전산천문학 HW-5

제출일 : 2017.5.22.월

2013-12239 서유경

1번. 쌍성의 운동

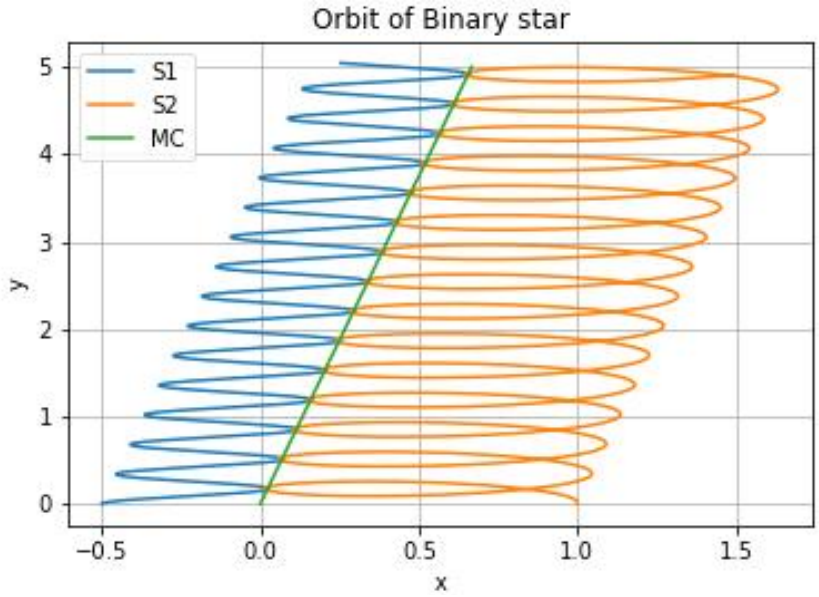
(a) 쌍성 궤도 그리기, (b) 질량중심의 이동경로 그리기

```
#Prob 5-1
from numpy import *
import numpy as np
import matplotlib.pyplot as plt

h=0.0001    <-시간 간격 값을 0.0001로 주었습니다
t=np.arange(0.,50.,h)

r1=np.zeros((len(t),2))   <-Star 1의 좌표array
r2=np.zeros((len(t),2))   <-Star 2의 좌표 array

r1[0,0]=-0.5    <- 초기상태에서 위치를 입력
r1[0,1]=0.
r2[0,0]=1.0
r2[0,1]=0.

v1=np.zeros((len(t),2))   <-Star 1과 2의 속도벡터 array. 마찬가지로 초기 속도를 입력
v2=np.zeros((len(t),2))
v1[0,0]=0.01
v1[0,1]=0.05
v2[0,0]=0.02
v2[0,1]=0.2

#half     <-leaf frog 방법을 이용할 때는, 속도의 경우 n단계와 n+1단계 사이의 - n+0.5단계 - 값이 필요하다. 때문에 Vn+0.5 array도 따로 지정해준다.
hv1=np.zeros((len(t),2))
hv2=np.zeros((len(t),2))

def a1(a,b):            <-Star 1의 가속도
    r=((a[0]-b[0])**2+(a[1]-b[1])**2)**0.5
    a1x=-0.5*(a[0]-b[0])/r**3
    a1y=-0.5*(a[1]-b[1])/r**3
    return np.array([a1x,a1y])
```

```
def a2(a,b): <-Star 2의 가속도
    r=((a[0]-b[0])**2+(a[1]-b[1])**2)**0.5
    a2x=(a[0]-b[0])/r**3
    a2y=(a[1]-b[1])/r**3
    return np.array([a2x,a2y])

hv1[0]=v1[0]+h/2*a1(r1[0],r2[0])
hv2[0]=v2[0]+h/2*a2(r1[0],r2[0])
mg=np.zeros((len(t),2))
mg[0]=r1[0]*1/1.5+r2[0]*0.5/1.5      <-가속도 함수로 0.5단계 상태일때 속도 값을 구한다.
```

<-t=0~50까지의 n단계의 위치백터 rn은 Vn+0.5속도를 이용해 구하고, 따로 Vn단계의 속도 값도 구한다. 그리고 그 위치백터를 구하면 다시 가속도 백터로 다시 속도백터를 구하고 다시 위치백터를 구하는 과정을 반복. 그리고 질량중심(mg)도 구한다

```
for n in range(1,len(t)):
    r1[n]=r1[n-1]+h*hv1[n-1]
    r2[n]=r2[n-1]+h*hv2[n-1]
    v1[n]=hv1[n-1]+h/2*a1(r1[n],r2[n])
    v2[n]=hv2[n-1]+h/2*a2(r1[n],r2[n])
    hv1[n]=hv1[n-1]+h*a1(r1[n],r2[n])
    hv2[n]=hv2[n-1]+h*a2(r1[n],r2[n])
    mg[n]=r1[n]*1/1.5+r2[n]*0.5/1.5

plt.plot(r1[:,0],r1[:,1],label='S1'),plt.plot(r2[:,0],r2[:,1],label='S2'),plt.plot(mg[:,0],mg[:,1],label='MC'),plt.legend(),plt.xlabel('x'),plt.ylabel('y'),plt.title('Orbit of Binary star'),plt.grid(),plt.savefig('Prob5-1-(a).png')
```



(MC가 질량중심)

(c) t=0~1000까지 각운동량과 에너지가 어떻게 변하는지 파악하기.

t2=np.arange(0,1000,h) <-0에서 1000까지 있는 array를 다시 지정해줍니다
L=np.zeros(len(t2)) <-L과 E의 scalar값을 입력할 L과 E array도 만들고
E=np.zeros(len(t2))

r1=np.zeros((len(t2),2))
r2=np.zeros((len(t2),2))
r1[0]=[-0.5,0.]
r2[0]=[1.0,0.]
v1=np.zeros((len(t2),2))
v2=np.zeros((len(t2),2))
v1[0]=[0.01,0.05]
v2[0]=[0.02,0.2]
hv1=np.zeros((len(t2),2))
hv2=np.zeros((len(t2),2))
hv1[0]=v1[0]+h/2*a1(r1[0],r2[0])
hv2[0]=v2[0]+h/2*a2(r1[0],r2[0])
mg=np.zeros((len(t2),2))
mg[0]=r1[0]*1/1.5+r2[0]*0.5/1.5

L[0]=(r1[0][0]*v1[0][1]-r1[0][1]*v1[0][0])+0.5*(r2[0][0]*v2[0][1]-r2[0][1]*v2[0][0])
E[0]=0.5*(v1[0][0]**2+v1[0][1]**2)+0.25*(v2[0][0]**2+v2[0][1]**2)-0.5/((r1[0][0]-r2[0][0])**2+(r1[0][1]-r2[0][1])**2)**0.5

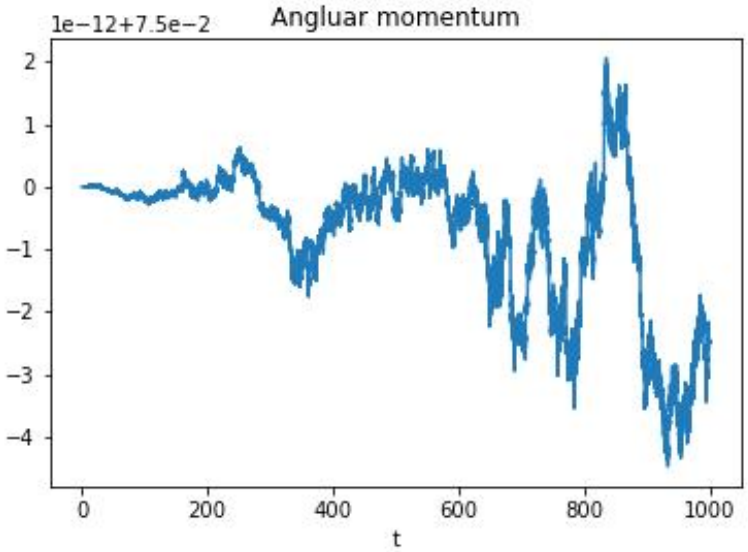
for n in range(1,len(t2)): <-속도, 위치 구하는 과정은 (a)랑 동일. 다만 거기서 L,E까지 구

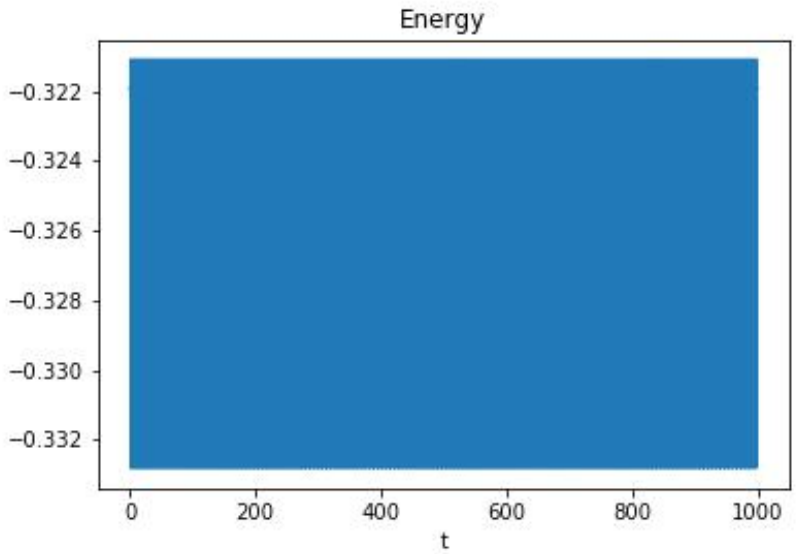
해준다. 시간 되게 오래 걸립니다.

```
r1[n]=r1[n-1]+h*hv1[n-1]
r2[n]=r2[n-1]+h*hv2[n-1]
v1[n]=hv1[n-1]+h/2*a1(r1[n],r2[n])
v2[n]=hv2[n-1]+h/2*a2(r1[n],r2[n])
hv1[n]=hv1[n-1]+h*a1(r1[n],r2[n])
hv2[n]=hv2[n-1]+h*a2(r1[n],r2[n])
mg[n]=r1[n]*1/1.5+r2[n]*0.5/1.5
L[n]=(r1[n][0]*v1[n][1]-r1[n][1]*v1[n][0])+0.5*(r2[n][0]*v2[n][1]-r2[n][1]*v2[n][0])
E[n]=0.5*(v1[n][0]**2+v1[n][1]**2)+0.25*(v2[n][0]**2+v2[n][1]**2)-0.5/((r1[n][0]-r2[n][0])**2+(r1[n][1]-r2[n][1])**2)**0.5
```

```
plt.plot(t2,L,label='L'),plt.xlabel('t'),plt.title('Angluar momentum'),plt.savefig('prob5-1-(d)L.png')
plt.plot(t2,E,label='E'),plt.xlabel('t'),plt.title('Energy'),plt.savefig('prob5-1-(d)E.png')
```

<결과>



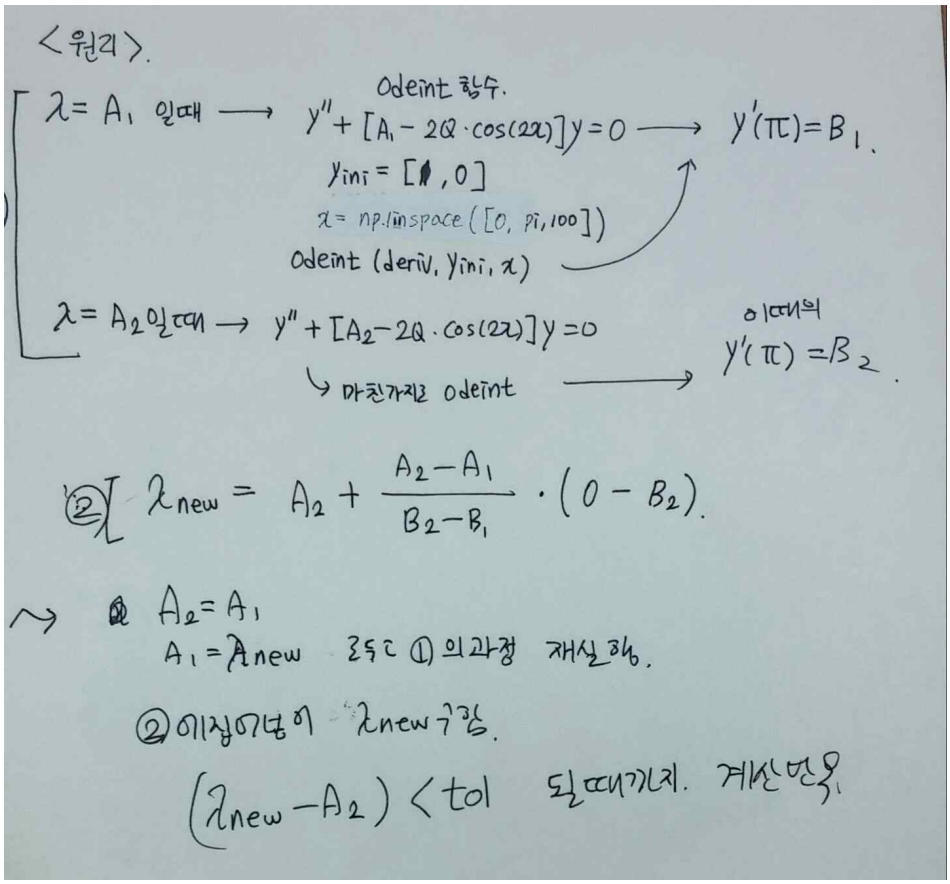


우선 Angular momentum의 경우 0.075라는 값에서 1e-12 scale로 변동이 있는 것을 알 수 있다. 그리고 Energy의 경우 화면을 채운 사각형 형태로 나타났는데 이건 sin,cos함수 마냥 조금씩 값이 변동이 존재하는 것이다. 오차는 0.01범위 내 이다.

이론상으로는 오차가 없이 Angular momentum과 Energy가 보존되어 상수 형태로 나타나야 한다. Angular momentum의 오차의 발생이유는 leaf frog 방법의 근본적인 한계(식을 적분해서 구하는 것이 아니라, 일종의 근사를 하므로 실제 값하고 완전히 들어 맞지 않기 때문에)로 인해 나타나는 결과로 파악된다. Energy역시 leaf frog 방법에 의한 계산오차 때문에 변동이 있는 것 처럼 나타나게 된다.

2번. Eigen value problem

**Eigenvalue problem의 경우, Q에 따른 lambda 값이 여러개가 나온다는것을 추측할 수 있지만, 여러개를 동시에 구하는 방법을 도저히 생각해 낼 수가 없고, 또 일종의 lambda에 대한 함수식이 있지 않을까 생각했지만, 결국 interpolation 방법으로 구하는 수 밖에 없었습니다.



<코드>

```
from numpy import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

#(1) Q=r lamda =?

Q=5
yini=np.array([1.,0.])
x=np.linspace(0.,pi,100)

I1=5.
I2=-5.
```

```
def  deriv1(y,x):
      dydt1=y[1]
      dydt2=-(l1-2*Q*cos(2*x))*y[0]
      return  [dydt1,dydt2]

def  deriv2(y,x):
      dydt1=y[1]
      dydt2=-(l2-2*Q*cos(2*x))*y[0]
      return  [dydt1,dydt2]


y1=odeint(deriv1,yini,x)
y2=odeint(deriv2,yini,x)
B1=y1[99,1]
B2=y2[99,1]

lnew=l2+(l2-l1)*(0.-B2)/(B2-B1)


while(abs(lnew-l2)>10**(-8)):
      l2=l1
      l1=lnew
      y1=odeint(deriv1,yini,x)
      y2=odeint(deriv2,yini,x)
      B1=y1[99,1]
      B2=y2[99,1]
      lnew=l2+(l2-l1)*(0.-B2)/(B2-B1)
```

<구한 lamda 값>

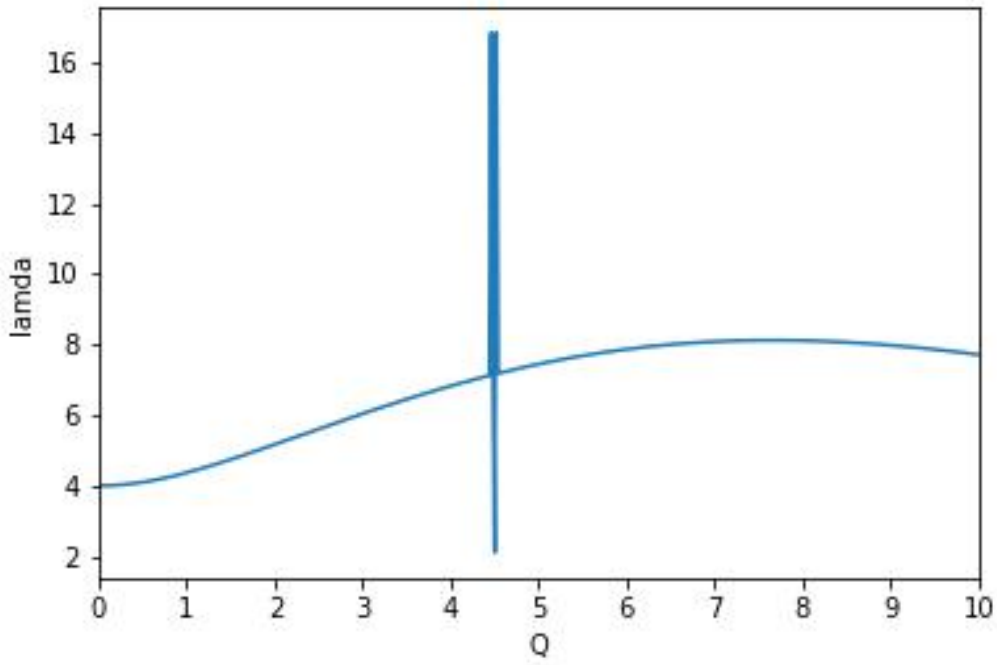
 # lnew=11.54883171536912

 # lnew=-5.80004600546176

 # lnew=7.4491098578413242

#(2) 0=<Q=<10 lambda ?

```
q=np.linspace(0.,10.,501)
lam=np.zeros(len(q))

for i in arange(len(q)):
    I1=5
    I2=-5
    def deriv1(y,x):
      dydt1=y[1]
      dydt2=-(I1-2*q[i]*cos(2*x))*y[0]
      return [dydt1,dydt2]

    def deriv2(y,x):
        dydt1=y[1]
        dydt2=-(I2-2*q[i]*cos(2*x))*y[0]
        return [dydt1,dydt2]

    y1=odeint(deriv1,yini,x)
    y2=odeint(deriv2,yini,x)
    B1=y1[99,1]
    B2=y2[99,1]
    Inew=I2+(I2-I1)*(0.-B2)/(B2-B1)

    while(abs(Inew-I2)>10**(-8)):
        I2=I1
        I1=Inew
        y1=odeint(deriv1,yini,x)
        y2=odeint(deriv2,yini,x)
        B1=y1[99,1]
        B2=y2[99,1]
        Inew=I2+(I2-I1)*(0.-B2)/(B2-B1)

    lam[i]=Inew

plt.plot(q,lam),plt.xlabel('Q'),plt.xlim(0,10),plt.xticks(np.arange(0.,10.5,1.)),plt.ylabel('lamda'),p
lt.savefig('prob5-2.png')
```



중간 Q=4.4부근에 lambda 값이 튀어 오르는 게 나오는데, 아마 연속적으로 이어지는 lambda 값이 아닌, 다른 영역의 lambda 값이 나온것으로 보입니다.

3번. #Prob 5-3

Tridiagonal system에 해당하는 식을 Gauss-Jordan elimination과 Forward and Backward substitution으로 풀기.

문제에 제시된 식을 256*256 정사각형 행렬로 이용해 나타내는 선형대수 식으로 만들면 다음과 같다.

$$\begin{pmatrix} -2 & 1 & 0 & 0 & \dots 0 \\ 1 & -2 & 1 & 0 & \dots 0 \\ 0 & 1 & -2 & 1 & \dots \dots 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & .. & .. & -2 & 1 \\ 0 & \dots & \dots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \dots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} -(2\pi/n)^2 \cos(2\pi/n) \\ -(2\pi/n)^2 \cos(2\pi * 2/n) \\ -(2\pi/n)^2 \cos(2\pi * 3/n) \\ \dots \\ -(2\pi/n)^2 \cos(2\pi * (n-1)/n) \\ -(2\pi/n)^2 \cos(2\pi * n/n) \end{pmatrix}$$

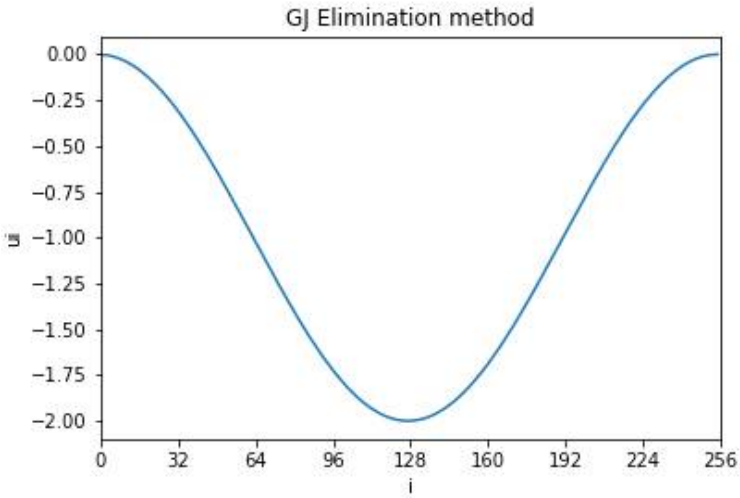
즉, u1~un까지의 값을 구하는 방식을 Gauss Jordan과, Tridiagonal 행렬의 문제를 푸는데 스이는 Forward-backward방법 둘 다 풀어보고 참값이라 할 수 있는 u(x)=cos(x),(단 x=2pi*i/n)값과 비교해본다.

```
from numpy import *
import numpy as np
import matplotlib.pyplot as plt

n=256
a=np.zeros((n,n))
b=np.zeros(n)

for i in range(1,n-1):
    a[i][i]=-2
    a[i][i-1]=1
    a[i][i+1]=1
    b[i]=-(2*pi/n)**2*cos(2*pi*(i+1)/n)

a[0][0]=-2
a[0][1]=1
a[255][255]=-2
a[255][254]=1
b[0]=-(2*pi/n)**2*cos(2*pi*1/n)
b[255]=-(2*pi/n)**2*cos(2*pi*256/n)    <-여기까지가 행렬들을 만들어 주는 과정
```

#(a)Gordon jordan elimination

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

g=np.zeros(n)
g[n-1]=b[n-1]/a[n-1][n-1]
for k in range(n-2,-1,-1):
    su=0.
    for i in range(k,n-1): su+=a[k][i+1]*g[i+1]
    g[k]=(b[k]-su)/a[k][k]
```

```
plt.plot(g),plt.xlim(1,256),plt.xticks(np.arange(0,257.,32)),plt.title('GJ Elimination method'),plt.xlabel('i'),plt.ylabel('ui'),plt.savefig('Prob5-3(a).png')
```

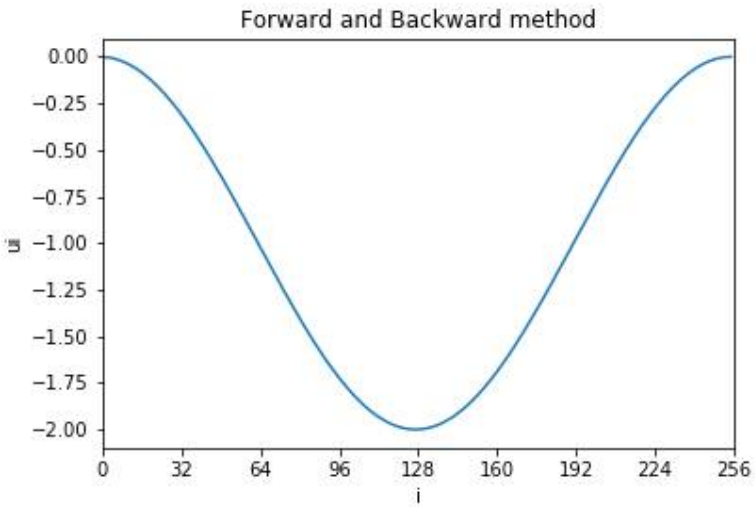


가로축을 i로 두고 세로축을 ui로 둬서, i에 따른 값 분포를 보았습니다. cos 함수의 모양과 상당히 비슷합니다(자세한 비교는 맨 뒤에)

#(b)forward and Backward substitution method

```
n=256
u=np.zeros((n,n))
v=np.zeros(n)
```

```
for i in range(1,n-1):
    u[i][i]=-2
    u[i][i-1]=1
    u[i][i+1]=1
    v[i]=-(2*pi/n)**2*cos(2*pi*(i+1)/n)

u[0][0]=-2
u[0][1]=1
u[255][255]=-2
u[255][254]=1
v[0]=-(2*pi/n)**2*cos(2*pi*1/n)
v[255]=-(2*pi/n)**2*cos(2*pi*256/n)   <-다시 한번더 행렬(array)들 입력해주고

bet=u[0][0]
rho=np.zeros(n)
rho[0]=v[0]/bet
for i in range(1,n):
    u[i-1][i]=u[i-1][i]/bet
    bet=u[i][i]-u[i][i-1]*u[i-1][i]
    rho[i]=(v[i]-u[i][i-1]*rho[i-1])/bet    <-여기까지가 foward substitution

x=np.zeros(n)
x[n-1]=rho[n-1]
for i in range(n-2,-1,-1):x[i]=rho[i]-u[i][i+1]*x[i+1]    <-여기가 backward substitution
plt.plot(x),plt.xlim(1,256),plt.xticks(np.arange(0,257.,32)),plt.xlabel('i'),plt.ylabel('ui'),plt.title('F
orward and Backward method'),plt.savefig('prob5-3(b).png')
#(np.linspace(0.,2*pi,9)
```

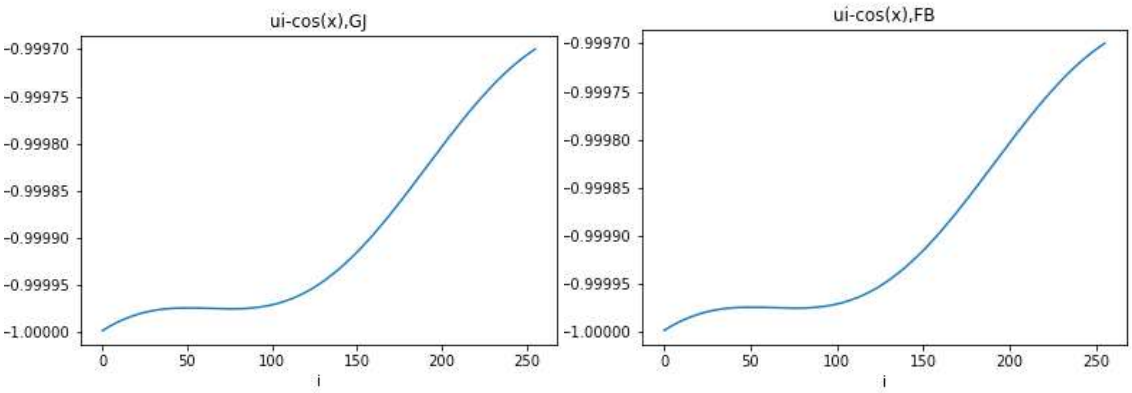


자세한 값을 알 수 없지만 forward backward 방법도 cos 함수 형태의 결과가 나옵니다.

comparing

```
t=np.zeros(n)
for i in range(n): t[i]=2*pi*(i+1)/n
gc=g-cos(t)
xc=x-cos(t)
plt.plot(gc),plt.xlabel('i'),plt.title('ui-cos(x),GJ'),plt.savefig('prob5-3(c)-1.png')
plt.plot(xc),plt.xlabel('i'),plt.title('ui-cos(x),FB'),plt.savefig('prob5-3(c)-2.png')
```

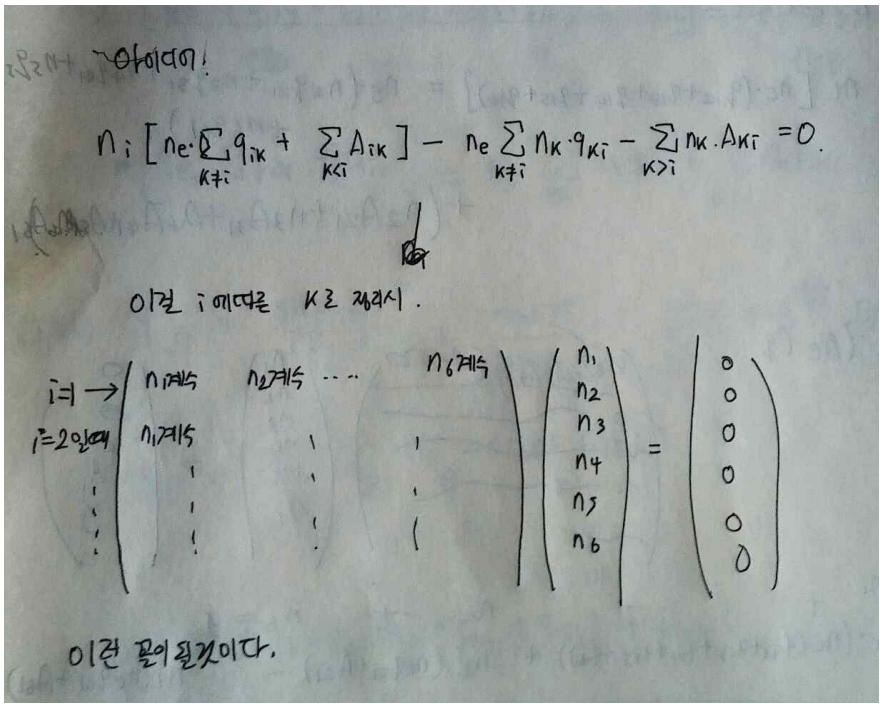
자세한 비교를 위해 참값이라고 할 수 있는 cos함수값과 비교해 보았습니다. u(x)=cos(x)+C (C는 상수)라고 할 때, ui에 대응되어야 할 cos값은 cos(2pi*i/n)이므로, i를 1~256까지 넣어서 구한 cos값들하고 ui값들하고 비교를 하였습니다.



만약 ui가 완벽한 cos 함수꼴로 나타내어 졌다면, ui와 cos(2pi*i/n)사이의 차이는 상수 C만큼만 존재해야 합니다. 즉 상수함수로 일직선의 형태로 나타나야 하는데, GJ와 FB 두 방법 모두 cos함수와의 차이가 일정한 값이 아니라, 증가하는 형태로 나타나는 것을 볼 수 있습니다.

4번.

(a) 코드




```
#prob 5-4
from numpy import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

beta=np.zeros((7,7)) <-식을 보면 전부 i,j로 되어 있는데, python에서 array는 1부터 시작하는게 아니라, 0부터 시작하기에 계산에 혼동이 와서 아예 1~6번 칸을 쓸 수 있게 행렬을 7x7로 잡았습니다.

```
beta[2][1]=0.125
beta[3][1]=0.21
beta[4][1]=0.048
beta[5][1]=0.050
beta[6][1]=0.025
beta[3][2]=0.17
beta[4][2]=0.048
beta[5][2]=0.050
beta[6][2]=0.025
beta[4][3]=0.048
beta[5][3]=0.050
beta[6][3]=0.025
beta[5][4]=-0.018
```

```
og1=np.zeros((7,7))
og1[2][1]=0.408
og1[3][1]=0.272
og1[4][1]=0.2934
og1[5][1]=0.0326
og1[6][1]=0.1323
og1[3][2]=1.120
og1[4][2]=0.8803
og1[5][2]=0.0977
og1[6][2]=0.3968
og1[4][3]=1.4672
og1[5][3]=0.1628
og1[6][3]=0.6613
og1[5][4]=0.8338

A=np.zeros((7,7))
A[2][1]=2.08*10**(-6)
A[3][1]=1.16*10**(-12)
A[4][1]=5.35*10**(-7)
A[5][1]=0.
A[6][1]=0.
A[3][2]=7.46*10**(-6)
A[4][2]=1.01*10**(-3)
A[5][2]=3.38*10**(-2)
A[6][2]=4.80*10
A[4][3]=2.99*10**(-3)
A[5][3]=1.51*10**(-4)
A[6][3]=1.07*100
A[5][4]=1.12

E=np.zeros((7,7))
E[1][2]=0.00609
E[1][3]=0.01628
E[1][4]=1.89879
E[1][5]=4.05244
E[1][6]=5.80061
E[2][3]=0.01019
E[2][4]=1.89270
E[2][5]=4.04635
E[2][6]=5.79452
```

```
E[3][4]=1.88251
E[3][5]=4.03616
E[4][5]=2.15365
E[4][6]=3.90182
E[5][6]=1.74817

w=np.array([0.,1.,3.,5.,5.,1.,1.])
<-여기까지 각종 상수 입력 완료

T=input('input the temperature : ')  <-온도와 전자 밀도 입력
ne=input('input the density of electron :')
#%%

t=T/10**4
q=np.zeros((7,7))
for i in range(1,7):
    for j in range(i+1,7):  <-qji에서 j는 항상 i보다 크니까 이렇게 범위를 지정
        q[j][i]=(8.629*10**(-8))/t**0.5*(og1[j][i]*t**(beta[j][i]))/w[j]
        q[i][j]=q[j][i]*w[j]/w[i]*e**(-1.160*E[i][j]/t)



p=np.zeros((7,7))
p[6]=np.array([0.,1.,1.,1.,1.,1.,1.])

for i in range(1,6):
    for k in range(1,7):
        if(k!=i):  p[i][i]+=ne*q[i][k]
    for k in range(1,i):
        p[i][i]+=ne*A[i][k]

for i in range(1,6):
    for k in range(1,7):
        if(k!=i): p[i][k]-=ne*q[k][i]
    for k in range(i+1,7):
        p[i][k]-=A[k][i]



#Gauss Jordan Elimination
a=p
b=np.zeros(7)
b[6]=1.
```

```
for k in range(1,7):
    for j in range(k+1,7):
        coef=a[j][k]/a[k][k]
        for i in range(k,7):
            a[j][i]-=coef*a[k][i]
        b[j]-=coef*b[k]

ni=np.zeros(7)
ni[6]=b[6]/a[6][6]
for k in range(5,0,-1):
    su=0.
    for i in range(k,6):
        su+=a[k][i+1]*ni[i+1]
    ni[k]=(b[k]-su)/a[k][k]
```

(b) 결과

<T=10000K으로 고정할 때>

ne	N1	N2	N3	N4	N5	N6
10^1	7.09234713e-01	1.94269737e-01	9.64947819e-02	7.02290368e-07	2.22185479e-11	6.54970508e-08
10^2	2.65022907e-01	4.55099463e-01,	2.79874983e-01,	7.05506574e-07,	2.23258940e-11,	1.94203440e-06
10^3	1.44716651e-01	5.24899147e-01	3.30360532e-01	7.06465350e-07,	2.23547232e-11	2.29635498e-05
10^4	1.30630495e-01	5.32934004e-01	3.36201061e-01	7.06431230e-07	2.23534374e-11,	2.33733874e-04
10^5	1.29269863e-01	5.32436413e-01	3.35957455e-01	7.04954881e-07	2.23067097e-11	2.33556379e-03

<ne=10000 으로 고정할 때>

	N1	N2	N3	N4	N5	N6
T=5000K	1.37772550e-01	5.28141125e-01	3.33854318e-01	1.07873182e-07	2.80302858e-13	2.31898246e-04
T=7000K	1.34927847e-01	5.30117657e-01	3.34721585e-01	3.24330244e-07	3.51536962e-12	2.32587563e-04
T=12000K	1.27963724e-01	5.34640732e-01	3.37160140e-01	9.37530006e-07	4.50031708e-11	2.34467058e-04
T=15000K	1.24417281e-01	5.36890410e-01	3.38455638e-01	1.22147966e-06	8.89533719e-11	2.35449842e-04
T=20000K	1.19656678e-01	5.39896354e-01	3.40208654e-01	1.54566784e-06	1.70791244e-10	2.36768971e-04

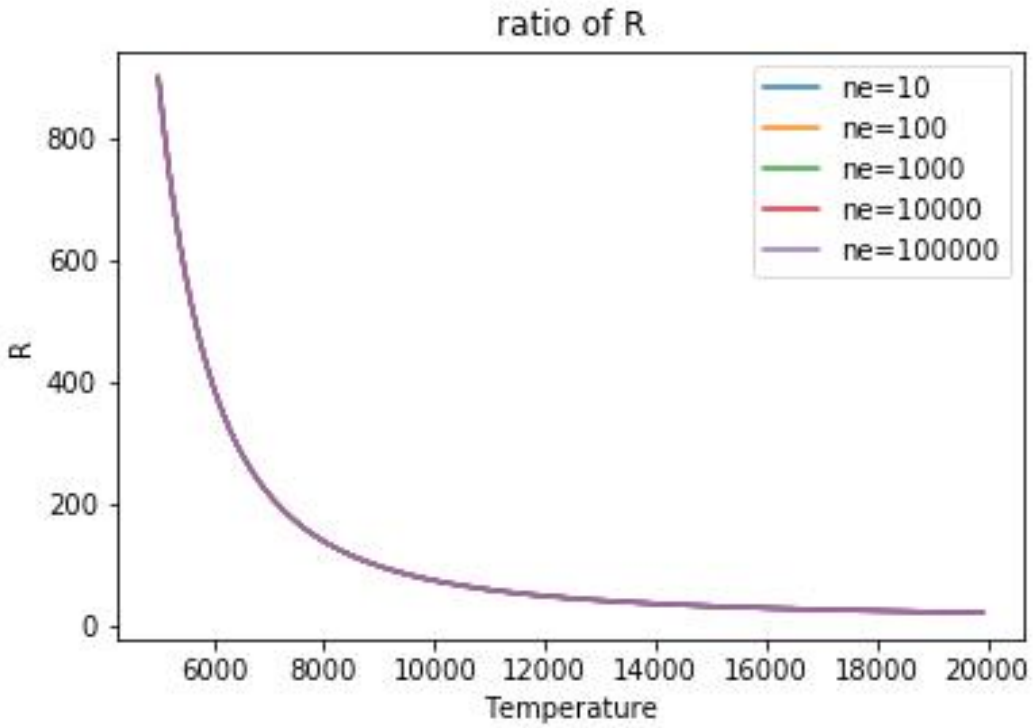
(c) 그래프 결과

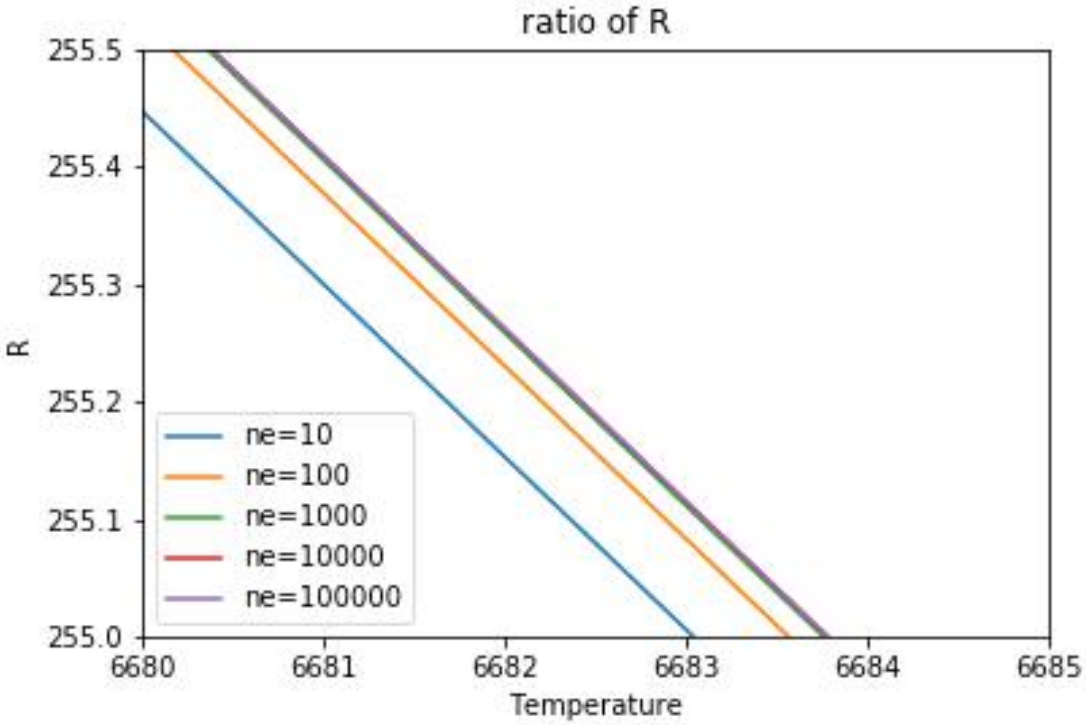
<코드>-앞에서 이미 A,E,w,beta 모두 입력 했다고 가정하고

```
Temp=np.arange(5000.,20000,100)
R=np.zeros((len(Temp),5))

for h in range(5):
    ne=10**(h+1)
    for u in range(len(Temp)):
        T=Temp[u]
        t=T/10**4
        q=np.zeros((7,7))
        for i in range(1,7):
            for j in range(i+1,7):
                q[j][i]=(8.629*10**(-8))/t**0.5*(og1[j][i]*t**(beta[j][i]))/w[j]
                q[i][j]=q[j][i]*w[j]/w[i]*e**(-1.160*E[i][j]/t)



        p=np.zeros((7,7))
        p[6]=np.array([0.,1.,1.,1.,1.,1.,1.])

        for i in range(1,6):
            for k in range(1,7):
                if(k!=i):   p[i][i]+=ne*q[i][k]
            for k in range(1,i):
                p[i][i]+=ne*A[i][k]

        for i in range(1,6):
            for k in range(1,7):
                if(k!=i): p[i][k]-=ne*q[k][i]
            for k in range(i+1,7):
                p[i][k]-=A[k][i]



        #Gauss Jordan Elimination
        a=p
        b=np.zeros(7)
        b[6]=1.
        for k in range(1,7):
            for j in range(k+1,7):
                coef=a[j][k]/a[k][k]
                for i in range(k,7):
```

```
                a[j][i]-=coef*a[k][i]
            b[j]-=coef*b[k]

    ni=np.zeros(7)
    ni[6]=b[6]/a[6][6]
    for k in range(5,0,-1):
        su=0.
        for i in range(k,6):
            su+=a[k][i+1]*ni[i+1]
        ni[k]=(b[k]-su)/a[k][k]

    R[u,h]=ni[4]*A[4][3]*5755/(ni[5]*A[5][4]*6584)
```

```
plt.plot(Temp,R[:,0],label='ne=10'),plt.plot(Temp,R[:,1],label='ne=100'),plt.plot(Temp,R[:,2],label='ne=1000'),plt.plot(Temp,R[:,3],label='ne=10000'),plt.plot(Temp,R[:,4],label='ne=100000'),plt.legend(),plt.xlabel('Temperature'),plt.ylabel('R'),plt.title('ratio of R'),plt.savefig('prob5-4(c)-1.png')
plt.plot(Temp,R[:,0],label='ne=10'),plt.plot(Temp,R[:,1],label='ne=100'),plt.plot(Temp,R[:,2],label='ne=1000'),plt.plot(Temp,R[:,3],label='ne=10000'),plt.plot(Temp,R[:,4],label='ne=100000'),plt.legend(),plt.xlabel('Temperature'),plt.ylabel('R'),plt.title('ratio of R'),plt.xlim(6680,6685),plt.ylim(255,255.5),plt.savefig('prob5-4(c)-2.png')
```





전자밀도 ne에 따른 차이가 매우 미세하다.