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

class ReflWidthStill:

	def __init__(self,amatfile,divv,divh,mosaic,dispersion,wavelength=1.24):
		self.amatfile=amatfile
		self.mosaic=mosaic
		self.dispersion=dispersion	#UNIT: degree
		self.divv=divv			# UNIT: degree
		self.divh=divh			# UNIT: degree
		self.cuspflag=1
		self.width_max=3.0		# UNIT: degree
		self.isPrepDELEPS=False
		self.isSetMisset=False
		self.wl=wavelength
		self.mosaic_block=0.0

		# 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 setMosaic(self,mosaic):
		self.mosaic=mosaic

	def setMosaicBlock(self,mosaic_block):
		self.mosaic_block=mosaic_block

	def setMisset(self,rx,ry,rz):
		rotx=self.makeRotX(rx)
		roty=self.makeRotY(ry)
		rotz=self.makeRotZ(rz)

		rot_xy=dot(rotx,roty)
		self.misset=dot(rot_xy,rotz)
		self.isSetMisset=True
		#print self.misset

	# hkl: array of reflection index (type: integer)
	# phistart: phi start angle [degrees]

	# phistart: start PHI angle
	# phiend  : end PHI angle
	# used in MOSFLM integration
	def setHKL(self,hkl,phistart,phiend):
		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

		self.phimid=(phistart+phiend)/2.0

		# Rotation matrix with phimid ((startphi+endphi)/2.0 in MOSFLM)
		phimid_matr=self.makeRotMat(self.phimid)
```

```
		# Amat x Rotation matrix (@end phi)
		self.amat_midphi=dot(phimid_matr,self.amat)
		if self.isSetMisset:
			self.amat_midphi=dot(self.misset,self.amat_midphi)

		# Amat*HKL -> XYZ in reciprocal space
		float_hkl=array(tmp_hkl)

		# XYZ1: RLP at start phi
		self.xyz1=dot(self.amat_midphi,float_hkl)

		# E1/E2/E3 vectors are calculated from XYZ1(RLP@ start phi)
		##	E2 vector (E3xRLP/|E3xRLP|)
		self.e2=cross(self.e3,self.xyz1)/linalg.norm(cross(self.e3,self.xyz1))
		#	E1 vector (E2 x E3)
		self.e1=cross(self.e2,self.e3)

		## d* value is calculated from XYZ1(RLP@start phi)
		## Calculating d* value
		self.dstar=linalg.norm(self.xyz1)
		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.xyz1,self.e1)
		# xe3: XRLP.E3
		xe3=dot(self.xyz1,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 getRLP(self):
```

```
		return self.xyz1

	def getD(self):
		# d=(wavelength/self.dstar)
		# dstar_true=(self.dstar/wavelength)
		return (1.0/self.dstar*self.wl)

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

		################
		# self.arg1 value is not in the reasonable range
		################
		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=self.phimid+phic
		# 2nd solutin in unit of degree
		self.phic=degrees(-t1+t2)
		phib=self.phimid+self.phic;

		# Choosing the solution
		diff1=fabs(phia-self.phimid)
		diff2=fabs(phib-self.phimid)

		# 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):
		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))
		return mat

	def makeRotZ(self,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))
		return mat

	def makeRotX(self,phix):
		phirad=radians(phix)
		#print "PHIRAD/COS(PHIRAD)/SIN(PHIRAD)=",phirad,cos(phirad),sin(phirad)
		tpl=matrix( (
			( 1., 0., 0.),
```

```
                ( 0.,cos(phirad), −sin(phirad)),
                ( 0.,sin(phirad), cos(phirad))
        ) )
        mat=array(matrix((tpl)).reshape(3,3))
        return mat

    def makeRotY(self,phiy):
        phirad=radians(phiy)
        #print "PHIRAD/COS(PHIRAD)/SIN(PHIRAD)=",phirad,cos(phirad),sin(phirad)
        tpl=matrix( (
                ( cos(phirad), 0., sin(phirad)),
                ( 0., 1., 0.),
                ( −sin(phirad), 0., cos(phirad)),
        ) )
        mat=array(matrix((tpl)).reshape(3,3))
        return mat

    def distEwaldToRLP(self) :
        if self.isSolved==False :
                self.solvePhi()
        # Ewald sphere (x+1)^2 + y^2 + z^2 = 1.0
    ######################
    ##  XRLP at start phi
    ######################
        x1=self.xyz1[0]
        y1=self.xyz1[1]
        z1=self.xyz1[2]

        # Some short cut variants
        # for XYZ1@start phi
        x1_2=(x1+1.0)*(x1+1.0)
        y1_2=y1*y1
        z1_2=z1*z1
        #print "XYZ1 %12.5f %12.5f %12.5f"%(x1,y1,z1)
        #print "XYZ2 %12.5f %12.5f %12.5f"%(x2,y2,z2)
        #####################
        # Distance from XYZ to Ewald sphere
        #####################
        self.del1=sqrt(x1_2+y1_2+z1_2)−1.0
        self.adel1=fabs(self.del1)

        return True

    def getPHI(self):
        return self.phi

    def getLorentz(self):
        return self.lorentz_factor

    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]
        #print "LOR: ",self.phic
        phicrotmat=self.makeRotMat(self.phic)
        xrlpe=dot(phicrotmat,self.xyz1)

        # Holton factor
        htmp=dot(self.e3,xrlpe)
        if htmp==0.0:
                self.holton_factor=0.0
        else:
                self.holton_factor=1.0/fabs(dot(self.e3,xrlpe))
        # Lorentz factor is estimated at the beginning rotation angle
        t3=dot(tmp1,xrlpe)
        if t3==0.0:
                self.lorentz_factor=0.0
                return 0.0
        else:
                self.lorentz_factor=fabs(1.0/t3);
                #print self.holton_factor,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.xyz1[2]
            # ymid : in the MOSFLM (Y@phistart+Y@phiend)/2.0
            ymid=self.xyz1[1]
            yms=ymid*ymid

            # Preparation of parameters required for divergence calculation
            esyn_h=delcor*self.dstar2+z1*divh
            esyn_v=divv*ymid

            # Divergence calculation
            divergence=sqrt((pow(esyn_h,2.0)+pow(esyn_v,2.0))/(yms+z1*z1))

            # mosaic block term
            if self.mosaic_block!=0.0:
                    mos_term=1.0/(self.dstar*self.mosaic_block)
            else:
                    mos_term=0.0

            # Reflection spot radius
            self.rspot=(divergence+mosaic)*sqrt(self.dstar2-self.dst4) \
                    +0.25*self.dispersion*self.dstar2+mos_term
            #print "DSTAR/RSPOT/MOS_TERM=",self.dstar,self.rspot-mos_term,mos_term
            #-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.xyz1[0],self.xyz1[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

    ## 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

        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):
                self.solvePhi()
                self.distEwaldToRLP()
                self.calcLorentz()
                self.calcRspot()
                self.diffWidth()
                self.cuspcheck()
                self.isPrepDELEPS=True

        def getPhiw(self):
                if self.isPrepDELEPS==False:
                        self.prepDELEPS()
                return self.phiw

        def calcDELEPS(self):
                # Preparation
                if self.isPrepDELEPS==False:
                        self.prepDELEPS()

        ## Calculate distance of edge of spot from sphere at end of rotation
                dist1=self.adel1-self.rspot;

                #print self.adel1,self.adel2
                #print "DIST1/DIST2=%12.5f %12.5f"%(dist1,dist2)

        ## 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.xyz1[0],self.xyz1[1],self.xyz1[2]

                self.deleps1=0.0

                # Flag for partial/full reflection
                self.isFull=True

                #print "DIST",dist1,dist2

                # ADEL1 -> Cross section between E.S and RLP (UNIT:radians)
                # Firstly, check CUSP flag
                if self.cuspflag<0:
                        self.isFull=False

                elif dist1<0.0:
                        self.isFull=False
                        if y1<0.0:
```

```
                        sign=-1
                else :
                        sign=1

                # DELEPS1 calculation
                self.deleps1=-(sign*self.del1/self.rspot+1.0)*0.5

        return self.deleps1

    def DEBUG_calcPartiality(self):
        print "====== PART CALC ====="
        dist1=self.adel1-self.rspot
        dist2=self.adel2-self.rspot

        if dist1<0.0:
                print "Start point is on ES %12.6f %12.6f"%(self.deleps1,self.deleps2)
                if dist2<0.0:
                        print "Both is on ES %12.6f %12.6f"%(self.deleps1,self.deleps2)
        elif dist2<0.0:
                print "After oscillation on ES %12.6f %12.6f"%(self.deleps1,self.deleps2)

        else:
                print "Hashinimo bounimo %12.6f %12.6f"%(self.deleps1,self.deleps2)

        return 1.0

    def getDel(self):
        return self.adel1,self.adel2

    def calcPartiality(self):
        dist1=self.adel1-self.rspot
        dist2=self.adel2-self.rspot

        ## Case1
        if self.isFull:
                p1=self.adel1/self.rspot
                p2=self.adel2/self.rspot
                self.pcalc=array([p1,p2]).min()
                #print self.pcalc
                return self.pcalc

        elif fabs(self.deleps2)<0.00001 :
                self.pcalc=0.5*(1.0-cos(self.deleps1*pi))
                return self.pcalc

        elif fabs(self.deleps1)<0.00001 :
                self.pcalc=0.5*(1.0-cos(self.deleps2*pi))
                return self.pcalc

        else:
                tmp=0.5*(1.0-cos(self.deleps1*pi))
                self.pcalc=tmp-(1.0-0.5*(1.0-cos(self.deleps2*pi)))
                return self.pcalc

    def calcP_Rossmann(self):

        mr_model=lambda a:3.0*a*a-2.0*a*a*a

        if self.isFull:
                p1=self.adel1/self.rspot
                p2=self.adel2/self.rspot
                self.pcalc=array([p1,p2]).min()
                #print self.pcalc
                return self.pcalc

        elif fabs(self.deleps2)<0.00001 :
                self.pcalc=mr_model(self.deleps1)
                return self.pcalc

        elif fabs(self.deleps1)<0.00001 :
                self.pcalc=mr_model(self.deleps2)
                return self.pcalc

        else:
```

```
                        tmp=mr_model(self.deleps1)
                        self.pcalc=tmp-(1.0-mr_model(self.deleps2))
                        return self.pcalc

        def calcP_tsuki(self):

                model=lambda q:(q-(sin(2.0*pi*q))/2.0/pi)

                if self.isFull:
                        p1=self.adel1/self.rspot
                        p2=self.adel2/self.rspot
                        self.pcalc=array([p1,p2]).min()
                        #print self.pcalc
                        return self.pcalc

                elif fabs(self.deleps2)<0.00001 :
                        self.pcalc=model(self.deleps1)
                        return self.pcalc

                elif fabs(self.deleps1)<0.00001 :
                        self.pcalc=model(self.deleps2)
                        return self.pcalc

                else:
                        tmp=model(self.deleps1)
                        self.pcalc=tmp-(1.0-model(self.deleps2))
                        return self.pcalc

if __name__=="__main__":
        tmp=ReflWidthBothEdge(sys.argv[1],0.02,0.02,0.5,0.0002,0.1)
        h,k,l=int(sys.argv[2]),int(sys.argv[3]),int(sys.argv[4])
        hklist=[array((h,k,l))]

        oscstart=0.0

        tmp.setMisseting(0.0,0.0,0.0)

        for hkl in hklist:
                #print "HKL type is ",type(hkl)
                if tmp.setHKL(hkl,oscstart)==True:
                        tmp.calcDELEPS()
                        print tmp.calcPartiality()
```