EE2703 -Week 8 : Cython
Aayush Patel, EE21B003

April 17, 2023

1 Cython Implementation for Factorial

[1]:
```
import Cython
%load_ext Cython
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%%cython --annotate
cpdef factorial_rec(int N):
    if(N<0):
        print("Invalid Input")
    if(N==1):
        return 1
    else:
        return N*factorial_rec(N-1)
```

[2]:
```
<IPython.core.display.HTML object>
```

[3]:
```
%%cython --annotate
cpdef factorial_iter(int N):
    cdef int fac
    cdef int num
    if(N<0):
        print("Invalid Input")
    fac=1
    for num in range(1,N+1):
        fac=fac*num
    return fac
```

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

[4]:
```
x=10
print("By Recursive Method : ")
%timeit factorial_rec(x)
print("By Iterative Method : ")
%timeit factorial_iter(x)
```

```
By Recursive Method :
165 ns ± 4.32 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
By Iterative Method :
42.2 ns ± 0.764 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

In Assignment 2, I got 1.12 micro-seconds for Recursive Method and 505 nano-seconds for Iterative Method. Here Cython is optimizing the code by converting the python code into C and thus reducting the time take to run the code. We can clearly see the improvement in the run-time.

2 Cython Implementation for Gauss Elimination (Matrix Solver)

[5]:
```
%%cython --annotate
import cython
import numpy
cimport numpy as np

# @cython.cdivision(True)
def No_Unique_Solution(list A,list B):
    cdef int N,M,counter
    cdef bint check
    N=len(A[0])     # Number of Variables
    M=len(A)        # Number of Equations
    counter=0
    for row in range(M-1,-1,-1):
        check=False
        for col in range(N):
            if(abs(A[row][col])>0.0000002):
                check=True
        if(check==False):
            counter+=1
            if(abs(B[row])>0.0000002):
                return "No Solution"
    if(counter<M-N):
        return "No Solution"
    elif(counter>M-N):
        return "Infinite Solution"
    else:
        return "Unique Solution"

# @cython.cdivision(True)
def Forward_Elimination(list A,list B):
    cdef int N,M
    cdef int row,dummy_row,col,next_rows
    cdef complex multiplier,divisor
    cdef bint swapped
    N=len(A[0])     # Number of Variables
    M=len(A)        # Number of Equations
    flag="Unique Solution"
    for row in range(0,N):
        #Check if Normalization Possible
        if(abs(A[row][row])<=0.0000002):
            #Find where its non-0
            swapped=False
            for dummy_row in range(row+1,M):
                if(abs(A[dummy_row][row])>0.0000002):
                    #Swap
                    A[dummy_row],A[row]=A[row],A[dummy_row]
                    B[dummy_row],B[row]=B[row],B[dummy_row]
                    swapped=True
                    break
            if(swapped==False):
                #No Unique Solution
                flag=No_Unique_Solution(A,B)
                return A,B,flag
        divisor=A[row][row]
        #Normalization
        for col in range(row,N):
            A[row][col]/=divisor
        B[row]/=divisor
        #Elimination
        for next_rows in range(row+1,M):
            multiplier=A[next_rows][row]
            for col in range(row,N):
                A[next_rows][col]-=multiplier*A[row][col]
            B[next_rows]-=multiplier*B[row]
    flag=No_Unique_Solution(A,B)
    return A,B,flag

@cython.cdivision(True)
def Backward_Substitution(list A,list B):
    cdef int N,M,row,cols
    cdef complex Sum
    cdef list x
    N=len(A[0])     # Number of Variables
    M=len(A)        # Number of Equations
    #Create the list x containing the values of the variables
    x=[0+0j for i in range(N)]
    for row in range(N-1,-1,-1):
        Sum=B[row]
        for cols in range(N-1,row,-1):
            Sum-=x[cols]*A[row][cols]
        x[row]=Sum
    return x
```

```
@cython.cdivision(True)
def Gauss_Elimination(A,B):
    if(not isinstance(A,list)):
        # A=A.astype(np.float32)
        # B=B.astype(np.float32)
        A=A.astype(numpy.clongdouble)
        B=B.astype(numpy.clongdouble)
    A=list(A)
    B=list(B)
    cdef list A1,B1,x
    A1,B1,flag=Forward_Elimination(A,B)
    if(flag=="Unique Solution"):
        x=Backward_Substitution(A1,B1)
        return x
    else:
        return flag
```

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

3 Explanation for the Cython Code

I have divided the explanation into several sub-parts:

- Here I have written all the code blocks in one cell only because, when we use Cython, it runs differently for different cell. Since my solution was divided into several sub-parts, I wrote the code in different functions. So all the functions need to be compiled in one code block only. The reason begin that since Python is an Intrepeted Language, it runs line by line. But when we convert it to C Language it needs to be compiled in the beginning. So we need to write all the required functions in one single cell of the Jupyter Notebook.
- Whilde defining the datatypes, I have used Complex Datatype because our aim is to have a Matrix Solver that solves MNA. For AC sources we would have complex numbers. Sadly in C there is no Complex Datatype. Due to this the efficiency of our coed decreases. (If you change the datatype form Complex to Float and run the Gauss_Elimination for float input, it would give the results faster.) Just using Complex datatype makes it run slower.
- As we can see there are a lot of yellow lines it's because of there is no inbuilt datatype like complex in c and the cython is converting that complex datatype using some inbuilt function or library into c language code.
- Converting any Python code to Cython doesn't really take much effort. We just need to change def to cpdef(or cdef) and initialize other datatypes. I did not use cdef because we are going to use the same function in a different cell too. Since cdef defines the function for one cell only, it would give issues if used in other cells. Also the reason I didn't use cpdef is that, then I couldn't use decorators. Here I used the decorator cdivision(True) .This decrease the checks that CYthon does for division by Zero. In Python there are ways to avoid the division by zero but in C if we have division by zero the program just crashes. So to avoid the extra time taken to check division by zero, I have used cdivision(True). But this means I couldn't use cpdef. Although using cpdef would have given better results.

I have changed the code very minimally for the Matrix Solver, so I need not give rigorous explanations for each function as this was already done in Week 2 Assignment. Let me just list the changes that I made:

- Using the original numpy would give very poor performance. Because numpy already works in C so to convert the Python Numpy code to C code is illogical. So I have used a different numpy which is a part of the Cython Package. This is done in the statement "cimport numpy as np".
- Before every function there is a decorator @cython.cdivision(True)
- Have initialized all the data types I will be using throughout that function.
- C doesn't have List data type(as in Python) and Python doesn't have array datatype(as in C), so I have just used List in the Cython code.
- C doesn't have complex datatype but to solve AC, we needed Complex. So instead of float datype I used Complex values.
- While initializing the list x in the function Backward_Substitution, I used none in my Assignment2, but here I have initialized it using 0+0j.
- For boolean datatypes I have used bint. Cython doesn't have a Boolean Datatype but bint works simillar to boolean. From some articles on internet I got to know that it stores integer but they are treated as boolean.
- There is no datatype for Strings in C. We have char datatype but for a collection for chars, we need to make an array in C. Here since I have used Python Strings, there is absolutely no way to convert it to its counterpart in C. So we need not initialize the string variable. Also most of the operations are taking place in Forward_Elimiation and Backward_Substitution, i.e. the operations are taking place mostly on the numbers and not on the strings. So I would not be of much effect even if we have a way to store Stings in C.

4 Running Cython & Comparision

[8]:
```
A=numpy.random.randint(low=-100000,high=100000,size=(10,10))
B=numpy.random.randint(low=-100000,high=100000,size=(10))

print("From linalg.solve() :")
print(numpy.linalg.solve(A,B))
# %timeit np.linalg.solve(A,B)

print("From my Method :")
print(Gauss_Elimination(A,B))
# %timeit Gauss_Elimination(A,B)
```

```
From linalg.solve() :
[  7.63823878   2.52955895  -2.97814995  12.60481027  -6.78103939
  11.71997259  12.48702502  -4.38190078 -10.04730157   5.22328781]
From my Method :
[(7.638238778468584+0j), (2.5295589509518166+0j), (-2.9781499489410788+0j),
(12.604810271979886+0j), (-6.781039389767209+0j), (11.719972592819522+0j),
(12.487025024221293+0j), (-4.381900781566766+0j), (-10.04730156854425+0j),
(5.223287809512507+0j)]
```

[9]:
```
%timeit numpy.linalg.solve(A,B)
```

```
19.5 µs ± 1.2 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[10]:
```
%timeit Gauss_Elimination(A,B)
```

```
128 µs ± 2.15 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

5 Original Gauss Elimination

This is the exact same code from Assignment 2. I have included it here just for Comparision as the matrix we are solving is randomly generated. So the runtime may be a bit different for a different 10x10 matrix.

[12]:
```
import numpy as np
def Forward_Elimination1(A,B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    flag="Unique Solution"
    for row in range(0,N):
        #Check if Normalization Possible
        if(abs(A[row][row])<=2e-19):
            #Find where its non-0
            swapped=False
            for dummy_row in range(row+1,M):
                if(abs(A[dummy_row][row])>2e-19):
                    #Swap
                    A[dummy_row],A[row]=A[row],A[dummy_row]
                    B[dummy_row],B[row]=B[row],B[dummy_row]
                    swapped=True
                    break
            if(swapped==False):
                #No Unique Solution
                flag=No_Unique_Solution1(A,B)
                return A,B,flag
        divisor=A[row][row]
        #Normalization
        for col in range(row,N):
            A[row][col]/=divisor
        B[row]/=divisor
        #Elimination
        for next_rows in range(row+1,M):
            multiplier=A[next_rows][row]
            for col in range(row,N):
                A[next_rows][col]-=multiplier*A[row][col]
            B[next_rows]-=multiplier*B[row]
    flag=No_Unique_Solution1(A,B)
    return A,B,flag
```

```python
def Backward_Substitution1(A, B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    #Create the list x containing the values of the variables
    x=['none' for i in range(N)]
    for row in range(N-1,-1,-1):
        Sum=B[row]
        for cols in range(N-1,row,-1):
            Sum-=x[cols]*A[row][cols]
        x[row]=Sum
    return x
def No_Unique_Solution1(A,B):
    N=len(A[0])      # Number of Variables
    M=len(A)         # Number of Equations
    counter=0
    for row in range(M-1,-1,-1):
        check=False
        for col in range(N):
            if(abs(A[row][col])>2e-19):
                check=True
        if(check==False):
            counter+=1
            if(abs(B[row])>2e-19):
                return "No Solution"
    if(counter<M-N):
        return "No Solution"
    elif(counter>M-N):
        return "Infinite Solution"
    else:
        return "Unique Solution"
def Gauss_Elimination1(A,B):
    if(not isinstance(A,list)):
        A=A.astype(np.float32)
        B=B.astype(np.float32)
    A1,B1,flag=Forward_Elimination1(A,B)
    if(flag=="Unique Solution"):
        x=Backward_Substitution1(A1,B1)
        return x
    else:
        return flag

print(Gauss_Elimination1(A,B))
%timeit Gauss_Elimination1(A,B)
```

```
[7.6382337, 2.5295503, -2.9781609, 12.604814, -6.7810364, 11.719982, 12.487032,
-4.381905, -10.04731, 5.2232933]
349 µs ± 5.89 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

This is the answer and time taken for my original Matrix Solver.

6 Conclusion : Better Performance

Here we observe that the 10x10 solver using the original implemenatation took 349 micro-seconds but using the cython implemenatation took 128 micro-seconds. The main reason for this increase in speed is due to the fact that I used a differernt Numpy which is a part of Cython Package. Otherwise I was getting very poor performance, using the normal Numpy wuth Cython. It is so because the normal Numpy is not made to work with Cython. Both Numpy and Cython use C in the backend, but Cython optimizes it more if we use the Cython's Numpy.