Week 8

Anshul Vaddiraju

April 16, 2023

1 Libraries

[1]:
```
import numpy as np
import math
import cython
%load_ext Cython
```

2 Matrix Solver

[3]:
```
def Shuffle(A,B):
    n = len(A)
    for i in range(n):
            max_val = abs(A[i][i])
            row = i
            for j in range(i+1,n):
                if abs(A[j][i]) > max_val:
                    max_val = abs(A[j][i])
                    row = j
            A[row],A[i] = A[i],A[row]
            B[row],B[i] = B[i],B[row]
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B

def UpperTriangle(A,B):
    n = len(A)
    for i in range(n):
        for j in range(i+1,n):
            norm = A[j][i] / A[i][i]
            for k in range(len(A[i])): A[j][k] = A[j][k] - A[i][k] * norm
            B[j] = B[j] - B[i] * norm
            #print(A,B)
    #print("Resultant Upper Triangle:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B
```

```python
def Normalise(A,B):
    n = len(A)
    for i in range(n):
        norm = A[i][i]
        for j in range(n):
            A[i][j] = A[i][j]/norm
        B[i] = B[i]/norm
    #print("Normalised Upper Triangle:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B

def Solver(A,B):
    n = len(A)
    for i in reversed(range(n)):
        for j in range(i):
            B[j] = B[j] - B[i]*A[j][i]
            A[j][i] = A[j][i] - A[i][i]*A[j][i]

    #print("Final Matrix:")
    #print(f"A = {A}")
    #print(f"B = {B}")
    return B

def CheckSolution(a,b):
    k = []
    n = len(a)
    for i in range(n):
        if a[i][i] == 0: k.append(i)
    for j in k:
        for i in range(n):
            if a[j][i] != 0:
                norm = a[j][i]/a[i][i]
                a[j][i] = a[j][i] - a[i][i]*norm
                b[j] = b[j] - b[i]*norm

        if abs(b[j]) < 1/1000 and b[j] != 0:
            b[j] = 0
    for j in k:
        if b[j] != 0:
            print("System is inconsistent and has no solutions")
            break
        else:
            if j == k[len(k)-1]: print("System is inconsistent and has infinite␣
 ↪solutions")

def GaussianSolver(A,B):
    a,b = Shuffle(A,B)
    n = len(a)
    try:
        a,b = UpperTriangle(a,b)
        a,b = Normalise(a,b)
    except ZeroDivisionError:
        CheckSolution(a,b)
        return None
    else:
        answer = Solver(a,b)
        return answer
```

- I am using shuffle here which takes the maximum value of a column below the current row and pushes it into the current location. This is to avoid any consistent set of eqn being treated inconsistent due to a zero in the diagonal element
- Creates a upper tringular matrix from the previous matrix
- It makes all the diagonal elements of the matrix 1 which makes it easier for back substitution in the next step
- It back substitutes the matrix from the last row to the first to find the solution of equations

Here, if the diagonal element of the upper triangular is 0. It means that the matrix is not singular i.e it is not consistent. - If, the row where the diagonal element is 0 has the value of b[row] as 0 that means there are infinite solutions.
- Else, the matrix has no solutions

If the diagonal element is 0, it gives a ZeroDivisionError in the above steps. Hence, the matrix is checked for consistency.

3 Cython Code

[5]:
```
%%cython --annotate
cimport numpy as np

# to find a pivot row with the largest absolute value and sort in descending␣
 ↪order
cpdef cShuffle(list A, list B):
    cdef int i, j, n, row
    cdef int max_val
    n = len(A)
    for i in range(n):
            max_val = abs(A[i][i])
            row = i
            for j in range(i+1,n):
                if abs(A[j][i]) > max_val:
                    max_val = abs(A[j][i])
                    row = j
            A[row],A[i] = A[i],A[row]
            B[row],B[i] = B[i],B[row]
    #print(f"A = {A}")
    #print(f"B = {B}")
    return A,B

# to generate a upper triangular matrix
cpdef cUpperTriangle(list A, list B):
    cdef int i,j,k,n
    cdef complex norm
    n = len(A)
    for i in range(n):
        for j in range(i+1,n):
            norm = A[j][i] / A[i][i]
            for k in range(len(A[i])): A[j][k] = A[j][k] - A[i][k] * norm
            B[j] = B[j] - B[i] * norm
    return A,B

#  performs the back substitution step
cpdef cNormalise(list A, list B):
    cdef int i, j,n
    cdef complex norm
    n = len(A)
    for i in range(n):
        norm = A[i][i]
        for j in range(n):
            A[i][j] = A[i][j]/norm
        B[i] = B[i]/norm
    return A,B

# To perform Gaussian elimination
cpdef cSolver(list A, list B):
    cdef int i, j, n
    n = len(A)
    for i in reversed(range(n)):
        for j in range(i):
            B[j] = B[j] - B[i]*A[j][i]
            A[j][i] = A[j][i] - A[i][i]*A[j][i]
    return B

cpdef cCheckSolution(list a, list b):
    cdef list k = list()
    cdef int i,j,n
    cdef complex norm
    n = len(a)
    for i in range(n):
        if a[i][i] == 0: k.append(i)
    for j in k:
        for i in range(n):
            if a[j][i] != 0:
                norm = a[j][i]/a[i][i]
                a[j][i] = a[j][i] - a[i][i]*norm
                b[j] = b[j] - b[i]*norm

        if abs(b[j]) < 1/1000 and b[j] != 0:
            b[j] = 0
    for j in k:
        if b[j] != 0:
            print("System is inconsistent and has no solutions")
            break
        else:
            if j == k[len(k)-1]: print("System is inconsistent and has infinite␣
↪solutions")
cpdef cGaussianSolver(list A, list B):
    cdef list a,b, answer
    cdef int n
    a,b = cShuffle(A,B)
    n = len(a)
    try:
        a,b = cUpperTriangle(a,b)
        a,b = cNormalise(a,b)
    except ZeroDivisionError:
        cCheckSolution(a,b)
        return None
    else:
        answer = cSolver(a,b)
        return answer
```

```
[5]: <IPython.core.display.HTML object>
```

```
[8]: a = [[1,1],[2,1]]
     b = [5,4]
     cGaussianSolver(a,b)
```

```
[8]: [(-1+0j), (6+0j)]
```

The two codes are essentially the same algorithm, but this code is optimized using Cython to speed up the computation.

The standard code is written in Python and uses Python's built-in data structures and functions. The code uses a for loop to iterate over the rows and columns of the matrix to swap rows, perform row operations, and solve for the unknowns using back substitution. The code also contains error handling to detect infinite and no solutions.

The second code uses Cython to optimize the Python code by converting it to C code, which can be compiled to machine code for faster execution. The Cython code defines C data types for the input matrices and vectors, which allows for direct memory access and faster computation. The Cython code also uses the cdef keyword to declare variables and functions as C data types, which further improves the speed of the code.

In addition, the Cython code uses the NumPy library to create and manipulate arrays, which is faster than using Python's built-in data structures. The NumPy library uses optimized C code for array operations, which further improves the performance of the code.

Overall, the Cython code is optimized for speed and can perform matrix operations much faster than the Python code.

The following changes have been made to optimize the code using Cython:

- The functions have been defined using `cpdef` instead of `def` to allow them to be called from both Python and C. This allows the code to be compiled into C and executed more efficiently.
- Type declarations have been added to variables and arrays wherever possible. This allows Cython to generate optimized C code, reducing the amount of Python overhead and increasing performance.

4 Netlist Conversion

[10]:
```
def NetlistConvert(file_path):
    with open(file_path, 'r') as file:
            netlist = file.readlines()
    net =[]
    t = 0
    for line in netlist:
        if line[0:2] == ".a": t = 1
        if line[0:2] == ".e": t = 0
        if t == 1:
            linesplit = line.split("#")[0].split('\n')[0].split('  ')
            net.append(linesplit[0])
        if line[0:2] == ".c": t = 1
    return net
net = NetlistConvert('ckt1.netlist')
print(net)
```

```
['R1 GND 1 1e3', 'R2 1 2 4e3', 'R3 2 GND 20e3', 'R4 2 3 8e3', 'R5 GND 4 10e3',
'V1 GND 4 dc 5']
```

[11]:
```
def MatrixSizeInc(MNA,b):
            k = np.zeros((1,MNA.shape[1]))
            m = np.zeros((MNA.shape[0]+1,1))
            l = [0]
            MNA = np.vstack((MNA,k))
            MNA = np.hstack((MNA,m))
            b = np.vstack((b,l))
            return MNA,b
```

To convert a (n x n) matrix to a (n+1 x n+1) matrix

[12]:
```
def addRes(MNAdc,value,i,j):
            MNAdc[i][i] += 1/float(value)
            MNAdc[j][j] += 1/float(value)
            MNAdc[i][j] -= 1/float(value)
            MNAdc[j][i] -= 1/float(value)
            return MNAdc
```

[13]:
```
def create_MNA_matrix(netlist):
    nodes = set()
    components = []
    v_type = set()
    k = 0
    t = complex(0,1)
    freq = 0

    if netlist[-1].startswith(".ac"):
        split = netlist[-1].split()
        freq = float(split[2])*2*math.pi
    for line in netlist:
        split_line = line.split()
        if len(split_line) == 3:
            continue
        component_type = split_line[0]
        nodes.update([split_line[1], split_line[2]])
        try:
            components.append((component_type, split_line[1], split_line[2],␣
↪split_line[3],split_line[4]))
        except IndexError:
            try:
                components.append((component_type, split_line[1], split_line[2],␣
↪split_line[3]))
            except IndexError:
                components.append((component_type, split_line[1], split_line[2],␣
↪split_line[3],split_line[4],split_line[5]))



    node_dict = {node: i for i, node in enumerate(nodes)}

    num_nodes = len(nodes)
    num_components = len(components)

    MNA = np.zeros((num_nodes, num_nodes))
    b = np.zeros((num_nodes,1))
    if freq !=0:
        MNA = np.zeros((num_nodes, num_nodes))*t
        b = np.zeros((num_nodes,1))*t
```

```
for component in components:
    try:
        component_type, node1, node2, acdc ,value = component
    except ValueError:
        try:
            component_type, node1, node2,value = component
        except ValueError:
            component_type, node1, node2, acdc ,value, phase = component

    i = node_dict[node1]
    j = node_dict[node2]
    k = node_dict["GND"]
    if component_type[0] == 'R':
        MNAdc = addRes(MNA,value,i,j)

    elif component_type[0] == 'I':
        v_type.update([acdc])

        if acdc.startswith("dc"):
            b[i] -= float(value)
            b[j] += float(value)

        if acdc.startswith("ac"):
            b[i] -= float(value)
            b[j] += float(value)

    elif component_type[0] == 'V':
        v_type.update([acdc])

        if acdc.startswith("dc"):
            MNA,b = MatrixSizeInc(MNA,b)
            l = MNA.shape[0]-1
            MNA[l][i] += 1
            MNA[l][j] -= 1
            MNA[i][l] += 1
            MNA[j][l] -= 1
            b[l] -= float(value)

        if acdc.startswith("ac"):
            MNA,b = MatrixSizeInc(MNA,b)
            l = MNA.shape[0]-1
            MNA[l][i] += 1
            MNA[l][j] -= 1
            MNA[i][l] += 1
            MNA[j][l] -= 1
```

```
                b[l] -= float(value)

        elif component_type[0] == 'C':
            if freq == 0:
                print("This function cannot compute DC circuit with a␣
↪capacitance")
                return None
            MNA[i][i] += t*float(value)*freq
            MNA[j][j] += t*float(value)*freq
            MNA[i][j] -= t*float(value)*freq
            MNA[j][i] -= t*float(value)*freq

        elif component_type[0] == 'L':
            if freq == 0:
                print("This function cannot compute DC circuit with a␣
↪capacitance")
                return None
            MNA[i][i] += 1/t*float(value)*freq
            MNA[j][j] += 1/t*float(value)*freq
            MNA[i][j] -= 1/t*float(value)*freq
            MNA[j][i] -= 1/t*float(value)*freq

    b = np.squeeze(b)
    MNA = np.delete(MNA, k, axis=0)
    MNA = np.delete(MNA, k, axis=1)

    b = np.delete(b, k)
    v_type = list(v_type)

    if len(v_type) == 1 and v_type[0] == 'dc':
        return MNA, b, node_dict,0
    elif len(v_type) == 1 and v_type[0] == 'ac':
        return MNA, b, node_dict,1
    else:
        return("The code doesn't work for DC+AC supply")
```

- This code implements a function that reads an electrical netlist and creates MNAac and MNAdc and bac and bdc to solve for the voltages and currents in an AC and DC circuit.

- The function splits each line in the netlist and processes components one-by-one, updating the MNA and b arrays based on the type of component. The MNA matrices contain information about the interconnections between the components and the b arrays contain the independent source information.
- The frequency information is taken from the netlist and used in the calculation of components such as capacitors and inductors.

- The function is implemented to handle resistors, current sources, voltage sources, capacitors, and inductors, and returns None if the function encounters a DC circuit with a capacitor,inductor.

[16]:
```
net = NetlistConvert('ckt1.netlist')
V,b,nodes,check = create_MNA_matrix(net)
V = V.tolist()
b = b.tolist()
a = GaussianSolver(V,b)
del a[len(a)-1]
del nodes['GND']
if check == 0:
    print(f"DC :{a}\nNodes:{nodes}")
if check == 1:
    print(f"AC :{a}\nNodes:{nodes}")
```

```
DC :[5.0, 0.0, 0.0, 0.0]
Nodes:{'4': 0, '2': 1, '1': 2, '3': 3}
```

[15]:
```
net = NetlistConvert('ckt1.netlist')
V,b,nodes,check = create_MNA_matrix(net)
V = V.tolist()
b = b.tolist()
a = cGaussianSolver(V,b)
del a[len(a)-1]
del nodes['GND']
if check == 0:
    print(f"DC :{a}\nNodes:{nodes}")
if check == 1:
    print(f"AC :{a}\nNodes:{nodes}")
```

```
DC :[(5+0j), 0j, 0j, 0j]
Nodes:{'4': 0, '2': 1, '1': 2, '3': 3}
```

Both the solvers give same results.

[17]:
```
net = NetlistConvert('ckt1.netlist')
V,b,nodes,check = create_MNA_matrix(net)
V = V.tolist()
b = b.tolist()
%timeit GaussianSolver(V,b)
%timeit cGaussianSolver(V,b)
%timeit np.linalg.solve(V,b)
```

```
15.7 µs ± 144 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
6.51 µs ± 154 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
9.27 µs ± 65.6 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As we can see , the time taken for the standard function is **15.7 µs** whereas for linalg solver it is **9.27 µs**

Cython is much better than both of them as it gives the result in **6.51 µs**