EE2703 : Circuit Solver

Annangi Shashnk Babu (ee21b021)

February 8, 2023

0.1 Comparison of factorial computation time.

[1]:
```
import numpy as np
x = np.random.randint(0,100)
```

[2]:
```
def first_factorial(n):
    if n == 0:
        return 1
    return first_factorial(n-1)*n
ans = first_factorial(x)
%timeit first_factorial(x)
```

```
693 ns ± 11.7 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

The above function is a recusive function, it calls it-self to reapeat the code based on the property of factorial function, $(n)! = (n-1)! * n$

[3]:
```
def second_factorial(n):
    ans = 1
    for i in range(n):
        ans *= (i+1)
    return ans
ans = second_factorial(x)
%timeit second_factorial(x)
```

```
408 ns ± 6.94 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

The above funciton is a iterative funciton and loop through some part of the code to find the factorial of a given number and as

Recursion has the overhead of repeated function calls, that is due to repetitive calling of the same function. So, recursive solution generally takes more time than the iterative solution (also conformed by timeit function)

[4]:
```
%timeit np.math.factorial(x)
```

```
59.7 ns ± 0.356 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

The numpy.math.factorial() function which is based on C is many-folds faster than that a code written in python.

0.2 Linear Equation Solver

Solving a electrical circuit essesntially comes down to solving a system of linear equation, hence we first build a function to solve linear equations

[5]:
```
#custom error class
class MyLinalError(Exception):
    """raises custom errors"""

    def __init__(self,message):
        """constructor for the error class

        :message: the message that is displayed when a error occurs

        """
        Exception.__init__(self)
        self._message = message
```

The solution to a system of equations do not change if we substract or add any 2 (distinct) of the equations picked randomly (infact any linear combination of the picked equations) and replace the newly created equation in place of any of the two picked equations.

for example : 5x + 6y = 11 …eq1 and x + y = 2 …eq2 have the same solutions as the system of equations : 4x + 5y = 9 (eq1 - eq2) and x + y = 2

exploiting this fact we first convert the coefficient matrix(n x n dimensions) to a upper triangular matrix by first picking the topmost row and substract all the row's below (with a multipling factor) such as to elemenate all the elements in 1st coloum except for the pivot element(picking multiplying factor becomes more easier if leftmost element in the topmost row is 1, this is achived by dividing the wholerow by a[0][0], including the constant matrix). (remember to perform the same substractions in the costant matrix also) now pick the (n-1)x(n-1) matrix(the solution to the this matrix combined with the top row gives the solution to the system of equation so now we have to find the solution to the n-1xn-1 matrix) and perform the same operations to make all the elements below a[1][1] in the column to be zero and repeat it until you reach (like reccursion)a 1x1 matrix, now we are left with a uppter triangular matrix

note :- assume a and b as augumented matrix,so any changes performed on 'a' shall also be performed on 'b'

we have to convert the uppter triangular matrix coefficient matrix to a identity matrix for finding the solution Now we have to modify process done to find the upper triangular matrix, instead of starting from the above row, we start from the bottomost row (n x n dimensions) and come to the topmost row (1x1 matrix)in a same recursive way , finaly the b matrix represents the solutions because 'a' matrix is identity matrix.

[24]:
```
def my_linal_solver(a,b):
    """will solve the system of equation given

    :a: The coefficient matrix
    :b: The constants matrix
    :returns: solution of the system of linear equation ax = b

    raises : MyLinalError when the matrix 'a' is singular
    """
    #first converting the input to a numpy array and check for dimentions of a␣
↪and b matrix
    #and raising appropriate errors

    try:
        a = np.array(a,dtype = complex)
        b = np.array(b,dtype = complex)
        if len(a) != len(b):
            raise MyLinalError
    except ValueError as e:
        print('error : The dimentions of the matrix are not correct')
        return
    except MyLinalError as e :
        print(format(e._message))
        return


    for i in range(0,len(a)):
        #partial pivoting (to decrease errors)
        #our method involves us to first divide the whole row by a[i][i]␣
↪(taking norm, for ixi matrix in recursive method)
        #and we dont want a[i][i] to be a small number as this leads to blowing␣
↪up of numbers in matrix so we swap
        #with a row which is in between (i,len(a)) (we cant select a row in␣
↪[0,i) as this wont lead to a upper triangular matrix)
        #and has the highest absolute value comperted to a[j][i] for j␣
↪belonging to [i,len(a)) this leads to stable solution rather that blowing up␣
↪in values.


        # finding the index of row which has the highest absolute value
        index = i # max_value index
        for j in range(i,len(a)):
            if abs(a[j][i]) > abs(a[index][i]) :
                index = j
        #swapping
        a[[i,index]] = a[[index,i]]
        b[[i,index]] = b[[index,i]]
        # if the maximum element is also zero then the matrix is singular,␣
↪consider ixi matrix formed with the first coloum entries all as
        #zeros. it will have the none or infinite solutions so the solution to␣
↪the previous matrix i+1 x i+1 also has none or infinite solution
```

```
        # similarly the nxn matrix also has none or infinite solution.
        try :
            if abs(a[i][i]) == 0:
                raise MyLinalError("Exception occured : Matrix is singular")
        except MyLinalError as e:
            print (format(e._message))
            return 1

        #taking norm now
        for j in range(i + 1,len(a[i])):
            a[i][j] /=  a[i][i]
        b[i] /= a[i][i]
        a[i][i] = 1

        #substracting a[i] row from a[j] row with suitable multiplying factor
        for j in range(i + 1,len(a)):
            temp = (a[j][i]/a[i][i])
            for k in range(i + 1,len(a[i])):
                a[j][k] -= (temp)*a[i][k]
            b[j] -= (temp)*b[i]
            a[j][i] = 0

    #now we have the upper triangular matrix so we reverse our process from␣
 ↪bottom to top, making elements in coefficient matrix zero
    for i in reversed(range(0,len(a))):
        for j in range(0,i):
            b[j] -= a[j][i]*b[i]
            a[j][i] = 0      #a[j][i] = a[j][i] - (a[j][i]/ a[i][i]) * a[i][i]␣
 ↪= 0)  a[i][i] == 1 so multiplying factor is a[j][i] itself
    return b
A = np.random.randint(1,100,size=(10,10))
B = np.random.randint(1,100,size=(10))
#A = [[0,1,1],[3,2,1],[2,3,7]]
#B = [5,10,29]

c= (my_linal_solver(A,B))
print(np.allclose(np.dot(A, c), B)) #Verification prints true if c is the␣
 ↪solution
#print(c)
%timeit my_linal_solver(A,B)
%timeit np.linalg.solve(A,B)
```

```
True
188 µs ± 806 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
12.7 µs ± 3.2 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

0.3 Circuit Solver

We have to build a matrix from the given netlist and pass it through our linear equation solver to get the unkown values the main task here is to build the circuit using the concept of modified nodal analysis. The stamp of each of the element (i.e the effect of each of the element) must be taken care of to get the final matrix.

here, we use the concept of object oriented programming to manage the data of each element until updated in the MNA matrix, in from of class attributes, to make the code less messier.

[22]:
```python
import numpy as np
error_flag = 0
class MyCircuitSolver(Exception):

    """Raises custom errors"""

    def __init__(self,message):
        """constructor
        :message: custom message to be shown

        """
        Exception.__init__(self)

        self._message = message

def parse_value(string):
    """converts the string in to a number to pursue computations

    :string: takes a string as input (ex : 2K, 3m ,5u )
    :returns: returns a float value representing the string (ex :  2k --> 2000)

    """
    convert_dict = {'K': '1e3','k': '1e3', 'M': '1e6', 'G': '1e9', 'n' :␣
 ↪'1e-9','N' : '1e-9' ,'m' : '1e-3','u':'1e-6','p' :
                        '1e-12'}
    t_val =''
    for letter in string.split() :
        if letter in convert_dict.keys():
            t_val = float(t_val)*float(convert_dict[letter])
            break
        t_val += letter
    return float(t_val)

class Circuit_Element():

    """General Class for R,L,C,V,I circuit elements"""

    def __init__(self,line):
        """constructor for this class

        :line: Takes the a line from spice netlist as a input and parses it

        """
        #attributes of this class
        self._line = line
        self._split_line = line.split()
        self._name = self._split_line[0]
        self._first_node = self._split_line[1]
        self._second_node = self._split_line[2]

        if len(self._split_line) == 4:  #ex : R1 GND 1 2k  # so passive element␣
↪will have 4 as length of split line active element
                                                        #    has 4 as the␣
↪length of the split line
            # r,l or c
            self._type = 'passive'
            self._value = complex(parse_value(self._split_line[3]),0)
        else :
            self._type = 'active'
            if self._split_line[3] == 'dc':
                # dc source v/i
                self._ac_flag = 0
                self._value = complex(parse_value(self._split_line[4]),0)
            else :
                #ac source  v/i
                self._ac_flag = 1
                #phase is assumed to be given in degres !!!
                self._phase = (float(self._split_line[5])/180)*(np.pi)
                self._value = complex(parse_value(self._split_line[4])*np.
↪cos(self._phase),parse_value(self._split_line[4])*np.sin(self._phase))

    def name(self):
        """conveys the name of the element
        """
        return self._name
    def impedance(self, omega):
        """calculates the impedance of the element for given omega
        :omega: the operating frequency
        :returns: impedance of the passive element

        """
        if self._name[0] == 'R':
            return self._value
        elif self._name[0] == 'L':
            return complex(0,self._value * omega)
        elif self._name[0] == 'C' :
            if omega == 0:
                # impedance is infinity for dc sources to get steady state value
                return complex(0,-np.finfo(np.float64).max)
            return complex(0,-(1)/(omega*self._value))
    def admittance(self, omega):
        """calculates the admitance of the passive element for given omega

        :omega: The operating frequency
        :returns: admittance of the passive element

        """
        if omega != 0 or self._name[0] == 'R' :
            return 1/self.impedance(omega)
        elif self._name[0] == 'L':
            return complex(0,np.finfo(np.float64).max)
        elif self._name[0] == 'C' :
            return complex(0,0)

    def type(self):
        """says whether the element is active or not
        """
        return self._type
    def first_node(self):
       """returns the first node from the parsed line
       """
       return self._first_node
    def second_node(self):
       """returns the second node from the parsed line
       """
       return self._second_node
    def value(self):
       """returns the value of the element from parsed line
       """
       return element._value

def _file_lines_of(filename):
    """used to divide the file into lines

    :filename: The name of the file that should be opened
    :returns: a list containing the lines of the given file

    """
    file_lines = []
    try :
        with open(f) as file_objects:
            for line in file_objects:
                file_lines.append(line)
    except FileNotFoundError :
        error_flag = 1
        print('error : File not present in the directory')
        return
    return file_lines


def find_frequency(file_lines):
    """finds the value of frequency (if ac circuit is given)

    :file_lines: a list containing lines from netlist
    :returns: return frequency if given circuit is ac, if there is no ac source␣
 ↪(or frequency not mentioned in netlist)

    """
    for line in file_lines:
        if len(line.split()) != 0 and line.split()[0] == '.ac':
            freq = line.split()[2]
            return parse_value(freq)
    return 0
def unknowns(dic):
    """this function creates a map to keep track of the information of unknowns

    :dic: the nodal_equation_dict is given as input.
    :returns: a map containing the unknowns (that have to be found).

    """
    unknowns_map = {}
    index = 0
    for node1 in dic.keys():
        for unknowns in dic[node1].keys():
            if unknowns not in unknowns_map.keys() and unknowns != 'CONSTANT':
                unknowns_map[unknowns] = index
                index += 1
    return unknowns_map
def add_to_dict(dictionary,first_node,second_node,val):
    """function helps in maintaining and adding elements to the dict

    :dictionary: dictionary to which element must be added
    :first_node: first_node represents where at which node is the nodal␣
 ↪equation is being written at
    """
    if first_node not in dictionary.keys():
        dictionary[first_node] = {}
    if second_node not in dictionary[first_node].keys():
        dictionary[first_node][second_node] = complex(float(0),float(0))
    dictionary[first_node][second_node] += complex(val)
    return

#now we will form the matrix that is obtained from nodal equation

#in nodal_equation_dict[key1][key2] (nodal_eqation_dict is a map of maps), the␣
 ↪key1 represensts the node at which the nodal eq is written and
# nodal_equation_dict[key1][key2] represents the coefficient of the "key2" in␣
 ↪the equation formed at node of key1


#read from file
f = input('Please enter your file name :')
file_lines = _file_lines_of(f)

#this dict maintains the system of nodal equations
nodal_equation_dict = {}

#find frequency
omega = find_frequency(file_lines)*2*np.pi

#finding the index at which .circuit  and .end are present
start = '.circuit'
end = '.end'
start_index = -1
end_index = -1
#should we give error for one dc and one ac source?
for line in file_lines:
    if start == line[:len(start)]:
        start_index = file_lines.index(line)
    elif end == line[:len(end)] :
        end_index = file_lines.index(line)
try :
    if end_index <= start_index:
        raise MyCircuitSolver('Error : the given circuit is invalid')
except MyCircuitSolver as e:
    error_flag = 1
    print(f'{e._message}')


for line in file_lines[start_index+1 : end_index] :
    #now we have to ignore the comments in the netlist
    line = line.split('#')[0]

    if line.split()[0][0] == '#':
        continue   #skips a line starting with comment between .circuit and .end
    try :
        if line.split()[0][0] != 'V' and line.split()[0][0] != 'I' and line.
↪split()[0][0] != 'R' and line.split()[0][0] != 'L' and line.split()[0][0] !=␣
↪'C':
            raise MyCircuitSolver('ERROR : unidentified component')
    except MyCircuitSolver as e:
        error_flag = 1
        print(f'{e._message}')
    element = Circuit_Element(line)

    #nodal equation --> sum of total current leaving a node is zero
    if element.type() == 'passive':
        #passive element stamp

        #second argument in add_to_dict represents the node at which nodal␣
↪equation is written, nodal_eqation_dict[node1][node2]
        #represents the coefficient of V_node2(unknown) in the nodal equation␣
↪at node1
        #effect of passive element on nodal equation at first_node
        add_to_dict(nodal_equation_dict,'V(' + element.first_node() +')','V(' +␣
↪element.first_node() +')',element.admittance(omega))
        add_to_dict(nodal_equation_dict,'V(' + element.first_node() +')','V(' +␣
↪element.second_node() +')',-element.admittance(omega))

       #effect of passive element on nodal equation at second_node
        add_to_dict(nodal_equation_dict,'V(' + element.second_node() +')','V('␣
↪+ element.second_node() +')',element.admittance(omega))
        add_to_dict(nodal_equation_dict,'V(' + element.second_node() +')','V('␣
↪+ element.first_node() +')',-element.admittance(omega))

    else :
        try:
            if (element._ac_flag == 1 and omega == 0) or (element._ac_flag == 0␣
↪and omega != 0):
                raise MyCircuitSolver('ERROR : both ac and dc sources are␣
↪involved')
        except MyCircuitSolver as e:
            error_flag = 1
            print(f'{e._message}')
            break
        if element.name()[0] == 'V':
            #voltage source stamp

            #the effect of the assumed current flown through the present␣
↪voltage source
            #on nodal equation at first_node (I_v1 assumed to be flown from␣
↪negitive termial of battery to positive termianl)
```

```
            add_to_dict(nodal_equation_dict,'V('+ element.first_node() + ')' ,␣
 ↪'I_' + element.name(), -1)

            #the effect of the assumed current flown through the present␣
 ↪voltage source
            # on nodal equation at second_node
            add_to_dict(nodal_equation_dict,'V('+ element.second_node() + ')',␣
 ↪'I_' + element.name(), +1)

            #auxilary equation  v(node1) - v(node2) - value_of_v = 0
            add_to_dict(nodal_equation_dict, 'AUX_' + element.name(),'V('+␣
 ↪element.first_node() + ')', 1)
            add_to_dict(nodal_equation_dict, 'AUX_' + element.name(),'V('+␣
 ↪element.second_node() + ')', -1)
            add_to_dict(nodal_equation_dict, 'AUX_' + element.
 ↪name(),'CONSTANT', -element.value())

        elif element.name()[0] == 'I' :
            #current source stamp
            #effect of current source on first_node
            add_to_dict(nodal_equation_dict,'V(' + element.first_node() +␣
 ↪')','CONSTANT',-element.value())


            #effect of current source on first_node
            add_to_dict(nodal_equation_dict,'V(' + element.second_node() +␣
 ↪')','CONSTANT',element.value())

no_of_unknowns = len(nodal_equation_dict.keys()) - 1  # aussuming V_GND = 0␣
 ↪hence the number of unknowns is reduces by one
unknowns = unknowns(nodal_equation_dict)  #unkowns are loaded in map and their␣
 ↪key is the name of the unknown (ex : V1) and their
                                          # value is their index at which their␣
 ↪coefficients are loaded in matrix

#initilizing empty numpy array of appropriate size to form  A (unkowns) = B
A = np.zeros((len(nodal_equation_dict.keys()),len(nodal_equation_dict.
 ↪keys())),dtype = complex)
B = np.zeros((len(nodal_equation_dict.keys())),dtype = complex)

#now we have to convert the nodal_equation_dict into a matrix for solving it
index = -1
for node in nodal_equation_dict.keys():
    index += 1
    if node == 'V(GND)':
        index_gnd = index    #storing index of ground to delete from the matrix␣
 ↪as V_gnd = 0
    for unknown in nodal_equation_dict[node].keys():
        if unknown != 'CONSTANT':
            A[index][unknowns[unknown]] += nodal_equation_dict[node][unknown]
        elif unknown == 'CONSTANT':
            B[index] -= nodal_equation_dict[node][unknown]
#now we have to solve A and B matrices

#deleting the nodal equation fromed at GND node and keeping V(GND) as zero
A = np.delete(A , index_gnd , axis = 0)
A = np.delete(A , unknowns['V(GND)'], axis = 1)
B = np.delete(B , index_gnd)
try :
    ans = np.linalg.solve(A,B)   # has inconsistent solution if 2 voltage␣
 ↪solutions are connected in a loop
except MyLinalError as e:
    error_flag = 1
    print('{e._message}')
if error_flag == 0 :
    print("The solution of unknown voltages (V(node_name)) and currents␣
 ↪(I_(name of source)) are :")
    left_heading = "Unknowns"
    Right_heading = "Complex form"
    Middle_heading = "Magnitude"
    print ( f'{left_heading : <10} {Middle_heading  :^30}{Right_heading : >40}')
    for unknown in unknowns.keys():
        if(unknown != 'V(GND)') :
            if unknowns[unknown] < unknowns['V(GND)'] :
                print ( f'{unknown : <10} {abs(ans[unknowns[unknown] ]) :
 ↪^30}{ans[unknowns[unknown]] : >40}')
            else :
                print ( f'{unknown : <10} {abs(ans[unknowns[unknown] - 1]) :
 ↪^30}{ans[unknowns[unknown] - 1] : >40}')
```

```
Please enter your file name : ckt5.netlist
```

```
The solution of unknown voltages (V(node_name)) and currents (I_(name of
source)) are :
Unknowns             Magnitude                                       Complex
form
V(1)                    10.0
(-10-0j)
I_V1                    1.0
(1+0j)
```