Submission

April 18, 2023

1 Problem Statement

The assignment for Week 8 is to use Cython to speed up the code you wrote for assignment 2 (SPICE simulation). Note that the grade is not based on how much you speed up, but on how well you are able to analyze the code and explain your optimizations. Your documentation here is particularly important. Even if you try techniques but don't see a speedup, you should try to explain why it didn't work.

[1]:
```
import numpy as np

#this is the previos code from week 2 assignment and all the documents are␣
 ↪there itself
class matrix_solver():
    def __init__(self, Ain, bin):
        self.A= Ain
        self.b= bin
        self.rowreduce=0
        self.sol=[]
        self.A, self.b= self.rearrange(self.A, self.b)


    def rearrange(self, matrix, b):
        len_row= len(matrix[0])
        len_column= len(matrix)
        if len_column>len_row:
            for row in range(len_row, len_column):
                for column in range(len_row):
                    matrix[0][column]+= matrix[row][column]
                b[0] += b[row]


        for diagonal in range(len_row):
            if matrix[diagonal][diagonal]== 0:
                for row in range(len_column):
                    if matrix[row][diagonal]!=0:
                        for column in range(len_row):
                            matrix[diagonal][column] += matrix[row][column]
                        b[diagonal] += b[row]
```

```
        return matrix, b

    def Row_reduce(self):

        lenrow= len(self.A[0])
        lencolumn= len(self.A)


        if lencolumn>=lenrow:
            for iter in range(lenrow):
                #normalizing
                for row_num in range(iter,lencolumn):

                    norm = self.A[row_num][iter]

                    if norm!=0:
                        for column_num in range(iter, lenrow):
                            self.A[row_num][column_num] /= norm
                        self.b[row_num] /= norm

                #row echelon
                for row_num in range(iter+1, lencolumn):

                    if self.A[row_num][iter]:
                        for column in range(iter, lenrow):
                            self.A[row_num][column] -= self.A[iter][column]
                        self.b[row_num] -= self.b[iter]
                self.rowreduce=1
        #rounding off
        for row in range(lencolumn):
            for column in range(lenrow):
                value_=self.A[row][column]
                self.A[row][column]= round(value_.real,20)
        for value in range(lencolumn):
            value_= self.b[value]
            self.b[value]= round(value_.real,10)
        return [self.A,self.b]


    def transpose(self, matrix):
        len_= len(matrix)
        newA=[]
        for row in reversed(matrix):
            new_row=[]
            for num in reversed(row):
                new_row.append(num)
```

```
            newA.append(new_row)
        return newA


    def reversed_(self, matrix):
        c=[]
        for i in reversed(range(len(matrix))):
            c.append(matrix[i])
        return c

    def check_solvability(self, matrix, Bin):
        len_row= len(matrix[0])
        len_column= len(matrix)
        prod=1
        if len_column> len_row:
            for row in range(len_row):
                prod *= matrix[row][row]
                if prod ==0:
                    print("No solution")
                    return 0
                for column in range(len_row):
                    if matrix[row][column] != 0:
                        break
                    if  matrix[row][column] != 0 and column== len_row:
                        if Bin[row]==0:
                            print("Infinite solution")
                            return 0
                        else:
                            print("No solution")
                            return 0
            if Bin[len_row]!= 0:
                print("It is unsolvable!")
                return 0
            return 1
        elif len_column== len_row:
            prod=1
            for diagonal in range(len_row):
                prod *= matrix[diagonal][diagonal]
            if prod==0 and Bin[len_row-1]!=0:
                print("No solution")
                return 0
            elif prod==0 and Bin[len_row-1]==0:
                print("Infinite solution")
                return 0
            else:
                return 1
        else:
```

```
            print("Infinite solution")
            return 0

    def return_required_matrix(self, matrix, Bin):
        len_row= len(matrix[0])
        mat= []
        res= []
        for row in range(len_row):
            mat.append(matrix[row])
            res.append(Bin[row])
        return mat, res

    def solve(self):
        self.A, self.b= self.rearrange(self.A, self.b)
        self.A, self.b= self.Row_reduce()

        if self.check_solvability(self.A, self.b):
            self.sol=[]
            self.A, self.b= self.return_required_matrix(self.A, self.b)
            self.A= self.transpose(self.A)
            self.b= self.reversed_(self.b)
            self.sol.append(self.b[0])
            len_= len(self.A[0])
            for row in range(1,len_):
                sum=0
                for column in range(0, row):
                    sum += (self.A[row][column])*(self.sol[column])
                self.sol.append(self.b[row]- sum)

            return self.reversed_(self.sol)
```

[2]:
```
A= [[7, 6, 6, 6, 7, 5, 5, 4, 2, 2],
 [8, 6, 5, 6, 1, 1, 5, 4, 9, 2],
 [5, 3, 9, 8, 6, 4, 8, 4, 1, 3],
 [3, 5, 8, 8, 6, 5, 6, 4, 8, 1],
 [6, 7, 2, 1, 1, 1, 9, 1, 9, 3],
 [1, 8, 5, 8, 5, 1, 1, 5, 3, 0],
 [9, 9, 0, 3, 9, 3, 4, 2, 7, 5],
 [4, 1, 6, 8, 4, 8, 9, 2, 2, 9],
 [7, 6, 0, 0, 1, 3, 1, 2, 1, 9],
 [7, 0, 7, 7, 5, 5, 4, 1, 1, 0]]
B= [5850, 6088, 6339, 6624, 6847, 7016, 7275, 7586, 7750, 7906]
```

[3]:
```
%timeit matrix_solver(A,B).solve()

print("THe above performace was from the week 2 Assingment!")
```

```
102 µs ± 1.97 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
THe above performace was from the week 2 Assingment!
```

So my first thought is to change all the types of the input into a defined one and all the other input types inside the class. but first lets make the important setups.

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

[5]:
```
import numpy as np
import cython
```

[6]:
```
%%cython --annotate

import cython
import numpy as np
cimport numpy as np

@cython.cdivision(True)
class cmatrix_solver1():
    def __init__(self,np.ndarray[np.double_t, ndim=2] Ain,np.ndarray[np.
 ↪double_t, ndim=1] bin):
        cdef np.ndarray[np.double_t, ndim=2] A
        cdef np.ndarray[np.double_t, ndim=2] B
        cdef int rowreduce
        cdef np.ndarray[np.double_t, ndim=1] sol

        self.A= Ain
        self.b= bin
        self.rowreduce=0
        self.sol=np.array([])
        self.A, self.b= self.rearrange(self.A, self.b)


    def rearrange(self, np.ndarray[np.double_t, ndim=2] matrix, np.ndarray[np.
 ↪double_t, ndim=1] b):
        cdef int len_row
        cdef int len_column

        len_row= len(matrix[0])
        len_column= len(matrix)
        if len_column>len_row:
            for row in range(len_row, len_column):
                for column in range(len_row):
                    matrix[0][column]+= matrix[row][column]
                b[0] += b[row]


        for diagonal in range(len_row):
```

```
            if matrix[diagonal][diagonal]== 0:
                for row in range(len_column):
                    if matrix[row][diagonal]!=0:
                        for column in range(len_row):
                            matrix[diagonal][column] += matrix[row][column]
                        b[diagonal] += b[row]

        return matrix, b

    def Row_reduce(self):
        cdef int lenrow
        cdef int lencolumn
        cdef float value_
        lenrow= len(self.A[0])
        lencolumn= len(self.A)


        if lencolumn>=lenrow:
            for iter in range(lenrow):
                #normalizing
                for row_num in range(iter,lencolumn):

                    norm = self.A[row_num][iter]

                    if norm!=0:
                        for column_num in range(iter, lenrow):
                            self.A[row_num][column_num] /= norm
                        self.b[row_num] /= norm

                #row echelon
                for row_num in range(iter+1, lencolumn):

                    if self.A[row_num][iter]:
                        for column in range(iter, lenrow):
                            self.A[row_num][column] -= self.A[iter][column]
                        self.b[row_num] -= self.b[iter]
                self.rowreduce=1
        #rounding off
        for row in range(lencolumn):
            for column in range(lenrow):
                value_=self.A[row][column]
                self.A[row][column]= round(value_,20)
        for value in range(lencolumn):
            value_= self.b[value]
            self.b[value]= round(value_,10)
        return [self.A,self.b]
```

```
def transpose(self, np.ndarray[np.double_t, ndim=2] matrix):
    cdef int len_
    cdef np.ndarray[np.double_t, ndim=2] newA
    cdef np.ndarray[np.double_t, ndim=1] new_row
    cdef int row
    cdef int roww
    roww = len(matrix[0])
    len_= len(matrix)
    newA= np.array([matrix[0]], dtype= np.double)
    for rownum in reversed(range(1,len_)):

        new_row=np.array([], dtype= np.double)
        for iter in reversed(range(roww)):
            new_row=np.append(new_row,[matrix[rownum][iter]])
        newA= np.append(newA,[new_row], axis=0)
    return newA


def reversed_(self, np.ndarray[np.double_t, ndim=1] matrix):
    cdef np.ndarray[np.double_t, ndim=1] c
    c=np.array([])
    for i in reversed(range(len(matrix))):
        c= np.append(c, [matrix[i]], axis=0) # c.append(matrix[i])
    return c

def check_solvability(self, matrix, Bin):

    cdef int len_row
    cdef int len_column
    cdef float prod
    cdef int row
    cdef int column

    len_row= len(matrix[0])
    len_column= len(matrix)
    prod=1
    if len_column> len_row:
        for row in range(len_row):
            prod *= matrix[row][row]
            if prod ==0:
                print("No solution")
                return 0
            for column in range(len_row):
                if matrix[row][column] != 0:
                    break
                if  matrix[row][column] != 0 and column== len_row:
```

```
                    if Bin[row]==0:
                        print("Infinite solution")
                        return 0
                    else:
                        print("No solution")
                        return 0
        if Bin[len_row]!= 0:
            print("It is unsolvable!")
            return 0
        return 1
    elif len_column== len_row:
        prod=1
        for diagonal in range(len_row):
            prod *= matrix[diagonal][diagonal]
        if prod==0 and Bin[len_row-1]!=0:
            print("No solution")
            return 0
        elif prod==0 and Bin[len_row-1]==0:
            print("Infinite solution")
            return 0
        else:
            return 1
    else:
        print("Infinite solution")
        return 0

def return_required_matrix( self,
                            np.ndarray[np.double_t, ndim=2] matrix,
                            np.ndarray[np.double_t, ndim=1] Bin
                            ):
    cdef int len_row
    len_row= len(matrix[0])
    cdef np.ndarray[np.double_t, ndim=2] mat
    cdef np.ndarray[np.double_t, ndim=1] res
    mat= np.array([matrix[0]])
    res= np.array([Bin[0]])

    for row in range(1,len_row):

        mat=np.append(mat,[matrix[row]], axis=0)
        res=np.append(res, [Bin[row]], axis=0)
    return mat, res

def solve(self):
    cdef int len_
    cdef int sum
    self.A, self.b= self.rearrange(self.A, self.b)
```

```
        self.A, self.b= self.Row_reduce()
        self.A= np.array(self.A)
        self.b= np.array(self.b)

        if self.check_solvability(self.A, self.b):
            self.sol=np.array([])
            self.A, self.b= self.return_required_matrix(self.A, self.b)
            self.A= self.transpose(self.A)
            self.b= self.reversed_(self.b)
            self.sol= np.append(self.sol,[self.b[0]])
            len_= len(self.A[0])

            for row in range(1,len_):

                sum=0
                for column in range(0, row):
                    sum += (self.A[row][column])*(self.sol[column])
                self.sol= np.append(self.sol,[self.b[row]-sum])

            return self.reversed_(self.sol)
```

```
In file included from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarraytypes.h:1940,
                 from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarrayobject.h:12,
                 from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/arrayobject.h:5,
                 from /home/btech/ee21b128/.cache/ipython/cython/_cython_magic_d
bd99b84e6ccaf155c902038e8c3adf8.c:769:
/usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:17:2: warning:
#warning "Using deprecated NumPy API, disable it with " "#define
NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-Wcpp]
   17 | #warning "Using deprecated NumPy API, disable it with " \
      |  ^~~~~~~
```

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

[7]:
```
#now since the input is only in array type I had to convert them
A=np.array(A, dtype= np.double)
B=np.array(B, dtype= np.double)
#now copying the opject so that it doesn't effect my orginal variable A and B
c= A.copy()
d= B.copy()
```

Finally running the timeit test, we can see that the time taken for compilation has reduced.

[8]:
```
%timeit obj= cmatrix_solver1(c,d).solve()
print("Here we can see that the performance has degraded from the previos case")
```

```
816 µs ± 63 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
Here we can see that the performance has degraded from the previos case
```

This is because we are using numpy array a lot in the code, and to define an array initially it takes more time that defining a list. Here is the demonstration.

[9]:
```
#the following code was written to understand which was faster defining list or␣
 ↪array
def a():
    m=[]
def b():
    m=np.array([])
```

[10]:
```
#now running the test
%timeit a()
%timeit b()
print("Thus defining an list is much faster than defining numpy array")
```

```
58.4 ns ± 1 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
307 ns ± 7.74 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Thus defining an list is much faster than defining numpy array
```

THe below codes are dummy code used to understand the cytho,. and doesnt relate to the problem

[11]:
```
%%cython --annotate

import cython
import numpy as np
cimport numpy as np

@cython.cdivision(True)
def create_array(np.ndarray[np.double_t, ndim=2] A):
    cdef np.ndarray[np.double_t, ndim=2] my_array
    my_array= A
    for i in range(10):
        print(my_array[i])
    my_array= np.append(my_array,[np.array([7., 0., 7., 7., 5., 5., 4., 1., 1.,␣
 ↪0.])], axis=0)
    print(" ")
    for i in range(11):
        print(my_array[i])
```

```
In file included from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarraytypes.h:1940,
                 from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarrayobject.h:12,
                 from /usr/local/lib/python3.9/dist-
```

```
packages/numpy/core/include/numpy/arrayobject.h:5,
                 from /home/btech/ee21b128/.cache/ipython/cython/_cython_magic_d
16780e74a6593cf5fe3ffee2b0b30ee.c:770:
/usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:17:2: warning:
#warning "Using deprecated NumPy API, disable it with " "#define
NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-Wcpp]
   17 | #warning "Using deprecated NumPy API, disable it with " \
      |  ^~~~~~~
```

[11]:
```
<IPython.core.display.HTML object>
```

[12]:
```
#thus using the function over here
create_array(A)
```

```
[1.         0.85714286 0.85714286 0.85714286 1.         0.71428571
 0.71428571 0.57142857 0.28571429 0.28571429]
[ 0.          1.          2.16666667  1.          8.16666667  5.5
  0.83333333  0.66666667 -7.83333333  0.33333333]
[ 0.          0.          1.          0.66666667  1.53333333  1.
  0.73333333  0.26666667 -1.4         0.26666667]
[ 0.          0.          0.          1.         -5.91538462 -3.69230769
  0.59230769  0.21538462  9.13846154 -0.24615385]
[ 0.          0.          0.          0.          1.          0.66104079
 -0.55743085  0.09095171 -1.39990624 -0.13877168]
[ 0.          0.          0.          0.          0.          1.
 -0.50830214 -0.0732316   3.15538919 -0.67338516]
[ 0.          0.          0.          0.          0.          0.
  1.         -0.3422571   2.38361467  0.49844103]
[  0.           0.           0.           0.           0.
   0.           0.           1.         -19.68533378   2.44586176]
[   0.           0.           0.           0.           0.
    0.           0.           0.           1.         -246.06949807]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

[1.         0.85714286 0.85714286 0.85714286 1.         0.71428571
 0.71428571 0.57142857 0.28571429 0.28571429]
[ 0.          1.          2.16666667  1.          8.16666667  5.5
  0.83333333  0.66666667 -7.83333333  0.33333333]
[ 0.          0.          1.          0.66666667  1.53333333  1.
  0.73333333  0.26666667 -1.4         0.26666667]
[ 0.          0.          0.          1.         -5.91538462 -3.69230769
  0.59230769  0.21538462  9.13846154 -0.24615385]
[ 0.          0.          0.          0.          1.          0.66104079
 -0.55743085  0.09095171 -1.39990624 -0.13877168]
[ 0.          0.          0.          0.          0.          1.
 -0.50830214 -0.0732316   3.15538919 -0.67338516]
[ 0.          0.          0.          0.          0.          0.
```

```
  1.         -0.3422571   2.38361467  0.49844103]
[  0.          0.          0.          0.          0.
   0.          0.          1.        -19.68533378   2.44586176]
[   0.          0.          0.          0.          0.
    0.          0.          0.          1.        -246.06949807]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
[7. 0. 7. 7. 5. 5. 4. 1. 1. 0.]
```

[13]:
```
%%cython --annotate

import cython
import numpy as np
cimport numpy as np
def sasa():
    cdef np.ndarray[np.double_t, ndim=1] a
    cdef np.ndarray[np.double_t, ndim=1] b
    a= np.array([1,2,3], dtype= float)
    b= np.array([4,5,6], dtype= float)
    print(np.append(a,[b]))
```

```
In file included from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarraytypes.h:1940,
                 from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/ndarrayobject.h:12,
                 from /usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/arrayobject.h:5,
                 from /home/btech/ee21b128/.cache/ipython/cython/_cython_magic_c
57258f58be52df836cf803cf3f030c3.c:771:
/usr/local/lib/python3.9/dist-
packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:17:2: warning:
#warning "Using deprecated NumPy API, disable it with " "#define
NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-Wcpp]
   17 | #warning "Using deprecated NumPy API, disable it with " \
      |  ^~~~~~~
```

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