EE21B005 Week 8 Assignment

Aditya Mallick EE21B005

April 16, 2023

1 Optimization using Cython

1.0.1 Running Instructions

All cells should be run one after another, just to be safe.

[1]:
```
# importing required libraries, as well as the Cython extension
%load_ext Cython
import numpy as np
import os
import math
from timeit import default_timer as timer
```

[2]:
```
# Simple factorial finder using repeated multiplication in for loop
def simple_factorial(n):
  ans = n
  if (n > 0):
    for i in range(1,n):
      ans *= i
    return ans
  elif (n == 0):
    return 1

print(simple_factorial(3))
%timeit simple_factorial(10)
```

```
6
424 ns ± 6.57 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

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

cpdef int c_simple_factorial(int n):
  cdef int ans = n
  cdef int i
  if (n > 0):
    for i in range(1,n):
      ans *= i
    return ans
  elif (n == 0):
```

```
    return 1
```

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

To implement cython, here we have:

- Declared the type of all variables before referencing them.
- Also declared the types of the function parameters, as well as the output of the function.

We also use %%cython –annotate to see how much of our code is actually getting converted to C effectively, and how much of it remains as python, as is shown by the yellow lines. Since there is not much python interaction, we can say we have optimized it successfully.

[4]:
```
print(c_simple_factorial(3))
%timeit c_simple_factorial(10)
```

```
6
36.4 ns ± 0.41 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

As can be seen from the above demonstrations, a Cython implementation of the simple factorial function is over 10 times faster than the regular Python, showcasing the power of Cython. We will see more optimizations below.

[5]:
```
def gausselim_partialpivoting(A,B):
    n = len(A)
    ans = [0 for i in range(n)]

    # iterate through the columns
    for counter in range(n - 1):
        # partial pivoting -> Swaps rows based on magnitude of leftmost␣
 ↪element, helps prevent division by zero
        if (A[counter][counter] == 0):
            B[[counter,i]] = B[[i, counter]] # shorthand notation for swapping␣
 ↪numpy rows
            A[[counter,i]] = A[[i, counter]]

        # Forward Elimination -> Converts to an upper triangular matrix
        for i in range(counter + 1,n):
            div = A[i][counter]/A[counter][counter]
            B[i] -= B[counter]*div
            for j in range(counter, n):
                A[i][j] -= A[counter][j]*div

    # Back Substitution
    for i in range(n - 1,-1,-1):
        sum = B[i]
        for j in range(n - 1, i - 1, -1):
            sum -= A[i][j]*ans[j]
        if (A[i][i] == 0):
```

```
                if (sum == 0):
                    return('Inf solutions')
                else:
                    return('No solutions')
        else:
            ans[i] = sum/A[i][i]

    return(ans)
```

[6]:
```
# Random 10x10 matrices for solving
a = np.array([[11, 15, 42, 49, 24, 40, 26, 5, 3, 37],
     [50, 40, -47, 4, -42, -18, -39, -3, -2, -1],
     [12, 18, 43, -45, 31, 48, -18, 8, -7, -38],
     [-28, 27, -49, -35, 5, -2, -38, 26, -22, -41],
     [23, -16, -1, -4, 33, -3, 37, 43, 30, -20],
     [1, 37, -7, 30, 10, 49, 36, 2, 18, -11],
     [41, 49, 19, 8, 43, -2, 36, 16, 23, 40],
     [44, 38, 37, 18, 11, -6, -18, 22, 8, -17],
     [-10, 36, 19, 27, 2, -17, -9, 11, -16, 25],
     [2, 19, 23, 27, -12, 21, 10, -13, 9, 42]], dtype='float32')
b = np.array([-4, 33, 29, -1, -1, 19, 42, 41, -3, 5], dtype='float32')

# Displaying the solutions
x = np.array([f"X{i + 1}" for i in range (len(b))])
solution1 = gausselim_partialpivoting(a,b)
for i in range(len(solution1)):
    print(f"{x[i]} = {solution1[i]}")
```

```
X1 = 0.26712258404673056
X2 = 0.5517940385795388
X3 = 0.040929817172932786
X4 = -0.06508414056730201
X5 = 0.40668447141378694
X6 = -0.12319645439928353
X7 = -0.3608562053154241
X8 = -0.6355867820117593
X9 = 0.7939021542149043
X10 = -0.2270072645680251
```

Above is my code imported from my Week 2 implementation of Gaussian Elimination.

[7]:
```
%%cython --annotate
import cython
import numpy as np

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

```
@cython.cdivision(True)
def c_gausselim_partialpivoting(complex[:,:] A, complex[:] B):

    cdef int n;
    cdef int i;
    cdef int j;
    cdef int counter;
    cdef complex div;
    cdef complex sum;
    n = len(A)
    cdef complex[:] ans = np.zeros((n,), dtype=complex)

    # iterate through the columns
    for counter in range(n - 1):
        # partial pivoting -> Swaps rows based on magnitude of leftmost␣
 ↪element, helps prevent division by zero
        if (A[counter, counter] == 0):
            B[counter],B[i] = B[i],B[counter] # shorthand notation for swapping␣
 ↪numpy rows
            A[counter],A[i] = A[i],A[counter]

        # Forward Elimination -> Converts to an upper triangular matrix
        for i in range(counter + 1,n):
            div = A[i, counter]/A[counter, counter]
            B[i] = B[i] - B[counter]*div
            for j in range(counter, n):
                A[i, j] = A[i, j] - A[counter, j]*div

    # Back Substitution
    for i in range(n - 1,-1,-1):
        sum = B[i]
        for j in range(n - 1, i - 1, -1):
            sum = sum - A[i, j]*ans[j]

        if (A[i, i] == 0):
                if (sum == 0):
                    return('Inf solutions')
                else:
                    return('No solutions')
        else:
            ans[i] = sum/A[i, i]

    return ans
```

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

Again to implement cython, here we have done slightly more:

- Declared the type of all variables before referencing them.
- Declared the types of the function parameters.
- Had to change some regular python notation (+= and -=), since it was throwing errors during compilation.
- Stopped cython from conducting some unnecessary checks such as raising division by zero errors, none type errors, etc.
- For arrays, we use a data type called a 'typed memoryview', which is a memory buffer. These are similar to NumPy arrays, so that is what we will be using for our inputs.
- We have ensured that all matrix values are of the complex (complex double) data type, so that we can use this function later on in our circuit solver.

We again use %%cython –annotate to see how much of our code is actually getting converted to C effectively, and how much of it remains as python, as is shown by the yellow lines. Since there is not much python interaction, we can say we have optimized it successfully.

[8]:
```
import numpy as np

# Random 10x10 matrices for solving
a = np.array([[11, 15, 42, 49, 24, 40, 26, 5, 3, 37],
     [50, 40, -47, 4, -42, -18, -39, -3, -2, -1],
     [12, 18, 43, -45, 31, 48, -18, 8, -7, -38],
     [-28, 27, -49, -35, 5, -2, -38, 26, -22, -41],
     [23, -16, -1, -4, 33, -3, 37, 43, 30, -20],
     [1, 37, -7, 30, 10, 49, 36, 2, 18, -11],
     [41, 49, 19, 8, 43, -2, 36, 16, 23, 40],
     [44, 38, 37, 18, 11, -6, -18, 22, 8, -17],
     [-10, 36, 19, 27, 2, -17, -9, 11, -16, 25],
     [2, 19, 23, 27, -12, 21, 10, -13, 9, 42]], dtype=complex)
b = np.array([-4, 33, 29, -1, -1, 19, 42, 41, -3, 5], dtype=complex)

# Displaying the solutions
x = [f"X{i + 1}" for i in range (len(b))]
solution1 = np.array(c_gausselim_partialpivoting(a,b))
for i in range(len(solution1)):
    print(f"{x[i]} = {solution1[i]}")
```

```
X1 = (0.267120930438556+0j)
X2 = (0.5517925649693345-0j)
X3 = (0.04092749676737994-0j)
X4 = (-0.06508234993398034+0j)
X5 = (0.4066898005472733+0j)
X6 = (-0.1231952440427521+0j)
X7 = (-0.3608635257851941+0j)
X8 = (-0.6355890541775159+0j)
X9 = (0.7939103512386102-0j)
X10 = (-0.2270058916396415+0j)
```

[9]:
```
%timeit c_gausselim_partialpivoting(a,b)
```

```
2.22 µs ± 59.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[10]:
```
%timeit gausselim_partialpivoting(a,b)
```

```
228 µs ± 17.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Above are implementations of gaussian elimination in python and cython.

- The cython implementation is about 100 times faster than the python implementation for solving a set of 10x10 matrix equations. This time difference will only get bigger as the size of the matrices increase, but will also be smaller for smaller matrices.
- Thus, using cython we have succesfully optimized our gaussian elimination function.
- We will use this in our cythonized SPICE circuit solver.

1.1 SPICE Netlist Solver - Python

[11]:
```
def mna_builder(components, ac_freq=0):

    # nodes dictionary for using later in filling array
    nodes = {}
    x = 0
    for component in components:
        if (components[component][0] not in nodes):
            nodes[components[component][0]] = x
            x += 1
        if (components[component][1] not in nodes):
            nodes[components[component][1]] = x
            x += 1

    #  initialize A and B matrices in AX = B
    nodecount = len(nodes)
    A = np.full((nodecount, nodecount), 0, dtype = complex)
    B = np.full(nodecount, 0, dtype = complex)

    # Filling up the matrices
    for component in components:
        if component[0] == 'R':

            val = float(components[component][2])

            # KCL at node 1 and node 2 creates 4 entries for each resistor, two␣
↪+ve and two -ve
            A[nodes[components[component][0]]][nodes[components[component][0]]]␣
↪+= 1/val
```

```
            A[nodes[components[component][0]]][nodes[components[component][1]]]␣
↪+= -1/val
            A[nodes[components[component][1]]][nodes[components[component][0]]]␣
↪+= -1/val
            A[nodes[components[component][1]]][nodes[components[component][1]]]␣
↪+= 1/val

        if (ac_freq):
            # AC analysis involves complex representation of V, I, L, C
            if component[0] == 'V':

                phase = float(components[component][3])

                # V value is taken in rectangular form
                val = float(components[component][2])*(math.cos(phase) +␣
↪1j*math.sin(phase))

                # Increasing dimensions to account for additional voltage␣
↪equation
                A = np.column_stack((A, np.full(nodecount, 0)))
                A = np.row_stack((A, np.full(nodecount + 1, 0)))
                B = np.append(B, val)

                # keeping up with the dimension change
                nodecount += 1

                # auxiliary current
                A[nodes[components[component][0]]][nodecount - 1] += 1
                A[nodes[components[component][1]]][nodecount - 1] += -1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] += -1
                A[nodecount - 1][nodes[components[component][0]]] += +1

            elif component[0] == 'L':

                val = float(components[component][2])*1j*ac_freq*2*math.pi

                # Reactance of inductor = j*w*L
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] += 1/val
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] += -1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] += -1/val
```

```
            ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] += 1/val

        elif component[0][0] == 'C':

            val = 1/(float(components[component][2])*1j*ac_freq*2*math.pi)

            # Reactance of capactitor = 1/j*w*C
            ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] += 1/val
            ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] += -1/val
            ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] += -1/val
            ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] += 1/val

        elif component[0] == 'I':

            phase = float(components[component][3])

            # I value is taken in rectangualar form
            val = float(components[component][2])*(math.cos(phase) +␣
↪1j*math.sin(phase))

            # Current source only needs to be filled in B
            B[nodes[components[component][0]]] += val
            B[nodes[components[component][1]]] += -val

    else:
        # DC current and voltage sources
        if component[0] == 'V':

            val = float(components[component][2])

            # Increasing dimensions to account for additional voltage␣
↪equation
            A = np.column_stack((A, np.full(nodecount, 0)))
            A = np.row_stack((A, np.full(nodecount + 1, 0)))
            B = np.append(B, val)

            # keeping up with the dimension change
            nodecount += 1

            # auxiliary current
            A[nodes[components[component][0]]][nodecount - 1] += 1
```

```
                A[nodes[components[component][1]]][nodecount - 1] += -1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] += -1
                A[nodecount - 1][nodes[components[component][0]]] += +1


            elif component[0] == 'I':

                val = float(components[component][2])

                # Current source only needs to be filled in B
                B[nodes[components[component][0]]] += val
                B[nodes[components[component][1]]] += -val

    # Delete GND row and column in A, and element in B
    A = np.delete(np.delete(A,0,0),0,1)
    B = np.delete(B,0)



    return([A,B,nodes])
```

[12]:
```
def circuit_simulator(file):
    start = timer()
    txt = file.read().splitlines()
    ac_freq = 0

    # checking for ac sources
    for line in txt:
        if line[:3] == '.ac':
            if (line.find('#') != -1):
                line = line[:line.find('#')].strip()
            ac_freq = float(line.split(" ")[-1])

    # cleaning data
    spice_code = txt[txt.index('.circuit') + 1:txt.index('.end')]
    for line in spice_code:
        newline = line
        if (line.find('#') != -1):
            newline = line[:line.find('#')].strip()
        spice_code[spice_code.index(line)] = newline.strip().split(" ")

    # creating component dictionary
    components = {}
    for line in spice_code:
```

```
        if (ac_freq):
            if line[0][0] == 'V' or line[0][0] == 'I':
                if line[3] != 'ac':
                    print("Cannot evaluate both DC and AC sources.")
                    return 0
                else:
                    components[line[0]] = [line[1], line[2], line[4], line[5]]
            else:
                components[line[0]] = [line[1], line[2], line[-1]]
        else:
            components[line[0]] = [line[1], line[2], line[-1]]


    # get A,B matrices
    matrices = mna_builder(components, ac_freq)

    # find solution
    solution = gausselim_partialpivoting(matrices[0],matrices[1])
    end = timer()
    '''
    nodes = list(matrices[2].keys())
    # print output
    for i in range(len(solution)):
        if (i < len(matrices[2]) - 1):
            print(f"Voltage at node {nodes[i + 1]} = {solution[i]} V")
        else:
            print(f"I{i - len(matrices[2]) + 2} = {solution[i]} A")
    print(f"Time taken for Python Implementation: {end - start:e} seconds\n")
    '''
    return end - start
```

This is the code imported from my Week 2 implementation of the SPICE circuit solver.

1.2 SPICE Netlist Solver - Cython

[13]:
```
%%cython --annotate
import cython
import numpy as np
import math

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)

def c_mna_builder(dict components, double ac_freq):
```

```
    # nodes dictionary for using later in filling array
    cdef dict nodes = {}
    cdef int x = 0
    for component in components:
        if (components[component][0] not in nodes):
            nodes[components[component][0]] = x
            x += 1
        if (components[component][1] not in nodes):
            nodes[components[component][1]] = x
            x += 1

    #  initialize A and B matrices in AX = B
    cdef int nodecount = len(nodes)
    cdef complex[:,:] A = np.zeros((nodecount,nodecount), dtype=complex)
    cdef complex[:] B = np.zeros((nodecount,), dtype=complex)


    cdef complex val;
    cdef double phase;
    # Filling up the matrices
    for component in components:
        if component[0] == 'R':

            val = (components[component][2])

            # KCL at node 1 and node 2 creates 4 entries for each resistor, two␣
↪+ve and two -ve
            A[nodes[components[component][0]]][nodes[components[component][0]]]␣
↪= A[nodes[components[component][0]]][nodes[components[component][0]]] + 1/val
            A[nodes[components[component][0]]][nodes[components[component][1]]]␣
↪= A[nodes[components[component][0]]][nodes[components[component][1]]] - 1/val
            A[nodes[components[component][1]]][nodes[components[component][0]]]␣
↪= A[nodes[components[component][1]]][nodes[components[component][0]]] - 1/val
            A[nodes[components[component][1]]][nodes[components[component][1]]]␣
↪= A[nodes[components[component][1]]][nodes[components[component][1]]] + 1/val

        if (ac_freq):
            # AC analysis involves complex representation of V, I, L, C
            if component[0] == 'V':

                phase = (components[component][3])

                # V value is taken in rectangular form
                val = (components[component][2])*(math.cos(phase) + 1j*math.
↪sin(phase))
```

```
                # Increasing dimensions to account for additional voltage␣
↪equation
                A = np.column_stack((A, np.full(nodecount, 0)))
                A = np.row_stack((A, np.full(nodecount + 1, 0)))
                B = np.append(B, val)

                # keeping up with the dimension change
                nodecount += 1

                # auxiliary current
                A[nodes[components[component][0]]][nodecount - 1] =␣
↪A[nodes[components[component][0]]][nodecount - 1] + 1
                A[nodes[components[component][1]]][nodecount - 1] =␣
↪A[nodes[components[component][1]]][nodecount - 1] - 1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] = A[nodecount␣
↪- 1][nodes[components[component][1]]] - 1
                A[nodecount - 1][nodes[components[component][0]]] = A[nodecount␣
↪- 1][nodes[components[component][0]]] + 1

            elif component[0] == 'L':

                val = (components[component][2])*1j*ac_freq*2*math.pi

                # Reactance of inductor = j*w*L
               ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] =␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] + 1/val
               ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] =␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] - 1/val
               ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] =␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] - 1/val
               ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] =␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] + 1/val

            elif component[0][0] == 'C':

                val = 1/((components[component][2])*1j*ac_freq*2*math.pi)

                # Reactance of capactitor = 1/j*w*C
```

```
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] =␣
↪A[nodes[components[component][0]]][nodes[components[component][0]]] + 1/val
                ␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] =␣
↪A[nodes[components[component][0]]][nodes[components[component][1]]] - 1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] =␣
↪A[nodes[components[component][1]]][nodes[components[component][0]]] - 1/val
                ␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] =␣
↪A[nodes[components[component][1]]][nodes[components[component][1]]] + 1/val

            elif component[0] == 'I':

                phase = (components[component][3])

                # I value is taken in rectangualar form
                val = (components[component][2])*(math.cos(phase) + 1j*math.
↪sin(phase))

                # Current source only needs to be filled in B
                B[nodes[components[component][0]]] =␣
↪B[nodes[components[component][0]]] + val
                B[nodes[components[component][1]]] =␣
↪B[nodes[components[component][1]]] - val

        else:
            # DC current and voltage sources
            if component[0] == 'V':

                val = (components[component][2])

                # Increasing dimensions to account for additional voltage␣
↪equation
                A = np.column_stack((A, np.full(nodecount, 0)))
                A = np.row_stack((A, np.full(nodecount + 1, 0)))
                B = np.append(B, val)

                # keeping up with the dimension change
                nodecount += 1

                # auxiliary current
                A[nodes[components[component][0]]][nodecount - 1] =␣
↪A[nodes[components[component][0]]][nodecount - 1] + 1
```

```
                A[nodes[components[component][1]]][nodecount - 1] =␣
↪A[nodes[components[component][1]]][nodecount - 1] - 1

                # voltage additional equation
                A[nodecount - 1][nodes[components[component][1]]] = A[nodecount␣
↪- 1][nodes[components[component][1]]] - 1
                A[nodecount - 1][nodes[components[component][0]]] = A[nodecount␣
↪- 1][nodes[components[component][0]]] + 1


            elif component[0] == 'I':

                val = (components[component][2])

                # Current source only needs to be filled in B
                B[nodes[components[component][0]]] =␣
↪B[nodes[components[component][0]]] + val
                B[nodes[components[component][1]]] =␣
↪B[nodes[components[component][1]]] - val

    # Delete GND row and column in A, and element in B
    A = np.delete(np.delete(A,0,0),0,1)
    B = np.delete(B,0)



    return([A,B,nodes])
```

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

[14]:
```
%%cython --annotate
import cython
from __main__ import c_mna_builder
from __main__ import c_gausselim_partialpivoting
from timeit import default_timer as timer
@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.cdivision(True)

def c_circuit_simulator(file):
    start = timer()
    cdef list txt = file.read().splitlines()
    cdef double ac_freq = 0
    # checking for ac sources
    for line in txt:
        if line[:3] == '.ac':
```

```
            if (line.find('#') != -1):
                line = line[:line.find('#')].strip()
            ac_freq = float(line.split(" ")[-1])

    # cleaning data
    cdef list spice_code = txt[txt.index('.circuit') + 1:txt.index('.end')]
    cdef str newline
    for line in spice_code:
        newline = line
        if (line.find('#') != -1):
            newline = line[:line.find('#')].strip()
        spice_code[spice_code.index(line)] = newline.strip().split(" ")

    # creating component dictionary
    cdef dict components = {}
    for line in spice_code:
        if (ac_freq):
            if line[0][0] == 'V' or line[0][0] == 'I':
                if line[3] != 'ac':
                    print("Cannot evaluate both DC and AC sources.")
                    return 0
                else:
                    components[line[0]] = [line[1], line[2],float(line[4]),␣
↪float(line[5])]
            else:
                components[line[0]] = [line[1], line[2], float(line[-1])]
        else:
            components[line[0]] = [line[1], line[2], float(line[-1])]

    # get A,B matrices
    cdef list matrices = c_mna_builder(components, ac_freq)

    # find solution
    cdef complex[:] solution =␣
↪c_gausselim_partialpivoting(matrices[0],matrices[1])
    end = timer()

    '''
    nodes = list(matrices[2].keys())
    # print output
    for i in range(len(solution)):
        if (i < len(matrices[2]) - 1):
            print(f"Voltage at node {nodes[i + 1]} = {solution[i]} V")
        else:
            print(f"I{i - len(matrices[2]) + 2} = {solution[i]} A")
    print(f"Time taken for Cython Implementation: {end - start:e} seconds\n")
    '''
```

```
    return end - start
```

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

To optimize the circuit solver using cython, we have implemented a variety of measures:

- First off, we have specified types for most variables, including dictionaries, complex matrices and so on.
- In our MNA builder function we have implemented the same techniques used in the gaussian elimination function, we have bypassed some Python checks and specified types for function parameters.
- We have also fixed the '+= and -=' notations that were throwing errors previously.
- In our base circuit solver, we have imported our cythonized functions to altogether use mostly only cython. These are the 'c_mna_builder' function and the 'c_gausselim_partialpivoting' function.
- We have defined the data type that our 'circuit_solver' function will return.
- List and string datatypes have also been used to slightly optimize file reading.
- **We are not printing the solution since we will be conducting multiple runs to find average time taken below. Therefore I have commented that part out. For checking solutions, go to the end.**

1.3 Comparison of time taken in the two methods

[15]:
```
iter = 10000 # take average time over 10000 runs
for i in range(1,9):
    if(i == 8):
        t1 = 0
        t2 = 0
        filename = 'test.netlist'
        print(f"\n\n{filename}\n")
        for j in range(iter):
            with open(f"spice_netlists/{filename}", "r") as f1:
                t1 += circuit_simulator(f1)/iter
            with open(f"spice_netlists/{filename}", "r") as f2:
                t2 += c_circuit_simulator(f2)/iter
        print(f"Time taken for Python Implementation: {t1: e} seconds\n")
        print(f"Time taken for Cython Implementation: {t2: e} seconds\n")
        print("\033[1m" + f"Cython Implementation is {t1/t2} times faster.\n" +␣
 ↪"\033[0m")
        for j in range(50): print("-", end="")
    elif(i != 2): # skipping the circuit having both AC and DC sources
        t1 = 0
        t2 = 0
        filename = 'ckt' + str(i) + '.netlist'
```

```
        print(f"\n\n{filename}\n")
        for j in range(iter):
            with open(f"spice_netlists/{filename}", "r") as f1:
                t1 += circuit_simulator(f1)/iter
            with open(f"spice_netlists/{filename}", "r") as f2:
                t2 += c_circuit_simulator(f2)/iter
        print(f"Time taken for Python Implementation: {t1: e} seconds\n")
        print(f"Time taken for Cython Implementation: {t2: e} seconds\n")
        print("\033[1m" + f"Cython Implementation is {t1/t2} times faster.\n" +␣
↪"\033[0m")
        for j in range(50): print("-", end="")
```

```
ckt1.netlist

Time taken for Python Implementation:  1.239904e-04 seconds

Time taken for Cython Implementation:  9.243723e-05 seconds

Cython Implementation is 1.3413466666151415 times faster.

--------------------------------------------------

ckt3.netlist

Time taken for Python Implementation:  1.616916e-04 seconds

Time taken for Cython Implementation:  1.040528e-04 seconds

Cython Implementation is 1.5539379607800452 times faster.

--------------------------------------------------

ckt4.netlist

Time taken for Python Implementation:  1.075779e-04 seconds

Time taken for Cython Implementation:  9.186474e-05 seconds

Cython Implementation is 1.1710462446002006 times faster.

--------------------------------------------------
```

```
ckt5.netlist

Time taken for Python Implementation:  8.549450e-05 seconds

Time taken for Cython Implementation:  8.684741e-05 seconds

Cython Implementation is 0.9844219634731118 times faster.

-------------------------------------------------

ckt6.netlist

Time taken for Python Implementation:  1.211955e-04 seconds

Time taken for Cython Implementation:  1.042879e-04 seconds

Cython Implementation is 1.1621242133046281 times faster.

-------------------------------------------------

ckt7.netlist

Time taken for Python Implementation:  6.788689e-05 seconds

Time taken for Cython Implementation:  6.436080e-05 seconds

Cython Implementation is 1.0547862957265435 times faster.

-------------------------------------------------

test.netlist

Time taken for Python Implementation:  9.572246e-05 seconds

Time taken for Cython Implementation:  8.960634e-05 seconds

Cython Implementation is 1.068255419325366 times faster.

-------------------------------------------------
```

To compare the time taken in the two methods, we have taken the average time over 10000 runs using both functions for each netlist.

As can be seen, we are **barely getting any speedup**. This could be due to a number of factors:

- Reading the file takes up a large portion of time, making the actual matrix solving time insignificant.

- The matrices that are to be solved are not large enough to create a significant time difference during computation.
- Of course, we can optimize even further as well. The native python objects such as dictionaries and lists are slowing down the cython program, and there are still many yellow lines in the annotated code.

[16]:
```
def anscheck_c_circuit_simulator(file):
    txt = file.read().splitlines()
    ac_freq = 0

    # checking for ac sources
    for line in txt:
        if line[:3] == '.ac':
            if (line.find('#') != -1):
                line = line[:line.find('#')].strip()
            ac_freq = float(line.split(" ")[-1])

    # cleaning data
    spice_code = txt[txt.index('.circuit') + 1:txt.index('.end')]
    for line in spice_code:
        newline = line
        if (line.find('#') != -1):
            newline = line[:line.find('#')].strip()
        spice_code[spice_code.index(line)] = newline.strip().split(" ")

    # creating component dictionary
    components = {}
    for line in spice_code:
        if (ac_freq):
            if line[0][0] == 'V' or line[0][0] == 'I':
                if line[3] != 'ac':
                    print("Cannot evaluate both DC and AC sources.")
                    return 0
                else:
                    components[line[0]] = [line[1], line[2],float(line[4]),␣
 ↪float(line[5])]
            else:
                components[line[0]] = [line[1], line[2], float(line[-1])]
        else:
            components[line[0]] = [line[1], line[2], float(line[-1])]

    # get A,B matrices
    matrices = c_mna_builder(components, ac_freq)
    nodes = list(matrices[2].keys())

    # find solution
    solution = c_gausselim_partialpivoting(matrices[0],matrices[1])
```

```
    # print output
    for i in range(len(solution)):
        if (i < len(matrices[2]) - 1):
            print(f"Voltage at node {nodes[i + 1]} = {solution[i]} V")
        else:
            print(f"I{i - len(matrices[2]) + 2} = {solution[i]} A")

    return 0
```

1.4 Validation of Cython Solver Output

[17]:
```
for i in range(1,9):
    if(i == 8):
        filename = 'test.netlist'
        print(f"\n\n{filename}\n")
        with open(f"spice_netlists/{filename}", "r") as f2:
            anscheck_c_circuit_simulator(f2)
        for j in range(50): print("-", end="")
    else: # skipping the circuit having both AC and DC sources
        filename = 'ckt' + str(i) + '.netlist'
        print(f"\n\n{filename}\n")
        with open(f"spice_netlists/{filename}", "r") as f2:
            anscheck_c_circuit_simulator(f2)
        for j in range(50): print("-", end="")
```

```
ckt1.netlist

Voltage at node 1 = 0j V
Voltage at node 2 = 0j V
Voltage at node 3 = 0j V
Voltage at node 4 = (-5+0j) V
I1 = (-0.0005-0j) A
--------------------------------------------------


ckt2.netlist

Cannot evaluate both DC and AC sources.
--------------------------------------------------


ckt3.netlist

Voltage at node 1 = (-10+0j) V
Voltage at node 2 = (-5.029239766081871+0j) V
```

```
Voltage at node 3 = (-2.5730994152046778+0j) V
Voltage at node 4 = (-1.4035087719298247+0j) V
Voltage at node 5 = (-0.9356725146198832+0j) V
I1 = (-0.004970760233918129-0j) A
-------------------------------------------------

ckt4.netlist

Voltage at node 1 = (-10+0j) V
Voltage at node 2 = (-5.555555555555556+0j) V
Voltage at node 3 = (-3.7037037037037037+0j) V
I1 = (-2.222222222222222-0j) A
-------------------------------------------------

ckt5.netlist

Voltage at node 1 = (-10+0j) V
I1 = (-1-0j) A
-------------------------------------------------

ckt6.netlist

Voltage at node n3 = (-5+0j) V
Voltage at node n1 = (-1.923920880145716e-10-3.1415926534719705e-05j) V
Voltage at node n2 = (-1.8751873949430728e-10-3.062015181929007e-05j) V
I1 = (-0.004999999999812482+3.0620151819290075e-08j) A
-------------------------------------------------

ckt7.netlist

Voltage at node n1 = (-1.333200087974177e-10+0.000816455782015824j) V
-------------------------------------------------

test.netlist

Voltage at node 1 = (-1+0j) V
Voltage at node 2 = (-2+0j) V
I1 = (-1-0j) A
-------------------------------------------------
```

Just to ensure that the Cythonized solver actually works, above are the outputs for the 8 netlists.