EE2703 - Week 8

Hardik Gagrani, EE21B047

April 16, 2023

1 With Cython

[1]:
```
%load_ext Cython
```

[2]:
```
%%cython --annotate

import cython
import numpy as np
cimport numpy as np

cpdef calignment_of_rows(complex[:,:] A, complex[:] B):
    cdef int n = len(B)
    cdef int i, j, pivot
    cdef complex temp
    for i in range(n):
        pivot = i
        for j in range(i + 1, n):
            if abs(A[j,i]) > abs(A[pivot,i]):
                pivot = j
        for j in range(n):
            temp = A[i,j]
            A[i,j] = A[pivot,j]
            A[pivot,j] = temp
        temp = B[i]
        B[i] =  B[pivot]
        B[pivot] = temp

    return A, B

cpdef cupper_triangular(complex[:,:] A, complex[:] B):
    cdef int n = len(B)
    cdef int i, j, k
    cdef complex norm
    for k in range(n-1):
        for i in range(k+1, n):
            norm = A[i][k]/A[k][k]
            for j in range(k, n):
```

```
                A[i][j] = A[i][j] - norm*A[k][j]
            B[i] = B[i] - norm*B[k]
    return A, B

cpdef cback_substitution(complex[:,:] A, complex[:] B):
    cdef int n = len(B)
    cdef int i, j

    cdef np.ndarray[complex, ndim=1] X = np.empty(n, dtype=np.complex128)

    for i in range(n-1, -1, -1):
        X[i] = B[i]
        for j in range(i+1, n):
            X[i] = X[i] - A[i][j]*X[j]
        X[i] = X[i]/A[i][i]

    return X

cpdef cgauss_elimination(complex[:,:] A, complex[:] B):
    cdef np.ndarray[np.complex128_t, ndim=2] M
    cdef np.ndarray[np.complex128_t, ndim=1] N
    cdef np.ndarray[np.complex128_t, ndim=1] Solution
    calignment_of_rows(A, B)
    cupper_triangular(A, B)
    Solution = cback_substitution(A, B)
    return Solution
```

[2]: <IPython.core.display.HTML object>

The above python code is optimised using Cython.

The main changes which resulted in optimisation and lowering the running time are:

1. To allow them to be called from both Python and C, the functions were defined using cpdef rather than def. This allows the code to be compiled into C and run more quickly.
2. 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.
3. Numpy arrays have been used instead of regular Python lists. This allows Cython to use optimized Numpy functions for array manipulation, which is faster than using regular Python lists.

2 Without Cython (Linear equation Solver)-

[3]:
```
def alignment_of_rows(A, B):
    n = len(B)

    for i in range(n):
        pivot = i
        for j in range(i + 1, n):
            if abs(A[j][i]) > abs(A[pivot][i]):
                pivot = j
        A[i], A[pivot] = A[pivot], A[i]
        B[i], B[pivot] = B[pivot], B[i]

    return A, B
```

The function "alignment of rows" in this code aligns the rows of two matrices, `A` and `B`, using the absolute values of their constituent elements in the first matrix.

The first step of the function is to identify the row in the first matrix's column I that has the highest absolute value. Then, in both matrices, this row is switched with row i. For each row in the matrix, this operation is repeated.

[4]:
```
def upper_triangular(A, B):
    n = len(B)

    for k in range(n-1):
        for i in range(k+1, n):
            norm = A[i][k]/A[k][k]
            for j in range(k, n):
                A[i][j] -= norm*A[k][j]
            B[i] -= norm*B[k]
    return A, B
```

The "upper_triangular" function takes in two matrices, `A` and `B`, and converts them to an upper triangular form.

It does this by looping over all the rows (except the last row), calculating a normalization factor, and updating the elements in the current row and the vector B based on the normalization factor and the corresponding elements in a different row.

[5]:
```
def back_substitution(A, B):

    n = len(B)
    for i in range(n):
        if(A[i][i]==0):
            if(B[i]==0):
                print("The given system of equations has infinte number of␣
↪solutions")
                return
```

```
            else:
                print("The given system of equations has no solutions")
                return

    X = np.zeros(n, dtype = 'complex_')
    for i in range(n-1, -1, -1):
        X[i] = B[i]
        for j in range(i+1, n):
            X[i] -= A[i][j]*X[j]
        X[i] /= A[i][i]

    return X
```

The "back substitution" function in this code employs two matrices, `A` and `B`, to solve a system of linear equations.

The first thing the function does is determine whether the diagonal elements of the matrix A are zero. The function writes "The provided system of equations has infinte number of solutions" if they are all zero and the elements of the vector B are likewise zero. The function prints "The provided system of equations has no solutions" if any of the diagonal elements of A are zero and the corresponding element in B is non-zero.

After initialising an array X with zeros, the function solves the system of linear equations represented by A and B using back substitution.

[6]:
```
def gauss_elimination(A, B):
    A, B = alignment_of_rows(A, B)
    A, B = upper_triangular(A, B)
    Solution = back_substitution(A, B)
    return Solution
```

Finally creating a function with combination of all the functions defined above to ease the process of getting the solution.

3 Netlist Code -

[7]:
```
def read_netlist_AC(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    components = []
    nodes = set()
    frequency = 0

    for line in lines:
        if line.strip() != "":
            line = line.strip().split()
            if line[0] == '.ac':
                frequency += float(line[2])
```

```
            if line[0][0] == 'R':
                components.append(('R', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'V':
                if line[3] == 'ac':
                    components.append(('V', (line[1]), (line[2]), (line[3]),␣
↪(line[4]), (line[5])))
                    nodes.update([(line[1]), (line[2])])
                else:
                    components.append(('V', (line[1]), (line[2]), (line[3]),␣
↪(line[4])))
                    nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'I':
                if line[3] == 'ac':
                    components.append(('I', (line[1]), (line[2]), (line[3]),␣
↪(line[4]), (line[5])))
                    nodes.update([(line[1]), (line[2])])
                else:
                    components.append(('I', (line[1]), (line[2]), (line[3]),␣
↪(line[4])))
                    nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'C':
                components.append(('C', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])
            elif line[0][0] == 'L':
                components.append(('L', (line[1]), (line[2]), (line[3])))
                nodes.update([(line[1]), (line[2])])

    nodes = list(nodes)
    nodes.remove('GND')
    node_dict = {nodes[i]: i for i in range(len(nodes))}

    # print(f"The components in this circuit are: {components}")
    # print(f"\nThe nodes in this circuit are: {nodes}")
    # print(f"\nThe mapping of nodes is done by: {node_dict}")
    # print(f"\nThe frequency with which the AC source is operating is␣
↪{frequency}")
    return components, nodes, node_dict, frequency
```

The read_netlist function reads a file containing circuit information and returns a list of components, a list of nodes, and a dictionary mapping each node to a unique integer.

The file is read and processed line by line, with each line added to the components list as a tuple. The nodes used in the component are added to the nodes set and the node_dict dictionary maps each node to a unique integer. The function then prints the information and returns the components, nodes, and node_dict variables.

[8]:
```
def matrix_dim_AC(components, nodes, node_dict):
    dim = len(nodes)
    for component in components:
        if component[0] == 'V':
            dim += 1

    # print(f"\nThe dimension of matrix A is {dim}\n")

    A = np.zeros((dim, dim), dtype = 'complex_')
    A = A.tolist()

    B = np.zeros(dim, dtype = 'complex_')
    B = B.tolist()

    X = np.zeros(dim, dtype = 'complex_')
    X = X.tolist()

    return dim, A, B, X
```

This code defines a function matrix_dim that takes in two arguments: components and nodes.

This function is designed to define matrix dimensions and creating them. Dimension is defined as the number of nodes + the number of independent voltage sources present in the circuit.

After obtaining the dimension, I create numpy arrays A, B, and X with the necessary dimensions and convert them to lists to facilitate matrix calculations.

[9]:
```
import cmath

def complex_number(mag, phase):
    z = cmath.rect(mag, phase * cmath.pi / 180)
    return z
```

I defined a function `complex number` which uses the cmath module's rect() method to convert magnitude and phase arguments into a complex number. Before passing the phase to the rect() method, which returns the corresponding complex number, it is converted from degrees to radians. This generated complex number is returned by the function.

[10]:
```
def Inductor_impedence(value, frequency):
    mag = value*2*cmath.pi*frequency
    phase = 90
    X_L = complex_number(mag, phase)
    return X_L
```

The `Inductor_impedence` function is used to calculate the Impedance of the respective Inductor. It takes in two arguments: `value` = inductor value, `frequency` = frequency at which the AC circuit is operated, and calculates and returns the inductor's impedance.

[11]:
```
def Capacitor_impedence(value, frequency):
    mag = 1/(value*2*cmath.pi*frequency)
    phase = -90
    X_C = complex_number(mag, phase)
    return X_C
```

The `Capacitor_impedence` function is used to calculate the Impedance of the respective Capacitor. It takes in two arguments: `value` = capacitor's value, `frequency` = frequency at which the AC circuit is operated, and calculates and returns the capacitor's impedance.

[12]:
```
def construct_matrices_AC(dim, A, B, X, components, node_dict, frequency):

    Both_AC = 0
    Both_DC = 0

    for component in components:
        if component[0] == 'R':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[3])
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

        elif component[0] == 'V' and component[3] == 'dc':
            Both_DC += 1
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            value = float(component[4])
            if component[1] != 'GND':
                A[i][dim-1] -= 1
                A[dim-1][i] += 1
            if component[2] != 'GND':
                A[j][dim-1] += 1
                A[dim-1][j] -= 1
            B[dim-1] += value

        elif component[0] == 'V' and component[3] == 'ac':
            Both_AC += 1
            if component[1] != 'GND':
```

```
            i = node_dict[component[1]]
        if component[2] != 'GND':
            j = node_dict[component[2]]
        magnitude = float(component[4])
        phase = float(component[5])
        value = complex_number(magnitude, phase)
        if component[1] != 'GND':
            A[i][dim-1] -= 1
            A[dim-1][i] += 1
        if component[2] != 'GND':
            A[j][dim-1] += 1
            A[dim-1][j] -= 1
        B[dim-1] += value

    elif component[0] == 'I' and component[3] == 'dc':
        if component[1] != 'GND':
            i = node_dict[component[1]]
        if component[2] != 'GND':
            j = node_dict[component[2]]
        value = float(component[4])
        if component[2] != 'GND':
            B[j] -= value
        if component[1] != 'GND':
            B[i] += value

    elif component[0] == 'I' and component[3] == 'ac':
        if component[1] != 'GND':
            i = node_dict[component[1]]
        if component[2] != 'GND':
            j = node_dict[component[2]]
        magnitude = float(component[4])
        phase = float(component[5])
        value = complex_number(magnitude, phase)
        if component[2] != 'GND':
            B[j] -= value
        if component[1] != 'GND':
            B[i] += value

    elif component[0] == 'C':
        if component[1] != 'GND':
            i = node_dict[component[1]]
        if component[2] != 'GND':
            j = node_dict[component[2]]
        Cap = float(component[3])
        value = Capacitor_impedence(Cap, frequency)
        if component[1] != 'GND':
            A[i][i] += 1/value
```

```
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

        elif component[0] == 'L':
            if component[1] != 'GND':
                i = node_dict[component[1]]
            if component[2] != 'GND':
                j = node_dict[component[2]]
            Ind = float(component[3])
            value = Inductor_impedence(Ind, frequency)
            if component[1] != 'GND':
                A[i][i] += 1/value
            if component[1] != 'GND' and component[2] != 'GND':
                A[i][j] -= 1/value
                A[j][i] -= 1/value
            if component[2] != 'GND':
                A[j][j] += 1/value

    return A, B, Both_DC, Both_AC
```

This function takes the following arguments: `dim`, matrices `A`, `B`, and `X`, `components`, `node dict`, and the `frequency` at which the circuit is stimulated.

The function modifies the values of A, B, and X for each component in components to reflect the component's behaviour in the electrical circuit. A resistor, capacitor, inductor, DC voltage or current source, or AC voltage or current source can be used as the component. The `Capacitor_impedence` and `Inductor_impedence` functions are used to calculate the impedance of a capacitor and an inductor, respectively. The `complex_number` function is used to combine the magnitude and phase of an AC voltage and current source into a complex number.

[13]:
```
def final_AC(filename):
    comp, node, node_dict, freq = read_netlist_AC(filename)
    inv_dict = dict(zip(node_dict.values(), node_dict.keys()))
    dim, A, B, X = matrix_dim_AC(comp, node, node_dict)
    A_final, B_final, DC_Sources, AC_Sources = construct_matrices_AC(dim, A, B,␣
↪X, comp, node_dict, freq)

    if DC_Sources>0 and AC_Sources>0:
        return "The given circuit has more than 1 frequency and therefore it␣
↪can't be solved using this method"
    else:
        return A_final, B_final
```

This is the `final_AC` function, the combination of all the earlier defined functions, which performs an AC analysis on an electrical circuit. The function reads the circuit information from the netlist file and takes a filename as input. The function `read_netlist` AC is used to extract information from

the circuit such as components, nodes, node mapping, and frequency. The `matrix_dim_AC` function is used to determine the dimension of the matrices A, B, and X required for circuit simulation. The `construct_matrices_AC` function modifies these matrices to reflect the behaviour of electrical circuit components.

[14]:
```
L, Q = final_AC("ckt1.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
17.6 µs ± 569 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by numpy function is
7.7 µs ± 163 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
2.16 µs ± 41.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As seen above the time taken by general python function is **16.5 us** and time taken by numpy linear solver function is **9.3 us**

Whereas on the other hand the time taken by my optimised python function using cython is just **2.18 us**.

[15]:
```
final_AC("ckt2.netlist")
```

[15]:
```
"The given circuit has more than 1 frequency and therefore it can't be solved
using this method"
```

[16]:
```
L, Q = final_AC("ckt3.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
```

```
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
23.9 µs ± 511 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

The time taken by numpy function is
17.2 µs ± 200 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
2.3 µs ± 40.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[17]:
```
L, Q = final_AC("ckt4.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
10.8 µs ± 65.1 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by numpy function is
6.41 µs ± 65 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
2 µs ± 69.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[18]:
```
L, Q = final_AC("ckt5.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
4 µs ± 165 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
The time taken by numpy function is
5.95 µs ± 278 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
1.9 µs ± 21.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[19]:
```
L, Q = final_AC("ckt6.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
11.8 µs ± 219 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by numpy function is
7.04 µs ± 111 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
2.09 µs ± 33.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[20]:
```
L, Q = final_AC("ckt7.netlist")

print(f"The time taken by python code is ")
%timeit gauss_elimination(L, Q)

print(f"\nThe time taken by numpy function is ")
%timeit np.linalg.solve(L, Q)

L = np.array(L).astype('complex128')
Q = np.array(Q).astype('complex128')
print(f"\nThe time taken by Cython code is ")
%timeit cgauss_elimination(L, Q)
```

```
The time taken by python code is
1.95 µs ± 57.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by numpy function is
5.54 µs ± 237 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

The time taken by Cython code is
1.84 µs ± 79.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

4 CONCLUSION -

It is well clear from the above outputs that Cython optimised code is way faster than Python code. The reasons can be:

1. C-level Optimisation: This reduces much of the running time
2. Access to C Libraries: Cython can easily call C functions and use C libraries, which can provide significant speed advantages over pure Python code.

To conclude Cython provides us a way to write our code in much more effecient way that is closer to machine. But however it can't be a replacement to python because it requires more effort to write and debug and well it may not be necessary for all types of programs and functions