18683927+0.j -277.94863784+0.j  -99.55969289-0.j  496.1019927 +0.j
 -231.0160736 +0.j   19.47956072+0.j -101.23098933+0.j -186.88277738-0.j
   33.49044218+0.j  192.28233913+0.j]
29.4 µs ± 460 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

- Here the linalg.solve() from the numpy library is used to used to solve the same set of linear equations and we can see that it just takes 29.4 μs which is very less compared to 43.4 ms.

3 Circuit

[9]:
```
def circuit_solver(filename):
    with open(filename,"r") as ckt:
        circuit=ckt.readlines()

        nodes=set()
        VS_count=0
        c=0
        e=0

        for l in circuit:
            l=l.split()
            if l[0] == ".circuit":
                c = 1
                continue
            if l[0]==".ac":
                freq=float(l[2])*2*(math.pi)
            elif l[0] == ".end":
                e = 1
                continue
            if c==1 and e==1:
                break
            if c ==1 and e==0:
                if l[0][0]=='V':
                    VS_count+=1
                if l[1]!='GND':
                    nodes.add(int(l[1]))
                if l[2]!='GND':
                    nodes.add(int(l[2]))

        n=max(nodes)
        N=n+VS_count

        A=[[complex(0) for _ in range(N)]for _ in range(N)]
        B=[complex(0) for _ in range(N)]
        c=0
        e=0
        r=n

        for l in circuit:
            l=l.split()
            if l[0] == ".circuit":
                c = 1
                continue
            if l[0]==".ac":
                w=float(l[2])
            elif l[0] == ".end":
                e = 1
                continue
            if c==1 and e==1:
                break
            if c ==1 and e==0:
                if l[0][0]=='R':
                    value=float(l[3])
                    if l[1]!='GND' and l[2]!='GND':
                        n_1=int(l[1])-1
                        n_2=int(l[2])-1
                        A[n_1][n_1]+=1/value
                        A[n_2][n_2]+=1/value
                        A[n_1][n_2]-=1/value
                        A[n_2][n_1]-=1/value
                    elif l[1]!='GND' and l[2]=='GND':
                        n_1=int(l[1])-1
                        A[n_1][n_1]+=1/value
                    elif l[1]=='GND' and l[2]!='GND':
                        n_2=int(l[2])-1
                        A[n_2][n_2]+=1/value

                if l[0][0]=='C':
                    value=float(l[3])
                    if l[1]!='GND' and l[2]!='GND':
                        n_1=int(l[1])-1
                        n_2=int(l[2])-1
                        A[n_1][n_1]+=value*freq*1j
                        A[n_2][n_2]+=value*freq*1j
                        A[n_1][n_2]-=value*freq*1j
                        A[n_2][n_1]-=value*freq*1j
                    elif l[1]!='GND' and l[2]=='GND':
                        n_1=int(l[1])-1
                        A[n_1][n_1]+=value*freq*j
                    elif l[1]=='GND' and l[2]!='GND':
                        n_2=int(l[2])-1
                        A[n_2][n_2]+=value*freq*1j

                if l[0][0]=='L':
                    value=float(l[3])
                    if l[1]!='GND' and l[2]!='GND':
                        n_1=int(l[1])-1
                        n_2=int(l[2])-1
                        A[n_1][n_1]+=1/(value*freq*1j)
                        A[n_2][n_2]+=1/(value*freq*1j)
                        A[n_1][n_2]-=1/(value*freq*1j)
                        A[n_2][n_1]-=1/(value*freq*1j)
            elif l[1]!='GND' and l[2]=='GND':
                n_1=int(l[1])-1
                A[n_1][n_1]+=1/(value*freq*1j)
            elif l[1]=='GND' and l[2]!='GND':
                n_2=int(l[2])-1
                A[n_2][n_2]+=1/(value*freq*1j)

        elif l[0][0]=='V':
            type = l[3]
            value = float(l[4])
            if type=='ac':
                phase = float(l[5])
            B[r]=value
            if l[1]!='GND' and l[2]!='GND':
                n_1=int(l[1])-1
                n_2=int(l[2])-1
                A[r][n_1]+=1
                A[r][n_2]-=1
                A[n_1][r]+=1
                A[n_2][r]-=1
            elif l[1]!='GND' and l[2]=='GND':
                n_1=int(l[1])-1
                A[r][n_1]+=1
                A[n_1][r]+=1
            elif l[1]=='GND' and l[2]!='GND':
                n_2=int(l[2])-1
                A[r][n_2]-=1
                A[n_2][r]-=1
            r+=1

        elif l[0][0]=='I':
            type = l[3]
            value = float(l[4])
            if type=='ac':
                phase = float(l[5])
            if l[1]!='GND' and l[2]!='GND':
                n_1=int(l[1])-1
                n_2=int(l[2])-1
                B[n_1]-=value
                B[n_2]+=value
            elif l[1]!='GND' and l[2]=='GND':
                n_1=int(l[1])-1
                B[n_1]-=value
            elif l[1]=='GND' and l[2]!='GND':
                n_2=int(l[2])-1
                B[n_2]+=value
        print(A)
        print(B)
    return lin_eqn_sol(A, B)
```

- The circuit_solver() function here takes a .netlist file as an argument and reads all the lines in it.
- Then it will create the coefficient matrix and constant matrix of the equations used to solve for the unknown vlues in the circuit.
- Now it will use the lin_eqn_sol() defined before to solve those equations and return the solutions.
- We initialize the coefficient and constant matrix to zeros.
- First it will calculate the no.of nodes(n) and the number of voltage sources(VS_count) and N=n+VS_count will be the no.of rows and columns in the coefficient matrix.
- Most of the required information will between .circuit and .end except the frequency. So we are we are checking each line between .circuit and .end and see which component(R,L,C,V,I) does the line correspond to and add their contribution to the coefficient matrix.
- It will print the coefficient matrix, constant martrix and the Solution matrix

[10]:
```
filename= input("Enter the file name: ")
circuit_solver(filename)
```

```
Enter the file name:  ckt1.netlist

[[(0.00125+0j), (-0.00025+0j), 0j, 0j, 0j], [(-0.00025+0j),
(0.00042500000000000003+0j), (-0.000125+0j), 0j, 0j], [0j, (-0.000125+0j),
(0.000125+0j), 0j, 0j], [0j, 0j, 0j, (0.0001+0j), (-1+0j)], [0j, 0j, 0j,
(-1+0j), 0j]]
[0j, 0j, 0j, 0j, 5.0]
```

[10]:
```
[0j, 0j, 0j, (-5+0j), (-0.0005-0j)]
```