EE21B144 Week 8

April 16, 2023

I have updated the cython coding in this complete assignment.

1 Speed of computation

Python, being an interpreted language, tends to be slower than compiled languages like C or Fortran. Some other languages like Java and Julia tend to use Just-in-Time compilation which can give speedups, but Python also has the problem of being dynamically typed, which eliminates the possibility of many optimizations.

The `timeit` library provides functions to estimate the time taken to run a piece of code. It can automatically run the code multiple times to get better average results, and can be used to identify bottlenecks in your program. However, it should be used with care as it is not a detailed function-call-level profiler.

It can either be `imported` as a module where you can then explicitly called `timeit.timeit(func)` to estimate time for a function, or you can use the *magic syntax* in Python notebooks as shown below.

[1]:
```
import numpy as np
x=np.random.rand(10000, 1)
# print(x)
```

[2]:
```
def sumarr(x):
    sum = 0
    for i in range(len(x)):
    # for i in x:
        sum += x[i]
    return sum
print(sumarr(x))
%timeit sumarr(x)
```

```
[4982.11677709]
9.93 ms ± 216 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[3]:
```
import numpy as np
def npsumarr(x):
    return np.sum(x)
print(npsumarr(x))
%timeit npsumarr(x)
```

```
4982.116777085048
5.53 µs ± 142 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

1.1 Using Cython

[4]:
```
%load_ext Cython
import cython
```

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

# Using cpdef the function can be implemented in both cython and python
cpdef float c_sumarr(x): # cpdef is used so that we can call the function later␣
 ↪as we do in python. It also somewhat optimises the speed of function calling.
    cdef float sum = 0.0
    cdef int i = 0
    for i from 0<= i < len(x):
        sum = sum + float(x[i])
    return sum
```

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

[6]:
```
print(c_sumarr(x))
%timeit c_sumarr(x)
```

```
4982.12158203125
714 µs ± 35.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

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

cpdef float c_npsumarr(x):
    return np.sum(x)
```

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

[8]:
```
print(c_npsumarr(x))
%timeit c_npsumarr(x)
```

```
4982.11669921875
5.63 µs ± 89.6 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

2 Solving equations by Gaussian elimination

Once you have constructed two matrices A and B to represent the system of linear equations

$$Ax = b$$

you can then proceed to solve the equations using the process known as Gaussian elimination.

It is assumed you already know how the process works, but to refresh your memory, you could use the reference material at LibreTexts.

Basically it involves making the A matrix *triangular* and ultimately into the shape of an identity matrix.

[9]:
```
# Input matrices - the set of equations - 2 variables x1 and x2z
A = [ [2,3], [1,-1] ]
B = [6,1/2]
print(A)
print(B)
```

```
[[2, 3], [1, -1]]
[6, 0.5]
```

[10]:
```
# Normalize row 1
norm = A[0][0]
for i in range(len(A[0])): A[0][i] /= norm
B[0] = B[0]/norm
print(A)
print(B)
```

```
[[1.0, 1.5], [1, -1]]
[3.0, 0.5]
```

[11]:
```
# Eliminate row 2 - A[1] - need to check and ensure div-by-zero etc doesnt happen
norm = A[1][0] / A[0][0]
for i in range(len(A[1])): A[1][i] = A[1][i] - A[0][i] * norm
B[1] = B[1] - B[0] * norm
print(A)
print(B)
```

```
[[1.0, 1.5], [0.0, -2.5]]
[3.0, -2.5]
```

[12]:
```
# Normalize row 2 - B[1] will now contain the solution for x2
norm = A[1][1]
for i in range(len(A[1])): A[1][i] = A[1][i] / norm
B[1] = B[1] / norm
print(A)
print(B)
```

```
[[1.0, 1.5], [-0.0, 1.0]]
[3.0, 1.0]
```

[13]:
```
# Sub back and solve for B[0] <-> x1
# This can be seen as eliminating A[0][1]
norm = A[0][1] / A[0][0]
# note that len(A) will give number of rows
for i in range(len(A)):
```

```
    A[i][1] = A[i][1] - A[i][0] * norm
    B[i] = B[i] - A[i][0] * norm
print(A)
print(B)
```

```
[[1.0, 0.0], [-0.0, 1.0]]
[1.5, 1.0]
```

2.1 Problems with Gaussian elimination

There are several obvious problems with the method outlined here. These include:

- Performance - Python lists are not the most efficient way to store matrices
- Zeros: the simple example does not consider a scenario where one of the values on the diagonal may be 0. Then some shuffling of rows is required.
- Numerical stability: there are several *normalization* steps involved, where it is quite possible for the values to blow up out of control if not managed properly. Usually some kind of pivoting techniques are used to get around these issues.

[14]:
```
import numpy as np
A1 = np.array( [ [2,3], [1,-1] ] )
B1 = np.array( [6, 1/2] )
np.linalg.solve(A1, B1)
```

[14]:
```
array([1.5, 1. ])
```

3 Library functions

Numeric Python or `numpy` is a library that allows Python code to directly call highly efficient implementations of several linear algebra routines (that have been written and optimized using C or Fortran and generally offer very high performance).

Although you can use the same `list` based approach to create matrices, it is better to declare them as the `array` data type:

- the numeric `type` (float, int etc.) can be specified for arrays
- memory allocation is done more efficiently by assuming they are not meant to be arbitrarily extended or changed

4 SPICE basics

Our goal is to implement a SPICE simulator. In order to do this, we first need to read in the circuit description from a text file. To start with, we will only consider the basic elements of SPICE: Voltage sources, Current sources, and Resistors. A typical SPICE netlist would look like this:

```
.circuit
R1 GND 1 1
R2 1 2 1
V1 GND 2 dc 2
.end
```

This is basically a *netlist* with 3 *nodes* - one of which is Ground (GND) which is assumed to be have a voltage of 0V. We can write down Kirchhoff's current law (KCL) equations at each node, to account for current balance. In addition, we will have some equations that specify the voltages between nodes having a direct voltage source, since there is no resistance there to provide an equation.

For the above example, the equations will be

$$\begin{aligned} \frac{V1-0}{R1} + \frac{V1-V2}{R2} &= 0 \\ \frac{V2-V1}{R2} + I1 &= 0 \\ V2 - 0 &= 2 \end{aligned}$$

which can be written in Matrix form as:

$$\begin{bmatrix} \frac{1}{R1}+\frac{1}{R2} & \frac{-1}{R2} & 0 \\ \frac{-1}{R2} & \frac{1}{R2} & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} V1 \\ V2 \\ I1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

At this point, you have reduced the solving of the SPICE equations to a known problem (linear equation solving) that you already know how to do.

4.1 AC sources

The assumption above is that the system consists only of Voltage or Current sources and resistors. What about capacitors, inductors, and AC sources? These can be handled in exactly the same way as long as the circuit is operating at a single frequency. We then replace the elements with their corresponding *impedance* values, which are frequency dependent complex numbers, but since there is only a single frequency they will still be constants.

4.2 Problem scenarios

- Voltage source loops
- Current sources at a node
- Circuit defined with both DC and AC sources
- Syntax errors

5 String and File manipulation

Given a SPICE netlist like the one above, you need to *read* it and construct an internal matrix as described above. For string manipulation, there are a few helpful utility functions that we can see here.

[15]:
```
circ = """.circuit
R1 GND 1 1
R2 1 2 1
V1 GND 2 dc 2
.end
```

```
""".splitlines()
# print(circ)
for l in circ:
    if l[0] == 'R':
        print("Found a resistor")
    elif l[0] == 'V':
        print("Found a voltage source with value: ", float(l.split()[4]))
```

```
Found a resistor
Found a resistor
Found a voltage source with value:  2.0
```

5.1 Files

You can read from a file using the `readlines()` method of file objects. One thing to keep in mind is how you open and close file objects. In particular, it is strongly recommended to use the pattern `with open("filename") as f:` to ensure that the file is closed once you are done with reading it.

6 Assignment

The following are the problems you need to solve for this assignment. You need to submit your code (either as standalone Python script or a Python notebook), a PDF document explaining your solution (either generated from the notebook or a separate LaTeX document), and any supporting files you may have (such as circuit netlists you used for testing your code).

- Write a function to find the factorial of N (N being an input) and find the time taken to compute it. This will obviously depend on where you run the code and which approach you use to implement the factorial. Explain your observations briefly.
- Write a linear equation solver that will take in matrices A and b as inputs, and return the vector x that solves the equation $Ax = b$. Your function should catch errors in the inputs and return suitable error messages for different possible problems.
  - Time your solver to solve a random 10×10 system of equations. Compare the time taken against the `numpy.linalg.solve` function for the same inputs.
- Given a circuit netlist in the form described above, read it in from a file, construct the appropriate matrices, and use the solver you have written above to obtain the voltages and currents in the circuit. If you find AC circuits hard to handle, first do this for pure DC circuits, but you should be able to handle both voltage and current sources.

6.1 Bonus assignments

- (Small bonus): after reading in the netlist, allow some or all sources and impedances to be controlled interactively - either using widgets or other mechanisms. On each change you should recompute the currents and voltages and display them.
- (Large bonus): make a solver that can do real-time transient simulations of a SPICE netlist and update the currents and voltages dynamically. They should also be plotted as a function of time and react to changes. This is something along the lines of https://www.falstad.com/circuit/. Ideally you should be able to do a real-time demo of some experiments you might conduct as part of a basic electronics lab, and simulate the behaviour of an oscilloscope and signal generator.

6.2 Libraries imported

Assuming numpy and math libraries have been installed on the machine

[16]:
```
import numpy as np
import math # To use math.pi in solving AC circuit netlist.
```

6.3 1] Factorial

6.3.1 Method 1

1] Here I have used a for loop to find the factorial of the number.

2] The variable prod calculates the factorial of number by multiplication of numbers from 1 to n.

3] The variable prod is returned by the function after completing the for loop.

[17]:
```
def factorial_1(N):
    if(N<0):
        return "Factorial does not exist"
    prod=1
    for i in range(1, N+1):
        prod=prod*i
    return prod

print(factorial_1(10))
%timeit factorial_1(10)
```

```
3628800
450 ns ± 7.31 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

6.4 Using Cython

[18]:
```
%load_ext Cython
```

```
The Cython extension is already loaded. To reload it, use:
  %reload_ext Cython
```

[19]:
```
%%cython --annotate
def c_factorial_1(N):
    if(N<0):
        return "Factorial does not exist"
    cdef int prod, i
    prod = 1
    for i in range(1, N+1):
        prod=prod*i
    return prod
```

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

[20]:
```
print(c_factorial_1(10))
%timeit c_factorial_1(10)
```

```
3628800
52.3 ns ± 1.67 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

6.4.1 Method 2

1] Here I have used the recursiveness of the loop, to calculate the factorial of number.

2] The factorial of N is calculated by multiplying N with factorial of (N-1), and so on.

3] If N=1, then 1 is returned.

At the end we get the factorial of number.

[21]:
```
def factorial_2(N):
    if(N<0):
        return "Factorial does not exist"

    if (N==1 or N==0):
        return N
    else:
        return N*factorial_2(N-1)

print(factorial_2(10))
%timeit factorial_2(10)
```

```
3628800
1.15 µs ± 27.7 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

6.5 Using Cython:

[22]:
```
%%cython --annotate

cpdef c_factorial_2(N):
    if(N<0):
        return "Factorial does not exist"

    if (N==1 or N==0):
        return N
    else:
        return N*c_factorial_2(N-1)
```

[22]:
```
<IPython.core.display.HTML object>
```

[23]:
```
print(c_factorial_2(10))
%timeit c_factorial_2(10)
```

```
3628800
273 ns ± 12.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

It can be seen that, according to %timeit, we get the factorial of a number faster by running a for loop than using the recursiveness of the function

6.6 2] Linear equation solver:

The idea that I have used to solve the linear equation is of Gaussian elimination, where we first create an upper triangular matrix, and check for the case of : a] Consistent, and b] Inconsistent solutions

In Inconsistent solutions, we can get : a] No solution, and b] Infinte solution.

When we get the case of Consistent solution, I have found the solution for the linear equation using back substitution, and printed them out.

shape_check(): 1] This function is used to check if the given matrix A, is a square matrix.

2] It also checks the equality of number of rows of A with the number of rows in B.

If the above condtions do not satisfy, its an invalid equation, and no solution is possible.

If the above conditions are satisfied, they are valid equations, and there might be a chance to get the solutions (Consistent and inconsistent).

[24]:
```
def shape_check(A, b):
    for i in range(len(A)):
        if(len(A)!=len(A[i])):
            print("Invalid linear equations, no solution is possible")
            return False
        if(len(b)!=len(A) or len(b)!=len(A[i])):
            print("Invalid linear equations, no solution is possible")
            return False
    else:
        print("Valid linear equations, solutions may be possible")
        return True
```

6.7 Using Cython

[25]:
```
%%cython --annotate
# cpdef to write the function both as using cdef and def
cpdef bint c_shape_check(list A, list b):
    cdef int i, n = len(A)

    for i in range(n):
        if len(A) != len(A[i]):
            print("Invalid linear equations, no solution is possible")
            return False
        if len(b) != n or len(b) != len(A[i]):
            print("Invalid linear equations, no solution is possible")
            return False
```

```
    print("Valid linear equations, solutions may be possible")
    return True
```

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

1] row_zero_check(A, b, row, col): This function is used to print out if the solution is a] No solution, or b] Infinite solution.

When we get the complete row of A as zero while simplification we can say that,
a] If the corresponding row element of B is zero, then there are infinite solutions,
b] If the corresponding row element of B is non-zero, then there are No solutions.

I have used the raise Exception for 'Infinte solution', which prints out 'Infinite solution' at the end, if we do not get any condition of 'No solution' after the current row. If there is a case of 'No solution' after this row, even if the 'Infinite solution' exception is raised, there is 'No solution' for the linear equations.

So, the condition of 'No solution' is dominant over the 'Infinite solution'. And, it will be raised over that of 'Infintite solution', if its raised.

2] col_swap(A, b, row, col): a] While moving in the diagonal, if we get a zero element, we will first try to swap it with the first of below rows containing non-zero element at the corresponding column of the the diagonal element.

b] If all the below row element at the corresponding diagonal element's column are zero, I will do a column swap.

In column swap, I will look for columns ahead of the current diagonal element, at the same corresponding row, and will swap the first one, if I get a non-zero element in that column. While doing this, if I get all the row elements below and column elements ahead as zero, the function row_zero_check(A, b, row, col) will be called.

3] row_swap(A, b, row, col): a] While moving in the diagonal, if I get the element as zero, I will do a row swap.

b] In row swap, I will look for rows below the corresponding row, whose column is same as the diagonal's, and swap with the first one, when it is non-zero.

While doing this, if I do not any get row whose corresponding column element is non-zero, the function col_swap(A, b, row, col) will be called.

4] function(A, b): This takes the matrix A and b.
a] try block :
i] The try block is used to normalize the row, and subtract this row from the other rows to get the upper triangular matrix needed for Gaussian elimination.
ii] There are two cases for the normalizing factor : one where it is zero, in which case I have called the row_swap(A, b, row, col) function to get the normalizing element as zero.
The other case is when the normalizing factor is non-zero, in which I have used it to subtract it from the other rows as it is done for Gaussian elimination.

b] except block :
This block is used to raise the Exception of "Infinite solution" or "No solution" when one of them is raised from the row_zer_check(A, b, row, col), and is printed when the requirements are satisfied.

At the end of running this function I have printed out the upper triangular matrix of A, and the correponding matrix b, that i get after running the above functions. A, and b are also returned.

[26]:
```
infinite=False
def row_zero_check(A, b, row, col):
    if(b[row]!=0):
        raise Exception("NO solutions")
    if(b[row]==0):
        infinite=True
        raise Exception("Infinite solution")

def col_swap(A, b, row, col):
    n=len(A)
    for i in range(row+1, n):
        if(A[row][i]!=0):
            for k in range(n):
                A[k][row], A[k][i] = A[k][i], A[k][row]
            print(A)
        return A, b
    return row_zero_check(A, b, row, col)

def row_swap(A, b, row, col): # checking in the same column
    n=len(A)
    for i in range(row+1, n):
        if(A[i][col]!=0):
            A[row], A[i] = A[i], A[row]
            b[row], b[i] = b[i], b[row]
            return A, b
    return col_swap(A, b, row, col)

def function(A, b):
    n=len(A)
    column_number=0

    try:
        for row in range(n):
            norm=A[row][row]
            if (norm==0):
                A, b = row_swap(A, b, row, row)

            if (norm!=0):
                A[row]=[ele/norm for ele in A[row]]
                b[row] = b[row]/norm
                for j in range(row+1, n):
```

```
                    factor = A[j][row]/(A[row][row])
                    for k in range(n):
                        A[j][k] = A[j][k] - factor*A[row][k]
                    b[j] = b[j]-factor*b[row]

        # print("A matrix = ", A, end="\n\n")
        # print("B matrix = ", b, end="\n\n")

    except Exception as t:
        return t
    return A, b
```

[27]:
```
############ COMMENTS(print statements to be removed while using␣
 ↪%timeit)###############
# In function(A, b)
#         print("A matrix = ", A, end="\n\n")
#         print("B matrix = ", b, end="\n\n")

# In solution(A, b)
#             print("Solutions are = ")
```

solution(A, b): This function finds the solution for matrix A and b which are returned by the function function(A, b), previously.
This function first runs the function(A, b), whose return type is tuple, and if it runs properly, the final solution of the equations are printed out by following back substitution.
In back substitution, we start from the bottomost row, and find its solution, then substitute this solution in the row above, and find its solution, and it continues until it has found all the solutions.
While doing this back substitution, there may be a possibility that the division is being done by zero, in which case I have made the except block for handling it.
The final output conatins the statement printed by the function(A, b) and its final solution.

[28]:
```
import numpy as np
def solution(A, b):
    try:
        if(type(function(A, b)) is tuple):
            n = len(A)
            sol=[0 for i in range(n)]
            sol[n-1] = b[n-1]/A[n-1][n-1]
            for i in range(n-2, -1, -1):
                sum=0
                for j in range(i+1, n):
                    sum = sum+A[i][j]*sol[j]
                sol[i]=(b[i]-sum)/A[i][i]
            # print("Solutions are = ")
            return sol
```

```
        elif(type(function(A, b)) is Exception):
            print(function(A, b))

    except:
        print("Division by zero error")


A = np.random.rand(10, 10)
b = np.random.rand(10)
solution(A, b)
%timeit solution(A, b)
# This timeit was ran on A = np.random.rand(10, 10), and b = np.random.rand(10)
 ↪without comments as above
# The %timeit should be ru without printing any statement above, as in the case
 ↪with printing statements the time increases
# a lot.
```

```
334 µs ± 7.21 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

6.8 Using Cython :

[29]:
```
%%cython --annotate
import numpy as np
cimport numpy as np
cimport cython

ctypedef np.float64_t dtype_t  # This could be used ahead c_solution(A, b) for
 ↪defining sol

cdef bint infinite = False # The cdef bint is equivalent to boolean of python


# boundscheck(False) tells Cython to skip bounds checking for all array accesses
 ↪in the decorated function. This can provide a performance boost,
# especially for code that performs many array accesses. However, it also means
 ↪that the code is more susceptible to runtime errors such as segmentation
 ↪faults or buffer
# overflows if the array indices are out of bounds.

# wraparound(False) tells Cython that arrays are not cyclic, which means that
 ↪negative indices and indices larger than the array size will result in an
 ↪error rather than
# wrapping around to the other end of the array. This can also provide a
 ↪performance boost, since it allows Cython to assume that array indices are
 ↪always within the bounds
# of the array. However, it may cause errors if the code relies on the cyclic
 ↪behavior of arrays.
```

```
# In general, it is recommended to use these directives only if we are sure that␣
 ↪the code does not require bounds checking or cyclic array behavior, and if
# the performance benefits outweigh the risks of potential runtime errors.


@cython.boundscheck(False)
@cython.wraparound(False)
cdef row_zero_check(A, b, row, col):
    cdef bint infinite
    if b[row] != 0:
        raise Exception("NO solutions")
    elif b[row] == 0:
        infinite = True
        raise Exception("Infinite solution")


@cython.boundscheck(False)
@cython.wraparound(False)
cdef col_swap(A, b, row, col):
    cdef int n = A.shape[0]
    cdef int i, k
    for i in range(row+1, n):
        if A[row, i] != 0:
            for k in range(n):
                A[k, row], A[k, i] = A[k, i], A[k, row]
            print(A)
        return A, b
    return row_zero_check(A, b, row, col)


@cython.boundscheck(False)
@cython.wraparound(False)
cdef row_swap(A, b, row, col):
    cdef int n = A.shape[0] # Defining integer n, i, k
    cdef int i, k
    for i in range(row+1, n):
        if A[i, col] != 0:
            A[row], A[i] = A[i], A[row]
            b[row], b[i] = b[i], b[row]
            return A, b
    return col_swap(A, b, row, col)


@cython.boundscheck(False)
@cython.wraparound(False)
cdef function(A, b):
```

```
    cdef int n = A.shape[0] # Defining integer n, i, k, and double norm, and␣
 ↪factor
    cdef int row, j, k
    cdef double norm, factor

    # global infinite
    # infinite = False

    try:
        for row in range(n):
            norm = A[row][row]
            if norm == 0:
                A, b = row_swap(A, b, row, row)

            if norm != 0:
                A[row] = [ele / norm for ele in A[row]]
                b[row] = b[row] / norm
                for j in range(row+1, n):
                    factor = A[j, row] / A[row, row]
                    for k in range(n):
                        A[j, k] = A[j, k] - factor * A[row, k]
                    b[j] = b[j] - factor * b[row]

        if infinite:
            raise Exception("Infinite solution")

    except Exception as t:
        return t
    return A, b


@cython.boundscheck(False)
@cython.wraparound(False)
def c_solution(A, b):
#     print("here")
    cdef int n = len(A)
#     A = np.array(A)
#     b = np.array(b)
    # Restrictions on np while using cython
#     cdef np.ndarray[dtype_t, ndim=2] c_A = np.asarray(A)
#     cdef np.ndarray[dtype_t, ndim=1] c_b = np.asarray(b)
#     cdef np.ndarray[dtype_t, ndim=1] sol = np.zeros(n)

    try:
        if isinstance(function(A, b), tuple):
            sol = [0 for i in range(n)]
            sol[n-1] = b[n-1]/A[n-1, n-1]
```

```
            for i in range(n-2, -1, -1):
                sum = 0
                for j in range(i+1, n):
                    sum += A[i, j]*sol[j]
                sol[i] = (b[i]-sum)/A[i, i]

            return sol

        elif isinstance(function(A, b), Exception):
            print(function(A, b))

    except ZeroDivisionError: # To identify the ZeroDivisionError
        return "Division by zero error"
```

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

[30]:
```
A = np.random.rand(10, 10)
b = np.random.rand(10)
# print(solution(A, b))
%timeit c_solution(A, b)
```

```
191 µs ± 2.59 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Solving using np.linalg.solve()

[31]:
```
np.linalg.solve(A, b)
%timeit np.linalg.solve(A, b)
# This timeit was ran on A = np.random.rand(10, 10), and b = np.random.rand(10)␣
 ↪commented as above
```

```
16.3 µs ± 200 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

7 3] Solving Netlist:

I have assumed that the syntax of the .netlist file follows the syntax as given in the SPICE syntax file shared in moodle. And, any of the values : name, n1, n2, dc/ac and value are not missing, respectively for R, I, V, L, and C. I have also assumed that L, C always are in AC and have a frequency associated with it.

7.0.1 File reading :

The below is used to open the file, read it, and store the proper SPICE syntax format in variable 'final_circ' as a list

[32]:
```
def file(file_name):
    file = open(file_name, 'r') # Opening the netlist file

    # Adding the string statements to circuit
    circuit=""
```

```
    for c in file.readlines():
        circuit=circuit+c

    circuit = circuit.splitlines()

    final_circ=[]
    length = len(circuit)
    end_index=0
    start_index=0

    # To get the index of starting and ending of circuit using '.circuit', and '.
↪end'
    for i in range(len(circuit)):
        for j in circuit[i].split():
            if(j=='.circuit'):
                start_index = i
                break
            if(j=='.end'):
                end_index = i
                break

    # Finding # for comment, and also the '.ac' to find the frequency of␣
↪operation of the ciruit.
    for i in range(len(circuit)):
        if(i>=start_index and i<=end_index):
            if(circuit[i].find('#')!=-1):
                index = circuit[i].find('#')
                final_circ.append(circuit[i][0:index].strip())
            else:
                final_circ.append(circuit[i].strip())
            start_index +=1

        if(circuit[i].find('.ac')!=-1):
            if(circuit[i].find('#')!=-1):
                index = circuit[i].find('#')
                final_circ.append(circuit[i][0:index].strip())
            else:
                final_circ.append(circuit[i].strip())

    # Printing the netlist with proper syntax using list

    return final_circ
```

7.0.2 Multiple frequencies error :

This is used to find the multiple frequency error, that might be present in netlist

[33]:
```
# Multiple frequencies error
def multiple_frequencies_error(final_circ):
    freq=[]
    circuit_dot =0 # To find the location of '.circuit'
    end_dot = 0 # To find the location of '.end'

    for i in range(len(final_circ)):
        if(final_circ[i].split()[0]=='.ac'):
            if(final_circ[i].split()[2] not in freq):
                freq.append(final_circ[i].split()[2])

        if(final_circ[i].split()[0]=='.circuit'):
            circuit_dot=i

        if(final_circ[i].split()[0]=='.end'):
            end_dot=i

    return freq, circuit_dot, end_dot
```

7.0.3 Frequency of AC, when only a single frequency are given :

[34]:
```
def frequency(freq):
    if(len(freq)>1):
        # print("Multiple frequencies are given ", freq)
        return False
    elif(len(freq)==0):
        # print("DC source")
        return True
    else:
        # print("Frequency = ", float(freq[0]))
        return True
```

7.0.4 Replacing GND with '0'

[35]:
```
# Printing the netlist with proper formatting
def replace_GND(final_circ):
    final=""
    for i in range(len(final_circ)):
        final += final_circ[i].replace('GND', '0')+"\n"
#     print(final)
    return final
```

7.0.5 Counting independent sources, and doing proper formatting of the netlist :

a] To count the number of independent sources.
b] To replace the nodes like n1, n2, ... with digits like '1', '2', ... Because the value of nodes should be a string containing only digits.

[36]:
```
def count_indpnt(final, circuit_dot, end_dot):
    nodes = []
    indpnt = 0
    final_circuit_copy = final.splitlines()
    final_circuit=[]

    # To find the number of independent sources, and the total number of nodes␣
 ↪in the circuit, including the Ground ('0')
    for i in range(len(final_circuit_copy)):
        final_split = final_circuit_copy[i].split()
        final_circuit.append(final_split)
        if(final_split[0][0]=='V'):
            indpnt +=1

        if(i>circuit_dot and i<end_dot):
            if(final_split[1] not in nodes):
                nodes.append(final_split[1])
            if(final_split[2] not in nodes):
                nodes.append(final_split[2])



    # All the values of nodes should be string containing only digits
    nodes_copy = nodes
    final_circuit_copy = final_circuit

    for i in range(circuit_dot, end_dot):
        for j in range(len(final_circuit_copy[i])):
            if(j==1 or j==2):
                s=""
                for k in final_circuit_copy[i][j]:
                    if(k.isdigit()==True):
                        s=s+str(int(k))
                final_circuit[i][j]=s


    for i in range(len(nodes_copy)):
        s =""
        for j in nodes_copy[i]:
            if(j.isdigit()==True):
                s=s+str(int(j))
        nodes[i]=s


    # print("Total number of nodes (excluding ground) = ", len(nodes)-1)
    # print("Total number of independent sources = ", indpnt)
```

```
    return final_circuit, nodes, indpnt
```

7.1 Method = Modified nodal analysis (MNA)

I have used the method of modified nodal analysis, which is used to determine the nodal voltage of each voltage, and the current passing through independent voltage sources.

We will apply this method to a circuit with n nodes, and m independent voltage sources. This is also applicable to independent current sources.

To apply this : 1] We will select a reference node, usually ground (node 0), and write the remaining node voltages(n-1) with respect to this reference node. 2] I have used the convention that : a] current leaving the node is taken as positive, and current entering the node is negative, b] the current flows from the positive node to the negative node. c] The value of voltage source is specified considering the first node to be positive terminal. 3] Apply Kirchoff's current law to each node. 4] Write an equation for the voltage of each of the independent voltage sources. 5] Solve the system of (n-1) unknowns.

Let the matrix equation being formed by MNA be:

GV = I

Let n = number of nodes(with respect to GND (0)), and m = number of independent voltage sourcces. 1]The G matrix will be of dimension (n+m)x(n+m): a] The upper left (nxn) part of matrix G : i] is determined by the interconnections of the passive elements, ii] the diagonal of this matrix consists of sum of conductance(1 over resistance or impedance) of element connected to corresponding node, iii] the off diagonal elements of the matrix consists of the sum of negative conductance of element connected to the pair of corresponding node

b] The upper right (nxm) part of matrix G : i] It consists of values 0, -1, and 1, ii] The values can be understood by the direction of current entering or leaving the node, according to the convention defined. The value 0 is written when there is no independent voltage source connected. The value 1 is written when there is voltage source connected, and the current is leaving the node. Similarly, for -1 the current is entering the node.

c] The lower left (mxn) part of matrix G : i] It consists of values 0, 1, -1. ii] The value tells us how the terminal of the corresponding voltage is connected to node. If no voltage is connected between the nodes, it is 0. If the positive terminal is connected to that node, then it is 1. Similarly, for negative terminal connected to that node it is -1.

d] The lower right (mxm) part of matrix G: This matrix is used to write for the dependent sources, but as we are only considering independent sources all its element will be 0.

2] The matrix V will be of dimension (n+m)x1 : It contains the nodal voltages, and the current through independent voltage sources i] The top n elements are the nodal voltages, ii] The bottom m elements contains the current passing through the independent voltage sources

3] The matrix I will be of dimension (n+m)x1 : This matrix holds the value of known quantities : independent voltage and current sources, The top n elements are either zero or the sum and difference of independent current soures in the circuit. The bottom m elements represent the m independent voltage sources in the circuit, entered according to the sign.

7.1.1 Generating G matrix (valid for both AC and DC) :

[37]:
```
def G_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq):
    # To use the complex functions, i have used the datatype '_complex'
    # G will be a complex matrix
    G = np.zeros((len(nodes)-1+indpnt, len(nodes)-1+indpnt), dtype = 'complex_')
    mm = 0
    vv = 0
    for i in range(circuit_dot+1, end_dot):
        final_split = final_circuit[i]
        node1 = int(final_split[1])
        node2 = int(final_split[2])
        symbol = final_split[0][0]

# Upper left (nxn) part of G matrix :
#         I have taken the current leaving node as positive, and current␣
 ↪entering node as negative
        if(symbol.upper() == 'R'):
            value = float(final_split[-1])
            if (node1!=0 and node2!=0):
                G[node1-1][node1-1] +=1/value
                G[node2-1][node2-1] +=1/value
                G[node1-1][node2-1] -=1/value
                G[node2-1][node1-1] -=1/value

            if(node1==0 and node2!=0):
                G[node2-1][node2-1] +=1/value

            if(node1!=0 and node2==0):
                G[node1-1][node1-1] +=1/value


#         Value of impedance of an inductor = complex of w*L (w=2*PI*freq)
        if(symbol.upper() == 'L'):
            val = float(final_split[-1])*float(freq[0])*2*math.pi
            value =  complex(0, val) # It convert the value to complex numbers
            if (node1!=0 and node2!=0):
                G[node1-1][node1-1] +=1/value
                G[node2-1][node2-1] +=1/value
                G[node1-1][node2-1] -=1/value
                G[node2-1][node1-1] -=1/value

            if(node1==0 and node2!=0):
                G[node2-1][node2-1] +=1/value

            if(node1!=0 and node2==0):
                G[node1-1][node1-1] +=1/value
```

```
#         Value of impedance of conductor = complex of (1/w*C), (w=2*PI*freq)
        if(symbol.upper() == 'C'):
            val = float(final_split[-1])*float(freq[0])*2*math.pi
            value =  complex(0, val) # It convert the value to complex numbers
            if (node1!=0 and node2!=0):
                G[node1-1][node1-1] +=value
                G[node2-1][node2-1] +=value
                G[node1-1][node2-1] -=value
                G[node2-1][node1-1] -=value

            if(node1==0 and node2!=0):
                G[node2-1][node2-1] +=value

            if(node1!=0 and node2==0):
                G[node1-1][node1-1] +=value


# Upper right (nxm) part of G matrix :
#         Node1 is positive, node2 is negative of the voltage source
        if(symbol.upper() == 'V'):
            if(node1!=0 and node2!=0):
                G[node2-1][len(nodes)-1+mm] = -1
                G[node1-1][len(nodes)-1+mm] = 1

            if(node1==0 and node2!=0):
                G[node2-1][len(nodes)-1+mm] = -1

            if(node1!=0 and node2==0):
                G[node1-1][len(nodes)-1+mm] = 1
            mm=mm+1


# Bottom left(mxn) part of G matrix :
#         I have assumed that the current flows from the positive to the␣
 ↪negative terminal of the voltage
#         I have taken node1 as positive and node2 as negative
        if(symbol.upper() == 'V'):
            if(node1!=0 and node2!=0):
                G[len(nodes)-1+vv][node1-1] = 1
                G[len(nodes)-1+vv][node2-1] = -1

            if(node1==0 and node2!=0):
                G[len(nodes)-1+vv][node2-1] = -1

            if(node1!=0 and node2==0):
```

```
                G[len(nodes)-1+vv][node1-1] = 1
            vv=vv+1

# Bottom right (mxm) part of G matrix = 0, as mentioned in the explanation above␣
↪
    return G

# print(G)
# G = G.tolist()
# print(type(G))
```

7.1.2 Generating I matrix (valid for both AC and DC) :

[38]:
```
# Forming the I matrix
def I_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq):
    I = np.zeros((len(nodes)-1+indpnt,1), dtype = 'complex_')
    ii=0
    vv=0
    for i in range(circuit_dot+1, end_dot):
        final_split = final_circuit[i]
        node1 = int(final_split[1])
        node2 = int(final_split[2])

        symbol = final_split[0][0]


# Top (nx1) part of I matrix :
        if(symbol.upper()=='I'):
            ac_or_dc = final_split[3]
            if(ac_or_dc.lower() == 'dc'): # If the independent current source is␣
↪'dc'
                value = float(final_split[-1])
                I[ii][0] = value

            if(ac_or_dc.lower() == 'ac'): # If the independent current source is␣
↪'ac'
                magnitude = float(final_split[4])
                phase = float(final_split[-1])
                real = magnitude*math.cos(phase)
                imaginary = magnitude*math.sin(phase)
                value = complex(real, imaginary)
                I[ii][0] = value
            ii=ii+1


# Bottom (mx1) part of I matrix :
        if(symbol.upper()=='V'):
```

```
            ac_or_dc = final_split[3]
            if(ac_or_dc.lower() == 'dc'): # If the independent voltage source is␣
↪'dc'
                value = float(final_split[-1])
                I[len(nodes)-1+vv][0] = value

            if(ac_or_dc.lower() == 'ac'): # If the independent voltage source is␣
↪'ac'
                magnitude = float(final_split[4])
                phase = float(final_split[-1])
                real = magnitude*math.cos(phase)
                imaginary = magnitude*math.sin(phase)
                value = complex(real, imaginary)
                I[len(nodes)-1+vv][0] = value
            vv=vv+1

    return I

# print(I)
```

7.1.3 Solution

[39]:
```
def netlist_solution(file_name):
    final_circ = file(file_name)
    freq, circuit_dot, end_dot = multiple_frequencies_error(final_circ)
    single_or_multiple = frequency(freq)
    final = replace_GND(final_circ)
    final_circuit, nodes, indpnt = count_indpnt(final, circuit_dot, end_dot)

    if(single_or_multiple == True):
#         I made the variables global, so that it can directly be solved in np.
↪linalg.solve(G, I) below
        global G # To make the vairable G, global
        global I # To make the vairable I, global
        G = G_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq)
        I = I_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq)
        # print(G)
        # print(I)

        sol = solution(G, I)
        # for s in sol:
            # print(s)

    else:
        # print("Multiple frequencies")
        pass
```

```
def c_netlist_solution(file_name):
    final_circ = file(file_name)
    freq, circuit_dot, end_dot = multiple_frequencies_error(final_circ)
    single_or_multiple = frequency(freq)
    final = replace_GND(final_circ)
    final_circuit, nodes, indpnt = count_indpnt(final, circuit_dot, end_dot)

    if(single_or_multiple == True):
#         I made the variables global, so that it can directly be solved in np.
 ↪linalg.solve(G, I) below
        global G # To make the vairable G, global
        global I # To make the vairable I, global
        G = G_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq)
        I = I_matrix(final_circuit, nodes, indpnt, circuit_dot, end_dot, freq)
        # print(G)
        # print(I)

        sol = c_solution(G, I)
        # for s in sol:
            # print(s)

    else:
        # print("Multiple frequencies")
        pass
```

To run the below statement on various files of .netlist, the .netlist file should be in the same folder as this jupyter notebook.

[40]:
```
netlist_solution("ckt6.netlist")
# %timeit netlist_solution("ckt6.netlist")
# The %timeit function must be used without printing of any statement by␣
 ↪netlist_solution(), as the time increases a lot.
```

[45]:
```
%timeit netlist_solution("ckt6.netlist")
%timeit c_netlist_solution("ckt6.netlist")
```

```
181 µs ± 3.27 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

/tmp/ipykernel_1324674/2197146062.py:41: ComplexWarning: Casting complex values
to real discards the imaginary part
  sol = c_solution(G, I)

168 µs ± 1.55 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[42]:
```
# np.linalg.solve(G, I)
# %timeit np.linalg.solve(G, I)
```

[43]:
```
##################COMMENTS for NETLIST (print statements to be removed while␣
 ↪using %timeit)####################
```

```
# def frequency(freq):
#         print("Multiple frequencies are given ", freq)
#         print("DC source")
#         print("Frequency = ", float(freq[0]))
# def count_indpnt(final, circuit_dot, end_dot):
#     print("Total number of nodes (excluding ground) = ", len(nodes)-1)
#     print("Total number of independent sources = ", indpnt)

# In function(A, b)
#         print("A matrix = ", A, end="\n\n")
#         print("B matrix = ", b, end="\n\n")

# In solution(A, b)
#             print("Solutions are = ")
```