EE2703: Week 8

Srikar Babu Gadipudi EE21B138

April 16, 2023

1 Prerequisites to run the notebook

A working jupyter notebook environment is expected aling with some simple libraries like numpy and cmath. Cython package is expected to convert code and compile to cython. In every section there are two sets of functions, - code that directly changes previous code to cython - code that it is optimized in cython, to perform better

1.1 Importing and Loading Cython

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

[2]:
```
import numpy as np
import cmath
```

2 Optimizing Gaussian Solver

2.1 Cython code from the previous code

[3]:
```
%%cython --annotate

def gaussian_c(A, B):

    if A.shape[0] != A.shape[1] or A.shape[0] != B.shape[0]:
        return ("Check your matrix.")
    n = A.shape[0]

    for i in range(n):

        #row swapping and checking pivot elements
        if A[i][i] == 0:
            for j in range(i, n):
                if A[j][j] !=0:
                    ls = A[i]
                    A[i] = A[j]
                    A[j] = ls
                    dum = B[i]
                    B[i] = B[j]
```

```
                    B[j] = dum

        #checking the value in the last row of the augmented matrix
        #if all zero then infinetly many solutions
        if all(l == 0 for l in A[n-1]) and B[n-1] == 0:
            return "Infinitely many solutions"
        #else if all zero in matrix A and non-zero value in B then no solution
        elif all(l == 0 for l in A[n-1]) and B[n-1] != 0:
            return "No solutions"

        #pivot element
        norm = A[i][i]
        #forward substitution for upper triangular matrix
        for j in range(n):
            A[i][j] = A[i][j]/norm
        B[i] = B[i]/norm

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

    #back substitution for row reduced echelon form
    for i in range(n-1, -1, -1):
        for j in range(i-1, -1, -1):
            norm = A[j][i]
            for k in range(n):
                A[j][k] = A[j][k] - norm*A[i][k]
            B[j] = B[j] - norm*B[i]
    #returns the matrices A in row reduced echelon form and B gives the values␣
 ↪of the variables
    return A, B
```

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

[4]:
```
A = np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
[6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]], dtype='float')
B = np.array([2, 3, 4, 1, 2, 2, 2, 9, 7, 5], dtype='float')
```

```
print(gaussian_c(A, B))

%timeit gaussian_c(A, B)
```

```
(array([[ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.]]), array([ 0.57592865,
-1.1434718 ,  1.700912  ,  1.56273285,  1.1733649 ,
        1.36712171, -1.35775437,  1.04556496, -1.97475077, -2.06722465]))
510 µs ± 12 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

2.2 Optimized Cython code

[5]:
```
%%cython --annotate

cimport numpy as np

def gaussian_c_op(double [:,:]A, double [:] B):

    # defing datatypes of the variables used
    cdef int a1 = len(A)
    cdef int a2 = len(A[0])
    cdef int b1 = len(B)
    cdef int n = a1

    cdef int i = 0
    cdef int j = 0
    cdef bint inf = True

    if a1 != a2 or a1 != b1:
         return ("Check your matrix.")

    # changed for loops to while loops
    while i < n:

        #row swapping and checking pivot elements
        if A[i][i] == 0:
            for j in range(i, n):
                if A[j][j] !=0:
                    ls = A[i]
```

```
                A[i] = A[j]
                A[j] = ls
                dum = B[i]
                B[i] = B[j]
                B[j] = dum

        #checking the value in the last row of the augmented matrix
        #if all zero then infinetly many solutions

        while j < a2:
            if A[n-1][j] != 0:
                inf = False
                break
            j += 1

        if inf == True and B[n-1] == 0:
            return "Infinitely many solutions"
        elif inf == True and B[n-1] != 0:
            return "No solutions"

        #pivot element
        norm = A[i][i]
        #forward substitution for upper triangular matrix
        for j in range(n):
            A[i][j] = A[i][j]/norm
        B[i] = B[i]/norm

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

    #back substitution for row reduced echelon form
    for i in range(n-1, -1, -1):
        for j in range(i-1, -1, -1):
            norm = A[j][i]
            for k in range(n):
                A[j][k] -= norm*A[i][k]
            B[j] -= norm*B[i]
    #returns the matrices A in row reduced echelon form and B gives the values
↪of the variables
    return A, B
```

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

[6]:
```
A = np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
[6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]], dtype='float')
B = np.array([2, 3, 4, 1, 2, 2, 2, 9, 7, 5], dtype='float')

print(gaussian_c_op(A, B))

A, B = gaussian_c_op(A, B)

%timeit gaussian_c_op(A, B)
```

```
(<MemoryView of 'ndarray' at 0x199d89a5040>, <MemoryView of 'ndarray' at
0x199d89a5380>)
19.4 µs ± 815 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[7]:
```
print(*A, *B)
```

```
<MemoryView of 'ndarray' object> <MemoryView of 'ndarray' object> <MemoryView of
'ndarray' object> <MemoryView of 'ndarray' object> <MemoryView of 'ndarray'
object> <MemoryView of 'ndarray' object> <MemoryView of 'ndarray' object>
<MemoryView of 'ndarray' object> <MemoryView of 'ndarray' object> <MemoryView of
'ndarray' object> 0.575928649457067 -1.1434717980768412 1.7009120009075736
1.5627328459907632 1.1733649044776104 1.3671217094329549 -1.3577543680410382
1.0455649580582582 -1.9747507747613304 -2.06722465307066
```

2.3 Explanation

Here optimization is done by initializing the datatypes of all variables to their required datatypes. This is done at the beginning of the function. Along with this in places of repetition of constants in a loop, we assign a new varaible to the constant and make use of this variable in the rest of the loop. Also the loops, which have `range` function have been changed to `while` loops to make the code run faster. The arguments to the function are also given specific datatypes of `double`.

Analysis in Time: | Function | Time | | ——— | ——— | | gaussian (from previous assignment) | 770 µs ± 20.9 µs | | gaussian_c | 492 µs ± 13.6 µs | | gaussian_c_op | 17.9 µs ± 663 ns |

3 DC circuit solver

3.1 Cython code from the previous code

[8]:
```
%%cython --annotate

import numpy as np

cimport numpy as np

def chan(str x):
    if x == "GND":
        x = '0'
    return x

def dc_c(filename):
    f = open(filename, "r")

    ls = f.readlines()

    #look for start and end of circuit
    for i in range(len(ls)):
        if ls[i] == ".circuit\n" or ls[i] == ".circuit": start = i+1
        if ls[i] == ".end\n" or ls[i] == ".end": end = i

    ls = ls[start:end]

    cnt=0

    arr = [[]]

    #splitting the terms in each line
    for i in range(len(ls)):
        arr.append(ls[i].split())

    arr = arr[1:]
    maxi = 0
    nv = 0

    #to calculate number of nodes and number of voltage sources
    for i in range(len(arr)):
        if arr[i][0][0] == 'V': nv += 1
        if maxi < int(chan(arr[i][1])):
            maxi = int(chan(arr[i][1]))
        if maxi < int(chan(arr[i][2])):
            maxi = int(chan(arr[i][2]))

    #initialize A and B amtrices accordingly
```

```
    A = np.zeros((maxi+nv, maxi+nv))
    B = np.zeros(maxi+nv)

    f.close()

    #go through each node and fill the matrix A and B
    for i in range(maxi):
        for j in range(len(arr)):
            #for resistive element
            if arr[j][0][0] == 'R':
                if int(chan(arr[j][1])) == i+1:
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if int(chan(arr[j][2])) != 0:
                        A[i][int(chan(arr[j][2]))-1] += -1/float(chan(arr[j][3]))
                if int(chan(arr[j][2])) == i+1:
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if int(chan(arr[j][1])) != 0:
                        A[i][int(chan(arr[j][1]))-1] += -1/float(chan(arr[j][3]))
            #for voltage source
            if arr[j][0][0] == 'V':
                if int(chan(arr[j][1])) == i+1:
                    A[i][maxi - 1 + int(arr[j][0][1])] -= 1
                if int(chan(arr[j][2])) == i+1:
                    A[i][maxi - 1 + int(arr[j][0][1])] += 1
            #for current source
            if arr[j][0][0] == 'I':
                if int(chan(arr[j][1])) == i+1:
                    B[i] -= int(chan(arr[j][4]))
                if int(chan(arr[j][2])) == i+1:
                    B[i] += int(chan(arr[j][4]))

    #auxillary equations
    for j in range(len(arr)):
        if arr[j][0][0] == 'V':
            if int(chan(arr[j][1])) != 0:
                A[maxi - 1 + int(arr[j][0][1])][int(chan(arr[j][1])) - 1] = 1
            if int(chan(arr[j][2])) != 0:
                A[maxi - 1 + int(arr[j][0][1])][int(chan(arr[j][2])) - 1] = -1
            B[maxi - 1 + int(arr[j][0][1])] = int(chan(arr[j][4]))

    return A, B, maxi, nv
```

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

[9]:
```
%%timeit
A, B, maxi, nv = dc_c("ckt3.netlist")
gaussian_c(A, B)
```

```
# print("Nodal Voltages and Auxillary Currents")
# for i in range(maxi):
#     print(f"V{i+1:} = %.5f" % B[i])
# for i in range(nv):
#     print(f"I{i+1:} = %.5f" % B[maxi + i])

# %timeit dc_c("ckt3.netlist")
```

```
240 µs ± 23 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

3.2 Optimised Cython code

[10]:
```
%%cython --annotate

import numpy as np

cimport numpy as np

def chan(str x):
    if x == "GND":
        x = '0'
    return x

def dc_op(str filename):
    f = open(filename, "r")

    ls = f.readlines()

    # defining datatypes
    cdef int maxi = 0
    cdef int nv = 0
    # variables defined for recursive use
    cdef int chk1
    cdef int chk2
    cdef float num
    cdef int a1
    cdef int c1
    cdef int c2
    cdef int c3
    # variables for indexing
    cdef int q = 0
    cdef int w = 0
    cdef int e = 0
    cdef int i = 0
    cdef int j = 0
    cdef int x = 0
```

```
# for loop converted to while loop
while q < len(ls):
    if ls[q] == ".circuit\n" or ls[q] == ".circuit": start = q+1
    if ls[q] == ".end\n" or ls[q] == ".end": end = q
    q+=1

ls = ls[start:end]

cdef int cnt=0

arr = [[]]

# splitting operation optimized
while w < len(ls):
    arr.append(ls[w].split())
    w+=1

arr = arr[1:]

# getting the number of nodes
while e < len(arr):
    a = int(chan(arr[e][1]))
    b = int(chan(arr[e][2]))
    if arr[e][0][0] == 'V': nv += 1
    if maxi < a:
        maxi = a
    if maxi < b:
        maxi = b
    e+=1

A = np.zeros((maxi+nv, maxi+nv))
B = np.zeros(maxi+nv)

f.close()

# circuit solver
while i < maxi:
    for j in range(len(arr)):

        chk1 = int(chan(arr[j][1]))
        chk2 = int(chan(arr[j][2]))
        if arr[j][0][0] == 'R':
            num = 1/float(chan(arr[j][3]))
            if chk1 == i+1:
                A[i][i] += num
                if chk2 != 0:
                    A[i][chk2-1] += -num
```

```
                if chk2 == i+1:
                    A[i][i] += num
                    if chk1 != 0:
                        A[i][chk1-1] += -num

            a1 = maxi - 1 + int(arr[j][0][1])

            if arr[j][0][0] == 'V':
                if chk1 == i+1:
                    A[i][a1] -= 1
                if chk2 == i+1:
                    A[i][a1] += 1

            if arr[j][0][0] == 'I' and arr[j][3] == "dc":
                if chk1 == i+1:
                    B[i] -= float(chan(arr[j][4]))
                if chk2 == i+1:
                    B[i] += float(chan(arr[j][4]))
        i += 1

    # auxilary equations
    while x < len(arr):
        c1 = int(chan(arr[x][1]))
        c2 = int(chan(arr[x][2]))
        c3 = maxi - 1 + int(arr[x][0][1])
        if arr[x][0][0] == 'V':
            if c1 != 0:
                A[c3][c1 - 1] = 1
            if c2 != 0:
                A[c3][c2 - 1] = -1
            if arr[x][3] == "dc":
                B[c3] = float(chan(arr[x][4]))
        x+=1

    return A, B, maxi, nv
```

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

[11]:
```
%%timeit
A, B, maxi, nv = dc_op("ckt3.netlist")
gaussian_c_op(A, B)
# print("Nodal Voltages and Auxillary Currents")
# for i in range(maxi):
#     print(f"V{i+1:} = %.5f" % B[i])
# for i in range(nv):
#     print(f"I{i+1:} = %.5f" % B[maxi + i])
```

```
# %timeit dc_op("ckt3.netlist")
```

103 µs ± 1.77 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

To check the answer and time only the circuit to matrix conversion, uncomment the comments in the timing cells.

3.3 Explanation

Similar to the gaussian solver, all the datatypes of the variables are specifies. Some of the `for` loops have been converted to `while` loops depending on their interaction with cython interface. Also all the recursive constants are now assigned to a variable to ensure that the same operation is not undergone multiple times in the same iteration in a loop.

Analysis in Time for netlist 3: | Cicuit Solver | Time | | ——— | ——— | | No Cython | 310 µs ± 12.5 µs | | Cython | 240 µs ± 6.65 µs | | Optimised Cython | 97.1 µs ± 1.62 µs |

4 AC Circuit Solver

4.1 Cython code from the previous code

[12]:
```
%%cython --annotate

import numpy as np

cimport numpy as np

def chan(str x):
    if x == "GND":
        x = '0'
    return x

def ac(filename):

    f = open(filename, "r")

    ls = f.readlines()

    fr = []

    #look for start and end of circuit and identify frequency
    for i in range(len(ls)):
        if ls[i] == ".circuit\n" or ls[i] == ".circuit": start = i+1
        if ls[i] == ".end\n" or ls[i] == ".end": end = i
        if ls[i][0:3] == ".ac":
            fr.append(i)
            chk = len(ls[i])
            for j in range(len(ls[i])):
                if ls[i][j] == '#':
```

```
                chk = j
        ls[i] = ls[i][:chk]

#to check for multiple frequencies
nl = [float(ls[fr[i]][7:]) - float(ls[fr[0]][7:]) for i in range(len(fr))]

#if not multiple frequencies
if all(i == 0.0 for i in nl):
    freq = float(ls[fr[0]][7:])
    #define omega
    omega = 2*np.pi*freq

    ls = ls[start:end]

    cnt=0

    arr = [[]]

    for i in range(len(ls)):
        arr.append(ls[i].split())

    arr = arr[1:]
    maxi = 0
    nv = 0
    #slice the array ad convert nodes accordingly
    for i in range(len(arr)):
        if arr[i][1][0] == 'n': arr[i][1] = arr[i][1][1]
        if arr[i][2][0] == 'n': arr[i][2] = arr[i][2][1]
    #finding the number of nodes and voltage sources
    for i in range(len(arr)):
        if arr[i][0][0] == 'V': nv += 1
        if maxi < int(chan(arr[i][1])):
            maxi = int(chan(arr[i][1]))
        if maxi < int(chan(arr[i][2])):
            maxi = int(chan(arr[i][2]))
    #initializing A and B matrices
    A = np.zeros((maxi+nv, maxi+nv), dtype = 'complex')
    B = np.zeros(maxi+nv, dtype = 'complex')

    f.close()

    #for each node update A and B matrices accordingly
    for i in range(maxi):
        for j in range(len(arr)):
            #if found resistor
            if arr[j][0][0] == 'R':
                if int(chan(arr[j][1])) == i+1:
```

```
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if int(chan(arr[j][2])) != 0:
                        A[i][int(chan(arr[j][2]))-1] += -1/
↪float(chan(arr[j][3]))
                if int(chan(arr[j][2])) == i+1:
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if int(chan(arr[j][1])) != 0:
                        A[i][int(chan(arr[j][1]))-1] += -1/
↪float(chan(arr[j][3]))
            #if found inductor
            if arr[j][0][0] == 'L':
                xl = float(chan(arr[j][3]))*complex(0, 1)*omega
                if int(chan(arr[j][1])) == i+1:
                    A[i][i] += 1/xl
                    if int(chan(arr[j][2])) != 0:
                        A[i][int(chan(arr[j][2]))-1] += -1/xl
                if int(chan(arr[j][2])) == i+1:
                    A[i][i] += 1/xl
                    if int(chan(arr[j][1])) != 0:
                        A[i][int(chan(arr[j][1]))-1] += -1/xl
            #if found capacitor
            if arr[j][0][0] == 'C':
                xc = 1/(float(chan(arr[j][3]))*complex(0, 1)*omega)
                if int(chan(arr[j][1])) == i+1:
                    A[i][i] += 1/xc
                    if int(chan(arr[j][2])) != 0:
                        A[i][int(chan(arr[j][2]))-1] += -1/xc
                if int(chan(arr[j][2])) == i+1:
                    A[i][i] += 1/xc
                    if int(chan(arr[j][1])) != 0:
                        A[i][int(chan(arr[j][1]))-1] += -1/xc
            #if found voltage source
            if arr[j][0][0] == 'V':
                if int(chan(arr[j][1])) == i+1:
                    A[i][maxi - 1 + int(arr[j][0][1])] -= 1
                if int(chan(arr[j][2])) == i+1:
                    A[i][maxi - 1 + int(arr[j][0][1])] += 1
            #if found current source
            if arr[j][0][0] == 'I':
                if int(chan(arr[j][1])) == i+1:
                    B[i] -= int(chan(arr[j][4]))
                if int(chan(arr[j][2])) == i+1:
                    B[i] += int(chan(arr[j][4]))
    #auxillary equations
    for j in range(len(arr)):
        if arr[j][0][0] == 'V':
            if int(chan(arr[j][1])) != 0:
```

```
                    A[maxi - 1 + int(arr[j][0][1])][int(chan(arr[j][1])) - 1] = 1
                if int(chan(arr[j][2])) != 0:
                    A[maxi - 1 + int(arr[j][0][1])][int(chan(arr[j][2])) - 1] =␣
↪-1
                if arr[j][3] == 'ac':
                    B[maxi - 1 + int(arr[j][0][1])] = int(chan(arr[j][4]))

        return A, B, maxi, nv

    #for multiple frequencies
    else:
        print("Skill issue : Multiple Frequencies")
        return 0, 0, 0, 0
```

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

[13]:
```
%%timeit
A, B, maxi, nv = ac("ckt7.netlist")

gaussian_c(A, B)

# print("Nodal Voltages and Auxillary Currents")

# for i in range(maxi):
#     print(f"V{i+1:}: Magnitude = %.5f" % cmath.polar(B[i])[0], "\t", "Phase =␣
↪%.5f" % cmath.polar(B[i])[1])
# for i in range(nv):
#     print(f"I{i+1:}: Magnitude = %.5f" % cmath.polar(B[maxi + i])[0], "\t",␣
↪"Phase = %.5f" % cmath.polar(B[maxi + i])[1])

# %timeit ac("ckt7.netlist")
```

```
90.5 µs ± 2.91 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

4.2 Optimised Cython code

[14]:
```
%%cython --annotate

import numpy as np

cimport numpy as np

def chan(str x):
    if x == "GND":
        x = '0'
    return x
```

```
def ckt2(str filename):

    f = open(filename, "r")

    ls = f.readlines()

    fr = []

    # defining datatypes
    cdef int max1
    cdef int max2
    cdef float freq
    cdef float omega
    # recursive constants to variables
    cdef int chk1
    cdef int chk2
    cdef int a1
    cdef int c1
    cdef int c2
    cdef int c3
    # indices for while loops
    cdef int q = 0
    cdef int w = 0
    cdef int e = 0
    cdef int x = 0
    cdef int i = 0
    cdef int x2 = 0

    while q < len(ls):
        if ls[q] == ".circuit\n" or ls[q] == ".circuit": start = q+1
        if ls[q] == ".end\n" or ls[q] == ".end": end = q
        if ls[q][0:3] == ".ac":
            fr.append(q)
            chk = len(ls[q])
            for j in range(len(ls[q])):
                if ls[q][j] == '#':
                    chk = j
            ls[q] = ls[q][:chk]
        q += 1

    #to check for multiple frequencies
    nl = [float(ls[fr[i]][7:]) - float(ls[fr[0]][7:]) for i in range(len(fr))]

    #if not multiple frequencies
    if all(l == 0.0 for l in nl):
        freq = float(ls[fr[0]][7:])
        #define omega
```

```
    omega = 2*np.pi*freq

    ls = ls[start:end]

    cnt=0

    arr = [[]]

    while w < len(ls):
        arr.append(ls[w].split())
        w += 1

    arr = arr[1:]
    maxi = 0
    nv = 0
    #slice the array ad convert nodes accordingly
    while e < len(arr):
        if arr[e][1][0] == 'n': arr[e][1] = arr[e][1][1]
        if arr[e][2][0] == 'n': arr[e][2] = arr[e][2][1]
        e += 1
    #finding the number of nodes and voltage sources
    while x < len(arr):
        max1 = int(chan(arr[x][1]))
        max2 = int(chan(arr[x][2]))
        if arr[x][0][0] == 'V': nv += 1
        if maxi < max1:
            maxi = max1
        if maxi < max2:
            maxi = max2
        x += 1
    #initializing A and B matrices
    A = np.zeros((maxi+nv, maxi+nv), dtype = 'complex')
    B = np.zeros(maxi+nv, dtype = 'complex')

    f.close()

    #for each node update A and B matrices accordingly
    while i < maxi:
        for j in range(len(arr)):
            #if found resistor
            chk1 = int(chan(arr[j][1]))
            chk2 = int(chan(arr[j][2]))
            if arr[j][0][0] == 'R':
                if chk1 == i+1:
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if chk2 != 0:
                        A[i][chk2-1] += -1/float(chan(arr[j][3]))
```

```
                if chk2 == i+1:
                    A[i][i] += 1/float(chan(arr[j][3]))
                    if chk1 != 0:
                        A[i][chk1-1] += -1/float(chan(arr[j][3]))
            #if found inductor
            if arr[j][0][0] == 'L':
                xl = float(chan(arr[j][3]))*complex(0, 1)*omega
                if chk1 == i+1:
                    A[i][i] += 1/xl
                    if chk2 != 0:
                        A[i][chk2-1] += -1/xl
                if chk2 == i+1:
                    A[i][i] += 1/xl
                    if chk1 != 0:
                        A[i][chk1-1] += -1/xl
            #if found capacitor
            if arr[j][0][0] == 'C':
                xc = 1/(float(chan(arr[j][3]))*complex(0, 1)*omega)
                if chk1 == i+1:
                    A[i][i] += 1/xc
                    if chk2 != 0:
                        A[i][chk2-1] += -1/xc
                if chk2 == i+1:
                    A[i][i] += 1/xc
                    if chk1 != 0:
                        A[i][chk1-1] += -1/xc
            #if found voltage source
            if arr[j][0][0] == 'V':
                a1 = maxi - 1 + int(arr[j][0][1])
                if chk1 == i+1:
                    A[i][a1] -= 1
                if chk2 == i+1:
                    A[i][a1] += 1
            #if found current source
            if arr[j][0][0] == 'I' and arr[j][3] == "ac":
                if chk1 == i+1:
                    B[i] -= float(chan(arr[j][4]))
                if chk2 == i+1:
                    B[i] += float(chan(arr[j][4]))
        i+=1
    #auxillary equations
    for j in range(len(arr)):
        c1 = int(chan(arr[j][1]))
        c2 = int(chan(arr[j][2]))
        c3 = maxi - 1 + int(arr[j][0][1])
        if arr[j][0][0] == 'V':
            if c1 != 0:
```

```
                A[c3][c1 - 1] = 1
            if c2 != 0:
                A[c3][c2 - 1] = -1
            if arr[j][3] == 'ac':
                B[c3] = int(chan(arr[j][4]))

    return A, B, maxi, nv

#for multiple frequencies
else:
    print("Skill issue : Multiple Frequencies")
    # return A, B, maxi, nv
    return 0, 0, 0, 0
```

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

[15]:
```
%%timeit
A, B, maxi, nv = ac("ckt7.netlist")

gaussian_c(A, B)

# print("Nodal Voltages and Auxillary Currents")

# for i in range(maxi):
#     print(f"V{i+1:}: Magnitude = %.5f" % cmath.polar(B[i])[0], "\t", "Phase =␣
↪%.5f" % cmath.polar(B[i])[1])
# for i in range(nv):
#     print(f"I{i+1:}: Magnitude = %.5f" % cmath.polar(B[maxi + i])[0], "\t",␣
↪"Phase = %.5f" % cmath.polar(B[maxi + i])[1])

# %timeit ac("ckt7.netlist")
```

```
91.2 µs ± 1.41 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

To check the answer and time only the circuit to matrix conversion, uncomment the comments in the timing cells.

4.3 Explanation

Similar to the gaussian solver, all the datatypes of the variables are specifies. Some of the `for` loops have been converted to `while` loops depending on their interaction with cython interface. Also all the recursive constants are now assigned to a variable to ensure that the same operation is not undergone multiple times in the same iteration in a loop. Here the constants which might involve complex numbers are left untouched as there are no particular equivalents to complex numbers in C. A similar issue is faced in gaussian solver as well. Used un-optimized gaussian solver to accomodate for complex numbers.

Analysis in Time for netlist 7: | Cicuit Solver | Time | | ——— | ——— | | No Cython | 117 µs ± 4.25 µs | | Cython | 91.9 µs ± 2.17 µs | | Optimised Cython | 90.7 µs ± 2.59 µs |