Week-8

Shah Jainam EE21B122

April 16, 2023

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

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

[3]:
```
%%cython --annotate
#Cython
# Only diffrence is using `int` N i.e C-type syntax
def factorial_cython(int N):
    if N == 1 or N == 0:
        return 1
    else:
        return N * factorial_cython(N-1)
```

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

[4]:
```
#Python
def factorial(N):
    if N==1 or N==0:
        return 1
    else:
        return N*factorial(N-1)
```

[5]:
```
try:
    N = int(input('Enter an integer number: '))
    print('This is Cython')
    print(factorial_cython(N))
    %timeit factorial_cython(N)
    print('This is Python')
    print(factorial(N))
    %timeit factorial(N)
except:
    print('Invalid Number')
```

```
Enter an integer number: 99
This is Cython
```

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
5.21 µs ± 33 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
This is Python
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
13.4 µs ± 53.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

0.1 Observation

As we can see from timeit command that the cython implentation takes very less time compared to the python implemtation, with only just one change this much amout of time we have saved

0.2 Understanding

I think that because of already defining the variable N as `int` we have saved a lot of time, the problem and blessing with python is that it is a dynamic language, that means we can write any variable we want and not care if its type, as we will define the type afterwards in the code when we use that vaiable, the down side of this is it makes the python slow compared to static languages like C or C++ as in them we first define the datatype and then use it accordingly, this makes C code long and hard to read/debug compare to python but because of this the speed of programs in C is much faster then Python.

1 Python Implementation

[6]:
```python
# This is week 2 code of Gauss Elemination in pure python
# I have removed the determinant part out because that was used to catch edge␣
 ↪case and used too much time
#, I wanted to see how my code
# compares to numpy,
import numpy as np
def row_swapp(A,r1,r2):         #Swapping rows r1 and r2 of matrix A
    A[r1],A[r2] = A[r2],A[r1]

def agumentedp(A,b):            #Making a Augumented Matrix using matrix A and b
    n = len(A)
    for i in range(len(A)):
        A[i].append(b[i][0])


def normalizationp(A,c):    #Normalise the row c of the matrix A
    m = len(A[0])
    n = len(A)
    #n = len(A)
    try:
        for i in range(m):
            if A[c][c] != 1:
```

```
                norm = A[c][c]
                for j in range(m):
                    A[c][j] = A[c][j]/norm
    except ZeroDivisionError:
        for i in range(n):
            for j in range(i+1,n):
                if A[j][i] != 0:
                    #swap A[j] with A[i]
                    row_swapp(A,j,i)
        normalizationp(A,c)


def eleminationp(A,r,c):   #make element rc of the matrix A zero
    #normalization(A,c)
    m = len(A[0])
    for i in range(m):
        if A[r][c] != 0:
            ele = A[r][c]
            for j in range(m):
                A[r][j] = A[r][j] - ((ele)*A[c][j])

def gaussp(A,b):           #Gauss eleminantion Algorithm
    agumentedp(A,b)
    m = len(A)
    for i in range(m):
        normalizationp(A,i)
        for j in range(m):
            if i != j:
                eleminationp(A,j,i)
    return A

def solutionsp(A,b):     #Extracting the solution after Gauss elemination and␣
 ↪handling edge cases
    n = len(b)
    #print(A)
    sol = []
    B = gaussp(A,b)
    for i in range(len(B)):
        sol.append(B[i][-1])

    return sol
```

- here i have used cdef as used in cython syntax but def or cpdef would have worked too,
- notice the use of void and arguments the syntax is similar as that of C
- The problem with using cdef is that it cannot be called in another jupyter cell, why? The answer is `%%cython --annotate` as this magic command makes this whole cell enviroment

to C-python enviroment this causes problems in other cells which work with their seprate cython enviroments, unlike python this is really problematic, the solution I thought was to add more mud in this by using def in my final function that calls all the other function and implement this function in cython cell, and then i can call the def function as its a pythonic function and implement the C functions in it from other cells

2 Cython Implementation

[7]:
```
%%cython --annotate
cdef void row_swap(list A, int r1, int r2):
    A[r1], A[r2] = A[r2], A[r1]


cdef void agumented(list A, list b):
    for i in range(len(A)):
        A[i].append(b[i][0])

cdef void normalization(list A, int c):
    #defing the datatype of variable beforehand unlike python
    cdef int m = len(A[0])
    cdef int n = len(A)
    ''' This is check snippet for timeit function as I am fundamentally changing
    input matrix repeataly running on timeit gives Index Error'''
    if c >= m or c >= n:  # Check if c is a valid column index of A
        return  # Return early if c is out of range
    try:
        for i in range(m):
            if A[c][c] != 1:
                norm = A[c][c]
                for j in range(m):
                    A[c][j] = A[c][j] / norm
    except ZeroDivisionError:
        for i in range(n):
            for j in range(i + 1, n):
                if A[j][i] != 0:
                    row_swap(A, j, i)
        normalization(A, c)


cdef elimination(A, r, c):
    cdef int m = len(A[0])
    cdef int n = len(A)
    try:
        for i in range(m):
            if A[r][c] != 0:
                ele = A[r][c]
                for j in range(m):
```

```
                    A[r][j] = A[r][j] - ((ele) * A[c][j])
    except IndexError:
        pass

cdef gauss(A, b):
    cdef int m = len(A[0])
    cdef int n = len(A)
    agumented(A, b)
    for i in range(m):
        try:
            normalization(A, i)
        except Exception as e:
            return ['No solutions']

        for j in range(n):
            if i != j:
                elimination(A, j, i)
    return A

#the use of def
def solutions(A, b):
    cdef int n = len(b)
    sol = []
    B = gauss(A, b)
    if B == ['No solutions']:
        return B
    else:
        for i in range(len(B)):
            sol.append(B[i][-1])

        return sol
```

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

[8]:
```
import copy
import numpy as np
a = np.random.rand(15,15).tolist()
b = np.random.rand(15,1).tolist()
A = copy.deepcopy(a)
B = copy.deepcopy(b)
print('This is solving Ax=b with NumPy')
print(np.linalg.solve(np.array(a), np.array(b)))
%timeit np.linalg.solve(np.array(a), np.array(b))
print()
print('This is solving Ax=b with Cython')
print(np.reshape(np.array(solutions(a,b)),(-1,1)))
%timeit -r 1 solutions(a,b)
```

```
print()
print('This is solving Ax=b with Python')
print(np.reshape(solutionsp(A,B),(-1,1)))
#print(np.reshape(np.array(solutionsp(a,b)),(-1,1)))
%timeit -r 1 solutionsp(A,B)
```

```
This is solving Ax=b with NumPy
[[ 0.99597357]
 [-0.53370016]
 [ 0.70838591]
 [ 0.08129779]
 [ 1.04385882]
 [ 0.1654204 ]
 [-0.65976206]
 [-0.1970988 ]
 [ 0.69724069]
 [ 0.84505954]
 [-1.77207775]
 [ 0.0738244 ]
 [ 0.73895165]
 [-0.92908233]
 [-0.46873932]]
41.5 µs ± 2.82 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

This is solving Ax=b with Cython
[[ 0.99597357]
 [-0.53370016]
 [ 0.70838591]
 [ 0.08129779]
 [ 1.04385882]
 [ 0.1654204 ]
 [-0.65976206]
 [-0.1970988 ]
 [ 0.69724069]
 [ 0.84505954]
 [-1.77207775]
 [ 0.0738244 ]
 [ 0.73895165]
 [-0.92908233]
 [-0.46873932]]
7.91 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)

This is solving Ax=b with Python
[[ 0.99597357]
 [-0.53370016]
 [ 0.70838591]
 [ 0.08129779]
```

```
 [ 1.04385882]
 [ 0.1654204 ]
 [-0.65976206]
 [-0.1970988 ]
 [ 0.69724069]
 [ 0.84505954]
 [-1.77207775]
 [ 0.0738244 ]
 [ 0.73895165]
 [-0.92908233]
 [-0.46873932]]
21.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
```

2.1 Observation

It looks like numpy is fasted followed by Cython implematation and then Python implentation, also the Cython implentation is considerably fater then the Python Implementation, but also very slow compared to the numpy function

2.2 Understanding

- Simliarly to factorial problem, Cython is faster is then Python not suprising since we are writing the code in static type instead dynamically liek python, while the code doenst look too cluttered all the functions were simple enough so it didnt caused problems later we will see the draw back of Cython in Spice implentation
- Intresting thing is speed of numpy function, now numpy documentation says all their function as coded in near C syntax and the numpy and python complier calls between the function and Interpreter.
- Now while Cython has C type syntax we are still coding in the Python enviroment and thus still constriced with Python Syntax, Cython is not C code with Python Complier or Python code in C complier but a mess of both with some plugins, so the reason why numpy is still faster is goes it goes full C in the backend of the complier, and thus a lot faster then Cython.

[9]:
```
'''
The parsing_circuit is a python script that contains all the functions I made␣
 ↪in the assigment 2, I am importing
rather copy pasting to avoid needless cluttering of cells.
'''
from parsing_circuit import *
```

[10]:
```
'''
In assigment 2 I have made the simulation circuit in broken cells, and it was␣
 ↪hard to run each cell repeatadly,
so I have combined them all in a single callabel function and also added␣
 ↪dictionary feature to give output in
dictionary format,rather than list as in assigment 2.
```

```
'''
def simuate_circuit(netlist,method):
    '''This is the cleaning and loading part of the week2'''
    #netlist = 'ckt6.txt'
    data_un = parse_circuit(netlist)
    data,flag_ac,flag_dc,w0,mode = process_net(data_un)
    rlc_count,v_count,i_count,line_count = Error(data)
    data_dic = {'element':[],'+node':[],'-node':[],'value':[],'phase':[]}
    load_data(data,data_dic,mode)
    num_nodes = count_nodes(len(data),data_dic)
    ################################################

    branch_count = line_count
    sources = 0   # count source number
    source = 0   # count source number
    sourcev = 0   # count source number
    '''This is the part of defining the matricies of the week2'''
    V = np.zeros((num_nodes,1),dtype='S5')
    I = np.zeros((num_nodes,1),dtype='complex_')
    G = np.zeros((num_nodes,num_nodes),dtype='complex_')
    # count the number of element types that affect the size of the B, C, D, E␣
 ↪and J arrays
    k = v_count
    kk = i_count#number of branches with unknown currents for MNA
    if k != 0 or kk!=0:
        B = np.zeros((num_nodes,k),dtype='complex_')
        C = np.zeros((k,num_nodes),dtype='complex_')
        D = np.zeros((k,k),dtype='complex_')
        E = np.zeros((k,1),dtype='complex_')
        J = np.zeros((k,1),dtype='S5')
    ################################################
    '''This is the multiple freqeuncy handling part '''
    if flag_ac and flag_dc:    #Cathing case of AC DC mixed circuit
        return '***AC and DC mixed circuit found***'
    ################################################
    '''This is the matrix making part of the assigment 2 most of the matrices␣
 ↪had same for loop so they all comnbine
    rather niciely'''
    for i in range(branch_count):
        ##############GGGGGGGGGGGGG#################
        n1 = data_dic['+node'][i] #first node is +
        n2 = data_dic['-node'][i] #second node is -
        # iterate through each element and save them in temporary variable g
        # then appy the rules to make G matrix
        comp = data_dic['element'][i][0]   #The first letter shows element type
        if comp == 'R':
```

```
            g = 1/data_dic['value'][i]      #1/R
        if comp == 'L':
            if mode == 'ac':
                g = 1/(1j*w0*data_dic['value'][i])  #1/jwL got AC
            else:
                g = 1e10                            #for dc L behaves as close␣
↪circuit in steady state
        if comp == 'C':
            if mode == 'ac':
                g = 1j*w0*data_dic['value'][i]      #jwC
            else:
                g = 0                               #for dc C behaves as open␣
↪circuit in steady state

        if (comp == 'R') or (comp == 'L') or (comp == 'C'):
            # If neither side of the element is connected to ground
            # then subtract it from appropriate location in matrix.
            if (n1 != 0) and (n2 != 0):
                G[n1-1,n2-1] += -g
                G[n2-1,n1-1] += -g

            # If node 1 is connected to ground, add element to diagonal of␣
↪matrix
            if n1 != 0:
                G[n1-1,n1-1] += g

            # same for for node 2
            if n2 != 0:
                G[n2-1,n2-1] += g
        ##############GGGGGGGGGGGGG#################
        ##############BBBBBBCCCCCCC#################
        if comp == 'V':
            if v_count > 1:
                if n1 != 0:
                    B[n1-1][sources] = 1
                    C[source][n1-1] = 1
                if n2 != 0:
                    B[n2-1][sources] = -1
                    C[source][n2-1] = -1
                sources += 1   #increment source count
                source += 1
            else:
                if n1 != 0:
                    B[n1-1] = -1
                    C[0][n1-1] = -1
                if n2 != 0:
                    B[n2-1] = +1
```

```
                    C[0][n2-1] = +1
        ##############BBBBBBCCCCCCC#################
        ##############IIIIIIIIIIII##################
        if comp == 'I':
            #g = data_dic['element'][i]
            g = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i]) # For AC␣
↪in, case dc phase = 0
            #g = data_dic['value'][i]
            # sum the current into each node
            if n1 != 0:
                I[n1-1] = I[n1-1] - g
            if n2 != 0:
                I[n2-1] = I[n2-1] + g
        ##############IIIIIIIIIIII##################
    ##############EEEEEEJJJJJJJ#####################
    for i in range(branch_count):
        # process all the passive elements
        #get 1st letter of element name
        comp = data_dic['element'][i][0]
        if comp == 'V':
            J[sourcev] = 'I_{:s}'.format(data_dic['element'][i])
            #E[source] = data_dic['element'][i]
            E[sourcev] = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i])
            sourcev += 1
    ##############EEEEEEJJJJJJJ#####################
    ##############AAAAAAAAAAAAAA####################
    n = num_nodes
    m = v_count
    A = np.zeros((m+n,m+n),dtype='complex_')
    for i in range(n):
        V[i] = np.string_('v') + np.string_(f'{i+1}')
        for j in range(n):
            A[i,j] = G[i,j]

    if v_count > 1:
        for i in range(n):
            for j in range(m):
                A[i,n+j] = B[i,j]
                A[n+j,i] = C[j,i]
    else:
        for i in range(n):
            A[i,n] = B[i]
            A[n,i] = C[0][i]
    ##############AAAAAAAAAAAAAA####################

    X = np.concatenate((V,J))
    Z = np.concatenate((I,E))
```

```
    X = X.astype(str).tolist()
    if method == 'cython':
        ans = solutions(A.tolist(),Z.tolist())
        #dictionary feature added
        solution_dictionary = {key[0]: value for key, value in zip(X, ans)}
        return solution_dictionary
    elif method == 'python':
        ans = solutionsp(A.tolist(),Z.tolist())
        #dictionary feature added
        solution_dictionary = {key[0]: value for key, value in zip(X, ans)}
        return solution_dictionary
    else:
        return 'Incorrect method'
```

2.3 Example netlists

[11]:
```
print(simuate_circuit('ckt1.txt','python'))
print()
print(simuate_circuit('ckt1.txt','cython'))
```

```
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}

{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
```

[12]:
```
print(simuate_circuit('ckt2.txt','python'))
print()
print(simuate_circuit('ckt2.txt','cython'))
```

```
***AC and DC mixed circuit found***

***AC and DC mixed circuit found***
```

[13]:
```
print(simuate_circuit('ckt3.txt','python'))
print()
print(simuate_circuit('ckt3.txt','cython'))
```

```
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}

{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}
```

[14]:
```
print(simuate_circuit('ckt4.txt','python'))
print()
```

```
print(simuate_circuit('ckt4.txt','cython'))
```

```
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}

{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
```

[15]:
```
print(simuate_circuit('ckt5.netlist','python'))
print()
print(simuate_circuit('ckt5.netlist','cython'))
```

```
{'v1': (10+0j), 'I_V1': (-1-0j)}

{'v1': (10+0j), 'I_V1': (-1-0j)}
```

[16]:
```
print(simuate_circuit('ckt6.txt','python'))
print()
print(simuate_circuit('ckt6.txt','cython'))
```

```
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}

{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
```

3 Cython Implementation

3.0.1 First re-factoring the parsing_circuit

The below code is cython implementation of cython of the functions of week2 assigment and the code has lots of syntax not to mention that unlike python, errors of the cython are not very well described by the devlopers, hence this is hard to read and debug but fast(?) then python

Overall I dont think this is much helpful in anyways as I am just parsing and refactoring/cleaning the data in diffrent format, Cython in my opininon is fater than Python because it makes CPU level computation faster here I am not doing any CPU related stuff like arthemeitic or matrix solving, so the speed of Cython can be fatser because I am still defining the variables before hand but, objectiviely speaking it could not be very fast as we will see later

[17]:
```
%%cython --annotate
# import libarires again bcz cython
import re
import numpy as np

class MoreFreqError(Exception):
    pass

'''I am using def here instead of cdef and that will become more clear␣
 ↪afterwards, it is realted to problem
```

```
I described in th Gauss code, the problem of importing cython functions in pure␣
 ↪python enviroments'''

'''

'''

def cy_parse_circuit(netlist):
    #read the file and load it contents to a list
    cdef list para
    try:
        with open(netlist,'r') as f:
            para = f.read().splitlines()
    except FileNotFoundError:
        print('Make sure the path of the .netlist or .txt file is same as this␣
 ↪notebook')

    #find .circuit and .ac or .end and remove others unneccesaary things
    '''intilialize the variable bcz cython'''
    cdef int index1,index2
    cdef list string
    cdef list w = []
    cdef str line
    cdef list w_st
    cdef list filtered = []
    cdef list l
    '''Cython doesnt allow using cdefs in try except block or if-else, somehow␣
 ↪no-one has raised this issue'''
    '''The below code is actuallly same as python code as this is just data␣
 ↪parsing and refactoring there is not
    so much of C involved'''
    try:
        index1 = para.index('.circuit')
        index2 = para.index('.end')
        string = re.findall('\.ac.*', '\n'.join(para))

        #update the content to have data between .circuit and .end using string␣
 ↪slicing method
        para = para[index1:index2+1]

        #update content to have data regarding frequencies
        para.extend(string)

        #to check if there is only one frequency
        if len(string)>=1:
            for line in string:
                w_st = line.split()
```

```
                w.append(int(w_st[2]))
        try:
            if max(w) != min(w):
                raise MoreFreqError
        except MoreFreqError:
            print('More than one AC frequencys')
        except ValueError:
            pass

    #remove comments from the paragraph or the strings after '#'
        for line in para:
            #l = re.sub(r'#.*','',line)
            l = line.split('#')
            filtered.append(l[0])
        #the proccesed content is filtered
        return filtered
    except ValueError:
        print('Invalid Netlist')
    #find .ac in the content as .ac comes after .end
# collect operating frequency


def cy_delete_headings(circ):
    '''defing the variables'''
    cdef list data
    cdef str mode
    cdef list k
    data = [' '.join(x.split()) for x in circ]
    mode = circ[-1]
    data = [n for n in data if not n.startswith('.')]
    if mode == '.end':
        return (0,'dc',data)
    else:
        k = mode.split()
        return (k[2],'ac', data)

def cy_process_net(netlist):
    (w0,mode,y) = cy_delete_headings(netlist)
    '''definng the variables'''
    w0 = int(w0)
    cdef int flag_ac = 0
    cdef int flag_dc = 0
    cdef str line
    cdef list psline
    #Making useful data list
    cdef list data = []
    try:
```

```
        for line in y:
            psline = line.split(' ')
            if psline[1] == 'GND':       # Changing the GND node to 0 node for␣
 ↪calculation purposes
                psline[1] = '0'
            if psline[2] == 'GND':
                psline[2] = '0'
            #Handling for non nuumeric nodes
            if psline[1][0] == 'n':
                psline[1] = psline[1][1:]
            if psline[2][0] == 'n':
                psline[2] = psline[2][1:]

            if psline[3] == 'dc' or psline[3] == 'ac': #Could have use mode
                if psline[3] == 'dc':
                    flag_dc = 1
                else:
                    flag_ac = 1
                del psline[3]                        # delete 'ac' or 'dc'␣
 ↪strings
            x = ' '.join(psline)
            data.append(x)
        return data,flag_ac,flag_dc,w0,mode
    except:
        print('Invalid Netlist')

#making a loading function that takes values from data
#and stores to dictionary for passive elements
def cy_load_rlc(line_number,data,data_dic):
    '''definng the variables'''
    cdef list line
    line = data[line_number].split()
    data_dic['element'] += [line[0]]
    data_dic['+node'] += [int(line[1])]
    data_dic['-node'] += [int(line[2])]
    data_dic['value'] += [float(line[3])]
    if line[0][0] == 'R':                #for ac circuit
        data_dic['phase'] += [0.0]
    elif line[0][0] == 'L':
        data_dic['phase'] += [np.pi/2]
    elif line[0][0] == 'C':
        data_dic['phase'] += [np.pi/2]

#making a loading function that takes values from data
#and stores to dictionary for active elements
def cy_load_sources(line_number,data,data_dic,mode):
    '''defing the variables'''
```

```
    cdef list line
    line = data[line_number].split()
    if mode == 'dc':
        data_dic['element'] += [line[0]]
        data_dic['+node'] += [int(line[1])]
        data_dic['-node'] += [int(line[2])]
        data_dic['value'] += [float(line[3])]
        data_dic['phase'] += [0.0]
    elif mode == 'ac':
            data_dic['element'] += [line[0]]
            data_dic['+node'] += [int(line[1])]
            data_dic['-node'] += [int(line[2])]
            data_dic['value'] += [float(line[3])]
            try:                                        #for ac mode
                data_dic['phase'] += [float(line[4])]
            except IndexError:
                data_dic['phase'] += [0.0]

    else:
        return('Unknown mode')


def cy_Error(data):
        # number of components and branches
    '''defing the variables'''
    cdef int rlc_count = 0
    cdef int v_count = 0
    cdef int i_count = 0
    cdef int branch_count = 0     #no need for this as line_count =␣
 ↪branch_count but done anyways
    cdef int line_count = len(data)
    cdef int value_count
    cdef str comp



    #counting number of passive elements rlc and active elemets V ,I
    for i in range(line_count):
        comp = data[i][0]      #the 0th index shows name of component
        value_count = len(data[i].split()) #value count is a nested list␣
 ↪containg a list
                                           #that contains name of component
                                           #postive node, negative node and␣
 ↪value of
                                           #the component
        if (comp == 'R') or (comp == 'L') or (comp == 'C'): #add capatlize␣
 ↪function
```

```
            if value_count != 4:           # handling errors
                print("Error in netlist")
                print('Error in passive elements RLC')
            rlc_count += 1
            branch_count += 1
        elif comp == 'V':
            if value_count != 5 and value_count != 4:
                print("Error in netlist ")
                print('Error in Voltage source/node')
            v_count += 1
            branch_count += 1
        elif comp == 'I':
            if value_count != 4 and value_count != 5:
                print("Error in netlist ")
                print('Error in Current source/branch')
            i_count += 1
            branch_count += 1
        else:
            print('Unknow element found in netlist')

    return rlc_count,v_count,i_count,branch_count

def cy_count_nodes(line_count,data_dic):
    cdef list n
    n = [([0]*(line_count+1)) for i in range(1)]
    for i in range(line_count - 1):
        n[0][data_dic['+node'][i]] = data_dic['+node'][i]
        n[0][data_dic['-node'][i]] = data_dic['-node'][i]
        #largetst node
    if max(data_dic['-node']) > max(data_dic['+node']):
        largest = max(data_dic['-node'])
    else:
        largest =  max(data_dic['+node'])

    # check for unfilled elements, skip node 0
    for i in range(1,largest):
        if n[0][i] == 0:
            print('Error in node order')
    return largest



def cy_load_data(data,data_dic,mode):
    cdef int line_count = len(data)
    cdef int i
    cdef str comp
    #using data to make a dictionary that can help in extracting data easily
```

```
    for i in range(line_count):
        comp = data[i][0]
        if (comp == 'R') or (comp == 'L') or (comp == 'C'):
            cy_load_rlc(i,data,data_dic)
        elif (comp == 'V') or (comp == 'I'):
            cy_load_sources(i,data,data_dic,mode)
        else:
            print('Unknown Elelment Error')

    # count number of nodes
    #num_nodes = count_nodes() #maximum node number in the circuit

    return data_dic
```

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

3.0.2 This is the simulation function that calls the above functions to get the data from the netlist and make matricis, which will be solved using the the gaussian solver of Python and Cython

- In the end I will compare all 4 combinations of simulators and the gaussian solvers.
- Below is the code fo simulator in Cython, I have to reimport the libraries bcz this is seprate python enviroment from the above Cython cells and that cause the importing errors, because of this I made all the above function using `def` instead `cdef` using cdef I could have called the functions in the same cells but not to other cells.
- Another problem with cdef is i think I cannot call it using `from __main__ import *`.
- What is `from __main__ import *`?
  - In the Gaussian elemination I solved the problem of calling functions from cython enviroment to python enviroment by making a python stype function(`solutions`) calling cython style functions(other funcs) in the same code cell and then called the my pythonic function in diffrent cells,because the `solutions` was small and short it didnt looked cluttery in the same cell and worked nicely.
  - Here the simulation function has to make matricis and get data form the above parsing cell, this if done in single cell would be teribble looking,reading and debugging.
  - To solve this I decided to think like this whole jupyter notebook as a libray and import every function it has in this cell, while this is really a bad idea if making a proffesional program for this assigment this is the easy way for it in my opinion.
  - I also have to import other libraies as needed the are not functions so they cannot be imported thinking like functions(I think this is called dependencies not so sure).

Again here I am fundamentally making a matrix, there is some arthemeitc operations like addition and subraction but ot CPU consuming parts like matrix multiplication, thus fundamentally this should be slighlty fatser than Python implementation but not so fast

[18]:
```
%%cython --annotate

import re
import numpy as np
from __main__ import *

def cython_simuate_circuit(netlist,method):
    #netlist = 'ckt6.txt'
    cdef list data_un
    data_un = cy_parse_circuit(netlist)
    cdef list data
    cdef int␣
 ↪flag_ac,flag_dc,line_countrlc_count,v_count,i_count,num_nodes,branch_count
    cdef dict data_dic
    cdef str mode
    data,flag_ac,flag_dc,w0,mode = cy_process_net(data_un)
    rlc_count,v_count,i_count,line_count = cy_Error(data)
    data_dic = {'element':[],'+node':[],'-node':[],'value':[],'phase':[]}
    cy_load_data(data,data_dic,mode)
    num_nodes = cy_count_nodes(len(data),data_dic)


    branch_count = line_count
    cdef int sources = 0   # count source number
    cdef int source = 0   # count source number
    cdef int sourcev = 0   # count source number
    cdef int k,kk,n1,n2
    cdef complex g
    cdef str comp
    V = np.zeros((num_nodes,1),dtype='S5')
    I = np.zeros((num_nodes,1),dtype='complex_')
    G = np.zeros((num_nodes,num_nodes),dtype='complex_')
    # count the number of element types that affect the size of the B, C, D, E␣
 ↪and J arrays
    k = v_count
    kk = i_count#number of branches with unknown currents for MNA
    if k != 0 or kk!=0:
        B = np.zeros((num_nodes,k),dtype='complex_')
        C = np.zeros((k,num_nodes),dtype='complex_')
        D = np.zeros((k,k),dtype='complex_')
        E = np.zeros((k,1),dtype='complex_')
        J = np.zeros((k,1),dtype='S5')

    if flag_ac and flag_dc:    #Cathing case of AC DC mixed circuit
        return '***AC and DC mixed circuit found***'
    for i in range(branch_count):
        n1 = data_dic['+node'][i] #first node is +
```

```
        n2 = data_dic['-node'][i] #second node is -
        # iterate through each element and save them in temporary variable g
        # then appy the rules to make G matrix
        comp = data_dic['element'][i][0]   #The first letter shows element type
        if comp == 'R':
            g = 1/data_dic['value'][i]     #1/R
        if comp == 'L':
            if mode == 'ac':
                g = 1/(1j*w0*data_dic['value'][i])  #1/jwL got AC
            else:
                g = 1e10                           #for dc L behaves as close␣
↪circuit in steady state
        if comp == 'C':
            if mode == 'ac':
                g = 1j*w0*data_dic['value'][i]      #jwC
            else:
                g = 0                               #for dc C behaves as open␣
↪circuit in steady state

        if (comp == 'R') or (comp == 'L') or (comp == 'C'):
            # If neither side of the element is connected to ground
            # then subtract it from appropriate location in matrix.
            if (n1 != 0) and (n2 != 0):
                G[n1-1,n2-1] += -g
                G[n2-1,n1-1] += -g

            # If node 1 is connected to ground, add element to diagonal of␣
↪matrix
            if n1 != 0:
                G[n1-1,n1-1] += g

            # same for for node 2
            if n2 != 0:
                G[n2-1,n2-1] += g


        if comp == 'V':
            if v_count > 1:
                if n1 != 0:
                    B[n1-1][sources] = 1
                    C[source][n1-1] = 1
                if n2 != 0:
                    B[n2-1][sources] = -1
                    C[source][n2-1] = -1
                sources += 1   #increment source count
                source += 1
            else:
```

```
                if n1 != 0:
                    B[n1-1] = -1
                    C[0][n1-1] = -1
                if n2 != 0:
                    B[n2-1] = +1
                    C[0][n2-1] = +1

        if comp == 'I':
            #g = data_dic['element'][i]
            g = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i]) # For AC␣
↪in, case dc phase = 0
            #g = data_dic['value'][i]
            # sum the current into each node
            if n1 != 0:
                I[n1-1] = I[n1-1] - g
            if n2 != 0:
                I[n2-1] = I[n2-1] + g

    # generate the E matrix
    for i in range(branch_count):
        # process all the passive elements
        #get 1st letter of element name
        comp = data_dic['element'][i][0]
        if comp == 'V':
            J[sourcev] = 'I_{:s}'.format(data_dic['element'][i])
            #E[source] = data_dic['element'][i]
            E[sourcev] = data_dic['value'][i]*np.exp(1j*data_dic['phase'][i])
            sourcev += 1
    #The A matrix is (m+n) by (m+n) and will be developed as the combination of␣
↪4 smaller matrices, G, B, C, and D.
    n = num_nodes
    m = v_count
    A = np.zeros((m+n,m+n),dtype='complex_')
    for i in range(n):
        V[i] = np.string_('v') + np.string_(f'{i+1}')
        for j in range(n):
            A[i,j] = G[i,j]

    if v_count > 1:
        for i in range(n):
            for j in range(m):
                A[i,n+j] = B[i,j]
                A[n+j,i] = C[j,i]
    else:
        for i in range(n):
            A[i,n] = B[i]
            A[n,i] = C[0][i]
```

```
    X = np.concatenate((V,J))
    Z = np.concatenate((I,E))

    X = X.astype(str).tolist()
    if method == 'cython':
        ans = solutions(A.tolist(),Z.tolist())
        solution_dictionary = {key[0]: value for key, value in zip(X, ans)}
        return solution_dictionary
    elif method == 'python':
        ans = solutionsp(A.tolist(),Z.tolist())
        solution_dictionary = {key[0]: value for key, value in zip(X, ans)}
        return solution_dictionary
    else:
        return ('Incorrect Method')
```

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

3.1 Example Circuits using Cython

[19]:
```
print(cython_simuate_circuit('ckt1.txt','cython'))
print()
print(cython_simuate_circuit('ckt1.txt','python'))
```

```
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}

{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
```

[20]:
```
print(cython_simuate_circuit('ckt2.txt','cython'))
print()
print(cython_simuate_circuit('ckt2.txt','python'))
```

```
***AC and DC mixed circuit found***

***AC and DC mixed circuit found***
```

[21]:
```
print(cython_simuate_circuit('ckt3.txt','cython'))
print()
print(cython_simuate_circuit('ckt3.txt','python'))
```

```
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}

{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}
```

[22]:
```
print(cython_simuate_circuit('ckt4.txt','cython'))
print()
print(cython_simuate_circuit('ckt4.txt','python'))
```

```
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}

{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
```

[23]:
```
print(cython_simuate_circuit('ckt5.netlist','cython'))
print()
print(cython_simuate_circuit('ckt5.netlist','python'))
```

```
{'v1': (10+0j), 'I_V1': (-1-0j)}

{'v1': (10+0j), 'I_V1': (-1-0j)}
```

[24]:
```
print(cython_simuate_circuit('ckt6.txt','cython'))
print()
print(cython_simuate_circuit('ckt6.txt','python'))
```

```
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}

{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
```

3.2 Comparsion Task

Here I will compare 4 combinations: * Simulator written in Python and Matric Solver in Python * Simulator written in Cython and Matric Solver in Cython * Simulator written in Cython and Matric Solver in Python * Simulator written in Python and Matric Solver in Cython

And then will observe the time analysis of each combination, I have not inculded numpy solver because it will be obsiouvly faster than all this implementation by large margin

[25]:
```
print('For Ckt1')
net = 'ckt1.txt'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
```

```
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt1
Simulator: Python, Matrix Solver: Python
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
63.5 µs ± 246 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
48.6 µs ± 490 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Python, Matrix Solver: Cython
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
57.7 µs ± 448 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
{'v1': (5+0j), 'v2': (4.999999999500001e-10+0j), 'I_V1': (-4.9999999995-0j)}
55.8 µs ± 510 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[26]:
```
print('For Ckt2')
net = 'ckt2.txt'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt2
Simulator: Python, Matrix Solver: Python
***AC and DC mixed circuit found***
47.8 µs ± 356 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
***AC and DC mixed circuit found***
35.2 µs ± 342 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
Simulator: Python, Matrix Solver: Cython
***AC and DC mixed circuit found***
46.5 µs ± 72.8 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
***AC and DC mixed circuit found***
34.8 µs ± 91.6 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[27]:
```
print('For Ckt3')
net = 'ckt3.txt'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt3
Simulator: Python, Matrix Solver: Python
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}
134 µs ± 1.25 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}
86.2 µs ± 2.13 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Python, Matrix Solver: Cython
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
(-0.004970760233918129-0j)}
100 µs ± 135 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
{'v1': (10+0j), 'v2': (5.029239766081871+0j), 'v3': (2.5730994152046778+0j),
'v4': (1.4035087719298247+0j), 'v5': (0.9356725146198832+0j), 'I_V1':
```

```
(-0.004970760233918129-0j)}
120 µs ± 1.29 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[28]:
```
print('For Ckt4')
net = 'ckt4.txt'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt4
Simulator: Python, Matrix Solver: Python
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
80.7 µs ± 571 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
56 µs ± 737 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Python, Matrix Solver: Cython
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
66.2 µs ± 512 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
{'v1': (10+0j), 'v2': (5.555555555555556+0j), 'v3': (3.7037037037037037+0j),
'I_V1': (-2.222222222222222-0j)}
69.9 µs ± 297 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[29]:
```
print('For Ckt5')
net = 'ckt5.netlist'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
```

```
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt5
Simulator: Python, Matrix Solver: Python
{'v1': (10+0j), 'I_V1': (-1-0j)}
47 µs ± 108 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
{'v1': (10+0j), 'I_V1': (-1-0j)}
38.7 µs ± 47.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Python, Matrix Solver: Cython
{'v1': (10+0j), 'I_V1': (-1-0j)}
43.7 µs ± 55 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
{'v1': (10+0j), 'I_V1': (-1-0j)}
41.8 µs ± 60.6 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[30]:
```
print('For Ckt6')
net = 'ckt6.txt'
print('Simulator: Python, Matrix Solver: Python')
print(simuate_circuit(net,'python'))
%timeit simuate_circuit(net,'python')
print()
print('Simulator: Cython, Matrix Solver: Cython')
print(cython_simuate_circuit(net,'cython'))
%timeit cython_simuate_circuit(net,'cython')
print()
print('Simulator: Python, Matrix Solver: Cython')
print(simuate_circuit(net,'cython'))
%timeit simuate_circuit(net,'cython')
print()
print('Simulator: Cython, Matrix Solver: Python')
print(cython_simuate_circuit(net,'python'))
%timeit cython_simuate_circuit(net,'python')
```

```
For Ckt6
Simulator: Python, Matrix Solver: Python
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
76.7 µs ± 98.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Cython
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
54 µs ± 26.4 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Python, Matrix Solver: Cython
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
63.5 µs ± 106 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Simulator: Cython, Matrix Solver: Python
{'v1': (-5+0j), 'v2': (-5+0j), 'v3': (-5-0j), 'I_V1': (-0-0j)}
67.7 µs ± 523 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

3.3 Observation:

- We can observe that Cython-Cython implementation is fastest, not suprsing sing all varable defining and solving is done in Cyton itself
- We also see that Python-Python implemenattion is slowest, as expected as Python being dynamical language slows it down consideraly compared to cython implemntaton
- Intresting stuff is of Python-Cython implementation, the matrix is made in python while solved using Cython, this has almost same speed as the Cython-Python implemetation, mayed code being taking imports and memory is cause of this?
- Lastly Cython-Python implemtation with slightly more speed that Python-Cython, this was also expected

3.4 Understanding:

- Obviosuly Cyton-Cython implemenatation is the fastest and Python-Python implemtation is slowest, the intresting stuff is in the 3rd and 4th case, the speed of 3rd and 4th case is rather close, but we observe that 4th case where matrix slover is in python takes long time than cython matrix solver, I think this boils down to computation of the code, the simulator is just a function that makes a matrix from data given there is logically no complex computation needed in this process, and thus speed of this either implemented in Cython or Python will be almost same,(python being slow due to dynamic), whereas matrix solver is computationally expensive part, here due to involvment of CPU resources Cython is faster and Python slows down.Thus if you compare Python-Python and Cython-Python you will notice diffrence of 10us and similar diffrence in other two.

3.5 Conculsion:

- Both Python and Cython have advantages and disadvantages, but it is upto the devloper to desgin what he/she wants. I my opinion Python should be used were there is not too much

use of CPU resources as it can be easy to read,faster to write and will have almost same speed as Cython but with less hassel to maintain.

- On the other hand Cython should be used when CPU resorces are extensively used as it will speed up the program considerably and most likey this wont be updated as frequently as other programs calling it, while it could be hard to debug devloper after making it once most lieky wont update it as frequenly as other functions, but the bottom line is if CPU is used extensively using Cython is wise choice