```
import os,sys,math
from numpy import *
from Amat import *

sys.path.append("/Users/kuntaro/00.Develop/Prog/02.Python/00.SimplyConverted/Lib
from ReadMtz import *


class ReflWidth:

 def __init__(self,amatfile,divv,divh,mosaic,dispersion,oscstep):
  self.amatfile=amatfile
  self.mosaic=mosaic
  self.dispersion=dispersion  #UNIT: degree
  self.divv=divv   #UNIT: degree
  self.divh=divh   #UNIT: degree
  self.cuspflag=0
  self.width_max=3.0   # UNIT: degree
  self.oscstep=0.1   # UNIT: degree
  self.isPrepDELEPS=False

  # Some flags
  self.isInit=False
  self.isSolved=False

 def init(self):
  # S0 vector (anti-parallel to the x-ray)
  self.s0=array((-1,0,0))
  # E3 vector (Z axis: rotation axis)
  self.e3=array((0,0,1))

  # A matrix file open and read 'A matrix'
         amatftmp=Amat(self.amatfile)
  self.amat=amatftmp.getAmat()
  self.isInit=True

 def setHKL(self,hkl): # hkl is a array of reflection indice (type:integer)
  if self.isInit==False:
   self.init()
  self.isPrepDELEPS=False

  ## HKL -> XYZ in reciprocal space
  # convert int -> float
  tmp_hkl=[]
  tmp_hkl.append(float(hkl[0]))
  tmp_hkl.append(float(hkl[1]))
  tmp_hkl.append(float(hkl[2]))
  self.hkl=hkl

  # Amat*HKL -> XYZ in reciprocal space
  float_hkl=array(tmp_hkl)
  self.xyz=dot(self.amat,float_hkl)

  ## E2 vector (E3xRLP/|E3xRLP|)
  self.e2=cross(self.e3,self.xyz)/linalg.norm(cross(self.e3,self.xyz))
  # E1 vector (E2 x E3)
  self.e1=cross(self.e2,self.e3)

  ## Calculating d* value
  self.dstar=linalg.norm(self.xyz)
  self.dstar2=self.dstar*self.dstar
  self.dst4=self.dstar2*self.dstar2*0.25

  ## XRLP .vs. Ewald sphere
         ## Diffraction condition
         # CEA.cos(phic)+CEB.sin(phic)=CEC
```

```
        # 1) CEA=(XRLP.E1)*(E1.S0)
        # 2) CEB=(XRLP.E1)*(E2.S0)
        # 3) CEC=0.5*(XRLP.E1)^2+0.5*(XRLP.E3)^2-(XRLP.E3)*(E3.S0)
        # 4) CEABSQ=CEA^2+CEB^2 (=0.0 -> XRLP on rotation axis)

 ## DEBUG
 #print "E1=",self.e1
 #print "E2=",self.e2
 #print "E3=",self.e3

 # Preparation
 # xe1: XRLP.E1
 xe1=dot(self.xyz,self.e1)
 # xe3: XRLP.E3
 xe3=dot(self.xyz,self.e3)
 # XE1D
 e1_dot_s0=dot(self.e1,self.s0)
 e2_dot_s0=dot(self.e2,self.s0)
 e3_dot_s0=dot(self.e3,self.s0)

 ## DEBUG
 #print "XE1/3=",xe1,xe3
 #print "E1.S0/E2.S0/E3.S0=",e1_dot_s0,e2_dot_s0,e3_dot_s0

 ####
 # CEA,CEB,CEC
 self.cea=xe1*e1_dot_s0
 self.ceb=xe1*e2_dot_s0
 self.cec=0.5*pow(xe1,2.0)+0.5*pow(xe3,2.0)-(xe3*e3_dot_s0)
 #print "CEA=",self.cea
 #print "CEB=",ceb
 #print "CEC=",cec

 self.ceabsq=pow(self.cea,2.0)+pow(self.ceb,2.0)
 #print "CEABS %12.5f"%self.ceabsq
 # There are 2 solutions where RLP crosses Ewald Sphere
 if self.ceabsq!=0.0:
  self.arg1=self.cec/sqrt(self.ceabsq)
  #print "ARG=",self.arg1
  return True
 else:
  return False

def solvePhi(self,phistart):
 dtor=4.0*arctan(1.0)/180.0

              if self.arg1>1.0:
   self.arg1=1.0
              elif self.arg1<-1.0:
   self.arg1=-1.0

 # solutions in unit of radians
 t1=arccos(self.arg1)
 t2=arctan2(self.ceb,self.cea)

              # 1st solution in unit of degree
              phic=degrees(t1+t2)
 phia=phistart+phic
              # 2nd solutin in unit of degree
 self.phic=degrees(-t1+t2)
              phib=phistart+self.phic;

              # Choosing the solution
 diff1=fabs(phia-phistart)
 diff2=fabs(phib-phistart)
```

```
 # DEBUG
 #-print "T1/T2=",t1,t2
 #print "PHIA/PHIB=",phia,phib
 #-print "DIFF1/DIFF2=",diff1,diff2

 # self.phi UNIT:degrees
 if diff1<diff2:
  self.phi=phia
 else:
  self.phi=phib

 #print "solved PHI",self.phi
 self.isSolved=True

def makeRotMat(self,phideg):
 #print "PHIDEG=",phideg
               phirad=radians(phideg)
 #print "PHIRAD/COS(PHIRAD)/SIN(PHIRAD)=",phirad,cos(phirad),sin(phirad)
               tpl=matrix( (
                       ( cos(phirad), -sin(phirad),0.),
                       ( sin(phirad), cos(phirad),0),
                       ( 0., 0., 1.)
               ) )
               mat=array(matrix((tpl)).reshape(3,3))
 #print "ROTMAT",mat

 return mat

def distEStoRLP(self) :
 if self.isSolved==False :
  self.solvePhi()
 # Ewald sphere (x+1)^2 + y^2 + z^2 = 1.0
       ##  XRLP at start phi
 #-print "XYZ:",self.xyz
 x1=self.xyz[0]
 y1=self.xyz[1]
 z1=self.xyz[2]
 x1_2=(x1+1.0)*(x1+1.0)
 y1_2=y1*y1
 z1_2=z1*z1
# Distance from XYZ to Ewald sphere
 self.del1=sqrt(x1_2+y1_2+z1_2)-1.0
 self.adel1=fabs(self.del1)
 #-print "DEL1:",self.del1
 #-print "ADEL1",self.adel1

def calcLorentz(self):
 # Matrix required for Lorentz factor estimation
               tmp1=cross(self.e3,self.s0)
 # RLP coordinates at the 'solved phi' angle
 # self.phic [degrees]
 phicrotmat=self.makeRotMat(self.phic)
 xrlpe=dot(phicrotmat,self.xyz)
 #print "XRLPE",xrlpe

        # Lorentz factor is estimated at the beginning rotation angle
 #print tmp1,xrlpe
               t3=dot(tmp1,xrlpe)
               self.lorentz_factor=fabs(1.0/t3);
 #print "T3=",t3
 #-print "Lorentz",self.lorentz_factor

def calcRspot(self):

 # Divergence
 #---- Conventional source geometry
```

```
 #  Radius of reciprocal lattice point along radius of Ewald sphere
 #     RSPOT = 0.5*(DIVERGENCE+dispersion*tan(theta) )*DSTAR*COS(THETA)
 #     and as DSTAR*COS(THETA) = SQRT(DSTAR**2-0.25*DSTAR**4)
 #  Note that the divergence parameters DIVH,DIVV and the mosaic spread
 #  are stored in the generate file as the FULL WIDTHS in degrees.
 #  These are converted to HALF WIDTHS in radians prior to the prediction
 #  calculations.
 #  Add the contribution due to finite block size.

 #print "DEG: divv,divh=",self.divv,self.divh
 #print "DEG: mosaic=",self.mosaic
 #print "RAD: divv,divh=",radians(self.divv),radians(self.divh)
 #print "RAD: mosaic=",radians(self.mosaic)

 # Mosaic/Divergence convertion which can match to MOSFLM
 # For strategy option ON ????
 divv=radians(self.divv)/2.0
 divh=radians(self.divh)/2.0
 mosaic=radians(self.mosaic)/2.0

 # Energy dispersion?
 delcor=0.0001
 # z1 of XRLP
 z1=self.xyz[2]
 # ymid : in the MOSFLM (phistart+phiend)/2.0 but
 # ymid -> y1 for still data collection
 ymid=self.xyz[1]
 yms=ymid*ymid
 # Preparation of parameters required for divergence calculation
 #-print "DEBUG:dstart2/z1/divh=",self.dstar2,z1,divh
 esyn_h=delcor*self.dstar2+z1*divh
 esyn_v=divv*ymid
 #-print "ESYNH/ESYNV/YMS/Z1S",esyn_h,esyn_v,yms,z1*z1
 # Divergence calculation
 divergence=sqrt((pow(esyn_h,2.0)+pow(esyn_v,2.0))/(yms+z1*z1))
 # Reflection spot radius
 self.rspot=(divergence+mosaic)*sqrt(self.dstar2-self.dst4) \
  +0.25*self.dispersion*self.dstar2
 #-print "DIVERG/ETA/DSTAR2/RSPOT=",divergence,mosaic,self.dstar2,self.rspot
 #print "RSPOT=",self.rspot
 return self.rspot

def cuspcheck(self):
               csmin=0.5*self.dstar2+self.rspot
               csmin2=self.dst4+self.dstar2*self.rspot
 x,y=self.xyz[0],self.xyz[1]
 xys=x*x+y*y
 #=====================
           # What should csimin test be ? The spot can still appear on image
           # even if the centre of the rlp never cuts the sphere (ie lies in
           # the cusp region) providing any part of the rlp touches the sphere.
           # In this case, the test on line below is the right one, but this
           # seems to overpredict in practice, so limit it to case where the
           # centre of the rlp must intersect the sphere.
 #=====================
               if xys<=csmin2:
               ##      A part of spot is in cusp region
                       self.cuspflag=-3
 elif(xys<self.dst4):
               ##      Whole spot is included in cusp region
                       self.cuspflag=-4
                       self.inCusp=true

 #print self.cuspflag

       ## Reflection width and start/end phi of diffraction
```

```
        def diffWidth(self) :
        ##      Full width of reflection condition (UNIT:radians)
  dtor=4.0*arctan(1.0)/180.0
  # self.phiw = spot angular radius in unit:degrees
                self.phiw=2.0*self.rspot*self.lorentz_factor/dtor
                self.phis=self.phi-0.5*self.phiw
                self.phie=self.phis+self.phiw
  #- print self.phiw,self.phis,self.phie

 def numframes(self) :
        ##      Estimating max frames of this reflections
                maxframes=(int)(self.width_max/oscstep)
  #- print maxframes

        ##      Find start BATCH in which this reflection is observed
  for i in range(1,maxframes+1):
                        if phistart-(i-1)*oscstep<=self.phis:
                                self.istart=i
                                break

                for i in range(1,maxframes+1):
                        if phiend+(i-1)*oscstep>=phie:
                                self.iend=i;
                                break;
                self.iwidth=self.istart-1+self.iend-1+1;
  #- print self.iwidth,self.istart,self.iend

        ## Estimation of deleps1,deleps2. this is the objective of this CLASS
 def prepDELEPS(self,phistart):
  self.solvePhi(phistart)
  self.distEStoRLP()
  self.calcLorentz()
  self.calcRspot()
  self.diffWidth()
  self.cuspcheck()
  self.isPrepDELEPS=True

 def calcDELEPS(self,phistart):
  if self.isPrepDELEPS==False:
   self.prepDELEPS(phistart)
        ## Is spot cut both ends? -> 120904 always TRUE
        ## Calculate distance of edge of spot from sphere at end of rotation
        #double dist2=adel2-rspot;

        ## Test if spot is cut at beginning of rotation
        ## Set DELEPS to a negative value
        ## NOTE: sign change depending on whether Y is +ve/-ve

  x1,y1,z1=self.xyz[0],self.xyz[1],self.xyz[2]
  self.deleps1=0.0

  # ADEL1 -> Cross section between E.S and RLP (UNIT:radians)
  if self.adel1-self.rspot<0.0:
   if y1<0.0:
    sign=-1
   else :
    sign=1
   self.deleps1=-(sign*self.del1/self.rspot+1.0)*0.5

  print "%20s %8.2f"%(self.hkl,self.deleps1)
  return self.deleps1


if __name__=="__main__":
 #amatfile,divv,divh,mosaic,dispersion,oscstep):
 tmp=ReflWidth(sys.argv[1],0.02,0.02,0.5,0.0002,0.1)
```

```
mtzf=ReadMtz(sys.argv[2])
miller=mtzf.getMillerOrig()

for phi in (0,30,60,90):
 for hkl in miller:
  #print "HKL type is ",type(hkl)
  if tmp.setHKL(hkl)==True:
   tmp.calcDELEPS(phi)
```