EE2703 Week 8

ANIRUDH B S (EE21B019)

April 15, 2023

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

[ ]:
```
import sys
import numpy as np
cimport numpy as np
import cmath
import cython
```

The above set of lines essentially imports the necessary set of libraries needed to run the program. If not installed, kindly install the above and proceed. It is interesting to note that numpy has been imported with both **import** and **cimport** as np. Though the name is the same, the compiler ensures that C versions run with the cimported numpy and the python aspects of the code run with the pythonic numpy. There is a slight difference between both in aspects of speed of computation and thus, it is necessary to import numpy as both cimport numpy and import numpy

[ ]:
```
@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.cdivision(True)
cpdef GaussianElimination(np.complex128_t [:,:] A,np.complex128_t [:] b):␣
 ↪#Solves a system of linear equations given matrices A and b using Gaussian␣
 ↪Elimination with parital pivoting (swapping of rows included)
    cdef int n,m, i, j,k
    n = A.shape[0]
    m = A.shape[1]
    if m!=n :
        sys.exit("Inconsistent Equation")
    cdef np.complex128_t [:,:] aug = np.array([[0 for j in range(n+1)] for i in␣
 ↪range(n)],dtype = complex)
    cdef np.complex128_t [:] x = np.array([0 for i in range(n)], dtype =␣
 ↪complex)  #Solution is initialised to zeros
    cdef complex r
    aug[:,:-1] = A
    aug[:,-1] = b     #Create the augumented matrix [A|b] by concatenating A and␣
 ↪b column wise
    try :
        for i in range(n-1):
            if aug[i][i] == 0: #Checks if pivot is zero
                for l in range(i+1, n):
```

```
                    if abs(aug[l][i]) > abs(aug[i][i]):
                        aug[i], aug[l] = aug[l], aug[i] #Swaps two rows of the␣
↪augmented matrix
            for j in range (i+1,n):
                if aug[i][i] == 0 :
                    raise ZeroDivisionError("Singular Matrix") #Raises exception
                else :
                    r = aug[j][i]/aug[i][i] #Find the ratio for each row
                for k in range(i,n+1):
                    aug[j][k] = aug[j][k] - r*aug[i][k] #Convert matrix to Row␣
↪Echelon Matrix
        if aug[n-1][n-1] == 0:
            raise ZeroDivisionError("Singular Matrix") #Raises exception
        else :
            x[n-1] = aug[n-1][n]/aug[n-1][n-1] #Start of Back Substitution
        for i in range(n-2,-1,-1):
            x[i] = aug[i][n]
            for j in range(i+1,n):
                x[i] = x[i] - aug[i][j]*x[j] #Back Substitution
            try :
                x[i] = x[i]/aug[i][i]
            except :
                print("Singular Matrix")
        return np.array(x, dtype = complex)
    except ZeroDivisionError:
        print("Singular Matrix")
```

0.0.1 Regarding Gaussian Solver

The following changes have been made to the Gaussian Solver - - Inclusion of compiler directives such as boundscheck, nonecheck and cdivision promote speed up of code. These have been explained below in greater depth as to what effect each one has. - cpdef GaussainElimination is necessary so that the function is compatible with both C and Python interfaces. Is there a tradeoff with speed for compatibility - NO ! - Now, the formal parameters are defined as np.complex128_t [:,:] for a 2D array and [:] for a 1D array. This means that we are taking a 2D complex np array A and 1 D complex np array B as input and solving the system of equations. - I have defined datatype int for n, m, i, j, k which are dimension variables (m,n) and looping variables (i,j,k) respectively. This is important as in C, data type is always specified. - Interestingly, the ratio r is defined as complex datatype. How does C handle a datatype it never saw ? The C-level complex number consists of two components, a real component and an imaginary component, which are stored as consecutive double precision floating-point values in memory. The real component is stored first, followed by the imaginary component. This means that the complex number is stored as an array of two double precision floating-point values. - The rest is simply Python code of Assignment 2, written in C by adding data types (on a higher level)

[ ]:
```
cdef int n = 0              #Number of nodes in circuit
cdef int k = 0              #Number of independent voltage sources in circuit
```

```
cdef list r = []           #Stores list of resistance objects
cdef list v = []           #Stores list of voltage sources objects
cdef list cur = []         #Stores list of current source objects
cdef list c = []           #Stores list of capacitor objects
cdef list ll = []          #Stores list of inductor objects
cdef dict nodes = {}       #Stores dictionary mapping of nodes
cdef set w = set()         #Stores set of distinct frequencies
cdef float frequency  = 0 #Stores last entered frequency of source
cdef float phase = 0       #Stores phase of complex quantity
```

The above set of lines essentially declare all the global variables to be used in the code as C variables. This is done with the cdef keyword. Interesting to note that int and float data types exist in both C and Python, however, list and dict variables do not exist in C. This is explained below - - When you use the cdef list statement in Cython, it creates a Python list object that is backed by a C array of pointers. The list object can hold any type of Python object, but the elements of the underlying C array must all be pointers of the same type. - When you use the cdef dict statement in Cython, it creates a Python dictionary object that is backed by a C data structure. The keys and values of the dictionary can be any Python object, but the keys must be hashable and comparable for equality. - set data structure exists both in C and Python and works on Binary Search Trees to enhance speed.

0.0.2 Why make variable definitons C ?

Python is significantly slower because of the overhead of Python's dynamic type checking and function call dispatching. In this area, C performs much better - static variable defintion and known function calls help achieve lesser run time. Thus, we can speed up this code significantly by using Cython and declaring the types of our variables.

[ ]:
```
cdef class R:              #Created a class for storing R objects nicely
    cdef float value
    cdef (int, int) nodes
    def __init__(self, int n1, int n2, float value) :
        self.value = value
        self.nodes = (n1, n2)
    cpdef (int, int) getnodes(self):
        return self.nodes
    cpdef float getvalue(self):
        return self.value

cdef class L:             #Created a class for storing L objects nicely
    cdef float value
    cdef (int, int) nodes
    def __init__(self, int n1, int n2, float value) :
        self.value = value
        self.nodes = (n1, n2)
    cpdef (int, int) getnodes(self):
        return self.nodes
    cpdef float getvalue(self):
        return self.value
```

```
cdef class C:            #Created a class for storing C objects nicely
    cdef float value
    cdef (int, int) nodes
    def __init__(self, int n1, int n2, float value) :
        self.value = value
        self.nodes = (n1, n2)
    cpdef (int, int) getnodes(self):
        return self.nodes
    cpdef float getvalue(self):
        return self.value

cdef class V:            #Created a class for storing V objects nicely
    cdef complex value
    cdef (int, int) nodes
    def __init__(self, int n1, int n2,complex value) :
        self.value = value
        self.nodes = (n1, n2)
    cpdef (int, int) getnodes(self):
        return self.nodes
    cpdef complex getvalue(self):
        return self.value

cdef class I:            #Created a class for storing I objects nicely
    cdef complex value
    cdef (int, int) nodes
    def __init__(self, int n1, int n2, complex value) :
        self.value = value
        self.nodes = (n1, n2)
    cpdef (int, int) getnodes(self):
        return self.nodes
    cpdef complex getvalue(self):
        return self.value
```

0.0.3 Reasons to make the above 'signifcantly visible' changes

- As visible, the definition of class has been defined using keyword cdef making each element R, L, C, V and I c-type classes rather than the usual pythonic classes. This helps in significant speed up.C classes contribute to speedup in three ways -
  - Performance - C classes are much faster compared to Python classes
  - Type Checking - C performs data type matching and then performs operations
  - Memory Management - C gives more freedom to clear up memory and use memory efficiently over python
- Another visible change is the type definition of variables which has been justified earlier.
- Python has tuples, how does C use tuples ? The tuple is stored in memory as a contiguous block of two integers. The first integer represents the first element of the tuple, and the second integer represents the second element of the tuple. Here, (int, int) is a tuple
- The **init**() is defined as a python function (using **def**). This is because class type functions

(called constructors in C) **cannot** be defined using cdef or cpdef. Thus, they need to be defined as def

- The getnodes() and getvalue() functions are cpdef so that both C and Python type variables, and functions can access them. The main question is why getnodes() or getvalue(), doesn't object.value work ? Yes, it doesn't work. This is because the classes are C classes and variables **cannot** be accessed outside of the class (by default). Thus, the getnodes() and getvalue() functions are defined, using cpdef()

0.0.4 Reasons for adopting an OOP based approach

I have adopted a classical OOP form of approaching this problem by creating classes and respective objects and storing them. This is because this would make my code look neater and more presentable. Moreover, it is quite intuitive from the name of the function or class itself the function it is meant to serve. Thus, the OOP form provides an edge over all other possible forms of approaching the given problem.

[ ]:
```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
cdef int findckt(list lines) : #Created a function to find .circuit in netlist
    cdef str line
    cdef int n
    cdef int i
    n = len(lines)
    for i in range(n) :
        line = lines[i]
        if line == ".circuit" or line == ".circuit\n" :
            return i
    return -1

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
cdef int findend(list lines) : #Created a function to find .end in netlist
    cdef str line
    cdef int n
    cdef int i
    n = len(lines)
    for i in range(n) :
        line = lines[i]
        if line == ".end" or line == ".end\n" :
            return i
    return -1
```

0.0.5 Use of Wrappers to reduce redundancy in code

The wrapper functions are essentially compiler directives that direct the C compiler to ignore few cases which we assume won't occur. - boundscheck(False) - If set to False, Cython is free to assume

that indexing operations in the code will not cause any IndexErrors to be raised. - wraparound(False) - In C, negative indexing is not supported. If set to False, Cython is allowed to neither check for nor correctly handle negative indices, possibly causing segfaults or data corruption - nonecheck(False) - If set to False, Cython is free to assume that None type will not occur in the code

Now commenting about the code - - The return type is mentioned to be int and the functions take in a list as formal argument - String line, int n and int i are defined similar to the way it is defined in C - The for loop is written only for numbers i.e for i in range(10), say, for example. This is because this results in optimized code. C cannot handle iterating over a list on its own like python does, thus, instead of iterating over the list directly, I have iterated over the indices of the list which as we shall see, results in significant speed up. - The lines inside the loop are just pythonic statements and I feel that simplifying them would not contribute to significant speed up as C and Python would take almost comparable time to do the instructions.

Its now quite obvious what my aim in speeding up is. **Yes, I will make sure that all loops are written for integers which results in significant speed up along with data type specification and function definition**

[ ]:
```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
def getMR (M, tuple nodes, float value) :                #Given a resistor␣
 ↪updates matrix M
    cdef int n1, n2
    n1, n2 = nodes
    M[n1][n1] += 1/value
    M[n1][n2] -= 1/value
    M[n2][n1] -= 1/value
    M[n2][n2] += 1/value
    return M

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
def getMC(tuple nodes, M, float frequency, float value):#Given a capacitor␣
 ↪updates matrix M
    cdef int n1, n2
    n1, n2 = nodes
    M[n1][n1] += complex(0,2*np.pi*frequency*value)
    M[n1][n2] -= complex(0,2*np.pi*frequency*value)
    M[n2][n1] -= complex(0,2*np.pi*frequency*value)
    M[n2][n2] += complex(0,2*np.pi*frequency*value)
    return M

@cython.boundscheck(False)
@cython.wraparound(False)
```

```
@cython.nonecheck(False)
@cython.cdivision(False)
def getML(tuple nodes, M, b,float frequency, float value, int idx): #Given a␣
 ↪inductor updates the matrix M
    cdef int n1, n2
    n1, n2 = nodes
    global n, k
    try :
        M[n1][n1] += complex(0,1/(6.28*frequency*value))
        M[n1][n2] -= complex(0,1/(6.28*frequency*value))
        M[n2][n1] -= complex(0,1/(6.28*frequency*value))
        M[n2][n2] += complex(0,1/(6.28*frequency*value))
    except ZeroDivisionError :
        M[n1][n+k+idx] += 1
        M[n+k+idx][n2] -= 1
        M[n+k+idx][n1] += 1
        M[n2][n+k+idx] -= 1
        b[n+k+idx] = 0
    return M, b

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
def getMV (M, b, tuple nodes, complex value, int n, int idx):#Created a function␣
 ↪to change M after considering voltage sources
    cdef int n1, n2
    n1, n2 = nodes
    M[n1][n+idx] += 1
    M[n+idx][n1] += 1
    M[n+idx][n2] -= 1
    M[n2][n+idx] -= 1
    b[n+idx] = value
    return M, b

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
def getMI (b, tuple nodes, complex value) : #Created a function to change M␣
 ↪after considering current sources
    cdef int n1, n2
    n1, n2 = nodes
    b[n1] += value
    b[n2] -= value
    return b
```

0.0.6 Why are the functions defined using def and not cdef ?

This is because of the following two reasons - - Flexibility - def provides flexibility to me over cdef which restrains me to specify all the formal parameters with their respective data types. This is avoided on purpose since, M and b are two dimensional complex matrices, it is difficult to specify a data type for them. Thus, def is used. - Return Type Escape - This is also one primary reason to avoid cdef here. As evident in some functions (example, getMV()) there are two objects returned. This is not permitted in C. However, Python permits the same. Thus, I have exploited this fact and used def.

0.0.7 Does this compromise on speed ?

Not significantly. There is very minimal difference in terms of speed between C and Python here, since it is a simple function call.

0.0.8 What is cdivision(True) ?

If set to False, Cython will adjust the remainder and quotient operators C types to match those of Python ints. If set to True, no checks are performed. This has up to a 35% speed penalty if set to False. Thus, we are introducing signifcant speedup owing to this.

[ ]:
```
cdef reset() : #Function to reset after all the operations with respect to a␣
 ↪netlist are completed.
    global r
    global v
    global cur
    global c
    global ll
    global n
    global k
    global w
    global frequency
    global phase
    #I have imported all the global variables used and reset them all to be zero
    nodes = {}
    r = []
    v = []
    cur = []
    c = []
    ll = []
    n = 0
    k = 0
    w.clear()
    frequency = 0
    phase = 0
```

0.0.9 Regarding reset()

The reset() function sets all variables to 0 so that the next netlist can be used without any issue. If reset() is not called then, all elements would get appended to the current netlist lists creating a mixture of both the netlists. To avoid this, we need to reset. The reset() function needs to be run before entering the next netlist. This is done by calling the reset() function after running the netlist and doing the required modifications. It is done in the main() function itself.

[ ]:
```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
cpdef main(str name):
    if name[-8:] != ".netlist" :       #Check for .netlist in ending
        print(".netlist not found")
        sys.exit("Netlist should end with .netlist")
    with open(name) as f:
        lines = f.readlines()
    cdef int cktcount                  #Used to check for number of .circuit in␣
↪netlist
    cdef int endcount                  #Used to check for number of .end in␣
↪netlist
    cdef str line                      #Used to store line of netlist
    cdef list s                        #Store words of a line of netlist
    cdef int a                         #Checks whether .circuit is present or not
    cdef int B                         #Checks whether .end is present or not
    cktcount = 0
    endcount = 0
    cdef int lv                        #Looping variable
    cdef int lenlines                  #Variable to store length og lines
    global w                          #Use the set of all W (frequencies)
    lenlines = len(lines)
    for lv in range(lenlines) :
        line = lines[lv]
        s = []
        if line == "" or line == "\n":
            continue
        s = line.split()
        if s[0] == ".ac" :            #If ac frequency is found, add it to w
            w.add(float(s[2]))
        if line == ".circuit" or line == ".circuit\n":
            cktcount +=1
        if line == ".end" or line == ".end\n":
            endcount +=1
    if cktcount > 1 : #Check for multiple .circuit
        print("Too many .circuit s")
        sys.exit("Netlist should contain only one .circuit")
    if endcount > 1 : #Check for multiple .end
```

```
        print("Too many .end s")
        sys.exit("Netlist should contain only one .end")
    a = findckt(lines)
    B = findend(lines)
    if a == -1 :
        print (".circuit not found")
    if B == -1 :
        print (".end not found")
    global nodes
    nodes = {} #Stores list of nodes in the circuit and their mapping
    cdef int count
    count = 1
    global n
    global k
    global r
    global c
    global ll
    global v
    global cur
    global frequency
    global phase
    #Using global keyword we access the lists and variables that are used␣
↪throughout the code
    cdef list circuit       #Stores the actual content of the netlist
    cdef list ele           #Parses each individual element of a line
    cdef int k_             #Runs through 1 and 2, that is the index of nodes␣
↪given a line in the netlist
    cdef str l              #Stores a line of circuit
    cdef int lenckt         #Stores length of circuit
    cdef int i              #Looping variable
    if a != -1 and B!=-1 :
        circuit = lines[a+1:B]
        lenckt = len(circuit)
        for i in range(lenckt) :
            l = circuit[i]
            ele = l.split()
            for k_ in range(1,3) :
                if ele[k_] == "GND" :       #GND is assigned a voltage of zero
                    nodes[ele[k_]] = 0
                if ele[k_] not in nodes.keys():
                    nodes[ele[k_]] = count
                    count += 1
            if l[0] == 'R': #Storing a resistor nicely
                print("Found a resistor")
                ro = R(nodes[ele[1]], nodes[ele[2]],  float(ele[3]))
                r.append(ro)
            if l[0] == 'C': #Storing a capacitor nicely
```

```
                print("Found a capacitor")
                co = C(nodes[ele[1]], nodes[ele[2]], float(ele[3]))
                c.append(co)
            if l[0] == 'L': #Storing a inductor nicely
                print("Found a inductor")
                lo = L(nodes[ele[1]], nodes[ele[2]], float(ele[3]))
                ll.append(lo)
            if l[0] == 'V': #Storing a voltage source nicely
                if ele[3] == "ac" : #AC Source
                    phase = float(ele[5])
                    value = complex(float(ele[4])*np.cos(phase),␣
↪float(ele[4])*np.sin(phase))
                elif ele[3] == "dc" : #DC source
                    value = float(ele[4])
                    w.add(0)
                else : #Checks if AC/ DC entry is left empty
                    sys.exit("Enter AC or DC")
                vo = V(nodes[ele[1]], nodes[ele[2]],value)
                v.append(vo)
                print("Found a voltage source with value: ", vo.value)
                k = k + 1
            if l[0] == 'I': #Storing a current source nicely
                if ele[3] == "ac" : #AC source
                    phase = float(ele[5])
                    value = complex(float(ele[4])*np.cos(phase),␣
↪float(ele[4])*np.sin(phase))
                elif ele[3] == "dc" : #DC source
                    value = float(ele[4])
                    w.add(0)
                else : #Checks if AC/ DC entry is left empty
                    sys.exit("Enter AC or DC")
                io = I(nodes[ele[1]], nodes[ele[2]], value)
                cur.append(io)
                print("Found a current source with value: ", io.value)
        n = len(nodes)
        cktsolver(n, k, r, c, ll, v, cur)
        reset()
```

0.0.10 Comments regarding the code snippet above

- **FAQ** - Why is main() cpdef ? main() has been kept to be compatible with both C and Python. Thus, it is cpdef(). Making C main() would compel me to take care of return type and would not allow me to use this file alone as a .py file. Making Python main() would compromise speed. Thus, the best plausible solution seemed to be in cpdef main(str name). Contrary to Assignement 2, this main takes in a file name to estimate time. If file name was not taken in as parameter, the timeit function later on would also include the time the user takes to type the file name and the user would need to type the name of the file for each of the 7000 times

timeit runs on average. To overcome this difficulty, I have resorted to this method.
- The standard Cython wrappers used throughout have also been used to speed up signifcantly such as boundscheck, wraparound, nonecheck and cdivision with appropriate True or False.
- Following the golden strategy to optimize - specify data types and make loops C, the main function looks signifantly optimized. The rest is exactly the same as Assignment 2. However, the same is documented here as well.

It can be observed that there is significant Python involvement here. This is because the base language involved here is just Python. The pythonic interactions are merely variable assignments, modifications, basic mathematical operations or function call or object creation. All these operations almost take time similar to C as in Python. Thus, more or less, no speed up can be achieved further from the lines in yellow.

For example, a if else statement in Python is structurally similar to a C if else. There may be a slight difference in time, however, this time is insignificant in comparision to the time used in loops.

0.0.11 Reason for code in the above manner

In the main() function we first check if the net list is valid and then proceed to get the elements and append them to an already existing empty list.

0.0.12 Use of a dictionary to store nodes

I have used a dictionary to store nodes. This is because the node entered by the user could either be a string like n1, n2, GND etc or integers like 1, 2, 0 etc. Either way while reading from the file, it is always a string. So extracting tokens as strings is a better approach as against directly converting them to int. Thus, I have adopted this style of coding.

[ ]:
```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
cpdef cktsolver(n, k, r, c, ll, v, cur): #Solves the circuit modified nodal␣
↪analysis Mx = b
    global w
    cdef int nlist           #Stores number of elements of a particular type at␣
↪one given time
    cdef int i               #Looping variable
    cdef int lenw = len(w)   #Number of distinct frequencies in netlist
    cdef int gnd = 0         #To check for GND specification
    cdef list nk             #List of nodes keys
    cdef int node_idx        #Looping variable
    cdef int node_values     #List of nodes values
    cdef int lenv            #Length of node_values
    global frequency
    cdef np.ndarray[np.complex128_t, ndim=2] M #Create matrix M
    cdef np.ndarray[np.complex128_t, ndim=1] b #Create matrix b
    cdef np.ndarray[np.complex128_t, ndim = 1] x
    if lenw == 1:
```

```
        frequency = next(iter(w))
        if frequency == 0:
            M = np.zeros((n+k+len(ll), n+k+len(ll)), dtype = np.complex128)  #M
↪= zero matrix
            b = np.zeros(n+k+len(ll), dtype = np.complex128)     #b = zero vector
            x = np.zeros(n+k+len(ll), dtype = np.complex128)
        else :
            M = np.zeros((n+k, n+k), dtype = np.complex128)  #M = zero matrix
            b = np.zeros(n+k, dtype = np.complex128)     #b = zero vector
            x = np.zeros(n+k, dtype = np.complex128)
        nlist = len(r)
        for i in range(nlist):
            res = r[i]
            M = getMR(M, res.getnodes(), res.getvalue())
↪#Create M for R elements
        nlist = len(c)
        for i in range(nlist):
            cap = c[i]
            M = getMC(cap.getnodes(), M, frequency, cap.getvalue())
↪#Create M for C elements
        nlist = len(ll)
        for i in range(nlist):
            ind = ll[i]
            M, b = getML(ind.getnodes(), M, b, frequency, ind.getvalue(), i)
↪#Create M, b for L elements
        nlist = len(v)
        for i in range(nlist):
            vol = v[i]
            M, b = getMV(M, b, vol.getnodes(), vol.getvalue(), i, n)
↪#Create M, b for V elements
        nlist = len(cur)
        for i in range(nlist) :
            curr = cur[i]
            b = getMI(b, curr.getnodes(), curr.getvalue())
↪#Create b for I elements
        M[0] = 0 #Set row corresponding to GND to 0
        M[0][0] = 1 #Set the matrix entry corresponding to GND to 1
        b[0] = 0
        for _ in nodes.keys():
            if _ == 'GND' :
                gnd = 1
        if gnd != 1:
            sys.exit("No GND specified")
        try :
            x = GaussianElimination(M,b)     #Solves the set of equations
↪generated
```

```
        except :
            sys.exit("No solution")          #Throws error that circuit is invalid
        nk = list(nodes.keys())
        node_values = len(nodes.values())
        lenv = len(v)
        if frequency == 0 :
            for node_idx in range(node_values) :
                print(f"The voltage at node {nk[node_idx]} is {x[node_idx].
↪real}")
            for i in range(lenv):
                vol = v[i]
                print(f"The current through voltage source {vol.getvalue()} V␣
↪between nodes {vol.getnodes()} is {x[n+v.index(vol)].real}")
        else :
            for node_idx in range(node_values) :
                print(f"The voltage at node {nk[node_idx]} is {abs(x[node_idx])}␣
↪at an angle {cmath.phase(x[node_idx])} with source.")
            for i in range(lenv):
                vol = v[i]
                print(f"The current through voltage source {vol.getvalue()} V␣
↪between nodes {vol.getnodes()} is {abs(x[n+v.index(vol)])} at an angle {cmath.
↪phase(x[n+v.index(vol)])} with source.")
    else :
        print("Enter a single frequency ONLY.")
```

0.0.13 About cpdef cktsolver() -

As evident, the wrapper functions have been called to promote speed up and reduce redundancy as explained earlier.

In continuation with the set trend, this function too has been made cpdef to promote interactability with both C and Python libraries. In particular, I need this function to work well at a good speed with the sys library to handle exceptions, thus, I wanted advantages of both the languages C and Python. Thus using cpdef is justified.

np.ndarray[np.complex128_t, ndim=2] helps in creating M and b which are 2 D Complex matrices. This is needed in this situation to optimize both space and time complexity. Also depending on whether its AC or DC, the dimensions of M and b are set. This is because in MNA, if DC, inductor is short which behaves like a zero volt source and needs one more row in the MNA matrix. This has been accounted for here.

Apart from that, loops have been made C type loops and variables have been defined with their data types to promote speed.

One important factor here that alters speed is np.linalg.solve(). Since this checks through various tests before giving the final answer, it is a rate determining factor in the code. Thus, this governs the speed up here. This has been left as it is without any modification as I prefer accuracy over speed. Numpy will give accurate answers though slightly slow. This slight sluggishness has actually been taken care by cimporting numpy !

The rest is just consistent with Assignement 2 with no changes in the overall structure of the code.

0.0.14 Cktsolver()

This function essentially develops the matrix M and b required to solve and also solves it. A separate function is definitely needed because on change of input, we need to recall this segment of code to solve for updated values. Thus, I have created it as a separate function.

cktsolver() also checks for a single frequency as this method of solving works only for single frequency sources.

0.0.15 How are problem scenarios solved in this code ?

- The first two problem scenarios are actually not a problem because a loop full of voltage sources and a node full of current sources are by definition inconsistent, they have either no solution if KCL or KVL is violated or infinite solution if KCL or KVL is satisfied.
- The third scenario of AC and DC is resolved by keeping a set w that keeps tracks of all frequencies, that includes DC whAich logically is AC with zero frequency.
- The fourth scenario of syntax errors have been appropriately addressed - like .netlist, .cirucit, .end and absence of GND. All these have been shown to work using netlists defined by me.

0.0.16 Additional Note : Regarding convention for V and I sources

- It is assumed that V n1 n2 means Vn1 - Vn2 = V
- It is assumed that I n1 n2 means I enters n1 and leaves n2

0.0.17 Additional Assumption :

It is assumed that magnitude of ac source by default means amplitude and not Vpp (peak to peak voltage).

0.0.18 Note on direction of current through voltage source -

The current through voltage source is the current entering into the battery through the positive terminal of the battery, consistent with passive sign convention.

0.1 Observations and Results

0.1.1 GaussianElimination

The results have been compiled below. The time taken by the Python Gaussian Elimination in Assignment 2 is mentioned in column 2, while the time taken by the Cython Gaussian Elimination implemented now is in column 3.

Run	Python (in us)	Cython (in us)
1	489	13.1
2	629	22.5
3	540	17.1

It is incredibly interesting to note that the time has almost gone down by roughly 25 times. This is a huge speed up in solving matrix equations. By cutting a few mathematical operations, mentioning

data types and defining functions in C form results in a speed up of 25 times. This is incredible and worth-noting!

0.1.2 SPICE Simulator

Netlist	Python (in us)	Cython (in us)
1	450	106
2	99.4	69.8
3	519.3	117.1
4	139	106
5	115	83.3
6	148	128
7	175.6	88.6

On an average, there is a huge speedup. Atleast it appears there is a factor of speedup of 4 for some and for some smaller netlists, the speed up is 1.5. As the netlist size increases the speed up factor increases. This can be somehow correlated to Amdahl's law in Computer Organization which states that as the number of cores increase, speed up increases. Similarly as the fraction of C content increases, speed up is higher.

0.1.3 Note -

Another Jupyter notebook has been submitted in which the code is written in one cell. Please run that notebook. Since Cython needs the code to be in ONLY one cell due to compilation, I have written everything in one cell. The above results are formulated by running that notebook only and not this notebook.

0.2 References

I referred to Cython's official documentation to get more information regarding the amazing mixture of C and Python - Cython !