copyweek2new

Varsha S P, EE21B154 <ee21b154@smail.iitm.ac.in>

April 16, 2023

1 Assignment

The gaussElim function is a Python implementation of the Gaussian elimination algorithm for solving a system of linear equations with a given coefficient matrix a and a constant vector b. The function performs the Gaussian elimination using nested loops over rows and columns of the matrix a, and it also calculates the determinant of a to check if the system has a unique solution. The function returns the solution vector b if the determinant is non-zero, or a message string if the determinant is zero.

On the other hand, c_gaussElim is a Cython implementation of the same algorithm, but with additional optimizations for performance. The function uses the cdef keyword to declare variables with C data types, and it also disables bounds checking and wraparound checking for array indexing to speed up the computation. The function uses nested loops over rows and columns of a to perform the Gaussian elimination, and it calculates the determinant of a as the product of its diagonal elements during the elimination process, instead of computing it separately at the beginning. The function returns the solution vector b if the determinant is non-zero, or a message string if the determinant is zero.

[6]:
```
%load_ext Cython
```

1.1 The gaussElim function

[7]:
```
import numpy as np
def dot(v1, v2):
    return sum(x*y for x, y in zip(v1, v2))
def gaussElim(a,b):
    n = len(b)

    if np.linalg.det(abs(a))!=0:
        for k in range(0,n-1):
            for i in range(k+1,n):
                if a[i,k] != 0.0:
                    lamb = a [i,k]/a[k,k]

                    a[i,k+1:n] = a[i,k+1:n] - lamb*a[k,k+1:n]
                    b[i] = b[i] - lamb*b[k]
        for k in range(n-1,-1,-1):
            b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
```

```
        return b
    else:
        return "No solution or infinite solution for the given matrix"
```

1.2 The cythonized gaussElim function

The `cpdef` keyword tells Cython to generate both C and Python versions of the function. The `cnp.ndarray[double, ndim=2]` and `cnp.ndarray[double, ndim=1]` specify the type and number of dimensions for the NumPy arrays a and b. The `cdef int n = b.shape[0]` declares and initializes the variable n to the length of the array b, which represents the number of rows in the matrix a. The `cdef double lamb, a_ik,a_kk, dot_prod` is also used to initialize the variable, because the slices lists idexing needs to be cythonized. The `cdef double det = 1.0` declares and initializes the determinant of the matrix a to 1.0. The determinant will be calculated in the loop below. Same you do to every variable used. In Cython, every variable must be declared beforehand because it is a statically-typed language, meaning that the data type of each variable must be explicitly defined at compile-time. This allows the Cython compiler to generate more efficient C code, which can be executed faster than pure Python code. The arrays `a[i, k+1:n] and a[k, k+1:n]` are sliced using Cython syntax to avoid the overhead of Python slicing. The determinant is calculated outside of the loop for efficiency. If the determinant is zero, the function returns an error message instead of the solution vector.

[8]:
```
%%cython --annotate
cimport cython
import numpy as np
cimport numpy as cnp

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True)

cpdef c_gaussElim(cnp.ndarray[double, ndim=2] a, cnp.ndarray[double, ndim=1] b):
    cdef int n = b.shape[0]
    cdef double lamb, a_ik,a_kk, dot_prod
    cdef int k
    cdef int i
    cdef double det = 1.0

    for k in range(n):
        det *= a[k, k]
        for i in range(k+1, n):
            a_ik = a[i, k]
            a_kk = a[k, k]
            lamb = a_ik / a_kk
            a[i, k+1:n] -= lamb*a[k, k+1:n]
            a[i, k] = 0.0
            b[i] -= lamb*b[k]
    if det != 0.0:
```

```
        for k in range(n-1, -1, -1):
            dot_prod = 0.0
            for i in range(k+1, n):
                dot_prod += a[k, i] * b[i]
            b[k] = (b[k] - dot_prod) / a[k, k]
        return b
    else:
        return "No solution or infinite solution for the given matrix"
```

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

1.2.1 Timing the defined function in python vs defined funtion in cython vs in.built fuction

The defined function in python

[9]:
```
A= np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
[6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]] ,float)
B= np.array( [[2],
[3],
[4],
[1],
[2],
[2],
[2],
[9],
[7],
[5]],float)

print(gaussElim(A,B))

%timeit gaussElim(A,B)
```

```
[[ 0.57592865]
 [-1.1434718 ]
 [ 1.700912  ]
 [ 1.56273285]
 [ 1.1733649 ]
 [ 1.36712171]
 [-1.35775437]
 [ 1.04556496]
```

```
 [-1.97475077]
 [-2.06722465]]
342 µs ± 5.61 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

The defined funtion in cython

[10]:
```
A= np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
             [7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
             [6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
             [8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
             [7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
             [9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
             [3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
             [8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
             [1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
             [6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]],dtype=np.double )
B= np.array( [[2],
              [3],
              [4],
              [1],
              [2],
              [2],
              [2],
              [9],
              [7],
              [5]],dtype=np.double)

# Flatten the B matrix
B = np.ravel(B)

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

```
[ 0.57592865 -1.1434718   1.700912    1.56273285  1.1733649   1.36712171
 -1.35775437  1.04556496 -1.97475077 -2.06722465]
83.3 µs ± 5.48 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The in.built fuction

[11]:
```
A= np.array([[1 ,3 ,1 ,2 ,6 ,6 ,0 ,1 ,3 ,5 ],
[7 ,0 ,2 ,0 ,5 ,5 ,6 ,3 ,3 ,3 ],
[6 ,0 ,6 ,0 ,0 ,8 ,4 ,5 ,3 ,7 ],
[8 ,5 ,4 ,9 ,3 ,5 ,3 ,5 ,8 ,7 ],
[7 ,6 ,3 ,8 ,9 ,2 ,3 ,8 ,7 ,8 ],
[9 ,5 ,7 ,0 ,7 ,7 ,0 ,1 ,8 ,6 ],
[3 ,9 ,7 ,9 ,2 ,1 ,7 ,6 ,7 ,1 ],
[8 ,5 ,6 ,4 ,4 ,0 ,3 ,7 ,2 ,5 ],
[1 ,2 ,7 ,6 ,1 ,5 ,2 ,0 ,8 ,1 ],
[6 ,4 ,4 ,3 ,6 ,2 ,7 ,8 ,5 ,2 ]] )
```

```
B= np.array( [[2],
[3],
[4],
[1],
[2],
[2],
[2],
[9],
[7],
[5]])
print(np.linalg.solve(A,B))
%timeit np.linalg.solve(A, B)
```

```
[[ 0.57592865]
 [-1.1434718 ]
 [ 1.700912  ]
 [ 1.56273285]
 [ 1.1733649 ]
 [ 1.36712171]
 [-1.35775437]
 [ 1.04556496]
 [-1.97475077]
 [-2.06722465]]
19.1 µs ± 317 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

1.2.2 Explanation

As you can see I have used a 10 * 10 matrix and solved for all 3 function and cython function gives a significant decrease in runtime compared to python, but not as good as pre-defined function. Now lets run this for the circuits that we have.

1.3 SPICE simulator

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.

[12]:
```
# %%cython --annotate
import numpy as np
import sys
import cmath
def ckt(file):
    try:
        with open(file,'r') as f:
            data=f.read().split("\n")
    except Exception as ex:
        print (ex)
```

```
        return
    fre=0

    for l in data:
        if l=='':
            continue
        i=l.split()

        if i[0]=='.circuit':
            start=data.index(l)
        if i[0]=='.end':
            end=data.index(l)
        if i[0]=='.ac':
            fre=float(i[2])

        #ac
    for l in data:
        if l=='':#ac dc
            continue
        i=l.split()
        if i[0][0]=='V' and i[3]=='dc' and fre!=0:
            print("ac and dc found")
            sys.exit()

    return data[start+1:end],fre
```

1.4 Explanation

We have defined a fuction called ckt(), to read a .netlist and return values. 'freq' returns the value of frequency of the circuit is at all there is an AC source. Start, end is used to determine the number of nodes in a given circuit. We also check if both AC and DC source if detected the same circuit and print an error message.

[25]:
```
def matrix():
    lines, fre=ckt('ckt3.netlist')
    nodes=[]
    nodes_var={}
    count=0
    curr=0

    for l in lines:
        if l=='':
            continue
        i=l.split()
        for k in range(1,3):
            if i[k] not in nodes:
                nodes.append(i[k])
```

```
            nodes_var[i[k]]=count
            count+=1

curr=0
curr_var={}
for l in lines:
    if l=='':
        continue
    i=l.split()
    if i[0][0]=='V':
        curr_var[i[0]]=(i[1],i[2],curr+count)
        curr+=1


mb=[0]*(curr+count)
ma=[]
for i in range(curr+count):
    ma.append([])
    for j in range(curr+count):
        ma[i].append(0)

for k in nodes_var.keys():
    for l in lines:
        if l=='':
            continue
        i=l.split()
        for m in range(1,3):
            if i[m]==k:
                if i[0][0]!='V' and i[0][0]!='I':
                    if i[0][0]=='R':
                        add=float(i[3])
                        if m==1:
                            ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                            ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                        elif m==2:
                            ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                            ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                    if i[0][0]=='C':
                        add=(complex(0,-1))*(1/(fre*2*np.pi*float(i[3])))
                        if m==1:
                            ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                            ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                        elif m==2:
                            ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                            ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                    if i[0][0]=='L':
                        add = (float(i[3])*fre*2*np.pi*(complex(0,1)))
```

```
                            if m==1:
                                ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                            elif m==2:
                                ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                    elif i[0][0]=='I':
                        if m==1:
                            mb[nodes_var[k]]= -float(i[4])
                        elif m==2:
                            mb[nodes_var[k]]= float(i[4])
                    elif i[0][0]=='V':
                        if m==1:
                            ma[nodes_var[k]][curr_var[i[0]][2]]=1
                        if m==2:
                            ma[nodes_var[k]][curr_var[i[0]][2]]= -1
                if i[0][0]=='V' :
                    ma[curr_var[i[0]][2]][nodes_var[i[1]]]=1
                    ma[curr_var[i[0]][2]][nodes_var[i[2]]]=-1
                    mb[curr_var[i[0]][2]]=float(i[4])

    return nodes,nodes_var,curr,curr_var,ma,mb,add,count


nodes,nodes_var,curr,curr_var,ma,mb,add,count = matrix()

reference = nodes_var['GND']
del mb[reference]
for i in range(len(ma)):
    del ma[i][reference]
del ma[reference]
#print(ma, mb)

a=np.array(ma, dtype = complex)
b=np.array(mb, dtype = complex)


x = gaussElim(a ,b)

for i in range(count-1):
    print("V",i+1,"=",b[i],"V")
for i in range(curr):
    print("I",i+1,"=",b[i+count-1],"A")
%timeit gaussElim(a ,b)
```

```
V 1 = (-10+0j) V
V 2 = (-5.029239766081871+0j) V
V 3 = (-2.5730994152046778+0j) V
```

```
V 4 = (-1.4035087719298247+0j) V
V 5 = (-0.9356725146198832+0j) V
I 1 = (-0.004970760233918129-0j) A
```

```
/tmp/ipykernel_1262465/1659613208.py:14: RuntimeWarning: overflow encountered in
scalar multiply
  b[i] = b[i] - lamb*b[k]
/tmp/ipykernel_1262465/1659613208.py:14: RuntimeWarning: invalid value
encountered in scalar subtract
  b[i] = b[i] - lamb*b[k]
```

```
51.9 µs ± 2.6 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[24]:
```python
def matrix():
    lines, fre=ckt('ckt3.netlist')
    nodes=[]
    nodes_var={}
    count=0
    curr=0

    for l in lines:
        if l=='':
            continue
        i=l.split()
        for k in range(1,3):
            if i[k] not in nodes:
                nodes.append(i[k])
                nodes_var[i[k]]=count
                count+=1

    curr=0
    curr_var={}
    for l in lines:
        if l=='':
            continue
        i=l.split()
        if i[0][0]=='V':
            curr_var[i[0]]=(i[1],i[2],curr+count)
            curr+=1


    mb=[0]*(curr+count)
    ma=[]
    for i in range(curr+count):
        ma.append([])
        for j in range(curr+count):
            ma[i].append(0)
```

```
    for k in nodes_var.keys():
        for l in lines:
            if l=='':
                continue
            i=l.split()
            for m in range(1,3):
                if i[m]==k:
                    if i[0][0]!='V' and i[0][0]!='I':
                        if i[0][0]=='R':
                            add=float(i[3])
                            if m==1:
                                ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                            elif m==2:
                                ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                        if i[0][0]=='C':
                            add=(complex(0,-1))*(1/(fre*2*np.pi*float(i[3])))
                            if m==1:
                                ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                            elif m==2:
                                ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                        if i[0][0]=='L':
                            add = (float(i[3])*fre*2*np.pi*(complex(0,1)))
                            if m==1:
                                ma[nodes_var[k]][nodes_var[i[1]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[2]]]-=1/add
                            elif m==2:
                                ma[nodes_var[k]][nodes_var[i[2]]]+=1/add
                                ma[nodes_var[k]][nodes_var[i[1]]]-=1/add
                    elif i[0][0]=='I':
                        if m==1:
                            mb[nodes_var[k]]= -float(i[4])
                        elif m==2:
                            mb[nodes_var[k]]= float(i[4])
                    elif i[0][0]=='V':
                        if m==1:
                            ma[nodes_var[k]][curr_var[i[0]][2]]=1
                        if m==2:
                            ma[nodes_var[k]][curr_var[i[0]][2]]= -1
                if i[0][0]=='V' :
                    ma[curr_var[i[0]][2]][nodes_var[i[1]]]=1
                    ma[curr_var[i[0]][2]][nodes_var[i[2]]]=-1
                    mb[curr_var[i[0]][2]]=float(i[4])
```

```
    return nodes,nodes_var,curr,curr_var,ma,mb,add,count


nodes,nodes_var,curr,curr_var,ma,mb,add,count = matrix()

reference = nodes_var['GND']
del mb[reference]
for i in range(len(ma)):
    del ma[i][reference]
del ma[reference]
#print(ma, mb)

a=np.array(ma, dtype = np.double)
b=np.array(mb, dtype = np.double)


x = c_gaussElim(a ,b)

for i in range(count-1):
    print("V",i+1,"=",b[i],"V")
for i in range(curr):
    print("I",i+1,"=",b[i+count-1],"A")
%timeit c_gaussElim(a ,b)
```

```
V 1 = -10.0 V
V 2 = -5.029239766081871 V
V 3 = -2.5730994152046778 V
V 4 = -1.4035087719298247 V
V 5 = -0.9356725146198832 V
I 1 = -0.004970760233918129 A
31 µs ± 945 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

As you can see that on cythonizing the Gaussian solver, we are able to optimise and reduce the runtime. Hence the purpose served. We will be able to get much better runtime when we apply cython for matrix() function.