Week2Pres

Sidharth S Kumar EE21B130

April 17, 2023

1 Assignment

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.

1.1 # Factorials

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

# Function to find the factorial of a number using recursion
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1) # function calling another instance of itself␣
 ↪in recurring manner

N = int(input("Enter the value of N: "))
start = time.time()
result = factorial(N)
end = time.time()

# Timing and printing the result of the function
%timeit factorial(N)
print("The factorial of", N, "is", result)
print("Time taken to compute the factorial:", end - start, "seconds")
```

```
Enter the value of N:  10
```

```
922 ns ± 23 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
The factorial of 10 is 3628800
Time taken to compute the factorial: 5.125999450683594e-05 seconds
```

This program is excecuted with a recurring function , that repeated call itself until the fucntion is completed.

1.2 Cython for factorials

1. No change (just direct cython)

[28]:
```
%%cython --annotate

# Function to find the factorial of a number using recursion
def factorial_direct_cython(n):
    if n == 0:
        return 1
    else:
        return n * factorial_direct_cython(n-1) # function calling another␣
 ↪instance of itself in recurring manner

N = 10
```

[28]:
```
<IPython.core.display.HTML object>
```

[29]:
```
%timeit factorial_direct_cython(N)
print(f"Factorial of the number is {factorial(N)}")
```

```
290 ns ± 6.52 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Factorial of the number is 3628800
```

Observations:
About 3x efficiency gain by simply adding cython

2. Optimise for C, Function defined in C

[30]:
```
%%cython --annotate

# we need to define the function in C
# recursively calling functions in Cython requires a lot of Python
# interaction
cdef int factorial_c(int x):
    if x == 0:
        return 1
    return x * factorial_c(x - 1)

# compute x factorial using recursion
def factorial_pyth(int x):
    if x == 0:
        return 1
    return x * factorial_c(x - 1)
```

[30]:
```
<IPython.core.display.HTML object>
```

[31]:
```
N = 10
print(factorial_pyth(N))
%timeit factorial_pyth(N)
```

```
3628800
46.4 ns ± 1.46 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Observations:
By optimising the function for C , we get a near 20X efficiency gain, grater than even the built-in functions

1.3 # Matrix Solver

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.

[32]:
```python
import numpy as np
from timeit import default_timer as timer

#defining a function to solve a set of consistent linear Equations
def Linear_solver(A,b):
    A = np.array(A,dtype=float)
    b = np.array(b,dtype=float)
    size = len(b) - 1

    # Exception for wrong format of matrix passed
    if A.shape[0] != b.shape[0]:
        raise Exception("The number of rows in A and b must be the same.")

    if np.linalg.det(A) == 0:
            raise Exception("The matrix A is singular, and cannot be inverted.")

    #Pivoting the matrix to avoid zero errors
    for j in range(size):
        Max = j
        for k in range(j,size):
            if np.abs(A[k][j]) > np.abs(A[Max][j]) :
                Max = k
        A[[j, Max]] = A[[Max, j]]

        for i in range(size-j):
            fact = A[j][j]/A[i+j+1][j]
            A[i+j+1] = fact*A[i+j+1] - A[j]
            b[i+j+1] = fact*b[i+j+1] - b[j]

    #Finding the diagonal matrix
    for j in range(size,0,-1):
        for i in range(size):
            fact = A[j-i-1][j]/A[j][j]
            A[j-i-1] = A[j-i-1] - fact*A[j]
            b[j-i-1] = b[j-i-1] - fact*b[j]

    for i in range(size):
        b[i] = b[i]/A[i][i]
    return b

# Generating randon 10 X 10 matrices to test the solver
A = np.random.random_sample(size = (10,10))*100
b = np.random.random_sample(size = (10))*10

#printing and timing the function againt np.linalg.solve()
print(Linear_solver(A,b))
%timeit Linear_solver(A,b)
print(np.linalg.solve(A,b))
%timeit np.linalg.solve(A,b)
```

```
[-1.02226286e+01  5.78247473e-01  5.30449257e+01 -2.10060837e+01
 -4.20609420e+01 -6.11973635e+01  1.61797604e+01  1.66544798e+01
  6.31885479e+00  1.41066162e+08]
504 µs ± 7.18 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
[-0.05572233  0.01483711  0.03881919  0.02875062  0.07697247  0.0386161
 -0.01149196 -0.00946318 -0.01001001  0.02358985]
16.1 µs ± 62.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Major part of the cell is a function defentition to solve a set of linear equations - It starts by converting the input list into array for computational ease - Then it checks if the dimensions of the arrays are consistent for a solution,and also if the determinant of the matrix is zero. - Then the matrix is pivoted to avoid the zero coefficient errors and to give more consistent answers - Then the matrix is converted into a diagonal matrix - In the last step the final solution to the linear system is found out by eqating the diagonal terms in the A matrix and the terms of the b matrix

1.4 Cython for Matrix solver

1. Direct Cython

[46]:
```
%%cython --annotate

import numpy as np
from timeit import default_timer as timer

#defining a function to solve a set of consistent linear Equations
def Linear_solver_cython(A,b):
    A = np.array(A,dtype=float)
    b = np.array(b,dtype=float)
    size = len(b) - 1

    # Exception for wrong format of matrix passed
    if A.shape[0] != b.shape[0]:
        raise Exception("The number of rows in A and b must be the same.")

    if np.linalg.det(A) == 0:
            raise Exception("The matrix A is singular, and cannot be inverted.")

    #Pivoting the matrix to avoid zero errors
    for j in range(size):
        Max = j
        for k in range(j,size):
            if np.abs(A[k][j]) > np.abs(A[Max][j]) :
                Max = k
        A[[j, Max]] = A[[Max, j]]

        for i in range(size-j):
            fact = A[j][j]/A[i+j+1][j]
            A[i+j+1] = fact*A[i+j+1] - A[j]
            b[i+j+1] = fact*b[i+j+1] - b[j]

    #Finding the diagonal matrix
    for j in range(size,0,-1):
        for i in range(size):
            fact = A[j-i-1][j]/A[j][j]
            A[j-i-1] = A[j-i-1] - fact*A[j]
            b[j-i-1] = b[j-i-1] - fact*b[j]

    for i in range(size):
        b[i] = b[i]/A[i][i]
    return b

# Generating randon 10 X 10 matrices to test the solver
A = np.random.random_sample(size = (10,10))*100
b = np.random.random_sample(size = (10))*10
```

[46]:
```
<IPython.core.display.HTML object>
```

[47]:
```
#printing and timing the function againt np.linalg.solve()
print(Linear_solver_cython(A,b))
%timeit Linear_solver_cython(A,b)
```

```
[-3.82278173e-03 -3.59851081e+01  2.32325106e+01 -3.08512079e+01
 -1.31678248e+01  2.35544138e+01  2.13721000e+01  2.69496822e+01
 -1.34256332e+01 -8.93869855e+08]
484 µs ± 8.09 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Observations: Not much difference as cython is not changing much to the code

2.Cython With C optimization

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

import numpy as np

# solves Ax = b and returns x
def solve_2(double complex[:, :] A, double complex[:] B):
    cdef Py_ssize_t n = A.shape[0] # number of unknowns

    cdef Py_ssize_t i, j

    # augmented matrix
    cdef double complex[:, :] b = np.concatenate(
            (A, np.expand_dims(B, axis=1)), axis=1
        )

    cdef double complex temp, ratio
    cdef Py_ssize_t k, Max



    for i in range(0, n):
        #Partial pivoting
        Max = i

        for k in range(i + 1, n):
            if abs(b[k, i]) > abs(b[Max, i]):
                Max = k

        #Check if the matrix is solvable
        if b[Max, i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        #Pivoting the matrix with the max elements
        for j in range(0, n + 1):
            temp = b[i, j]
            b[i, j] = b[Max, j]
            b[Max, j] = temp

        #Reduction by ratio
        for j in range(i + 1, n):
            ratio = b[j, i] / b[i, i]

            for k in range(i, n + 1):
                b[j, k] = b[j, k] - ratio * b[i, k]

    #Dividing to get the actual variables
    for i in range(n - 1, -1, -1):
        b[i, n] = b[i, n] / b[i, i]
        for j in range(0, i):
            b[j, n] = b[j, n] - b[i, n] * b[j, i]

    return b[0:n,n]

# Generating randon 10 X 10 matrices to test the solver
A = np.random.rand(10, 10).astype('complex128')
b = np.random.rand(10).astype('complex128')
```

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

[44]:
```
print(np.real(solve_2(A, b)))
%timeit solve_2(A, b)
```

```
[ 0.3802889  -0.37754793 -0.13453924 -0.80319293  0.20842589 -0.54590669
 -0.73790896  0.96452394  1.41555153  0.25876679]
10.5 µs ± 328 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Observations: After the optimization for C ,(data type specification) is about 40 times faster . This shows that the dynamic dtaa type allocation in python takes a lot of time to excecute

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.

The thrid problem is solved in another file uploade with this **netlist_solver.py**

1.5 Explanation for the third problem

- The program starts by defining class for all the components.
- It checks if the file has been passed to it throught the command line properly
- File is read along with registering the values of the start and end of the file as well as the ac check data. Errors int his part are also rooted out with a try except block
- All the data given in through the netlist file is now transfered to a list l for easy access, this is done simultaniously while addressing variety in the data recieved for each component
- Now the number of nodes is decided along with the information about them
- Penultimate step of the program is to make the MNA matrix to be solved, this is done by taking different scenarious for various components and depending on the kind of voltage they might encounter.

Note: The following cell must be running making sure that the FILE associated with the netlist is in the same directory as the ipynb file.

[6]:
```
from numpy import *

# CHANGE THE FILE NAME HERE
FILE = "ckt1.netlist"
```

```
# Class declared for components.
class Component:
        def __init__(self,name,nA,nB,value):
                self.name = name
                self.nA = nB
                self.nB = nA
                self.value = value


# Assigning constants variables to .circuit and .end
CIRCUIT = ".circuit"
END = ".end"
AC = ".ac"

# Global exception for invalid file type
try:

        # Opening the file mentioned in the commandline.
        with open(FILE) as f:
                lines = f.readlines()

        # These are parameters to check the errors in the file format.␣
 ↪
                start = -1; start_check = -1; end = -2; end_check = -1; ac = -1␣
 ↪; ac_check = -1

        # The program will traverse through the file and take out only the␣
 ↪required part.
                for line in lines:
                        if CIRCUIT == line[:len(CIRCUIT)]:
                                start = lines.index(line)
                                start_check = 0

                        elif END == line[:len(END)]:
                                end = lines.index(line)
                                end_check = 0
        #This part is to check if the circuit has an AC or a DC source.
                        elif AC == line[:len(AC)]:
                                ac = lines.index(line)
                                ac_check = 1

        # The program will throw in an error if the circuit definition format␣
 ↪is not proper.
                if start >= end or start_check == -1 or end_check == -1:
                        print("Invalid circuit definition.")
                        exit()
```

```
    # Creating a list and storing the necessary information into it.
↪
        l = [] ; k=0
    # In case of an AC circuit, the required information is collected.
↪
        try:
            if ac_check ==1:
                _,ac_name,freq = lines[ac].split("#")[0].split()
                freq = 2*pi*float(freq)

            for line in (lines[start+1:end]):
                name,nA,nB,*value = line.split("#")[0].split()

                if name[0] == 'R' or name[0] == 'C' or name[0]␣
↪== 'L' or name[0] == 'I':
                    element = Component(name,nA,nB,value)

                elif name[0] == 'V':
                    element = Component(name,nA,nB,value)
                    k = k+1

        # Converting the values of the components into real numbers
                if len(element.value) == 1:
                        element.value = float(element.
↪value[0])

        # In case of an AC source, the voltage and phase are assigned␣
↪properly.
                elif value[0] == "ac":
                    element.value = (float(element.
↪value[1])/2)*complex(cos(float(element.value[2])),sin(float(element.
↪value[2])))

                else:
                    element.value = float(element.value[1])


                l.append(element)

    # The program will throw an error if the netlist is not written␣
↪properly.
        except IndexError:
            print("Please make sure the netlist is written properly.
↪")
            exit()
    # Nodes are creating using a dictionary.
    node ={}
    for element in l:
        if element.nA not in node:
            if element.nA == 'GND':
                node['n0'] = 'GND'
            else:
                name = "n" + element.nA
                node[name] = int(element.nA[-1])

        if element.nB not in node:
            if element.nB == 'GND':
                node['n0'] = 'GND'
            else:
                name = "n" + element.nB
                node[name] = int(element.nB[-1])

    node['n0'] = 0
    n = len(node)

    # Creating the N and b matrices for solving the equations.
    N = zeros(((n+k-1),(n+k-1)),dtype="complex_")
    b = zeros(((n+k-1),1),dtype="complex_")
    p=0

    # This part of code will fill the matrices N and b taking into␣
↪consideration if it is an AC or a DC source.
    for element in l:

    # In case of a resistor, the matrix N is filled in a certain way as␣
↪shown below.
        if element.name[0] == 'R':
            if element.nB == 'GND':
                N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value

            elif element.nA == 'GND':
                N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value

            else:
                N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value
                N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value
```

```
                              N[int(element.nA[-1])-1][int(element.nB[-1])-1]␣
↪+= -1/element.value
                              N[int(element.nB[-1])-1][int(element.nA[-1])-1]␣
↪+= -1/element.value

      # In case of a capacitor, the impedance is calculated first and then␣
↪the matrix N is filled.
              elif element.name[0] == 'C':
                      if ac_check ==1:
                              Xc = -1/(float(element.value)*freq)
                              element.value = complex(0,Xc)

                      if element.nB == 'GND':
                              N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value
                      elif element.nA == 'GND':
                              N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value

                      else:
                              N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value
                              N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value
                              N[int(element.nA[-1])-1][int(element.nB[-1])-1]␣
↪+= -1/element.value
                              N[int(element.nB[-1])-1][int(element.nA[-1])-1]␣
↪+= -1/element.value

      # In case of an inductor, the impedance is calculated first and then␣
↪the matrix N is filled.
              elif element.name[0] == 'L':
                      if ac_check ==1:
                              Xl = (float(element.value)*freq)
                              element.value = complex(0,Xl)

                      if element.nB == 'GND':
                              N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value
                      elif element.nA == 'GND':
                              N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value

                      else:
                              N[int(element.nA[-1])-1][int(element.nA[-1])-1]␣
↪+= 1/element.value
```

```
                                N[int(element.nB[-1])-1][int(element.nB[-1])-1]␣
↪+= 1/element.value
                                N[int(element.nA[-1])-1][int(element.nB[-1])-1]␣
↪+= -1/element.value
                                N[int(element.nB[-1])-1][int(element.nA[-1])-1]␣
↪+= -1/element.value

        # In case of a current source, the matrix b is filled as shown.
                elif element.name[0] == 'I':
                        if element.nB == 'GND':
                                b[int(element.nA[-1])-1][0] += element.value

                        elif element.nA == 'GND':
                                b[int(element.nB[-1])-1][0] += -element.value

                        else:
                                b[int(element.nA[-1])-1][0] += element.value
                                b[int(element.nB[-1])-1][0] += -element.value

        # In case of a voltage source, the matrices N and b are filled as shown.
                elif element.name[0] == 'V':
                        if element.nB == 'GND':
                                N[int(element.nA[-1])-1][n-1+p] += 1
                                N[n-1+p][int(element.nA[-1])-1] += 1
                                b[n-1+p] += element.value
                                p = p+1
                        elif element.nA == 'GND':
                                N[int(element.nB[-1])-1][n-1+p] += -1
                                N[n-1+p][int(element.nB[-1])-1] += -1
                                b[n-1+p] += element.value
                                p = p+1
                        else:
                                N[int(element.nA[-1])-1][n-1+p] += 1
                                N[int(element.nB[-1])-1][n-1+p] += -1
                                N[n-1+p][int(element.nA[-1])-1] += 1
                                N[n-1+p][int(element.nB[-1])-1] += -1
                                b[n-1+p] += element.value
                                p = p+1

        # I tried using both linalg.solve() and my function Linear_solver(),
        # but due to some issue the Linear_solver funciton is not working from␣
↪this particular case
        V = linalg.solve(N,b)
        # V = Linear_solver(N,b)

        print(V,"\n")
        for i in range(n-1):
                print("V",i+1,"=",V[i],"\n")
        for j in range(k):
                print("I",j+1,"=",V[j+n-1],"\n")

# The program will throw in this error if the name of the netlist file is not␣
 ↪proper
# or if the netlist file is not found in the same directory as the program.

except FileNotFoundError:
        print("Invalid File.")
        exit()
```

```
[[ 0.e+00+0.j]
 [ 0.e+00+0.j]
 [ 0.e+00+0.j]
 [ 5.e+00+0.j]
 [-5.e-04+0.j]]

V 1 = [0.+0.j]

V 2 = [0.+0.j]

V 3 = [0.+0.j]

V 4 = [5.+0.j]

I 1 = [-0.0005+0.j]
```

[9]:
```
V = real(solve_2(N,b))

%timeit solve_2(N,b)
print(V,"\n")

for i in range(n-1):
        print("V",i+1,"=",V[i],"\n")
for j in range(k):
        print("I",j+1,"=",V[j+n-1],"\n")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 V = real(solve_2(N,b))
      3 get_ipython().run_line_magic('timeit', 'solve_2(N,b)')
      4 print(V,"\n")
```

```
File ~/.cache/ipython/cython/_cython_magic_f42d4653456ae92ad83fd3ad680451e6.pyx
↪5, in _cython_magic_f42d4653456ae92ad83fd3ad680451e6.solve_2()
      3
      4 # solves Ax = b and returns x
----> 5 def solve_2(double complex[:, :] A, double complex[:] B):
      6     cdef Py_ssize_t n = A.shape[0] # number of unknowns
      7

ValueError: Buffer has wrong number of dimensions (expected 1, got 2)
```

```
[ ]:
```