hexrd Documentation

Release 0.2.2

Joel Bernier, et. al.

December 13, 2014

Contents

CHAPTER 1

Welcome to HEXRD's documentation!

Contents:

1.1 The HEXRD API

hexrd.arrayutil	
hexrd.fileutil	
hexrd.gridutil	
hexrd.matrixutil	
hexrd.orientations	
hexrd.pfigutil	
hexrd.plotwrap	***
hexrd.quadrature.q1db	
hexrd.quadrature.q2db	
hexrd.quadrature.q3db	
hexrd.tens	
hexrd.valunits	Module for associating units with scalar quantities
hexrd.xrd.crystallography	
hexrd.xrd.detector	
hexrd.xrd.distortion	
hexrd.xrd.experiment	Module for wrapping the main functionality of the xrd package.
hexrd.xrd.fitting	
hexrd.xrd.grain	
hexrd.xrd.hydra	Hydra detector tools
hexrd.xrd.indexer	
hexrd.xrd.material	Module for XRD material class
hexrd.xrd.rotations	
hexrd.xrd.spacegroup	Interface with sglite for hkl generation and Laue group determination
hexrd.xrd.spotfinder	
hexrd.xrd.symmetry	
hexrd.xrd.transforms	
hexrd.xrd.transforms_CAPI	
hexrd.xrd.xrdbase	
hexrd.xrd.xrdutil	

1.1.1 Module: `arrayutil`

6 Functions

`hexrd.arrayutil.`**`getMem`**(*shape*, *asOnes=False*, *asZeros=False*, *typeInt=False*)

`hexrd.arrayutil.`**`toArray`**(*a*)

`hexrd.arrayutil.`**`writeArray`**(*fid*, **args*, ***dargs*)
Print to file, pasting arrays together; eventually replace with numpy.savetxt?

`hexrd.arrayutil.`**`arrayToString`**(*a*)

`hexrd.arrayutil.`**`structuredSort`**(*order*, *things*)
sort things by order, return sorted things

`hexrd.arrayutil.`**`histoFit`**(*data*, *nBins*, *plot=False*)
Fit data using histogramming; useful for stuff pulled in using DataThief

1.1.2 Module: `fileutil`

5 Classes

class `hexrd.fileutil.`**`Log`**(*logFileName=None*, *toScreen=True*)
for logging

`__init__`(*logFileName=None*, *toScreen=True*)

class `hexrd.fileutil.`**`FileDescr`**(**args*)
base class

`__init__`(**args*)

class `hexrd.fileutil.`**`FileLink`**(*fileName*)
Bases: `hexrd.fileutil.FileDescr`

`__init__`(*fileName*)

class `hexrd.fileutil.`**`FileLinkWild`**(*fileNameWild*)
Bases: `hexrd.fileutil.FileDescr`

must be given an absolute path

`__init__`(*fileNameWild*)

class `hexrd.fileutil.`**`FileForm`**(*fileName*, *formFile=None*, *form=None*, *dictForDefs=None*)
Bases: `hexrd.fileutil.FileDescr`

`__init__`(*fileName*, *formFile=None*, *form=None*, *dictForDefs=None*)

22 Functions

`hexrd.fileutil.`**`resolveWild`**(*fname*)
might be better to go to glob.glob() call here

`hexrd.fileutil.`**`catList`**(*lines*, *sep=''*)

`hexrd.fileutil.`**`fileToForm`**(*fname*)

hexrd.fileutil.**readFloatDataA**(*fname*)
This is definitely faster than readFloatData when it is appropriate, which is pretty much any time the data can be interpreted as an array and numpy is available

hexrd.fileutil.**readFloatData**(*fname=None*)

hexrd.fileutil.**readDataFlat**(*fname*)

hexrd.fileutil.**archiveDir**(*dirName*)

hexrd.fileutil.**archiveFile**(*fileName*)

hexrd.fileutil.**getFromPipe**(*command*, *werr=False*)

hexrd.fileutil.**getSysType**()

hexrd.fileutil.**getNCorePerNode**()

hexrd.fileutil.**getHostName**()

hexrd.fileutil.**getBankNames**()

hexrd.fileutil.**rmSafe**(*fileName*)

hexrd.fileutil.**rmDirF**(*dirName*)

hexrd.fileutil.**rmWorkDir**(*workdir*)

hexrd.fileutil.**listFiles**(*filename*)
most useful if filename contains a wildcard

hexrd.fileutil.**rmWild**(*filename*)

hexrd.fileutil.**getScratchBaseDir**()

hexrd.fileutil.**argListToStr**(*argv*)

hexrd.fileutil.**dictToDefs**(*d*)

hexrd.fileutil.**readFLT**(*fname*, *structured=False*)
Read a Fable-style .flt file

JVB 2011/03/24

1.1.3 Module: `gridutil`

7 Functions

hexrd.gridutil.**cellIndices**(*edges*, *points_1d*)
get indices in a 1-d regular grid.

edges are just that:

point: x (2.5)

edges: |1 |2 |3 |4 |5

indices: | 0 | 1 | 2 | 3 |

above the deltas are + and the index for the point is 1

point: x (2.5)

edges: |5 |4 |3 |2 |1

indices: | 0 | 1 | 2 | 3 |

here the deltas are - and the index for the point is 2

- can handle grids with +/- deltas
- be careful when using with a cyclical angular array! edges and points must be mapped to the same branch cut, and abs(edges[0] - edges[-1]) = 2*pi

`hexrd.gridutil.`**`cellConnectivity`**(*m*, *n*, *p=1*, *origin='ul'*)
p x m x n (layers x rows x cols)

origin can be upper left – 'ul' <default> or lower left – 'll'

choice will affect handedness (cw or ccw)

`hexrd.gridutil.`**`cellCentroids`**(*crd*, *con*)
con.shape = (nele, 4) crd.shape = (ncrd, 2)

con.shape = (nele, 8) crd.shape = (ncrd, 3)

`hexrd.gridutil.`**`computeArea`**(*polygon*)
must be ORDERED and CONVEX!

`hexrd.gridutil.`**`computeIntersection`**(*line1*, *line2*)
compute intersection of two-dimensional line intersection

this is an implementation of two lines:

line1 = [[x0, y0], [x1, y1]] line1 = [[x3, y3], [x4, y4]]

<http://en.wikipedia.org/wiki/Line-line_intersection>

`hexrd.gridutil.`**`isinside`**(*point*, *boundary*, *ccw=True*)
Assumes CCW boundary ordering

`hexrd.gridutil.`**`sutherlandHodgman`**(*subjectPolygon*, *clipPolygon*)

1.1.4 Module: `matrixutil`

22 Functions

`hexrd.matrixutil.`**`columnNorm`**(*a*)
normalize array of column vectors (hstacked, axis = 0)

`hexrd.matrixutil.`**`rowNorm`**(*a*)
normalize array of row vectors (vstacked, axis = 1)

`hexrd.matrixutil.`**`unitVector`**(*a*)
normalize array of column vectors (hstacked, axis = 0)

`hexrd.matrixutil.`**`nullSpace`**(*A*, *tol=1e-14*)
computes the null space of the real matrix A

`hexrd.matrixutil.`**`blockSparseOfMatArray`**(*matArray*)
Constructs a block diagonal sparse matrix (csc format) from a (p, m, n) ndarray of p (m, n) arrays

...maybe optional args to pick format type?

`hexrd.matrixutil.`**`symmToVecMV`**(*A*)
convert from symmetric matrix to Mandel-Voigt vector representation (JVB)

hexrd.matrixutil.**vecMVToSymm**(*A*)
convert from Mandel-Voigt vector to symmetric matrix representation (JVB)

hexrd.matrixutil.**vecMVCOBMatrix**(*R*)
GenerateS array of 6 x 6 basis transformation matrices for the Mandel-Voigt tensor representation in 3-D given by:

[A] = [[A_11, A_12, A_13], [A_12, A_22, A_23], [A_13, A_23, A_33]]

{A} = [A_11, A_22, A_33, sqrt(2)*A_23, sqrt(2)*A_13, sqrt(2)*A_12]

where the operation $R * A * R.T$ (in tensor notation) is obtained by the matrix-vector product [T]*{A}.

USAGE

T = vecMVCOBMatrix(R)

INPUTS

1.R is (3, 3) an ndarray representing a change of basis matrix

OUTPUTS

1.T is (6, 6), an ndarray of transformation matrices as described above

NOTES

1.Compoments of symmetric 4th-rank tensors transform in a manner analogous to symmetric 2nd-rank tensors in full matrix notation.

SEE ALSO

symmToVecMV, vecMVToSymm, quatToMat

hexrd.matrixutil.**nrmlProjOfVecMV**(*vec*)
Gives vstacked p x 6 array to To perform n' * A * n as [N]*{A} for p hstacked input 3-vectors using the Mandel-Voigt convention.

Nvec = normalProjectionOfMV(vec)

*) the input vector array need not be normalized; it is performed in place

hexrd.matrixutil.**rankOneMatrix**(*vec1*, **args*)
Create rank one matrices (dyadics) from vectors.

r1mat = rankOneMatrix(vec1) r1mat = rankOneMatrix(vec1, vec2)

vec1 is m1 x n, an array of n hstacked m1 vectors vec2 is m2 x n, (optional) another array of n hstacked m2 vectors

r1mat is n x m1 x m2, an array of n rank one matrices formed as c1*c2' from columns c1 and c2

With one argument, the second vector is taken to the same as the first.

Notes:

*) **This routine loops on the dimension m, assuming this** is much smaller than the number of points, n.

hexrd.matrixutil.**skew**(*A*)
skew-symmetric decomposition of n square (m, m) ndarrays. Result is a (squeezed) (n, m, m) ndarray

hexrd.matrixutil.**symm**(*A*)
symmetric decomposition of n square (m, m) ndarrays. Result is a (squeezed) (n, m, m) ndarray.

hexrd.matrixutil.**skewMatrixOfVector**(*w*)
given a (3, n) ndarray, w, of n hstacked axial vectors, computes the associated skew matrices and stores them in an (n, 3, 3) ndarray. Result is (3, 3) for w.shape = (3, 1) or (3,).

See also: vectorOfSkewMatrix

hexrd.matrixutil.**vectorOfSkewMatrix**(*W*)
given an (n, 3, 3) or (3, 3) ndarray, W, of n stacked 3x3 skew matrices, computes the associated axial vector(s) and stores them in an (3, n) ndarray. Result always has ndim = 2.

See also: skewMatrixOfVector

hexrd.matrixutil.**multMatArray**(*ma1*, *ma2*)
multiply two 3-d arrays of 2-d matrices

hexrd.matrixutil.**uniqueVectors**(*v*, *tol=1e-12*)
Sort vectors and discard duplicates.

USAGE:

uvec = uniqueVectors(vec, tol=1.0e-12)

v – tol – (optional) comparison tolerance

4. (a)Boyce 2010-03-18

hexrd.matrixutil.**findDuplicateVectors**(*vec*, *tol=1e-14*, *equivPM=False*)
Find vectors in an array that are equivalent to within a specified tolerance

USAGE:

eqv = DuplicateVectors(vec, *tol)

INPUT:

1.**vec is n x m, a double array of m horizontally concatenated** n-dimensional vectors.

***2) tol is 1 x 1, a scalar tolerance. If not specified, the default** tolerance is 1e-14.

*3) set equivPM to True if vec and -vec are to be treated as equivalent

OUTPUT:

1.eqv is 1 x p, a list of p equivalence relationships.

NOTES:

Each equivalence relationship is a 1 x q vector of indices that represent the locations of duplicate columns/entries in the array vec. For example:

1 2 2 2 1 2 7 |

vec = | |

2 3 5 3 2 3 3 |

eqv = [[1x2 double] [1x3 double]], where

eqv[0] = [0 4] eqv[1] = [1 3 5]

hexrd.matrixutil.**normvec**(*v*)

hexrd.matrixutil.**normvec3**(*v*)

hexrd.matrixutil.**normalized**(*v*)

hexrd.matrixutil.**cross**(*v1*, *v2*)

hexrd.matrixutil.**determinant3**(*mat*)

1.1.5 Module: orientations

8 Classes

class hexrd.orientations.**RotationParameterization**(*args*)
template for rotation parameterization class

__init__(*args*)

toMatrix()
use matrix as common representation all classes should have a constructor that works using this

** Construct [C] matrix (Kocks convention) ** {a} = [C] {a} ** sm cr

class hexrd.orientations.**RotInv**(**args*)
Bases: hexrd.orientations.RotationParameterization

rotation invariants

__init__(**args*)

class hexrd.orientations.**CanovaEuler**(**args*)
Bases: hexrd.orientations.RotationParameterization

__init__(**args*)

class hexrd.orientations.**KocksEuler**(**args*)
Bases: hexrd.orientations.RotationParameterization

Kocks Euler angles see equation 6 page 65 of koc-tom-wen-98a (Kocks, Tome, & Wenk; Texture and Anisotropy)

__init__(**args*)

toBunge()
trusting Table 1a of koc-tom-wen-98a (Kocks, Tome, & Wenk; Texture and Anisotropy)

class hexrd.orientations.**BungeEuler**(**args*)
Bases: hexrd.orientations.RotationParameterization

Bunge Euler angles see koc-tom-wen-98a (Kocks, Tome, & Wenk; Texture and Anisotropy)

__init__(**args*)

toKocks()
trusting Table 1a of koc-tom-wen-98a (Kocks, Tome, & Wenk; Texture and Anisotropy)

class hexrd.orientations.**Quat**(**args*)
Bases: hexrd.orientations.RotationParameterization

quaternions, normalized; parameterization of SO(3); do not bother making unique (is 2-to-1)

__init__(**args*)

T()
return transposed quaternion

static getRandQuat(*n=1*)
sample uniform orientation distribution to get quaternion parameters

misorAng(*other*)
compute the misorientation angle, in radians, in [0,2*pi]

normalize()
normalize the quaternion

static normalizeQuat(*q*)
normalize in place

normalized()
return normalize quaternion

transpose()
transpose the quaternion in place; retains metadata

transposed()
return transposed quaternion

class hexrd.orientations.**SymmGroup**(**args*)
symmetry group

__init__(**args*)

findDistinct(*qList*)
given a list of quaternions, find those which are distinct under the symmetry group

class hexrd.orientations.**Fiber**(*latVec*, *recipVec*)
Like John Edmiston's MakeFiber class, but with the implementation more tightly coupled to the rest of the code base

__init__(*latVec*, *recipVec*)

distBetweenFibers(*other*)
Compute the distance between two fibers using the polar decomposition of the projection operator taking one geodesic plane to the other. input: instance of MakeFiber class output: (max_eigenvalue, Rotation at max_eigenvalue), intersecting fibers would have max_eigenvalue ~ 1. Rotation at max_eigenvalue would be the 'closest' Rotation which would relate the two fibers.

25 Functions

hexrd.orientations.**arccosSafe**(*temp*)

hexrd.orientations.**orthogonalize**(*rMatIn*)

hexrd.orientations.**traceToAng**(*tr*)
given trace of a rotation matrix find the angle or rotation

hexrd.orientations.**matToCanova**(*r*)

hexrd.orientations.**invToRodr**(*inv*)
do not check for divide-by-zero

hexrd.orientations.**rodrToInv**(*rodr*)
do not check for divide-by-zero

hexrd.orientations.**rodrToQuat**(*rodr*)

hexrd.orientations.**invToM**(*theta*, *n0*, *n1*, *n2*)

hexrd.orientations.**invToQuat**(*inv*)

hexrd.orientations.**bungeToMat**(*euler*)

hexrd.orientations.**matToQuat**(*r*)

based on Spurrier's algorithm for quaternion extraction, as described in cite{sim-vuq-85a}

returns a 4-vector, not a Quat instance

BibTeX: @TechReport{sim-vuq-85a, author = {J. C. Simo and L. {Vu Quoc}}, title = {Three dimensional finite strain rod model part

{II}: computational aspects, Memorandum No. {UCB/ERL M85/31}},

institution = {Electronics Research Laboratory, College of Engineering, University of California, Berkeley},

year = {1985} }

hexrd.orientations.**matToThetaN**(*r*)

2nd order tensor => angle/axis

to see that angle is right take R in a basis so that R11=1; get tr(R) = 1 + 2 cos(theta), solve for theta and use argument about invariance of tr(R)

references: 1) box 4 in Simo and VuQuoc, 1985, ERL Berkeley memorandum no. UCB/ERL M85/31 2) Marin and Dawson 98 part 1, equation for update d_rstar (exponential mapping)

n is in vector notation of a skew tensor according to w_i = 1/2 epsilon_jik W_jk

hexrd.orientations.**quatToInv**(*q*)

hexrd.orientations.**quatToMat**(*quat*)

Take an array of n quats (numpy ndarray, 4 x n) and generate an array of rotation matrices (n x 3 x 3)

Uses the truncated series expansion for the exponential map; divide-by-zero is checked using the global 'tinyRotAng'

hexrd.orientations.**quatToProdMat**(*quat*, *mult='right'*)

Form 4 x 4 arrays to perform the quaternion product

USAGE qmats = quatToProdMat(quats, mult='right')

INPUTS

1. quats is (4, n), a numpy ndarray array of n quaternions horizontally concatenated
2. mult is a keyword arg, either 'left' or 'right', denoting the sense of the multiplication:

   / quatToProdMat(h, 'right') * q

   q * h –> < quatToProdMat(q, 'left') * h

OUTPUTS

1. qmats is (n, 4, 4), the left or right quaternion product operator

NOTES

*) **This function is intended to replace a cross-product based** routine for products of quaternions with large arrays of quaternions (e.g. applying symmetries to a large set of orientations).

hexrd.orientations.**sampleToLatticeT2**(*A_sm*, *C*)

T

[A_sm]=[C][A_lat][C]

hexrd.orientations.**latticeToSampleT2**(*A_lat*, *C*)

T

[A_sm]=[C][A_lat][C]

hexrd.orientations.**latticeToSampleV**(*V_lat*, *C*)

hexrd.orientations.**rotMatrixFromCrystalVectors**(*cvs1=None*, *cvs2=None*, *cvs3=None*)
Make a rotation matrix in the RotationParameterization convention from components of crystal vectors that are along given sample directions

hexrd.orientations.**transposeQuats**(*qList*)

hexrd.orientations.**stripQuatList**(*qList*)

hexrd.orientations.**writeQuats**(*qList*, *f*)

hexrd.orientations.**makeQuatsBall**(*qRef*, *thetaScale*, *n*)
make n quaternions in a ball around qRef, with ball size scaled by thetaScale

hexrd.orientations.**makeQuatsComponents**(*nGrain*, *scale=None*)

hexrd.orientations.**millerBravais2Normal**(*invec*, **args*)

Generate the normal(s) for a plane(s) given in the Miller-Bravais convention for the hexagonal basis {a1, a2, a3, c}. The basis for the output {o1, o2, o3} is chosen such that:

o1 || a1 o3 || c o2 = o3 ^ o1

returns a (3, n) array of horizontally concatenated unit vectors

1.1.6 Module: pfigutil

6 Functions

hexrd.pfigutil.**sph2n**(*coords_sph*)
convert from chi/eta spherical coordinates to normal vectors; can use with coords from femODF.FemHemisphere

hexrd.pfigutil.**n2sph**(*nVectors*)

hexrd.pfigutil.**n2eap**(*nVectors*, *flip=True*)
unit vectors to equal-area projection

hexrd.pfigutil.**renderEAProj**(*nVecs*, *vals*, *n*, *patch=False*, *sum=False*, *nzByContrib=True*, *northernOnly=False*)
render an equal-area projects of general pole values, on an n-by-n grid; if sum=True, then sum contributions, otherwise average them; returns a masked array

hexrd.pfigutil.**fromSouthern**(*nVecs*, *invert*)

hexrd.pfigutil.**drawLines**(*pw*, *pointLists=[]*, *netStyle=None*, *netNDiv=12*, *netAlpha=0.5*, *rMat=None*, *southern=False*, *invertFromSouthern=True*, *origin=(0.0, 0.0)*, *r=1.0*)

1.1.7 Module: plotwrap

*** For now, plotwrap is hardwired for TkAgg this is not great, but also not a high priority to fix for now; PlotWinP might be the start of a decent fix

but plotwrap was built with the idea that you could have a plot without a window, and pyplot relies on the new_figure_manager functions in the backends, which always make a window; what we need is something to make the figure and the canvas without the figure manager ... but FigureCanvasMac

... how does one get a drawable figure that is not necessarily drawn?!

4 Classes

class hexrd.plotwrap.**PlotWin**(*numRows=1*, *numCols=-1*, *title='PlotWin window'*, *figure=None*, *relfigsize=(3, 3)*, *axesList=None*, *noAutoAxesList=False*, *dpi=100*)

__init__(*numRows=1*, *numCols=-1*, *title='PlotWin window'*, *figure=None*, *relfigsize=(3, 3)*, *axesList=None*, *noAutoAxesList=False*, *dpi=100*)
If pass negative numCols, then numRows is the number of plots and the layout is done automatically

getAxes(*plotNum*, *rect=None*, *withPW=False*, ***axprops*)
careful: plotNum is from 0, not 1 as is the case for subplot axprops is useful for this like setting sharex and sharey

class hexrd.plotwrap.**PlotWinLite**(*canvas*, *figure*, *axes*)
Lightweight PlotWin substitute for when windows are being controlled by code outside of plotwrap

__init__(*canvas*, *figure*, *axes*)

haveXLabels()
may want to turn off any functionality in this method

haveYLabels()
may want to turn off any functionality in this method

class hexrd.plotwrap.**PlotWinP**(*axes=None*, *as3D=False*, *title=None*, ***kwargs*)
Just wrap pyplot

__init__(*axes=None*, *as3D=False*, *title=None*, ***kwargs*)

haveXLabels()
may want to turn off any functionality in this method

haveYLabels()
may want to turn off any functionality in this method

pwList = None
for now, punt on setting a title

class hexrd.plotwrap.**PlotWrap**(***keyArgs*)
Bases: object

__init__(***keyArgs*)

colorbar(*rect=(0.8, 0.1, 0.05, 0.8)*, *adjustPos=True*, *thing=None*, ***kwargs*)
if set rect to None, then colorbar steals self.a

destroy()
does not clean up self.a, just kills the window if this plot owns the window

ownCanvas = None
checking self.showByDefault here causes trouble because it is hard to attach a figure to a window later for a general backend ***

save(***keyArgs*)
make hardcopy of the current figure; specify filename or prefix keyword argument

6 Functions

hexrd.plotwrap.**autoTicks**(*x*, *n*, *format='%0.2e'*)

hexrd.plotwrap.**argToPW**(*arg*)

hexrd.plotwrap.**hist2D**(*xVals*, *yVals*, *bins*, *hRange=None*, *weights=None*, ***kwArgs*)
Plot 2D histogram of data yVals versus xVals, with number of bins given by bins (int or 2-tuple)

hexrd.plotwrap.**plotHist2D**(*xedges*, *yedges*, *H*, *hRange=None*, *logScale=False*, *minCount=1*, *win=None*, *xlabel=None*, *ylabel=None*, *xformat=None*, *yformat=None*, *nXTics=0*, *nYTics=0*, *winArgs={}*)
winArgs can include things like title, relfigsize, dpi

hexrd.plotwrap.**makeHist2D**(*xVals*, *yVals*, *bins*, *hRange=None*, *weights=None*)

hexrd.plotwrap.**main**()

1.1.8 Module: quadrature.q1db

7 Functions

hexrd.quadrature.q1db.**qloc1**()

hexrd.quadrature.q1db.**qloc2**()

hexrd.quadrature.q1db.**qloc3**()

hexrd.quadrature.q1db.**qloc4**()

hexrd.quadrature.q1db.**qloc5**()

hexrd.quadrature.q1db.**qloc8**()

hexrd.quadrature.q1db.**qLoc**(*quadr*, *promote=False*)

1.1.9 Module: quadrature.q2db

5 Functions

hexrd.quadrature.q2db.**qloc1**()

hexrd.quadrature.q2db.**qloc4**()

hexrd.quadrature.q2db.**qloc9**()

hexrd.quadrature.q2db.**qLocFrom1D**(*quadr1d*)
product of 1d quadrature rules; given accuracy may be available with fewer quadrature points using a native 3D rule

hexrd.quadrature.q2db.**qLoc**(*quadr*)

1.1.10 Module: quadrature.q3db

5 Functions

hexrd.quadrature.q3db.**qloc1**()

hexrd.quadrature.q3db.**qloc8**()

hexrd.quadrature.q3db.**qloc27**()
3x3x3 quadrature, product of qloc1d03 in three directions

hexrd.quadrature.q3db.**qLocFrom1D**(*quadr1d*)
product of 1d quadrature rules; given accuracy may be available with fewer quadrature points using a native 3D rule

hexrd.quadrature.q3db.**qLoc**(*quadr*)

1.1.11 Module: tens

4 Classes

class hexrd.tens.**T2Symm**(*args*)
template for symmetric second order tensor components

__init__(*args*)

class hexrd.tens.**T2Vecds**(*vecds*)
Bases: hexrd.tens.T2Symm

__init__(*vecds*)

class hexrd.tens.**T2Svec**(*val*)
Bases: hexrd.tens.T2Symm

__init__(*val*)

class hexrd.tens.**T2SvecP**(*svecp*)
Bases: hexrd.tens.T2Symm

__init__(*svecp*)

23 Functions

hexrd.tens.**vecdvToVecds**(*vecdv*)
convert from [t1,...,t5,v] to vecds[:] representation, where v is the relative volume

hexrd.tens.**vecdsToSymm**(*vecds*)
convert from vecds representation to symmetry matrix

hexrd.tens.**traceToVecdsS**(*Akk*)

hexrd.tens.**vecdsSToTrace**(*vecdsS*)

hexrd.tens.**trace3**(*A*)

hexrd.tens.**symmToVecds**(*A*)
convert from symmetry matrix to vecds representation

hexrd.tens.**matxToSkew**(*A*)

hexrd.tens.**skewOfMatx**(*A*)

hexrd.tens.**matxToSymm**(*A*)

hexrd.tens.**symmOfMatx**(*A*)

hexrd.tens.**symmToMVvec**(*A*)
convert from symmetric matrix to Mandel-Voigt vector representation (JVB)

hexrd.tens.**MVvecToSymm**(*A*)
convert from Mandel-Voigt vector to symmetric matrix representation (JVB)

hexrd.tens.**MVCOBMatrix**(*R*)

GenerateS array of 6 x 6 basis transformation matrices for the Mandel-Voigt tensor representation in 3-D given by:

[A] = [[A_11, A_12, A_13],

[A_12, A_22, A_23], [A_13, A_23, A_33]]

V

{A} = [A_11, A_22, A_33, sqrt(2)*A_23, sqrt(2)*A_13, sqrt(2)*A_12]

where the operation R * A *R.T (in tensor notation) is obtained by the matrix-vector product [T]*{A}.

USAGE

T = MVCOBMatrix(R)

INPUTS

1.R is (3, 3) an ndarray representing a change of basis matrix

OUTPUTS

1.T is (6, 6), an ndarray of transformation matrices as described above

NOTES

1.Compoments of symmetric 4th-rank tensors transform in a manner analogous to symmetric 2nd-rank tensors in full matrix notation.

SEE ALSO

symmToMVvec, quatToMat

hexrd.tens.**NormalProjectionOfMV**(*vec*)

hexrd.tens.**svecToVecds**(*svec*)

convert from svec to vecds representation

hexrd.tens.**symmPlusI**(*Ain*)

add the identity to a symmetric matrix

hexrd.tens.**svecpToSvec**(*svecp*)

hexrd.tens.**symmToSvec**(*symm*)

hexrd.tens.**matxToSvec**(*matx*)

hexrd.tens.**svecToMatx**(*svec*)

hexrd.tens.**svecToSymm**(*svec*)

hexrd.tens.**dAiAoH_svecList**(*aInv*)

derivative of inverse of symmetric matrix wrt svec components of that matrix; aInv is the inverse of the matrix

hexrd.tens.**svecToSvecP**(*svec*)

1.1.12 Module: `valunits`

Module for associating units with scalar quantities

This module has been modified from its original form by removing the call to the "units" executable and restricting the units to only those used by the heXRD package.

2 Classes

class `hexrd.valunits.`**`UNames`**
Bases: `object`

Units used in this module

class `hexrd.valunits.`**`valWUnit`**(*name*, *unitType*, *value*, *unit*)
Value with units

`__init__`(*name*, *unitType*, *value*, *unit*)
Initialization

INPUTS name

(str) name of the item

unitType (str) class of units, e.g. 'length', 'angle'

value (float) numerical value

unit (str) name of unit

`getVal`(*toUnit*)
Return value in requested units

INPUTS

toUnit (str) requested unit for output

`isAngle`()
Return true if quantity is an angle

`isEnergy`()
Return true if quantity is an energy

`isLength`()
Return true if quantity is a length

2 Functions

`hexrd.valunits.`**`toFloat`**(*val*, *unitName*)
Return the raw value of the object

INPUTS

val (floatlvalWUnit) object with value

unitName (str) name of unit

This function returns the raw value of the object, ignoring the unit, if it is numeric or converts it to the requested units and returns the magnitude if it is a valWUnit instance.

For example:

```
>>> print toFloat(1.1, 'radians')
1.1
>>> v = valWUnit('vee', 'angle', 1.1, 'radians')
>>> print toFloat(v, 'degrees')
63.0253574644
```

hexrd.valunits.**valWithDflt**(*val*, *dflt*, *toUnit=None*)
Return value or default value

1.1.13 Module: xrd.crystallography

1 Class

class hexrd.xrd.crystallography.**PlaneData**(*hkls*, **args*, ***kwargs*)
Bases: object

Careful with ordering: Outputs are ordered by the 2-theta for the hkl unless you get self.__hkls directly, and this order can change with changes in lattice parameters (lparms); setting and getting exclusions works on the current hkl ordering, not the original ordering (in self.__hkls), but exclusions are stored in the original ordering in case the hkl ordering does change with lattice parameters

if not None, tThWidth takes priority over strainMag in setting two-theta ranges; changing strainMag automatically turns off tThWidth

__init__(*hkls*, **args*, ***kwargs*)

getDD_tThs_lparms()
derivatives of tThs with respect to lattice parameters; have not yet done coding for analytic derivatives, just wimp out and finite difference

getHKLID(*hkl*)
can call on a single hkl or list of hkls

getHKLs(*asStr=False*, *thisTTh=None*, *allHKLs=False*)
if pass thisTTh, then only return hkls overlapping the specified 2-theta; if set allHKLs to true, the ignore exlcusions, tThMax, etc

getLatticeOperators()
gets lattice vector operators as a new (deepcopy)

getLatticeType()
This is the lattice type

getLaueGroup()
This is the Schoenflies tag

getNhklRef()
does not use exclusions or the like

getPhaseID()
may return None if not set

getPlaneNormals()
gets both +(hkl) and -(hkl) normals

getPlaneSpacings()
gets plane spacings

getSymHKLs(*asStr=False*, *indices=None*)
new function that returns all symmetric hkls

getTThRanges (*strainMag=None*, *lparms=None*)
Return 2-theta ranges for included hkls

return array is n x 2

get_hkls ()
do not do return self.__hkls, as everywhere else hkls are returned in 2-theta order; transpose is to comply with lparm convention

hkls
do not do return self.__hkls, as everywhere else hkls are returned in 2-theta order; transpose is to comply with lparm convention

latVecOps
gets lattice vector operators as a new (deepcopy)

static makePlaneData (*hkls*, *lparms*, *qsym*, *symmGroup*, *strainMag*, *wavelength*)
hkls : need to work with crystallography.latticePlanes lparms : need to work with crystallography.latticePlanes laueGroup : see symmetry module wavelength : wavelength strainMag : swag of strian magnitudes

static makeScatteringVectors (*hkls*, *rMat_c*, *bMat*, *wavelength*, *chiTilt=None*)
modeled after QFromU.m

13 Functions

hexrd.xrd.crystallography.**hklToStr** (*x*)

hexrd.xrd.crystallography.**tempSetOutputDegrees** (*val*)

hexrd.xrd.crystallography.**revertOutputDegrees** ()

hexrd.xrd.crystallography.**cosineXform** (*a*, *b*, *c*)
Spherical trig transform to take alpha, beta, gamma to expressions for cos(alpha*). See ref below.

[1] R. J. Neustadt, F. W. Cagle, Jr., and J. Waser, ''Vector algebra and the relations between direct and reciprocal lattice quantities''. Acta Cryst. (1968), A24, 247–248

hexrd.xrd.crystallography.**processWavelength** (*arg*)
Convert an energy value to a wavelength. If argument has units of length or energy, will convert to globally specified unit type for wavelength (dUnit). If argument is a scalar, assumed input units are keV.

hexrd.xrd.crystallography.**latticeParameters** (*lvec*)
Generates direct and reciprocal lattice vector components in a crystal-relative RHON basis, X. The convention for fixing X to the lattice is such that a ∥ x1 and c* ∥ x3, where a and c* are direct and reciprocal lattice vectors, respectively.

hexrd.xrd.crystallography.**latticePlanes** (*hkls*, *lparms*, *ltype='cubic'*, *wavelength=1.54059292*, *strainMag=None*)
Generates lattice plane data in the direct lattice for a given set of Miller indices. Vector components are written in the crystal-relative RHON basis, X. The convention for fixing X to the lattice is such that a ∥ x1 and c* ∥ x3, where a and c* are direct and reciprocal lattice vectors, respectively.

USAGE:

planeInfo = latticePlanes(hkls, lparms, **kwargs)

INPUTS:

1. hkls (3 x n float ndarray) is the array of Miller indices for the planes of interest. The vectors are assumed to be concatenated along the 1-axis (horizontal).

2.lparms (1 x m float list) is the array of lattice parameters, where m depends on the symmetry group (see below).

3.The following optional keyword arguments are recognized:

*) **ltype=(string) is a string representing the symmetry type of** the implied Laue group. The 11 available choices are shown below. The default value is 'cubic'. Note that each group expects a lattice parameter array of the indicated length and order.

latticeType lparms ——— ——— 'cubic' a 'hexagonal' a, c 'trigonal' a, c 'rhombohedral' a, alpha (in degrees) 'tetragonal' a, c 'orthorhombic' a, b, c 'monoclinic' a, b, c, beta (in degrees) 'triclinic' a, b, c, alpha, beta, gamma (in degrees)

*) **wavelength=<float> is a value represented the wavelength in** Angstroms to calculate bragg angles for. The default value is for Cu K-alpha radiation (1.54059292 Angstrom)

*) strainMag=None

OUTPUTS:

1.planeInfo is a dictionary containing the following keys/items:

normals (3, n) double array array of the components to the unit normals for each {hkl} in X (horizontally concatenated)

dspacings (n,) double array array of the d-spacings for each {hkl}

2thetas (n,) double array array of the Bragg angles for each {hkl} relative to the specified wavelength

NOTES:

*) **This function is effectively a wrapper to 'latticeVectors'.** See 'help(latticeVectors)' for additional info.

*) **Lattice plane d-spacings are calculated from the reciprocal** lattice vectors specified by {hkl} as shown in Appendix 1 of [1].

REFERENCES:

[1] B. D. Cullity, ''Elements of X-Ray Diffraction, 2 ed.''. Addison-Wesley Publishing Company, Inc., 1978. ISBN 0-201-01174-3

hexrd.xrd.crystallography.**latticeVectors**(*lparms*, *tag='cubic'*, *radians=False*, *debug=False*)

Generates direct and reciprocal lattice vector components in a crystal-relative RHON basis, X. The convention for fixing X to the lattice is such that a ∥ x1 and c* ∥ x3, where a and c* are direct and reciprocal lattice vectors, respectively.

USAGE:

lattice = LatticeVectors(lparms, <symmTag>)

INPUTS:

1.lparms (1 x n float list) is the array of lattice parameters, where n depends on the symmetry group (see below).

2.symTag (string) is a case-insensitive string representing the symmetry type of the implied Laue group. The 11 available choices are shown below. The default value is 'cubic'. Note that each group expects a lattice parameter array of the indicated length and order.

latticeType lparms ——— ——— 'cubic' a 'hexagonal' a, c 'trigonal' a, c 'rhombohedral' a, alpha (in degrees) 'tetragonal' a, c 'orthorhombic' a, b, c 'monoclinic' a, b, c, beta (in degrees) 'triclinic' a, b, c, alpha, beta, gamma (in degrees)

OUTPUTS:

1.lattice is a dictionary containing the following keys/items:

F (3, 3) double array transformation matrix taking componenents in the direct lattice (i.e. {uvw}) to the reference, X

B (3, 3) double array transformation matrix taking componenents in the reciprocal lattice (i.e. {hkl}) to X

BR (3, 3) double array transformation matrix taking componenents in the reciprocal lattice to the Fable reference frame (see notes)

U0 (3, 3) double array transformation matrix (orthogonal) taking componenents in the Fable reference frame to X

vol double the unit cell volume

dparms (6,) double list the direct lattice parameters: [a b c alpha beta gamma]

rparms (6,) double list the reciprocal lattice parameters: [a* b* c* alpha* beta* gamma*]

NOTES:

*) **The conventions used for assigning a RHON basis,** X -> {x1, x2, x3}, to each point group are consistent with those published in Appendix B of [1]. Namely: a ∥ x1 and c* ∥ x3. This differs from the convention chosen by the Fable group, where a* ∥ x1 and c ∥ x3 [2].

*) **The unit cell angles are defined as follows:** alpha=acos(b'*c/|b||c|), beta=acos(c'*a/|c||a|), and gamma=acos(a'*b/|a||b|).

*) **The reciprocal lattice vectors are calculated using the** crystallographic convention, where the prefactor of 2*pi is omitted. In this convention, the reciprocal lattice volume is 1/V.

*) **Several relations from [3] were employed in the component** calculations.

REFERENCES:

[1] J. F. Nye, ‘‘Physical Properties of Crystals: Their Representation by Tensors and Matrices’‘. Oxford University Press, 1985. ISBN 0198511655

[2] E. M. Lauridsen, S. Schmidt, R. M. Suter, and H. F. Poulsen, ‘‘Tracking: a method for structural characterization of grains in powders or polycrystals’‘. J. Appl. Cryst. (2001). 34, 744–750

[3] R. J. Neustadt, F. W. Cagle, Jr., and J. Waser, ‘‘Vector algebra and the relations between direct and reciprocal lattice quantities’‘. Acta Cryst. (1968), A24, 247–248

`hexrd.xrd.crystallography.`**`hexagonalIndicesFromRhombohedral`**(*hkl*)
converts rhombohedral hkl to hexagonal indices

`hexrd.xrd.crystallography.`**`rhombohedralIndicesFromHexagonal`**(*HKL*)
converts hexagonal hkl to rhombohedral indices

`hexrd.xrd.crystallography.`**`rhombohedralParametersFromHexagonal`**(*a_h*, *c_h*)
converts hexagonal lattice parameters (a, c) to rhombohedral lattice parameters (a, alpha)

`hexrd.xrd.crystallography.`**`getFriedelPair`**(*tth0*, *eta0*, **ome0*, ***kwargs*)
Get the diffractometer angular coordinates in degrees for the Friedel pair of a given reflection (min angular distance).

AUTHORS:

10. (a)Bernier – 10 Nov 2009

USAGE:

ome1, eta1 = getFriedelPair(tth0, eta0, *ome0, display=False, units='degrees', convention='hexrd')

INPUTS:

1. tth0 is a list (or ndarray) of 1 or n the bragg angles (2theta) for the n reflections (tiled to match eta0 if only 1 is given).
2. eta0 is a list (or ndarray) of 1 or n azimuthal coordinates for the n reflections (tiled to match tth0 if only 1 is given).
3. ome0 is a list (or ndarray) of 1 or n reference oscillation angles for the n reflections (denoted omega in [1]). This argument is optional.
4. Keyword arguments may be one of the following:

Keyword Values|{default} Action ———— ———— ———— 'display' True|{False} toggles display info to cmd line 'units' 'radians'|{'degrees'} sets units for input angles 'convention' 'fable'|{'hexrd'} sets conventions defining

the angles (see below)

'chiTilt' None the inclination (about Xlab) of the oscillation axis

OUTPUTS:

1. ome1 contains the oscialltion angle coordinates of the Friedel pairs associated with the n input reflections, relative to ome0 (i.e. ome1 = <result> + ome0). Output is in DEGREES!
2. eta1 contains the azimuthal coordinates of the Friedel pairs associated with the n input reflections. Output units are controlled via the module variable 'outputDegrees'

NOTES:

JVB) The ouputs ome1, eta1 are written using the selected convention, but the units alway degrees. May change this to work with Nathan's global...

JVB) In the 'fable' convention [1], {XYZ} form a RHON basis where X is downstream, Z is vertical, and eta is CCW with +Z defining eta = 0.

JVB) In the 'hexrd' convention [2], {XYZ} form a RHON basis where Z is upstream, Y is vertical, and eta is CCW with +X defining eta = 0.

REFERENCES:

[1] E. M. Lauridsen, S. Schmidt, R. M. Suter, and H. F. Poulsen, ''Tracking: a method for structural characterization of grains in powders or polycrystals''. J. Appl. Cryst. (2001). 34, 744–750

[2] J. V. Bernier, M. P. Miller, J. -S. Park, and U. Lienert, ''Quantitative Stress Analysis of Recrystallized OFHC Cu Subject to Deformed In Situ'', J. Eng. Mater. Technol. (2008). 130. DOI:10.1115/1.2870234

`hexrd.xrd.crystallography.`**`getDparms`**(*lp*, *lpTag*, *radians=True*)
Utility routine for getting dparms, that is the lattice parameters without symmetry – 'triclinic'

1.1.14 Module: `xrd.detector`

24 Classes

class `hexrd.xrd.detector.`**`FmtCoordIdeal`**(*planeData*, *workDist*)

`__init__`(*planeData*, *workDist*)

class `hexrd.xrd.detector.`**`ThreadReadFrame`**(*img*, *readArgs*, *castArgs*)
Bases: `threading.Thread`

__init__(*img*, *readArgs*, *castArgs*)

class hexrd.xrd.detector.**Framer2DRC**(*ncols*, *nrows*, *dtypeDefault='int16'*, *dtypeRead='uint16'*, *dtypeFloat='float64'*)

Bases: object

Base class for readers.

You can make an instance of this class and use it for most of the things a reader would do, other than actually reading frames

__init__(*ncols*, *nrows*, *dtypeDefault='int16'*, *dtypeRead='uint16'*, *dtypeFloat='float64'*)

getDark()
needed in findSpotsOmegaStack

getDeltaOmega()
needed in findSpotsOmegaStack

getEmptyMask()
convenience method for getting an emtpy mask or bin frame

getFrameOmega(*iFrame=None*)
needed in findSpotsOmegaStack

getNFrames()
number of total frames with real data, not number remaining needed in findSpotsOmegaStack

classmethod maxVal(*dtypeRead*)
maximum value that can be stored in the image pixel data type; redefine as desired

read(*nskip=0*, *nframes=1*, *sumImg=False*)
needed in findSpotsOmegaStack

class hexrd.xrd.detector.**FrameWriter**(**args*, ***kwargs*)

Bases: hexrd.xrd.detector.Framer2DRC

__init__(**args*, ***kwargs*)

class hexrd.xrd.detector.**ReadGeneric**(*filename*, *ncols*, *nrows*, **args*, ***kwargs*)

Bases: hexrd.xrd.detector.Framer2DRC

may eventually want ReadGE to inherit from this, or pull common things off to a base class

__init__(*filename*, *ncols*, *nrows*, **args*, ***kwargs*)

getDark()
no dark yet supported

getFrameOmega(*iFrame=None*)
if iFrame is none, use internal counter

makeNew()
return a clean instance for the same data files useful if want to start reading from the beginning

read(*nskip=0*, *nframes=1*, *sumImg=False*)
sumImg can be set to True or to something like numpy.maximum

subtractDark = None
keep things for makeNew convenience

class hexrd.xrd.detector.**ReadGE**(*fileInfo*, **args*, ***kwargs*)

Bases: hexrd.xrd.detector.Framer2DRC

Read in raw GE files; this is the class version of the foregoing functions

NOTES

*) **The flip axis ('v'ertical) was verified on 06 March 2009 by** JVB and UL. This should be rechecked if the configuration of the GE changes or you are unsure.

*) **BE CAREFUL! nframes should be < 10 or so, or you will run out of** memory in the namespace on a typical machine.

*) The header is currently ignored

*) **If a dark is specified, this overrides the use of empty frames as** background; dark can be a file name or frame

*) **In multiframe images where background subtraction is requested but no** dark is specified, attempts to use the empty frame(s). An error is returned if there are not any specified. If there are multiple empty frames, the average is used.

__init__(*fileInfo*, **args*, ***kwargs*)
meant for reading a series of frames from an omega sweep, with fixed delta-omega for each frame

omegaStart and omegaDelta can follow fileInfo or be specified in whatever order by keyword

fileInfo: string, (string, nempty), or list of (string, nempty) for multiple files

for multiple files and no dark, dark is formed only from empty frames in the first file

classmethod display(*thisframe*, *roi=None*, *pw=None*, ***kwargs*)
this is a bit ugly in that it sidesteps the dtypeRead property

getFrameOmega(*iFrame=None*)
if iFrame is none, use internal counter

getFrameUseMask()
this is an optional toggle to turn the mask on/off

getNFrames()
number of total frames with real data, not number remaining

indicesToMask(*indices*)
Indices can be a list of indices, as from makeIndicesTThRanges

makeNew()
return a clean instance for the same data files useful if want to start reading from the beginning

classmethod maxVal(*dummy*)
maximum value that can be stored in the image pixel data type

rawRead(**args*, ***kwargs*)
wrapper around readRaw that does the same flipping as the reader instance from which it is called

read(*nskip=0*, *nframes=1*, *sumImg=False*, *mask=None*)
sumImg can be set to True or to something like numpy.maximum

readBBox(*bbox*, *raw=True*, *doFlip=None*)
with raw=True, read more or less raw data, with bbox = [(iLo,iHi),(jLo,jHi),(fLo,fHi)]

careful: if raw is True, must set doFlip if want frames potentially flipped; can set it to a reader instance to pull the doFlip value from that instance

classmethod readDark(*darkFile*, *nframes=1*)
dark subtraction is done before flipping, so do not flip when reading either

classmethod readRaw(*fname*, *mode='raw'*, *headerlen=0*)
read a raw binary file; if specified, headerlen is in bytes; does not do any flipping

useThreading
turn threading on or off

class hexrd.xrd.detector.**ReadMar165**(*mode*)
Bases: hexrd.xrd.detector.Framer2DRC

placeholder; not yet really implemented

__init__(*mode*)

class hexrd.xrd.detector.**ReadMar165NB1**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.ReadMar165

__init__(**args*, ***kwargs*)

class hexrd.xrd.detector.**ReadMar165NB2**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.ReadMar165

__init__(**args*, ***kwargs*)

class hexrd.xrd.detector.**ReadMar165NB3**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.ReadMar165

__init__(**args*, ***kwargs*)

class hexrd.xrd.detector.**ReadMar165NB4**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.ReadMar165

__init__(**args*, ***kwargs*)

class hexrd.xrd.detector.**LineStyles**(*lt=None*)
do not want to just cycle through default plot line colors, as end up with black lines

__init__(*lt=None*)

class hexrd.xrd.detector.**Peak1DAtLoc**(*centers*, *xVecDflt=None*)
base class for 1D peak shapes at fixed location; fixed that is unless newCenter is passed to the __call__ method

__init__(*centers*, *xVecDflt=None*)
If __init__ is called with a list, then put one peak at each location

d_dCenters(*xVec*, *p*)
derivative of call with respect to centers

d_dp(*xVec*, *p*)
derivative of call with respect to p assuming each eval depends only on its own point!

d_dx(*xVec*, *p*)
derivative of call with respect to xVec

eval(*xVec*, *p*)
xVec is parameters, p is positions

fitFloatingCenter(*tThVals*, *intensityVals*, *xVecGuess=None*, *centersGuess=None*, *weights=4*, *tThWidth=None*, *fitGoodnessTol=0.5*)
Note that centers are kept as they are – if you want to actually change the centers of the function you need to call setCenters(cFit) after calling this function

class hexrd.xrd.detector.**PeakPV1DAtLoc**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Peak1DAtLoc

the pseudo-Voigt: f = A*(n*fl + (1 - n)*fg)

__init__(**args*, ***kwargs*)

d_dx(*xVec*, *p*)
allow p to be general shape

class hexrd.xrd.detector.**PeakLorentzian1DAtLoc**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Peak1DAtLoc

__init__(**args*, ***kwargs*)

d_dx(*xVec*, *p*)
allow p to be general shape

getNParams()
2 parameters for background, 2 for intensity and width of each peak

class hexrd.xrd.detector.**PeakGauss1DAtLoc**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Peak1DAtLoc

__init__(**args*, ***kwargs*)

d_dCenters(*xVec*, *p*)
allow p to be general shape

d_dx(*xVec*, *p*)
allow p to be general shape

getNParams()
2 parameters for background, 2 for intensity and width of each peak

class hexrd.xrd.detector.**MultiRingBinned**(*detectorGeom*, *planeData*, *dataFrame*, *funcType='pv'*, *refineParamsDG=True*, *refineParamsL=False*, *targetNRho=None*, *polarRebinKWArgs={}*, *quadr=4*, *npdivMax=8*, *samplingFactor=1*, *singleRebin=True*, *distortionFreeRefDG=False*, *log=None*)
like MultiRingEval, but works with polar rebinned (or 'caked') images

no funcXVecList to init because expectation is that always pulled from dataFrame

should work fine whether or not corrected is True in polarRebinKWArgs; but note that default is changed to True

if etaRange is not specified in polarRebinKWArgs (or is None), then the etaRange is calculated based on numEta so that the first eta bin is centered around an angle of zero

note that this works by matching intesities for binned 1D functions in a least sqaures problem; one could probably instead form a residual on the two-theta values of the image frame positions for the peak centers found with independtly floating centers (on the output of the getTThErrors method)

KEYWORD ARGS

funcType = funcTypeDflt, refineParamsDG = True, refineParamsL = False, targetNRho = 30, polarRebinKWArgs = { }, quadr = 4, npdivMax = 4, samplingFactor = 0.25, singleRebin = True, distortionFreeRefDG = False, log=None: if not None, then a file-like object with a "write" method;

__init__(*detectorGeom*, *planeData*, *dataFrame*, *funcType='pv'*, *refineParamsDG=True*, *refineParamsL=False*, *targetNRho=None*, *polarRebinKWArgs={}*, *quadr=4*, *npdivMax=8*, *samplingFactor=1*, *singleRebin=True*, *distortionFreeRefDG=False*, *log=None*)

doFit(*xVec0=None*, ***lsKWArgs*)
lsKWArgs can have things like ftol and xtol for leastsq

eval(*xVec*)
careful: this updates the settings in detectorGeom and planeData

floatingCentersIJ = None
make a reference detector geom

getTThErrors (*plot=False*, *units='strain'*, *outputFile=None*)
convenient way of looking at the errors, though not how the errors are actually measured in the fitting procedure; get the tTh values at the image frame locations deemed to be the centers with the floating-center fits

units can be: 'mm' <radius>, 'd-spacing', 'strain', 'radians' <tTh>, 'degrees' <tTh>

logfile
file for log messages

plotByRingEta (*iRingSet*, *iEta*, *win=None*, *sqrtIntensity=True*, *alpha=0.25*)
may have redundant work here, but assume this is not a big deal if doing plots

polImg = None
tth values for figuring out where the rings fall

prbkw = None
things from the user

rhoPxRange = None
and now compute number of rho bins across all ring sets

ticMethod = None
defaults which want different from those for polarRebin's defaults

wQP = None
leastsq to do fit, with floating center

class hexrd.xrd.detector.**MultiRingEval** (*detectorGeom*, *planeData*, *indicesList=None*, *iHKLLists=None*, *dataFrame=None*, *funcType='pv'*, *refineParamsDG=True*, *refineParamsL=False*, *funcXVecList=None*, *copyFrame=False*, *quadr=3*)
For working with data as rings, particularly for fitting detector geometry or lattice parameters.

__init__ (*detectorGeom*, *planeData*, *indicesList=None*, *iHKLLists=None*, *dataFrame=None*, *funcType='pv'*, *refineParamsDG=True*, *refineParamsL=False*, *funcXVecList=None*, *copyFrame=False*, *quadr=3*)
Mostly meant for use with DetectorGeomGE.fit

If funcXVecList is passed, then entries in this list are used for peak function forms, and these peak function forms do not appear in the degrees of freedom

Note that ranges for 2-thetas from planeData need to be such that rings are adequately covered

Can optionally pass indicesList and iHKLLists if they are already handy

if copyFrame is True, then data in dataFrame is copied

deval (*xVec*)
useful to pass, for example, as Dfun to leastsq; bit of a misnomer in that deval is the derivative of __call__, not eval

doFit (*xVec0=None*, ***lsKWArgs*)
lsKWArgs can have things like ftol and xtol for leastsq

eval (*xVec*, *thisframe=None*)
if thisframe is passed, the put values on the frame

jQP = None
do not worry about dvQP kinds of contributions for now

radialFitXVec(*dataFrame=None*, *plot=False*, *plotTitlePrefix=''*, *quadr1d=None*)
if dataFrame is not provided, use self.dataFrame

if quadr1d is not specified, use quadr specified in init

radialPlotData(*dataFrame=None*, *plotTitlePrefix=''*)
for simple radial plotting, useful if other things are mysteriously breaking

setFuncXVecList(*funcXVecList*)
only okay if funcXVecList set on init

class hexrd.xrd.detector.**DetectorBase**(*reader*)
Bases: object

base class for a detector

__init__(*reader*)

getPVecScaling()
scaling, suitable for scaling perturbations for finite differencing

getParamScalings()
scalings, suitable for scaling perturbations for finite differencing

class hexrd.xrd.detector.**Detector2DRC**(*ncols*, *nrows*, *pixelPitch*, *vFactorUnc*, *vDark*, *reader*, **args*, ***kwargs*)
Bases: hexrd.xrd.detector.DetectorBase

base class for 2D row-column detectors

__init__(*ncols*, *nrows*, *pixelPitch*, *vFactorUnc*, *vDark*, *reader*, **args*, ***kwargs*)

angOnDetector(*tTh*, *eta*, **args*)
note: returns a scalar if tTh and eta have single entries

angToXYO(*x0*, *y0*, **args*, ***kwargs*)
convert Cartesian to polar

uses blocking to call vectorized version

angToXYOBBox(**args*, ***kwargs*)
given either angBBox or angCOM (angular center) and angPM (+-values), compute the bounding box on the image frame

if forSlice=True, then returned bbox is appropriate for use in array slicing

if reader or omegas is passed, then convert from omegas to frames; and if doWrap=True, then frames may be a list for an omega range that spans the branch cut

angToXYO_V(*tth*, *eta_l*, **args*, ***kwargs*)
opposite of xyoToAng

cartesianCoordsOfPixelIndices(*row*, *col*, *ROI=None*)
converts [i, j] pixel array indices to cartesian spatial coords where the lower left corner of the image is (0, 0)

Output units are in the pixel pitch units (see self.pixelPitch)

Will optionally take the upper left-hand corner (min row, min col) of a ROI when dealing with subregions on the detector as in when zooming in on diffraction spots...

*) explicitly enforce this to be self-consistent with radial distortion correction, etc...

display(*thisframe*, *planeData=None*, ***kwargs*)
wrap reader display method; display coordinates as 2-theta and eta given that self knows how to do this

if pass planeData, then it is used to list HKLs overlapping the given 2-theta location

...*** option for drawing lab-frame glyph

displayIdeal (*thisframe*, *planeData=None*, *workDist=None*, *nlump=None*, ***kwargs*)
render and display frame on ideal detector plane; if workDist is not specified, then use self.workDist

drawRings (*drawOn*, *planeData*, *withRanges=False*, *legendLoc=(0.05*, *0.5)*, *legendMaxLen=10*, *ideal=None*, *range=None*, *lineType=None*, *lineWidth=1.0*)
If drawOn is a PlotWrap instance, draw on the existing instance, otherwise pass drawOn to display and return the resulting PlotWrap instance

planeData.exclusions can be used to work with a subset of rings;

set legendLoc to None or False to skip making the legend

removes any existing lines in the axes

if pass ideal, then display rings on an ideal detector with the working distance taken from the value of the ideal argument

drawRingsGUI (*thisframe*, *planeData*, *displayKWArgs={}*, *sliderRangeFactor=1.0*, *funcType='pv'*)
a simple GUI

getAngPixelSize (*xyo*, *delta_omega*)
get pixel size in angular coordinates at a given cartesian coordinate position

getPRBOverlay (*polarRebinKWArgs*)
Return plottable coordinates of rebinning sector.

Takes in dictionary of keyword args for polarRebin

for etas, right now assumes stopEta > startEta, CCW

getParamScalings ()
scalings, suitable for scaling perturbations for finite differencing

getRings (*planeData*, *ranges=False*)
Return a list of rings for the given hkls

Already filters on the exclusions.

getVScale (*vThese*)
get scale factors for use in uncertainty quantification

makeIndicesTThRanges (*planeData*, *cullDupl=False*)
return a list of indices for sets of overlaping two-theta ranges; to plot, can do something like:

mask = self.reader.getEmptyMask()

mask[indices] = True

With cullDupl set true, eliminate HKLs with duplicate 2-thetas

makeMaskTThRanges (*planeData*)
Mask in the sense that reader with the mask will exclude all else

pixelIndicesOfCartesianCoords (*x*, *y*, *ROI=None*)
converts [i, j] pixel array indices to cartesian spatial coords where the lower left corner of the image is (0, 0)

Output units are in the pixel pitch units (see self.pixelPitch)

Will optionally take the upper left-hand corner (min row, min col) of a ROI when dealing with subregions on the detector as in when zooming in on diffraction spots...

*) explicitly enforce this to be self-consistent with radial distortion correction, etc...

polarRebin(*thisFrame, npdiv=2, rhoRange=[100, 1000], numRho=1200, etaRange=array([-0.08726646, 6.19591884]), numEta=36, ROI=None, corrected=False, verbose=True, log=None*)
Caking algorithm

INPUTS

thisFrame npdiv=2, pixel subdivision (n x n) to determine bin membership rhoRange=[100, 1000] - radial range in pixels numRho=1200 - number of radial bins etaRange=num.pi*num.r_[-5, 355]/180. – range of eta numEta=36 - number of eta subdivisions ROI=None - region of interest (four vector) corrected=False - uses 2-theta instead of rho verbose=True,

renderIdeal(*thisframe*, *nlump=None*, *workDist=None*)
render the frame on an ideal detector plane; returns interpolated frame data zi on a regular grid xi, yi; suitable for use with pcolormesh(xim, yim, zi), with xim, yim = num.meshgrid(xi, yi); note that pcolormesh is used instead of pcolor because zi may be a masked array

xyoToAng(*x0*, *y0*, **args*, ***kwargs*)
convert Cartesian to polar

uses blocking to call vectorized version

xyoToAngAll()
get angular positions of all pixels

xyoToAngCorners()
get angular positions of corner pixels

xyoToAngMap(*x0*, *y0*, **args*, ***kwargs*)
eta by default is in [-pi,pi] if all data are in the left quadrants, remap eta into [0,2*pi]

xyoToAng_V(*x0*, *y0*, **args*, ***kwargs*)
Convert radial spectra obtained from polar rebinned powder diffraction images to angular spectra.

USAGE: mappedData = XFormRadialSpectra(t, D, tilt, xydata, azim, tthRange, radDistFuncHandle, radDistArgs)

INPUTS:

1) t is 2 x 1 (double), the origin translation. The convention is from ‘true’ to ‘estimated’ centers. 2) D is 1 x 1 (double), the sample-to-detector distance in mm. 3) gammaYprime is 1 x 1 (double), the angle between the ‘ideal’ and ‘experimental’ X-axes (horizontal). 4) gammaXhatPrime is 1 x 1 (double), the angle between the ‘ideal’ and ‘experimental’ Y-axes (vertical). 5) xydata is 1 x n (cell), the cell array of data 6) azim 7) tthRange 8) radDistFuncHandle

OUTPUT:

1) mappedData is 1 x n (cell), the cell array of mapped radial data corresponding to the input ‘xydata’.

class hexrd.xrd.detector.**DetectorGeomMar165**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Detector2DRC

__init__(**args*, ***kwargs*)

radialDistortion(*xin*, *yin*, *invert=False*)
no distortion correction

class hexrd.xrd.detector.**DetectorGeomGE**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Detector2DRC

handle geometry of GE detector, such as geometric and radial distortion corrections; x and y are in pixels, as is rho; pixels are numbered from (0,0);

__init__(**args*, ***kwargs*)

radialDistortion(*xin*, *yin*, *invert=False*)
Apply radial distortion to polar coordinates on GE detector

xin, yin are 1D arrays or scalars, assumed to be relative to self.xc, self.yc Units are [mm, radians]. This is the power-law based function of Bernier.

Available Keyword Arguments :

invert = True or >False< :: apply inverse warping

class hexrd.xrd.detector.**DetectorGeomFrelon**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.Detector2DRC

handle geometry of GE detector, such as geometric and radial distortion corrections; x and y are in pixels, as is rho; pixels are numbered from (0,0);

__init__(**args*, ***kwargs*)

radialDistortion(*xin*, *yin*, *invert=False*)
Apply radial distortion to polar coordinates on GE detector

xin, yin are 1D arrays or scalars, assumed to be relative to self.xc, self.yc Units are [mm, radians]. This is the power-law based function of Bernier.

(p[0]*(ri/rx)**p[3] * num.cos(2.0 * ni) + p[1]*(ri/rx)**p[4] * num.cos(4.0 * ni) + p[2]*(ri/rx)**p[5] + 1)*ri

1 + p[0]*(ri/rx)**p[2] * num.cos(p[4] * ni) + p[1]*(ri/rx)**p[3]

Available Keyword Arguments :

invert = True or >False< :: apply inverse warping

class hexrd.xrd.detector.**DetectorGeomQuadGE**(**args*, ***kwargs*)
Bases: hexrd.xrd.detector.DetectorBase

No global parameters – all detector parameters hang off of the sub-detectors; although some data are stored off of this higher level class for convenience

__init__(**args*, ***kwargs*)
pass ReadGE instance as the reader for now; perhaps make a ReadQuadGE class later if it turns out to be needed

dgDummy = None
cleanup after auto-parsing of keyword args

displayIdeal(*framesQuad*, *planeData=None*, *workDist=None*, *nlump=None*, *doFmtCoord=True*, ***kwargs*)
display all sub-detectors on an idealized detector plane

If matplotlib gets around to enabling the transform argument to imshow, that might be a much faster approach than what is currently done here, although what is done here is nice in that it takes account of all of the distortions, not just the in-plane rotation. The idea would be that the in-plane rotation would be, by far, the biggest effect. import matplotlib.transforms as mtransforms tr = mtransforms.Affine2D() tr.rotate(self.zTilt) imshow(, transform=tr)

displaySub(*iQuad*, *thisframe*, *planeData=None*, ***kwargs*)
convenience for displaying a sub-detector ...*** need to code labAxesGlyph support in display

drawRings(*drawOn*, *planeData*, *workDist=None*, ***kwargs*)
assumes ideal geometry, as from displayIdeal

fitProcedureA(*planeData*, *framesQuad*, *iRefQuad=0*, *funcType='pv'*, *funcXVecList=None*, *quadr=1*, *doGUI=0*, *doMRingPlot=False*)
A procedure for fitting the set of detectors; do not need to click 'fit' button in GUI – done inside the procedure.

Watch out – MultiRingEval instances a memory hogs, especially while creating Jacobian matrices!

If want to just refine detector geometry and not the functional forms for the rings, pass funcXVecList as True or as something like a list of arrays from MultiRingEval.getFuncXVecList()

classmethod getRefineFlagsDflt()
no parameters to refine for this detector; call fitProcedureA for a procedure to refine individual sub-detectors

getTThMax()
min over sub-detectors, where for each sub-detector max two-theta is evaluated as the max over points checked in getTThMax for the sub-detector

setCentersFromRef(*iRefQuad=0*)
this assumes all of the tilts are the same

setQuadOffsets(*iRefQuad=0*)
this assumes all of the tilts are the same

13 Functions

hexrd.xrd.detector.**angToXYIdeal**(*tTh*, *eta*, *workDist*)

hexrd.xrd.detector.**mapAngs**(*eta*, *doMap=None*)

hexrd.xrd.detector.**getCentered**(*vmin*, *vmax*)

hexrd.xrd.detector.**getCMap**(*spec*)

hexrd.xrd.detector.**omeRangeToFrameRange**(*omeA*, *omeB*, *omegaStart*, *omegaDelta*, *nFrames*, *checkWrap=True*, *slicePad=1*)
assumes omegas are evenly spaced omegaDelta may be negative

hexrd.xrd.detector.**frameInRange**(*iFrame*, *frameRange*)
for use with output from omeRangeToFrameRange; trust that slicePad=1 was used in omeRangeToFrameRange

hexrd.xrd.detector.**getNFramesFromBytes**(*fileBytes*, *nbytesHeader*, *nbytesFrame*)

hexrd.xrd.detector.**mar165IDim**(*mode*)

hexrd.xrd.detector.**getOmegaMMReaderList**(*readerList*, *overall=False*)
get omega min/max information from a list of readers

hexrd.xrd.detector.**detectorList**()

hexrd.xrd.detector.**newDetector**(*detectorType*, **args*, ***kwargs*)
Return a detector of the requested type

INPUTS

detectorType - a string in the detector type list [see detectorList()]

hexrd.xrd.detector.**newGenericReader**(*ncols*, *nrows*, **args*, ***kwargs*)
currently just returns a Framer2DRC

hexrd.xrd.detector.**newGenericDetector**(*ncols*, *nrows*, *pixelPitch*, **args*, ***kwargs*)
If reader is passed as None, then a generic reader is created

Keyword Arguments: vFactorUnc vDark reader readerKWArgs getDParamDflt setDParamZero getDParamScalings getDParamRefineDflt radialDistortion

If *args is an existing detector geometry, then additional keyword arguments may include:

pVec

If *args is (xc, yc, workDist, xTilt, yTilt, zTilt) detector parameters, then additional keyword arguments may include:

distortionParams

1.1.15 Module: xrd.distortion

2 Functions

hexrd.xrd.distortion.**dummy**(*xy_in*, *params*, *invert=False*)

hexrd.xrd.distortion.**GE_41RT**(*xy_in*, *params*, *invert=False*)

Apply radial distortion to polar coordinates on GE detector

xin, yin are 1D arrays or scalars, assumed to be relative to self.xc, self.yc Units are [mm, radians]. This is the power-law based function of Bernier.

Available Keyword Arguments :

invert = True or >False< :: apply inverse warping

1.1.16 Module: xrd.experiment

Module for wrapping the main functionality of the xrd package.

The Experiment class is the primary interface. Other classes are helpers.

9 Classes

class hexrd.xrd.experiment.**FitModes**

Bases: object

Indicators for single-frame or multiframe data files

class hexrd.xrd.experiment.**ImageModes**

Bases: object

Indicators for single-frame or multiframe data files

class hexrd.xrd.experiment.**Experiment**(*cfgFile='/home/docs/checkouts/readthedocs.org/user_builds/hexrd/envs/v0.2.2/lo packages/hexrd/data/materials.cfg'*, *matFile='/home/docs/checkouts/readthedocs.org/user_builds/hexrd/envs/v0.2.2/lo packages/hexrd/data/all_materials.cfg'*)

Bases: object

Wrapper for xrd functionality

__init__(*cfgFile='/home/docs/checkouts/readthedocs.org/user_builds/hexrd/envs/v0.2.2/local/lib/python2.7/site-packages/hexrd/data/materials.cfg'*, *matFile='/home/docs/checkouts/readthedocs.org/user_builds/hexrd/envs/v0.2.2 packages/hexrd/data/all_materials.cfg'*)

Constructor for Experiment

INPUTS

cfgFile – name of the config file to use for initialization; an empty string indicates that default values for options are used

matFile – name of the materials data file; a real file name is required here

activeImage
Active image

activeMaterial
Active Material

Can be set by number (index in material list) or by name.

On output, it is always a material instance.

activeReader
Get method for activeReader

Reader is set by using index in reader list or by name.

active_img
Current image

add_to_img_list(*name*)
Append the active image to the image list

calInput
(read only) Calibration input data

calibrate(*log=None*)
Calibrate the detector

Currently, uses polar rebin only.

clear_reader()
Close current reader

clear_spots()
Reset the list of spots

curFrameNumber
Current frame number

detector
(read only) detector

dump_grainList(*f*)
dump grainList to cPickle

export_grainList(*f*, *dspTol=None*, *etaTol=None*, *omeTol=None*, *doFit=False*, *sort=True*)
export method for grainList

find_raw_spots()
find spots using current reader and options

fitRMats
(get-only) Rotation matrices from indexing

getSavedReader(*which*)
Get a specified reader

get_spots_ind()
Select spots for indexing

hydra
(read only) hydra image class

img_list
(get only) List of saved images (get only)

img_names
(get-only) List of names for saved images

index_opts
(get-only) Options for indexing

loadDetector (*fname*)
Load the detector information from a file

INPUTS fname – the name of the file to load from

loadMaterialList (*fname*)
Load the pickled material list from a file

INPUTS fname – the name of the file to load from

loadRawSpots (*fname*)
Load the detector information from a file

INPUTS fname – the name of the file to load from

loadReaderList (*fname*)
Load the reader list from a file

INPUTS fname – the name of the file to load from

matDict
(read only) Dictionary mapping material names to material

matList
List of materials

matNames
(read only) List of material names

newDetector (*gp*, *dp*)
Create a new detector with given geometry and distortion parameters

gp - initial geometric parameters *dp* - initial distortion parameters

newMaterial ()
Create a new material and add it to the list

newReader ()
Add new reader to the list and make it active

Changes name if necessary.

numFramesTotal
Number of frames available for reading

raw_spots
(get-only) spots from image before culling and association with rings

readImage (*frameNum=1*)
Read and return an image

DESCRIPTION

This reads an image according to the active reader specification, saving it in the activeImage attribute.

readerListAddCurrent()
Add current list to list of saved readers

readerNames
Return list of saved readers

refineFlags
(read only) refinement flags for calibration

refine_grains(*minCompl*, *nSubIter=3*, *doFit=False*, *etaTol=valWUnit("etaTol", "ANGLE", 1.0, "degrees")*, *omeTol=valWUnit("etaTol", "ANGLE", 1.0, "degrees")*, *fineDspTol=0.005*, *fineEtaTol=valWUnit("etaTol", "ANGLE", 0.5, "degrees")*, *fineOmeTol=valWUnit("etaTol", "ANGLE", 0.5, "degrees")*)
refine a grain list

run_indexer()
Run indexer

saveDetector(*fname*)
Save the detector information to a file

INPUTS fname – the name of the file to save in

saveRMats(*f*)
save rMats to npy file

saveRawSpots(*fname*)
Save the detector information to a file

INPUTS fname – the name of the file to save in

saveReaderList(*fname*)
Save the reader list to a file

INPUTS fname – the name of the file to save in

savedReaders
Return list of saved readers

Experiment.simulateGrain(rMat=array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]]), vMat=array([1., 1., 1., 0., 0., 0.]), planeData=None, detecto
Simulate a grain with choice of active material

spotOpts
(get-only) spot finding options

spot_readers
(get-only) list of readers used to generate spots

spots_for_indexing
(get-only) spots associated with rings

class hexrd.xrd.experiment.**ReaderInput**(*name='reader'*, *desc='no description'*)
Bases: object

ReaderInput

This class is for holding input required to instantiate a reader object. Currently, only GE reader is supported.

__init__(*name='reader'*, *desc='no description'*)
Constructor for ReaderInput

INPUT name – [optional] (str) name desc – [optional] (str) description

NOTES * currently only GE reader is supported

RC
alias of `ReadGE`

aggMode
Mode identifier for frame aggregation

aggModeOp
(read only) option to pass to GE reader instances for aggregation mode

darkFile
Full pathname of dark file

getNumberOfFrames()
Return number of frames available in data files

hasImages
(get only) true if list of images has been set

imageNames
Get method for imageNames

makeReader()
Return a reader instance based on self

setOmegaInfo(*imgName*, *omin*, *omax*, *odel*)
Set omega info for the specified image

class hexrd.xrd.experiment.**CalibrationInput**(*mat*, *xtol=1e-06*)
Bases: object

CalibrationInput

__init__(*mat*, *xtol=1e-06*)
Constructor for CalibrationInput

cakeArgs
(get only) Keyword arguments for polar rebinning

calData
(get only) Lattice parameter data for calibrant

This provides a deepcopy with wavelength, strain magnitude and two-theta width set.

calMat
Calibration material (calibrant)

fitType
fit type: direct or caked

class hexrd.xrd.experiment.**DetectorInfo**(*gParms=[]*, *dParms=[]*)
Bases: object

Class for detector and associated data

__init__(*gParms=[]*, *dParms=[]*)
Constructor for detectorInfo

calibrate(*calInp*, *rdrInp*, *mat*, *log=None*)
Calibrate this detector using specified reader and options

class hexrd.xrd.experiment.**PolarRebinOpts**
Bases: object

Options for polar rebinning

__init__()
Constructor for PolarRebinOpts

This routine sets default values for caking options.

The following attributes (with initial values) can be modified directly. etaMin = 0 etaMax = 360 rhoMin = 100 rhoMax = 1000 numEta = 36 numRho = 500 correct = True

kwArgs
(get only) Return keyword args to pass to polarRebin

class hexrd.xrd.experiment.**SpotOptions**
Bases: object

Manage options available for spot finding and analysis

Mainly, this manages the keyword options to the findSpotsOmegaStack() static method in the Spots class.

__init__()
SpotOptions Constructor

class hexrd.xrd.experiment.**IndexOptions**
Bases: object

indexOptions

__init__()
Constructor for indexOptions

5 Functions

hexrd.xrd.experiment.**newName**(*name*, *nlist*)
return a name not in the list, but based on name input

hexrd.xrd.experiment.**saveExp**(*e*, *f*)
save experiment to file

hexrd.xrd.experiment.**loadExp**(*inpFile*, *matFile='/home/docs/checkouts/readthedocs.org/user_builds/hexrd/envs/v0.2.2/loca*
packages/hexrd/data/materials.cfg')
Load an experiment from a config file or from a saved exp file

inpFile – the name of either the config file or the saved exp file; empty string means start new experiment

matFile – name of the materials file

hexrd.xrd.experiment.**refineDetector**(*grainList*, *scl=None*, *gtol=1e-06*)

hexrd.xrd.experiment.**objFunc**(*x*, *grainList*, *scl*)

1.1.17 Module: xrd.fitting

7 Functions

matchOmegas(xyo_det, hkls_idx, chi, rMat_c, bMat, wavelength, vInv=array([1., 1., 1., 0
[-0.],
[-1.]]), etaVec=array([[1.],
[0.],

[0.]]), omePeriod=None)
For a given list of (x, y, ome) points, outputs the index into the results from oscillAnglesOfHKLs, including the calculated omega values.

hexrd.xrd.fitting.**geomParamsToInput**(*tiltAngles*, *chi*, *expMap_c*, *tVec_d*, *tVec_s*, *tVec_c*, *dParams*)

hexrd.xrd.fitting.**inputToGeomParams**(*p*)

calibrateDetectorFromSX(xyo_det, hkls_idx, bMat, wavelength, tiltAngles, chi, expMap_c, tVe
[-0.],
[-1.]]), etaVec=array([[1.],
[0.],
[0.]]), distortion=(<function GE_41RT at 0x7f7100cd5d70>, [0.0, 0.0, 0.0, 2.0, 2.0, 2], ar
False, False, False, False, False, False, False], dtype=bool), pScl=array([1, 1, 1, 1, 1, 1

hexrd.xrd.fitting.**objFuncSX**(*pFit*, *pFull*, *pFlag*, *dFunc*, *dFlag*, *xyo_det*, *hkls_idx*, *bMat*, *vInv*, *wavelength*, *bVec*, *eVec*, *omePeriod*, *simOnly=False*, *returnScalarValue=False*)

fitGrain(xyo_det, hkls_idx, bMat, wavelength, detectorParams, expMap_c, tVec_c, vInv, beamV
[-0.],
[-1.]]), etaVec=array([[1.],
[0.],
[0.]]), distortion=(<function GE_41RT at 0x7f7100cd5d70>, [0.0, 0.0, 0.0, 2.0, 2.0, 2]), g
True, True, True], dtype=bool), gScl=array([True, True, True, True, True, True, Tr
True, True, True], dtype=bool), omePeriod=None, factor=0.1, xtol=0.0001, ftol=0.0001)

hexrd.xrd.fitting.**objFuncFitGrain**(*gFit*, *gFull*, *gFlag*, *detectorParams*, *xyo_det*, *hkls_idx*, *bMat*, *wavelength*, *bVec*, *eVec*, *dFunc*, *dParams*, *omePeriod*, *simOnly=False*, *returnScalarValue=False*)

gFull[0] = expMap_c[0] gFull[1] = expMap_c[1] gFull[2] = expMap_c[2] gFull[3] = tVec_c[0] gFull[4] = tVec_c[1] gFull[5] = tVec_c[2] gFull[6] = vInv_MV[0] gFull[7] = vInv_MV[1] gFull[8] = vInv_MV[2] gFull[9] = vInv_MV[3] gFull[10] = vInv_MV[4] gFull[11] = vInv_MV[5]

detectorParams[0] = tiltAngles[0] detectorParams[1] = tiltAngles[1] detectorParams[2] = tiltAngles[2] detectorParams[3] = tVec_d[0] detectorParams[4] = tVec_d[1] detectorParams[5] = tVec_d[2] detectorParams[6] = chi detectorParams[7] = tVec_s[0] detectorParams[8] = tVec_s[1] detectorParams[9] = tVec_s[2]

1.1.18 Module: xrd.grain

1 Class

class hexrd.xrd.grain.**Grain**(*spots*, *refineFlags=None*, *pVec=None*, *grainData=None*, ***kwargs*)
Bases: object

A (maybe) indexed grain

method to fit: centroid, orientation, strain, strain+orientation; small and large-strain versions indices into spots a reference to spots? reference lattice parameters – not planeData in case it gets changed with pressure

fitting methods for orientation, stretch, and centroid?

what happens if fit a spot and the fit is bad? what if decide to refine the spot into two spots for clear cases of modest overlap? does that happen often enough that we need to worry about it? should Spots class handle a change in spot numbers: NO can Spot fit methods easily be generalized? Spot should probably barf if asked for fit center if multiple peaks found unless an index is given for which peak set claimbedBy for spots that are found to be bad? – yes, and then if another grain wants to claim the spot, it can ask the claiming grain to hand over the spot or tell it whether there are multiple peaks or whatever

__init__(*spots*, *refineFlags=None*, *pVec=None*, *grainData=None*, ***kwargs*)

bMat
Returns the reciprocal lattice vector components consistent with the stretch tensor. components are written in the CRYSTAL FRAME.

checkClaims()
useful if have not done claims yet and want to check and see if spots are still available; updates completeness too

claimSpots(*asMaster=None*)
claim spots; particularly useful if claimingSpots was False on init; assume conflicts are handled elsewhere or ignored if want to claim spots using this method;

fMat
Returns the lattice vector components consistent with the stretch tensor. Components are written in the CRYSTAL FRAME.

fit(*xtol=1e-12*, *ftol=1e-12*, *fitPVec=True*, *display=True*, *fout=None*)
Fits the cell distortion and orientation with respect to the reference in terms of the deformation gradient F = R * U where R is proper orthogonal and U is symmetric positive-definite; i.e. the right polar decomposition factors.

fitPrecession(*weighting=False*, *display=True*, *xtol=1e-12*, *ftol=1e-12*, *fout=None*)
Fit the Center-Of-Mass coordinates of the grain in the sample frame

getAlignmentRotation()
num.dot(q, num.eye(3) - 2 * num.diag(num.diag(num.dot(r.T, fMat)) < 0))

getCellVolume()
Returns the volume of the direct lattice consistent with the stretch tensor.

getFitResid(*fitPVec=True*, *norm=None*)
returns as shape (n,3), so that len of return value is number of vectors

getLatticeParams()
Returns the lattice parameters consistent with stretch tensor

getLatticeVectors()
Returns the lattice vector components consistent with the stretch tensor. Components are written in the CRYSTAL FRAME.

getReciprocalAlignmentRotation()

getReciprocalLatticeVectors()
Returns the reciprocal lattice vector components consistent with the stretch tensor. components are written in the CRYSTAL FRAME.

getReferenceLatticeParams()
Return the reference lattice parameters stored on self

getRightStretchTensor()
Returns the components of the right stretch tensor, which is symmetric positive-definite. Components are written in the CRYSTAL FRAME. The output is calculated as: U = R^T * V * R This is for convenience and cannot be set independently to preserve self-consistency.

getStretchTensor()
Returns the components of the left stretch tensor, which is symmetric positive-definite. Components are written in the SAMPLE FRAME. This is the primary representation of the stretch tensor in the code base

latticeParameters
Returns the lattice parameters consistent with stretch tensor

minimizeFiberDistance(*xtol=1e-12*, *ftol=1e-12*)
find orientation by minimizing distance to all fibers

newGrain(*newSpots*, *claimingSpots=False*, *lineage=None*, *phaseID=None*, *rMatTransf=None*, *vMat=None*, *omeTol=None*, *etaTol=None*, ***kwargs*)
return a new grain instance without changing self; the new instance will use newSpots;

NOTE: claimingSpots is False by default, so if a grain is to be kept, may want to call claimSpots() method

phaseID and rMatTransf are useful for twins or phase transformations

referenceLatticeParameters
Return the reference lattice parameters stored on self

setStretchTensor(*vVec*)
Sets stretch tensor properly from a 6-vector in the Mandel-Voigt notation.

SEE ALSO: matrixutil.vecMVToSymm()

set_pVec(*pVec*)
sets pVec properly

strip()
meant for multiprocessing, to strip out things that do not really need to be pickled and sent

uMat
Returns the components of the right stretch tensor, which is symmetric positive-definite. Components are written in the CRYSTAL FRAME. The output is calculated as: U = R^T * V * R This is for convenience and cannot be set independently to preserve self-consistency.

updateGVecs(*rMat=None*, *bMat=None*, *chiTilt=None*)
special routine for updating the predicted G-vector angles for subsequent fitting *) need to do this after updating chiTilt, or fixed bMat, etc... *) assumption is that the changes are SMALL so that the existing list of

valid reflection is still valid...

vMat
Returns the components of the left stretch tensor, which is symmetric positive-definite. Components are written in the SAMPLE FRAME. This is the primary representation of the stretch tensor in the code base

vol
Returns the volume of the direct lattice consistent with the stretch tensor.

1.1.19 Module: xrd.hydra

Hydra detector tools

This is just a first pass at laying out a class for the hydra reader. Needs much more development.

1 Class

class hexrd.xrd.hydra.**Hydra**
Bases: object

Hydra image processing

__init__()
Constructor for Hydra.

loadImages()
Load the four hydra images

1.1.20 Module: xrd.indexer

1 Class

class hexrd.xrd.indexer.**GrainSpotter**
Interface to grain spotter, which must be in the user's path

__init__()

9 Functions

hexrd.xrd.indexer.**convertUToRotMat**(*Urows*, *U0*, *symTag='Oh'*, *display=False*)
Takes GrainSpotter gff ouput in rows

U11 U12 U13 U21 U22 U23 U13 U23 U33

and takes it into the hexrd/APS frame of reference

Urows comes from grainspotter's gff output U0 comes from xrd.crystallography.latticeVectors.U0

convertRotMatToFableU(rMats, U0=array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]]), symTag='Oh', display=False)
Makes GrainSpotter gff ouput

U11 U12 U13 U21 U22 U23 U13 U23 U33

and takes it into the hexrd/APS frame of reference

Urows comes from grainspotter's gff output U0 comes from xrd.crystallography.latticeVectors.U0

hexrd.xrd.indexer.**testThisQ**(*thisQ*)
NOTES: (*) doFit is not done here – in multiprocessing, that would end

up happening on a remote process and then different processes would have different data, unless spotsArray were made to be fancier

(*) kludge stuff so that this function is outside of fiberSearch

hexrd.xrd.indexer.**fiberSearch**(*spotsArray*, *hklList*, *iPhase=0*, *nsteps=120*, *minCompleteness=0.6*, *minPctClaimed=0.95*, *preserveClaims=False*, *friedelOnly=True*, *dspTol=None*, *etaTol=0.025*, *omeTol=0.025*, *etaTolF=0.00225*, *omeTolF=0.00875*, *nStdDev=2*, *quitAfter=None*, *doRefinement=True*, *debug=True*, *doMultiProc=True*, *nCPUs=None*, *outputGrainList=False*)
This indexer finds grains by performing 1-d searches along the fibers under the valid spots associated with each reflection order specified in hklList. The set of spots used to generate the candidate orientations may be restricted to Friedel pairs only.

hklList *must* have length > 0; Dach hkl entry in hklList *must* be a tuple, not a list

the output is a concatenated list of orientation matrices ((n, 3, 3) numpy.ndarray).

hexrd.xrd.indexer.**pgRefine**(*x*, *etaOmeMaps*, *omegaRange*, *threshold*)

hexrd.xrd.indexer.**paintGrid**(*quats*, *etaOmeMaps*, *threshold=None*, *bMat=None*, *omegaRange=None*, *etaRange=None*, *omeTol=0.017453292519943295*, *etaTol=0.017453292519943295*, *omePeriod=(-3.141592653589793, 3.141592653589793)*, *doMultiProc=False*, *nCPUs=None*, *debug=False*)

do a direct search of omega-eta maps to paint each orientation in quats with a completeness

bMat is in CRYSTAL frame

etaOmeMaps is instance of xrd.xrdutil.CollapseOmeEta

omegaRange=([-num.pi/3., num.pi/3.],) for example

*) lifted mainly from grain.py

*) **self.etaGrid, self.omeGrid = num.meshgrid(self.etaEdges, self.omeEdges)** this means that ETA VARIES FASTEST!

...make a new function that gets called by grain to do the g-vec angle computation?

hexrd.xrd.indexer.**paintGridThis**(*quat*)

hexrd.xrd.indexer.**writeGVE**(*spotsArray*, *fileroot*, ***kwargs*)

write Fable gve file from Spots object

fileroot is the root string used to write the gve and ini files

Outputs:

No return value, but writes the following files:

<fileroot>.gve <fileroot>_grainSpotter.ini (points to –> <fileroot>_grainSpotter.log)

Keyword arguments:

Mainly for GrainSpotter .ini file, but some are needed for gve files

'sgNum': <225> 'phaseID': <None> 'cellString': <F> 'omeRange': <-60, 60, 120, 240> the oscillation range(s)

**currently pulls from spots

'deltaOme': <0.25, 0.25> the oscillation delta(s) **currently pulls from spots

'minMeas': <24> 'minCompl': <0.7> 'minUniqn': <0.5> 'uncertainty': <[0.10, 0.25, .50]> the min [tTh, eta, ome] uncertainties

in degrees

'eulStep': <2> 'nSigmas': <2> 'minFracG': <0.90> 'nTrials': <100000> 'positionfit': <True>

Notes:

*) **The omeRange is currently pulled from the spotsArray input; the kwarg** has no effect as of now. Will change this to 'override' the spots info if the user, say, wants to pare down the range.

*) **There is no etaRange argument yet, but presumably GrainSpotter knows** how to deal with this. Pending feature...

1.1.21 Module: xrd.material

Module for XRD material class

Use the Material class directly for new materials. Known materials are defined by name in materialDict.

1 Class

class hexrd.xrd.material.**Material**(*name='material'*, *cfgP=None*)
Bases: object

Simple class for holding lattice parameters, accessible by name.

The class references materials by name and contains lattice and space group data.

__init__(*name='material'*, *cfgP=None*)
Constructor for Material

name – (str) name of material cfgP – (instance) configuration file parser with

– the material name as a section

beamEnergy
Beam energy in keV

hklMax
Max sum of squares for HKLs

latticeParameters
Lattice parameters

On output, all six paramters are returned.

On input, either all six or a minimal set is accepted.

The values have units attached, i.e. they are valWunit instances.

name
Name of material

planeData
(read only) Return the planeData attribute (lattice parameters)

sgnum
Space group number

spaceGroup
(read only) Space group

1 Function

hexrd.xrd.material.**loadMaterialList**(*cfgFile*)
Load a list of materials from a file

The file uses the config file format. See ConfigParser module.

1.1.22 Module: xrd.rotations

20 Functions

hexrd.xrd.rotations.**arccosSafe**(*temp*)
Protect against numbers slightly larger than 1 in magnitude due to round-off

hexrd.xrd.rotations.**fixQuat**(*q*)
flip to positive q0 and normalize

hexrd.xrd.rotations.**invertQuat**(*q*)
silly little routine for inverting a quaternion

hexrd.xrd.rotations.**misorientation**(*q1*, *q2*, **args*)
sym is a tuple (crystal_symmetry, *sample_symmetry) generally coded, may split up special cases for no symmetry or crystal/sample only...

hexrd.xrd.rotations.**quatProduct**(*q1*, *q2*)
Product of two unit quaternions.

qp = quatProduct(q2, q1)

q2, q1 are 4 x n, arrays whose columns are quaternion parameters

qp is 4 x n, an array whose columns are the quaternion parameters of the product; the first component of qp is nonnegative

If R(q) is the rotation corresponding to the quaternion parameters q, then

R(qp) = R(q2) R(q1)

hexrd.xrd.rotations.**quatProductMatrix**(*quats*, *mult='right'*)
Form 4 x 4 arrays to perform the quaternion product

USAGE qmats = quatProductMatrix(quats, mult='right')

INPUTS

1. quats is (4, n), a numpy ndarray array of n quaternions horizontally concatenated
2. mult is a keyword arg, either 'left' or 'right', denoting the sense of the multiplication:

   / quatProductMatrix(h, mult='right') * q

   q * h –> < quatProductMatrix(q, mult='left') * h

OUTPUTS

1. qmats is (n, 4, 4), the left or right quaternion product operator

NOTES

*) **This function is intended to replace a cross-product based** routine for products of quaternions with large arrays of quaternions (e.g. applying symmetries to a large set of orientations).

hexrd.xrd.rotations.**quatOfAngleAxis**(*angle*, *rotaxis*)
make an hstacked array of quaternions from arrays of angle/axis pairs

hexrd.xrd.rotations.**quatOfExpMap**(*expMap*)

hexrd.xrd.rotations.**quatOfRotMat**(*R*)

hexrd.xrd.rotations.**rotMatOfExpMap_opt**(*expMap*)
Optimized version of rotMatOfExpMap

hexrd.xrd.rotations.**rotMatOfExpMap_orig**(*expMap*)
Original rotMatOfExpMap, used for comparison to optimized version

hexrd.xrd.rotations.**rotMatOfQuat**(*quat*)
Take an array of n quats (numpy ndarray, 4 x n) and generate an array of rotation matrices (n x 3 x 3)

Uses the truncated series expansion for the exponential map; didvide-by-zero is checked using the global 'tinyRotAng'

hexrd.xrd.rotations.**angleAxisOfRotMat**(*R*)

hexrd.xrd.rotations.**distanceToFiber**(*c*, *s*, *q*, *qsym*, ***kwargs*)

discreteFiber(c, s, B=array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]]), ndiv=120, invert=False, csym=None, ssym=None)

hexrd.xrd.rotations.**mapAngle**(*ang*, **args*, ***kwargs*)
Utility routine to map an angle into a specified period

hexrd.xrd.rotations.**angularDifference_orig**(*angList0*, *angList1*, *units='radians'*)
Do the proper (acute) angular difference in the context of a branch cut.

*) Default angular range in the code is [-pi, pi] *) ... maybe more efficient not to vectorize?

hexrd.xrd.rotations.**angularDifference_opt**(*angList0*, *angList1*, *units='radians'*)
Do the proper (acute) angular difference in the context of a branch cut.

*) Default angular range in the code is [-pi, pi]

hexrd.xrd.rotations.**printTestName**(*num*, *name*)

hexrd.xrd.rotations.**testRotMatOfExpMap**(*numpts*)
Test rotation matrix from axial vector

1.1.23 Module: xrd.spacegroup

Interface with sglite for hkl generation and Laue group determination

This module contains mappings from space group number to either Hall or Hermann-Mauguin notation, as well as the inverse notations.

Space groups can be mapped to crystal class (one of 32 point groups) and then to crystal system .

NOTES:

- Laue group is the cyrstal class if you add a center of symmetry. There are 11 Laue groups, determined directly from the point group.
- This module avoids the use of numpy and uses math module instead. That means the hkl lists are not numpy arrays, but simple lists of tuples.
- Rhombohedral lattices:

REFERENCES

1. Mappings among space group number, Hall symbols and Hermann-Mauguin symbols. http://cci.lbl.gov/sginfo/hall_symbols.html

2. For mapping space group number to point group (crystal class in Schonflies notation)

http://en.wikipedia.org/wiki/Space_group

3. Crystallography and crystal defects By Anthony Kelly, G. W. Groves, P. Kidd

4. Contains point group from sgnum. http://en.wikipedia.org/wiki/Space_group#Classification_systems_for_space_groups

5. Point group to laue group http://www.ruppweb.org/Xray/tutorial/32point.htm

6. For discussion of rhombohedral lattice and "obverse" and "reverse" settings for lattice parameters. Crystal structure determination (book) By Werner Massa

TESTING

Run this module as main to generate all space groups and test the HKL evaluation.

1 Class

class hexrd.xrd.spacegroup.**SpaceGroup**(*sgnum*)
Bases: object

Wrapper on sglite

__init__(*sgnum*)
Constructor for SpaceGroup

INPUTS sgnum – (int) space group number (between 1 and 230)

HallSymbol
(read only) Hall symbol

SgOps
(read only) An sglite.SgOps instance

getHKLs(*ssmax*)
Return a list of HKLs with a cutoff sum of square

INPUTS ssmax – cutoff sum of squares

OUTPUTS hkls – a list of all HKLs with sum of squares less than

or equal to the cutoff, excluding systematic absences and symmetrically equivalent hkls

DESCRIPTION

hermannMauguin
(read only) Hermann-Mauguin symbol

latticeType
Lattice type

Possible values are 'cubic', 'hexagonal', 'trigonal', 'tetragonal', 'orthorhombic', 'monoclinic' and 'triclinic'

Rhombohedral lattices are treated as trigonal using the "obverse" setting.

laueGroup
Schonflies symbol for Laue group (read only)

pointGroup
Schonflies symbol for point group (read only)

reqParams
(read only) Zero-based indices of required lattice parameters

sgnum
Space group number

sixLatticeParams(*lparams*)
Return the complete set of six lattice parameters from the abbreviated set

INPUTS lparams – (tuple) the abbreviated set of lattice parameters

OUTPUTS sparams – (tuple) the complete set of lattice parameters;

(a, b, c, alpha, beta, gamma)

DESCRIPTION * Output angles are in degrees

1 Function

hexrd.xrd.spacegroup.**testHKLs**()

1.1.24 Module: xrd.spotfinder

12 Classes

class hexrd.xrd.spotfinder.**IntensityFunc3D**(**args*, ***kwargs*)
Bases: object

This is just a template for intensity distribution functions in 3D

__init__(**args*, ***kwargs*)

eval(*xVec*, *x*, *y*, *z*, *w=None*, *vSub=None*, *vScale=None*, *diff=False*, *noBkg=False*)
if w is None: x, y, and z are 1D; if w has 1D weights: x, y, and z are 2D

if vSub is present, it is subtracted from results – useful for forming least-squares residuals; and vScale is used to scale the results if it is present

class hexrd.xrd.spotfinder.**IntensityFuncGauss3D**
Bases: hexrd.xrd.spotfinder.IntensityFunc3D

8 parameters: centers (3) FWHMs (3) scaling (1) background (1)

__init__()

class hexrd.xrd.spotfinder.**IntensityFuncGauss3DGenEll**
Bases: hexrd.xrd.spotfinder.IntensityFunc3D

generalization of IntensityFuncGauss3D to have principal axes generally aligned 11 parameters:

centers (3) diagonal "fwhm" (3) scaling (1) off-diagonal (3) background (1)

__init__()

classmethod get2DFunc()
if drop to 2D, also drop back to aligned ellipsoid

guessXVec(*x*, *y*, *z*, *w=None*, *v=None*, *noBkg=False*)
guess is for aligned ellipsoid

class hexrd.xrd.spotfinder.**IntensityFuncMulti3D**(*subFuncClass*, *nSubFuncs*, *minWidth=None*)
Bases: hexrd.xrd.spotfinder.IntensityFunc3D

combination of multiple overlapped functions

__init__(*subFuncClass*, *nSubFuncs*, *minWidth=None*)

guessXVecPureNDImage(*x*, *y*, *z*, *pxlCenterList*, *w=None*, *v=None*, *pxl=None*, *gaussFilterSigma=None*, *noBkg=False*)
for now, rely on specified centers; assumes that v can be made into integers without too much loss of precision

class hexrd.xrd.spotfinder.**IntensityFunc2D**(**args*, ***kwargs*)
Bases: object

This is just a template for intensity distribution functions in 2D; It could be unified with the 3D version, but that would potentially make the interface harder to understand

__init__(**args*, ***kwargs*)

eval(*xVec*, *x*, *y*, *w=None*, *vSub=None*, *vScale=None*, *diff=False*, *noBkg=False*)
if w is None: x and y are 1D; if w has 1D weights: x and y are 2D

if vSub is present, it is subtracted from results – useful for forming least-squares residuals; and vScale is used to scale the results if it is present

class hexrd.xrd.spotfinder.**IntensityFuncGauss2D**
Bases: hexrd.xrd.spotfinder.IntensityFunc2D

6 parameters: centers (2) FWHMs (2) scaling (1) background (1)

__init__()

class hexrd.xrd.spotfinder.**IntensityFuncMulti2D**(*subFuncClass*, *nSubFuncs*)
Bases: hexrd.xrd.spotfinder.IntensityFunc2D

combination of multiple overlapped functions

__init__(*subFuncClass*, *nSubFuncs*)

guessXVec(*x*, *y*, *pxlCenterList*, *w=None*, *v=None*, *pxl=None*, *gaussFilterSigma=None*, *noBkg=False*, *minWidth=None*, *wtrshd=None*)
for now, rely on specified centers; assumes that v can be made into integers without too much loss of precision

class hexrd.xrd.spotfinder.**UnfitableError**(*err*, *msg*)
Bases: exceptions.Exception

__init__(*err*, *msg*)

class hexrd.xrd.spotfinder.**FitFailedError**(*err*, *msg*)
Bases: exceptions.Exception

__init__(*err*, *msg*)

class hexrd.xrd.spotfinder.**Spot**(*key*, *delta_omega*, *data=None*, *omega=None*, *iFrame=None*, *detectorGeom=None*)
Bases: object

__init__(*key*, *delta_omega*, *data=None*, *omega=None*, *iFrame=None*, *detectorGeom=None*)
key should be unique among all spots in the collection; can call with initial data or not

if have tuple from getDataMinimal call, should be able to init directly with that tuple

angCOM(*useFit=True*, *detectorGeom=None*, *getUncertainties=False*, *iSubSpot=None*)
Get center-of-mass in twotheta, eta, omega coordinates. If useFit, then return value from function fit instead of simple estimate

Could look at detectorGeom to see if it has a pVec and do something more elaborate if detectorGeom is already set without a pVec, but that is probably asking for trouble

append(*dataDict*, *omega*, *iFrame*)
dataDict should be like one of the entries in spotDataList coming back from spotFinderSingle

cleanFit()
clean up data associated with having done fit()

static cullSpots(*spots*, *tests*)
Apply a list of tests to spots and return a list with spots that fail culled

spots is a list of Spot instances; tests is a list of tests to be applied;

For convenience, some tests have been defined as static methods off of the Spot class. If a test is tests has a length, then the non-first entries are used as arguments.

display (*cmap=None, relfigsize=(1, 1), vAll=None, markersize=2, xyoPointsList=[([], {})], **kwargs*)
vAll, if present, is used instead of self.vAll; useful for results from fitting

displayFlat (*vAll=None*, *cmap=None*, *markersize=2*, ***kwargs*)
Flatten in Omega for display, no loss of x-y resolution

displayFrames (*reader*, *nFramesPad=0*, *sumImg=<ufunc 'maximum'>*, ***kwargs*)
display the frame(s) from which a spot came

doMap
centralize this decision so that it does not change with the use of quadrature and the like

exceptionOnFit ()
return the exception if stored on fit; note that not all exceptions get stored

finalize (*flatten=False*, *modifyDeltaOmega=False*, *cullDupl=True*)
Could potentially get rid of self.data once finalize is called, but leave it around just in case it is useful for subsequent operations. It might, for example, be needed if we go back and read more intensity data in the vicinity of the spot.

fit (*funcType=None*, *quadr='auto'*, *full_output=False*, *uncertainties=False*, *confidence_level=0.95*, *debug=None*, *detectorGeom=None*, *fout=<open file '<stdout>', mode 'w' at 0x7f710be9a150>*)
fit a distribution to a spot;

may throw a UnfitableError Exception if it is not possible to fit the spot with the given function

if want to re-fit a spot, call the cleanFit method first

if just want to make sure an attempt was made to fit the spot, call fitWrap

fitWrap (**args*, ***kwargs*)
useful if just want to make sure an attempt was made to fit the spot; only lets exceptions through if the spot is suspect

flattenOmega (*modifyDeltaOmega=False*)
flatten spot in Omega;

unless modifyDeltaOmega is set, do not change delta_omega, under the assumption that the spot is being flattened if it does not significantly spill across multiple omega frames

getDoMap ()
centralize this decision so that it does not change with the use of quadrature and the like

getFrames (*reader=None*)
if have self.data, do not need reader

isMarked (*marks*)
return true of spot has been marked as indicated; marks can be an integer or list; if marks is a list, returns true if any of marks are true

static loadSpots (*f*, *minimal=True*, *closeLocal=True*)
load spots stored with storeSpots

note that is a detectorGoem was stored by storeSpots, then all spots loaded here will have it attached to them

merge (*other*)
merge in other's data

setupMulti (*xyoList*, *wtrshd=None*)
for now, rely on candidate centers having been provided; probably want to do a fit

setupQuardFromFWHM (*fwhm*, *fout=<open file '<stdout>', mode 'w' at 0x7f710be9a150>*)
so far, written only for withOmega being True

static spotFromDataList(*key*, *delta_omega*, *dataList*)
useful for merging data from spots that have already been finalized

static storeSpots(*f*, *spotList*, *closeLocal=True*)
store spots to f, which can be a filename or something which behaves like a shelve instance

stores a somewhat minimal set of data

all spots must have the same detectorGeom

consider changing to something like: import cPickle as pickle s = open(f,'w') pickle.dump(stuff, f) f.close()

xyoCOM(*useFit=True*, *iSubSpot=None*)
Get center-of-mass in x, y, omega coordinates on the image stack. If useFit, then return value from function fit instead of simple estimate

class hexrd.xrd.spotfinder.**Spots**(*planeData*, *data*, *detectorGeom*, *omegaMM*, ***kwargs*)
Bases: object

Meant for holding spot data in a form useful for indexing

__init__(*planeData*, *data*, *detectorGeom*, *omegaMM*, ***kwargs*)
planeData : an instance or list (for multiple phases) of the PlaneData class data : can any of:

spots : list of Spot instances, all of which must have been finalized spotAngCoords : spot positions in angular coordinates None

detectorGeom : an instance of DetectorGeomGE or the like omegaMM : (min, max) of omega range, or list of such min/max tuples

checkClaim(*index*)
careful: unlike claimSpots, does not check grain association

checkClaims(*indices=None*)
careful: unlike claimSpots, does not check grain association; careful: indices should not be a boolean array

claimSpots(*indices*, *newClaimedBy*, *checkOnly=False*, *asMaster=False*)
careful: indices should not be a boolean array;

returns bool of same length as indices, with True entries for indices that are in conflict; if checkOnly, then do not actaully claim the spots

cleanMulti()
clean up spots for which split fitting was done, as they might need to change to single spots

static enslaveSpotData(*d*, *dMaster*)
ends up, after potentially doing recursive calls, slaving master of d to master of dMaster

static findSpotsOmegaStack(*reader*, *nFrames*, *threshold*, *minPx*, *discardAtBounds=True*, *keepWithinBBox=True*, *overlapPixelDistance=None*, *nframesLump=1*, *padOmega=True*, *padSpot=True*, *debug=False*, *pw=None*, *fout=None*, *sumImg=None*)
... if merge at bounds, would need to do that before culling based on size (*) spots from beginning would have already been finalized – can merge with them without trouble? (*) add comments here about what happens when reader.wrapsAround

This method does not necessarily need to hang off of the Spots class, but Spots is a convenient place to put it.

If nFrames == 0, read all frames.

if pass overlapPixelDistance, then overlap is determined based on centroid distances in pixel units; with overlapPixelDistance being set to the tolerance for overlap; probably most useful when omega steps are large

reader has been created by doing something like: fileInfo = [('RUBY_4537.raw', 2), ('RUBY_4538.raw', 2)] reader = detector.ReadGE(fileInfo, subtractDark=True)

if go to parallel processing, perhaps return first and last lables for doing merges

fitHasFailed(*index*, *subSpotOnly=False*)
if subSpotOnly the only return true if it was a subspot that failed to fit

fitSpots(*indices*, **args*, ***kwargs*)
fit spots with given indices and update spotAngCoords; for indices is None, all spots are considered for fitting; indices may be a single integer instead of a list of spot indices

set claimsBased if want to base method on spots that have been claimed

fitSpotsMulti(*indices*, *threshold*, **args*, ***kwargs*)
like fitSpots, but consider splitting up spots with multiple peaks

getAngCoords(*indices=None*)
... make returned thing unwritable even if it is a slice

getHKLSpots(*hkl*, *phaseID=None*, *unclaimedOnly=False*, *disallowMasterWhenSplit=True*)
get boolean array that is True for spots associated with a given hkl (and optionally phase)

getIntegratedIntensity(*index*, ***kwargs*)
wrapper that takes care of subspot details

getIterHKL(*hkl*, *phaseID=None*, *unclaimedOnly=True*, *friedelOnly=False*, *iSpotLo=0*, *returnBothCoordTypes=False*)
Returns an iterator over a given set of spots

getIterPhase(*phaseID=None*, *unclaimedOnly=True*, *friedelOnly=False*, *iSpotLo=0*, *returnBothCoordTypes=False*)
Returns an iterator over a given set of spots

getPixelIsInSpots(*indices*, *xyo*, *pixelDist=0*)
find spots that are within pixelDist of containing a given pixel; indices may be a single integer, a list of spot indices, or a boolean array;

may want to use pixelDist of 1 or 2 to protect against dead pixels

getXYOCoords(*indices*)
... make returned thing unwritable even if it is a slice

static mergeSpotsOmegaStack(*spotsA*, *spotsB*, *readerA*, *readerB*, *keepWithinBBox=True*, *overlapPixelDistance=None*, *padOmega=True*, *logger=None*)
see findSpotsOmegaStack (documentation in header and in the body) for comments about padOmega and overlapPixelDistance

spotsB and readerB can be None if this is for wrap-around

A and B can be the same – as would be the case for omega wrapping all the way around

resetDetectorGeom(*detectorGeom*, *doFits=False*, *fitArgs=[]*, *fitKWArgs={}*)
update detector geometry; also fits all spots; probably only want to call this for a single-grain data set or the like; updates angular and cartesian coordinates, but not 2-theta associations or other meta-data

class hexrd.xrd.spotfinder.**SpotsIterator**(*spots*, *hkl*, *phaseID*, *unclaimedOnly*, *friedelOnly*, *iSpotLo*, *returnBothCoordTypes*)
iterator over a given set of spots in a Spots instance, note that the iterator ignores relations among split spots

__init__(*spots*, *hkl*, *phaseID*, *unclaimedOnly*, *friedelOnly*, *iSpotLo*, *returnBothCoordTypes*)

18 Functions

hexrd.xrd.spotfinder.**getBin**(*thisframe*, *threshold*, *padSpot*)

hexrd.xrd.spotfinder.**dilateObj**(*obj*, *shape*, *nDilate=1*)

hexrd.xrd.spotfinder.**emptyBox**(*obj*, *dtype*)

hexrd.xrd.spotfinder.**getDtype**(*this*)

hexrd.xrd.spotfinder.**copyBox**(*inpt*, *obj*)
deepcopy does not seem to do what we want with masked arrays

hexrd.xrd.spotfinder.**getSpot**(*inpt*, *labels*, *objs*, *index*, *keepWithinBBox*, *padSpot*, *darkframe=None*)
labels is from ndimage.label; objs is from ndimage.find_objects

hexrd.xrd.spotfinder.**getSpotFromPixels**(*inpt*, *xThese*, *yThese*, *index*, *keepWithinBBox*, *padSpot*, *darkframe=None*)
this is a bit of an oddball function, mostly for data uniformity of data structures;

hexrd.xrd.spotfinder.**cullSpotUsingBin**(*spot*, *bin*)
remove data for pixels where bin is true; vbox does not change, just change the locations that are marked as belonging with the spot

hexrd.xrd.spotfinder.**getValuesOnly**(*inpt*, *labels*, *objs*, *index*)
labels is from ndimage.label; objs is from ndimage.find_objects

hexrd.xrd.spotfinder.**getIndices**(*inpt*, *labels*, *obj*, *index*, *mode='default'*, *minlabel=0*)

hexrd.xrd.spotfinder.**getImCOM**(*inpt*, *labels*, *objs*, *index*, *floor=None*, *getVSum=False*)
labels is from ndimage.label; objs is from ndimage.find_objects

set floor for a minimum intensity to use in intensity weighted COM

return sum of intensity as well if getVSum is True

hexrd.xrd.spotfinder.**getObjSize**(*labels*, *objs*, *index*)
labels is from ndimage.label; objs is from ndimage.find_objects

hexrd.xrd.spotfinder.**spotFinderSingle**(*thisframe*, *threshold*, *minPx*, *keepWithinBBox*, *padSpot*, *weightedCOM=True*, *pw=None*, *debug=False*, *darkframe=None*)
find spots in thisframe; threshold can be a scalar or defined over of same dimension as thisframe; minPx is the minimum number of pixels to be kept as a spot; weightedCOM is true to use thisframe data for weighting center-of-mass position; if pw is present, it is used for plotting (assumed to be plotwrap.PlotWrap instance)

hexrd.xrd.spotfinder.**getWtrShd**(*inp*, *threshold*, *gaussFilterSigma=None*, *footPrint=None*, *fpi=5*, *numPeaks=None*)
fpi only used if footPrint is None

hexrd.xrd.spotfinder.**doSpotThis**(*iSpot*)
meant for use with multiprocessing

hexrd.xrd.spotfinder.**testSingle**(*fileInfo*)

hexrd.xrd.spotfinder.**testSpotFinder**(*fileInfo*, *delta_omega*, *omega_low*, *howMuch=0.1*)

hexrd.xrd.spotfinder.**main**(*argv=[]*)

1.1.25 Module: xrd.symmetry

4 Functions

hexrd.xrd.symmetry.**toFundamentalRegion**(*q*, *crysSym='Oh'*, *sampSym=None*)

hexrd.xrd.symmetry.**ltypeOfLaueGroup**(*tag*)
See quatOfLaueGroup

hexrd.xrd.symmetry.**quatOfLaueGroup**(*tag*)
Generate quaternion representation for the specified Laue group.

USAGE:

qsym = quatOfLaueGroup(schoenfliesTag)

INPUTS:

1) schoenfliesTag 1 x 1, a case-insensitive string representing the Schoenflies symbol for the desired Laue group. The 14 available choices are:

Class Symbol n

Triclinic Ci (S2) 1 Monoclinic C2h 2 Orthorhombic D2h (Vh) 4 Tetragonal C4h 4

D4h 8

Trigonal C3i (S6) 3 D3d 6

Hexagonal C6h 6 D6h 12

Cubic Th 12 Oh 24

OUTPUTS:

1) qsym is (4, n) the quaterions associated with each element of the chosen symmetry group having n elements (dep. on group – see INPUTS list above).

NOTES:

*) The conventions used for assigning a RHON basis, {x1, x2, x3}, to each point group are consistent with those published in Appendix B of [1].

REFERENCES:

[1] Nye, J. F., ‘‘Physical Properties of Crystals: Their Representation by Tensors and Matrices’‘, Oxford University Press, 1985. ISBN 0198511655

hexrd.xrd.symmetry.**applySym**(*vec*, *qsym*, *csFlag=False*, *cullPM=False*, *tol=1e-08*)
apply symmetry group to a single 3-vector (columnar) argument

csFlag : centrosymmetry flag cullPM : cull +/- flag

1.1.26 Module: xrd.transforms

22 Functions

hexrd.xrd.transforms.**makeGVector**(*hkl*, *bMat*)
take a CRYSTAL RELATIVE B matrix onto a list of hkls to output unit reciprocal lattice vectors (a.k.a. lattice plane normals)

Required Arguments: hkls – (3, n) ndarray of n hstacked reciprocal lattice vector component

triplets

bMat – (3, 3) ndarray representing the matirix taking reciprocal lattice vectors to the crystal reference frame

Output: gVecs – (3, n) ndarray of n unit reciprocal lattice vectors

(a.k.a. lattice plane normals)

To Do: * might benefit from some assert statements to catch improperly shaped

input.

hexrd.xrd.transforms.**anglesToGVec**(*angs*, *bHat_l*, *eHat_l*, *rMat_s=None*, *rMat_c=None*)
from 'eta' frame out to lab (with handy kwargs to go to crystal or sample)

gvecToDetectorXY(gVec_c, rMat_d, rMat_s, rMat_c, tVec_d, tVec_s, tVec_c, beamVec=array([[-0.],
[-0.],
[-1.]]))
Takes a list of unit reciprocal lattice vectors in crystal frame to the specified detector-relative frame, subject to the conditions:

1.the reciprocal lattice vector must be able to satisfy a bragg condition

2.the associated diffracted beam must intersect the detector plane

Required Arguments: gVec_c – (3, n) ndarray of n reciprocal lattice vectors in the CRYSTAL FRAME rMat_d – (3, 3) ndarray, the COB taking DETECTOR FRAME components to LAB FRAME rMat_s – (3, 3) ndarray, the COB taking SAMPLE FRAME components to LAB FRAME rMat_c – (3, 3) ndarray, the COB taking CRYSTAL FRAME components to SAMPLE FRAME tVec_d – (3, 1) ndarray, the translation vector connecting LAB to DETECTOR tVec_s – (3, 1) ndarray, the translation vector connecting LAB to SAMPLE tVec_c – (3, 1) ndarray, the translation vector connecting SAMPLE to CRYSTAL

Outputs: (3, m) ndarray containing the intersections of m <= n diffracted beams associated with gVecs

detectorXYToGvec(xy_det, rMat_d, rMat_s, tVec_d, tVec_s, tVec_c, distortion=(<function GE_
[-0.],
[-1.]]), etaVec=array([[1.],
[0.],
[0.]]))
Takes a list cartesian (x, y) pairs in the detector coordinates and calculates the associated reciprocal lattice (G) vectors and (bragg angle, azimuth) pairs with respect to the specified beam and azimth (eta) reference directions

Required Arguments: xy_det – (n, 2) ndarray or list-like input of n detector (x, y) points rMat_d – (3, 3) ndarray, the COB taking DETECTOR FRAME components to LAB FRAME rMat_s – (3, 3) ndarray, the COB taking SAMPLE FRAME components to LAB FRAME tVec_d – (3, 1) ndarray, the translation vector connecting LAB to DETECTOR tVec_s – (3, 1) ndarray, the translation vector connecting LAB to SAMPLE tVec_c – (3, 1) ndarray, the translation vector connecting SAMPLE to CRYSTAL

Optional Keyword Arguments: beamVec – (3, 1) mdarray containing the incident beam direction components in the LAB FRAME etaVec – (3, 1) mdarray containing the reference azimuth direction components in the LAB FRAME

Outputs: (n, 2) ndarray containing the (tTh, eta) pairs associated with each (x, y) (3, n) ndarray containing the associated G vector directions in the LAB FRAME associated with gVecs

oscillAnglesOfHKLs(hkls, chi, rMat_c, bMat, wavelength, vInv=array([[1.],
[1.],
[1.],
[0.],
[0.],

```
[ 0.]]), beamVec=array([[-0.],
[-0.],
[-1.]]), etaVec=array([[ 1.],
[ 0.],
[ 0.]]))
```

Takes a list of unit reciprocal lattice vectors in crystal frame to the specified detector-relative frame, subject to the conditions:

1.the reciprocal lattice vector must be able to satisfy a bragg condition

2.the associated diffracted beam must intersect the detector plane

Required Arguments: hkls – (3, n) ndarray of n reciprocal lattice vectors in the CRYSTAL FRAME chi – float representing the inclination angle of the oscillation axis (std coords) rMat_c – (3, 3) ndarray, the COB taking CRYSTAL FRAME components to SAMPLE FRAME bMat – (3, 3) ndarray, the COB taking RECIPROCAL LATTICE components to CRYSTAL FRAME wavelength – float representing the x-ray wavelength in Angstroms

Optional Keyword Arguments: beamVec – (3, 1) ndarray containing the incident beam direction components in the LAB FRAME etaVec – (3, 1) ndarray containing the reference azimuth direction components in the LAB FRAME vInv – (6, 1) ndarray containing the indep. components of the inverse left stretch tensor

in the SAMPLE FRAME in the Mandel-Voigt notation

Outputs: ome0 – (3, n) ndarray containing the feasible (tTh, eta, ome) triplets for each input hkl (first solution) ome1 – (3, n) ndarray containing the feasible (tTh, eta, ome) triplets for each input hkl (second solution)

The reciprocal lattice vector, G, will satisfy the the Bragg condition when:

b.T * G / ||G|| = -sin(theta)

where b is the incident beam direction (k_i) and theta is the Bragg angle consistent with G and the specified wavelength. The components of G in the lab frame in this case are obtained using the crystal orientation, Rc, and the single-parameter oscillation matrix, Rs(ome):

Rs(ome) * Rc * G / ||G||

The equation above can be rearranged to yeild an expression of the form:

a*sin(ome) + b*cos(ome) = c

which is solved using the relation:

a*sin(x) + b*cos(x) = sqrt(a**2 + b**2) * sin(x + alpha)

—> sin(x + alpha) = c / sqrt(a**2 + b**2)

where:

alpha = arctan2(b, a)

The solutions are:

/ | arcsin(c / sqrt(a**2 + b**2)) - alpha

x = <

pi - arcsin(c / sqrt(a**2 + b**2)) - alpha

There is a double root in the case the reflection is tangent to the Debye-Scherrer cone (c**2 = a**2 + b**2), and no solution if the Laue condition cannot be satisfied (filled with NaNs in the results array here)

```
polarRebin(thisFrame, npdiv=2, mmPerPixel=(0.2, 0.2), convertToTTh=False, rMat_d=array([[ 1
```

```
[ 0.,  1.,  0.],
[ 0.,  0.,  1.]]), tVec_d=array([    0.,     0., -1000.]), beamVec=array([[-0.],
[-0.],
[-1.]]), etaVec=array([[ 1.],
[ 0.],
[ 0.]]), rhoRange=array([ 20, 200]), numRho=1000, etaRange=array([-0.08726646,  6.19591884]
```

Caking algorithm

INPUTS

thisFrame npdiv=2, pixel subdivision (n x n) to determine bin membership rhoRange=[100, 1000] - radial range in pixels numRho=1200 - number of radial bins etaRange=np.pi*np.r_[-5, 355]/180. – range of eta numEta=36 - number of eta subdivisions ROI=None - region of interest (four vector) corrected=False - uses 2-theta instead of rho verbose=True,

hexrd.xrd.transforms.**arccosSafe**(*temp*)

Protect against numbers slightly larger than 1 in magnitude due to round-off

hexrd.xrd.transforms.**angularDifference**(*angList0*, *angList1*, *units='radians'*)

Do the proper (acute) angular difference in the context of a branch cut.

*) Default angular range is [-pi, pi]

hexrd.xrd.transforms.**mapAngle**(*ang*, **args*, ***kwargs*)

Utility routine to map an angle into a specified period

hexrd.xrd.transforms.**reg_grid_indices**(*edges*, *points_1d*)

get indices in a 1-d regular grid.

edges are just that:

point: x (2.5)

edges: |1 |2 |3 |4 |5

indices: | 0 | 1 | 2 | 3 |

above the deltas are + and the index for the point is 1

point: x (2.5)

edges: |5 |4 |3 |2 |1

indices: | 0 | 1 | 2 | 3 |

here the deltas are - and the index for the point is 2

- can handle grids with +/- deltas
- be careful when using with a cyclical angular array! edges and points must be mapped to the same branch cut, and abs(edges[0] - edges[-1]) = 2*pi

hexrd.xrd.transforms.**columnNorm**(*a*)

normalize array of column vectors (hstacked, axis = 0)

hexrd.xrd.transforms.**rowNorm**(*a*)

normalize array of row vectors (vstacked, axis = 1)

hexrd.xrd.transforms.**unitVector**(*a*)

normalize array of column vectors (hstacked, axis = 0)

hexrd.xrd.transforms.**makeDetectorRotMat**(*tiltAngles*)
Form the (3, 3) tilt rotations from the tilt angle list:

tiltAngles = [gamma_Xl, gamma_Yl, gamma_Zl] in radians

hexrd.xrd.transforms.**makeOscillRotMat**(*oscillAngles*)
oscillAngles = [chi, ome]

hexrd.xrd.transforms.**makeRotMatOfExpMap**(*expMap*)

hexrd.xrd.transforms.**makeBinaryRotMat**(*axis*)

hexrd.xrd.transforms.**makeEtaFrameRotMat**(*bHat_l*, *eHat_l*)
make eta basis COB matrix with beam antiparallel with Z

takes components from ETA frame to LAB

hexrd.xrd.transforms.**validateAngleRanges**(*angList*, *startAngs*, *stopAngs*, *ccw=True*)
A better way to go. find out if an angle is in the range CCW or CW from start to stop

There is, of course, an ambigutiy if the start and stop angle are the same; we treat them as implying 2*pi having been mapped

hexrd.xrd.transforms.**rotate_vecs_about_axis**(*angle*, *axis*, *vecs*)
Rotate vectors about an axis

INPUTS *angle* - array of angles (len == 1 or n) *axis* - array of unit vectors (shape == (3, 1) or (3, n)) *vec* - array of vectors to be rotated (shape = (3, n))

Quaternion formula: if we split v into parallel and perpedicular components w.r.t. the axis of quaternion q,

v = a + n

then the action of rotating the vector dot(R(q), v) becomes

v_rot = (q0**2 - |q|**2)(a + n) + 2*dot(q, a)*q + 2*q0*cross(q, n)

hexrd.xrd.transforms.**quat_product_matrix**(*q*, *mult='right'*)
Form 4 x 4 array to perform the quaternion product

USAGE qmat = quatProductMatrix(q, mult='right')

INPUTS

1. quats is (4,), an iterable representing a unit quaternion horizontally concatenated
2. mult is a keyword arg, either 'left' or 'right', denoting the sense of the multiplication:

   / quatProductMatrix(h, mult='right') * q

   q * h –> < quatProductMatrix(q, mult='left') * h

OUTPUTS

1. qmat is (4, 4), the left or right quaternion product operator

NOTES

*) **This function is intended to replace a cross-product based** routine for products of quaternions with large arrays of quaternions (e.g. applying symmetries to a large set of orientations).

hexrd.xrd.transforms.**quat_distance**(*q1*, *q2*, *qsym*)

1.1.27 Module: xrd.transforms_CAPI

19 Functions

hexrd.xrd.transforms_CAPI.**makeGVector**(*hkl*, *bMat*)

take a CRYSTAL RELATIVE B matrix onto a list of hkls to output unit reciprocal lattice vectors (a.k.a. lattice plane normals)

Required Arguments: hkls – (3, n) ndarray of n hstacked reciprocal lattice vector component

triplets

bMat – (3, 3) ndarray representing the matirix taking reciprocal lattice vectors to the crystal reference frame

Output: gVecs – (3, n) ndarray of n unit reciprocal lattice vectors

(a.k.a. lattice plane normals)

To Do: * might benefit from some assert statements to catch improperly shaped

input.

gvecToDetectorXY(gVec_c, rMat_d, rMat_s, rMat_c, tVec_d, tVec_s, tVec_c, beamVec=array([[-0.],
[-0.],
[-1.]]))

Takes a list of unit reciprocal lattice vectors in crystal frame to the specified detector-relative frame, subject to the conditions:

1.the reciprocal lattice vector must be able to satisfy a bragg condition

2.the associated diffracted beam must intersect the detector plane

Required Arguments: gVec_c – (n, 3) ndarray of n reciprocal lattice vectors in the CRYSTAL FRAME rMat_d – (3, 3) ndarray, the COB taking DETECTOR FRAME components to LAB FRAME rMat_s – (3, 3) ndarray, the COB taking SAMPLE FRAME components to LAB FRAME rMat_c – (3, 3) ndarray, the COB taking CRYSTAL FRAME components to SAMPLE FRAME tVec_d – (3, 1) ndarray, the translation vector connecting LAB to DETECTOR tVec_s – (3, 1) ndarray, the translation vector connecting LAB to SAMPLE tVec_c – (3, 1) ndarray, the translation vector connecting SAMPLE to CRYSTAL

Outputs: (m, 2) ndarray containing the intersections of m <= n diffracted beams associated with gVecs

detectorXYToGvec(xy_det, rMat_d, rMat_s, tVec_d, tVec_s, tVec_c, beamVec=array([[-0.],
[-0.],
[-1.]]), etaVec=array([[1.],
[0.],
[0.]]))

Takes a list cartesian (x, y) pairs in the detector coordinates and calculates the associated reciprocal lattice (G) vectors and (bragg angle, azimuth) pairs with respect to the specified beam and azimth (eta) reference directions

Required Arguments: xy_det – (n, 2) ndarray or list-like input of n detector (x, y) points rMat_d – (3, 3) ndarray, the COB taking DETECTOR FRAME components to LAB FRAME rMat_s – (3, 3) ndarray, the COB taking SAMPLE FRAME components to LAB FRAME tVec_d – (3, 1) ndarray, the translation vector connecting LAB to DETECTOR tVec_s – (3, 1) ndarray, the translation vector connecting LAB to SAMPLE tVec_c – (3, 1) ndarray, the translation vector connecting SAMPLE to CRYSTAL

Optional Keyword Arguments: beamVec – (3, 1) mdarray containing the incident beam direction components in the LAB FRAME etaVec – (3, 1) mdarray containing the reference azimuth direction components in the LAB FRAME

Outputs: (n, 2) ndarray containing the (tTh, eta) pairs associated with each (x, y) (n, 3) ndarray containing the associated G vector directions in the LAB FRAME associated with gVecs

oscillAnglesOfHKLs(hkls, chi, rMat_c, bMat, wavelength, vInv=None, beamVec=array([[-0.], [-0.], [-1.]]), etaVec=array([[1.], [0.], [0.]]))

Takes a list of unit reciprocal lattice vectors in crystal frame to the specified detector-relative frame, subject to the conditions:

1.the reciprocal lattice vector must be able to satisfy a bragg condition

2.the associated diffracted beam must intersect the detector plane

Required Arguments: hkls – (n, 3) ndarray of n reciprocal lattice vectors in the CRYSTAL FRAME chi – float representing the inclination angle of the oscillation axis (std coords) rMat_c – (3, 3) ndarray, the COB taking CRYSTAL FRAME components to SAMPLE FRAME bMat – (3, 3) ndarray, the COB taking RECIPROCAL LATTICE components to CRYSTAL FRAME wavelength – float representing the x-ray wavelength in Angstroms

Optional Keyword Arguments: beamVec – (3, 1) mdarray containing the incident beam direction components in the LAB FRAME etaVec – (3, 1) mdarray containing the reference azimuth direction components in the LAB FRAME

Outputs: ome0 – (n, 3) ndarray containing the feasible (tTh, eta, ome) triplets for each input hkl (first solution) ome1 – (n, 3) ndarray containing the feasible (tTh, eta, ome) triplets for each input hkl (second solution)

The reciprocal lattice vector, G, will satisfy the the Bragg condition when:

b.T * G / ||G|| = -sin(theta)

where b is the incident beam direction (k_i) and theta is the Bragg angle consistent with G and the specified wavelength. The components of G in the lab frame in this case are obtained using the crystal orientation, Rc, and the single-parameter oscillation matrix, Rs(ome):

Rs(ome) * Rc * G / ||G||

The equation above can be rearranged to yield an expression of the form:

a*sin(ome) + b*cos(ome) = c

which is solved using the relation:

a*sin(x) + b*cos(x) = sqrt(a**2 + b**2) * sin(x + alpha)

—> sin(x + alpha) = c / sqrt(a**2 + b**2)

where:

alpha = atan2(b, a)

The solutions are:

/ | arcsin(c / sqrt(a**2 + b**2)) - alpha

x = <

pi - arcsin(c / sqrt(a**2 + b**2)) - alpha

There is a double root in the case the reflection is tangent to the Debye-Scherrer cone (c**2 = a**2 + b**2), and no solution if the Laue condition cannot be satisfied (filled with NaNs in the results array here)

hexrd.xrd.transforms_CAPI.**arccosSafe**(*temp*)
Protect against numbers slightly larger than 1 in magnitude due to round-off

hexrd.xrd.transforms_CAPI.**angularDifference**(*angList0*, *angList1*, *units='radians'*)
Do the proper (acute) angular difference in the context of a branch cut.

*) Default angular range is [-pi, pi]

hexrd.xrd.transforms_CAPI.**mapAngle**(*ang*, **args*, ***kwargs*)
Utility routine to map an angle into a specified period

hexrd.xrd.transforms_CAPI.**columnNorm**(*a*)
normalize array of column vectors (hstacked, axis = 0)

hexrd.xrd.transforms_CAPI.**rowNorm**(*a*)
normalize array of row vectors (vstacked, axis = 1)

hexrd.xrd.transforms_CAPI.**unitRowVector**(*vecIn*)

hexrd.xrd.transforms_CAPI.**makeDetectorRotMat**(*tiltAngles*)
Form the (3, 3) tilt rotations from the tilt angle list:

tiltAngles = [gamma_Xl, gamma_Yl, gamma_Zl] in radians

hexrd.xrd.transforms_CAPI.**makeOscillRotMat**(*oscillAngles*)
oscillAngles = [chi, ome]

hexrd.xrd.transforms_CAPI.**makeRotMatOfExpMap**(*expMap*)
make a rotation matrix from an exponential map

hexrd.xrd.transforms_CAPI.**makeRotMatOfQuat**(*quat*)
make rotation matrix from a unit quaternion

...check to set if input is unit magnitude?

hexrd.xrd.transforms_CAPI.**makeBinaryRotMat**(*axis*)

hexrd.xrd.transforms_CAPI.**makeEtaFrameRotMat**(*bHat_l*, *eHat_l*)

hexrd.xrd.transforms_CAPI.**validateAngleRanges**(*angList*, *angMin*, *angMax*, *ccw=True*)

hexrd.xrd.transforms_CAPI.**rotate_vecs_about_axis**(*angle*, *axis*, *vecs*)

hexrd.xrd.transforms_CAPI.**quat_distance**(*q1*, *q2*, *qsym*)
qsy coming from hexrd.xrd.crystallogray.PlaneData.getQSym() is C-contiguous

1.1.28 Module: xrd.xrdbase

2 Functions

hexrd.xrd.xrdbase.**dataToFrame**(*data*, *sumImg=True*)
utility function to allow flexibility in input

data can be: (*) an instance of ReadGE or the like, which is already set up, in which

case all frames are used and flattened

(*) a frame

hexrd.xrd.xrdbase.**getGaussNDParams**(*xList*, *w=None*, *v=None*)

1.1.29 Module: xrd.xrdutil

6 Classes

class hexrd.xrd.xrdutil.**FormatEtaOme**(*etas*, *omes*, *A*, *T=False*, *debug=False*)
for plotting data as a matrix, with ijAsXY=True

__init__(*etas*, *omes*, *A*, *T=False*, *debug=False*)

class hexrd.xrd.xrdutil.**OmeEtaPfig**(*omeEdges*, *etaEdges*, *cmap=None*, *vMM=None*, *doRot=False*, *invertFromSouthern=True*, *netStyle=None*, *netNDiv=12*, *netAlpha=0.5*, *pointLists=[]*, *drawColorbar=True*)
Bases: object

works with data from with CollapseOmeEta

__init__(*omeEdges*, *etaEdges*, *cmap=None*, *vMM=None*, *doRot=False*, *invertFromSouthern=True*, *netStyle=None*, *netNDiv=12*, *netAlpha=0.5*, *pointLists=[]*, *drawColorbar=True*)

clearLines()
get rid of any existing lines and legends

display(*data, tTh, nP=True, opacity=[None], rangeVV_w=None, winKWArgs={}*)
if want to do interpolated results, matplotlib.delaunay has some triangulation and interpolation stuff, but have not sorted that out yet

drawLines(*nP*, *rMat=None*, *pointLists=None*)
if specify a pointLists argument then it replaces self.pointLists

class hexrd.xrd.xrdutil.**CollapseOmeEta**(*readerList*, *planeData*, *hklList*, *detectorGeom*, *strainMag=None*, *nEtaBins=480*, *mask=None*, *threshold=None*, *applyLorenz=True*, *nframesLump=1*, *debug=False*, *computeMeanTwoTheta=False*)
Bases: object

Can pass a mask to use in addition to whatever the readers are already set up to do; with frames set zero where mask is True

If pass pw: after plot is made, can do calls on self.pw to plot other things on top of the image

while this takes a list of readers, it is not yet really coded for disjoint omega ranges

__init__(*readerList*, *planeData*, *hklList*, *detectorGeom*, *strainMag=None*, *nEtaBins=480*, *mask=None*, *threshold=None*, *applyLorenz=True*, *nframesLump=1*, *debug=False*, *computeMeanTwoTheta=False*)

display(*iData=0*, *rangeVV=None*, *cmap=None*, *pw=None*, *debug=False*, *pfig=False*, *comment=''*, *tTh=False*, *rangeVV_w=None*, *winKWArgs={}*, *pfigKWArgs={}*)
iData is index into hklList passed on initialization, NOT into planeData

pointList, if specified is a list of tuples of the form (3xn ndarray, style) where style gets passed to plotwrap

can set tTh to 'withOpacity' if the 2-theta plot should be made with opacity (alpha) constructed from the intensity weights; so far this only works for pfig=False

etas = None
count frames

iEtaBin = None
protect against numerical silliness

indices = None
make valsThese into float so that do not overrun max int value

tThRanges = None
set up data

class hexrd.xrd.xrdutil.**GrainPoles**(*omeEtaPfigs*)

__init__(*omeEtaPfigs*)

class hexrd.xrd.xrdutil.**MultiSlopeFunc**(*m1*, *m2*, *xcrit*, *power=0.2*)
function that transitions smoothly from slope of m1 to slope of m2 in the vacinity of xcrit, with the smoothness of the transition dictated by a specified power; for large values of power, might eventually want to put in code to protect against numerical overflow

__init__(*m1*, *m2*, *xcrit*, *power=0.2*)

class hexrd.xrd.xrdutil.**MultiSlopeFuncSmooth**(*m1*, *w1*, *p1*, *m2*, *xcrit*, *power=0.2*)
function that transitions smoothly from slope of m1 to slope of m2 in the vacinity of xcrit, with the smoothness of the transition dictated by a specified power;

__init__(*m1*, *w1*, *p1*, *m2*, *xcrit*, *power=0.2*)

48 Functions

hexrd.xrd.xrdutil.**fitLParm**(*data*, *detectorGeom*, *planeData*, *funcType='pv'*, *lParm0=None*, *funcXVecList=None*, *quadr1d=8*, *debug=False*)
fit lattice parameters to data; using dataToFrame to map the data to a frame

input planeData is not changed

hexrd.xrd.xrdutil.**fitDG**(*data*, *detectorGeom*, *planeData*, *funcType='pv'*, *funcXVecList=None*, *quadr1d=8*, *debug=False*)
fit detector geometry parameters to data; using dataToFrame to map the data to a frame

pass funcXVecList as True or as something like detectorGeom.fitRingsFunc.getFuncXVecList() if want to just refine detector geometry and not the functional forms for the rings

input detectorGeom is used to guess parameters and is not modified – a new detectorGeom is returned

hexrd.xrd.xrdutil.**fitDGX**(*data*, *detectorGeom*, *planeData*, *funcType='pv'*, *quadr1d=8*, *debug=False*, *nGlIter=2*, *xFuncs=None*, *xDG=None*)
fit detector geometry parameters to data; using dataToFrame to map the data to a frame

uses a procedure that might end up being more robust than fitDG

input detectorGeom is used to guess parameters and is not modified – a new detectorGeom is returned

hexrd.xrd.xrdutil.**textureToSpots**(*texture*, *planeData*, *detectorGeom*, *omeMM=None*, *etaMM=None*, *pVecs=None*)
take texture as returned from pyMps and make spots

hexrd.xrd.xrdutil.**makeSynthSpots**(*rMats, pVecs, bMats, planeData, detectorGeom, omeMM=[-3.141592653589793, 3.141592653589793], etaMM=None, hklList=None, beamSize=None*)
make synthetic spots

hexrd.xrd.xrdutil.**makePathVariantPoles**(*rMatRef*, *fromPhase*, *pathList*, *planeDataDict*, *hkllID*)

hexrd.xrd.xrdutil.**displayPathVariants**(*data, rMatRef, fromPhase, pathList, planeDataDict, detectorGeom, omeMin, omeMax, phaseForDfltPD=None, markerList=['D', 'o', 'p', 's', 'v', 'x', '+', '*', '<', '>', '1', '2', '3', '4', '^'], hklList=None, color=None, pointKWArgs={}, hklIDs=None, pw=None*)

hexrd.xrd.xrdutil.**makeRMatList**(*pathList*, *fromPhase*)

hexrd.xrd.xrdutil.**findGrainsNewSpots**(*grainList0*, *spots*, *doFitting=True*, *minCompleteness=None*, *pathList=[]*, *eosDict=None*, *refPDDict=None*, *tK=300.0*, *debug=True*, *indepFitPVec=False*, *findByPixelDist=1*, *maxIterRefit=3*, *pVecTol=None*)

see if grains in grainList0 show up in spots; meant to be useful for taking existing grains from a load step and looking for them in a new load step; returns a new list of the same length, with None wherever a grain near the existing one was not found, and a list of grains for each path in pathList; grains in grainList0 are not modified; spots should be a Spots instance

hexrd.xrd.xrdutil.**stretchToLV**(*V*, *fMat*)

from stretch V in crystal frame and fMat, compute new lattice parameters; fMat can be the 'F' from lparms.latticeVectors

V = B + I, where B is the Biot strain

hexrd.xrd.xrdutil.**calculateBiotStrain**(*initLP*, *finalLP*, *display=False*)

hexrd.xrd.xrdutil.**makeMNConn**(*m*, *n*, *tri=True*)

m and n are number of edges, so 1 more than number of zones

hexrd.xrd.xrdutil.**makeNVecs**(*tth*, *etaIn*, *omeIn*, *asGrid=False*)

hexrd.xrd.xrdutil.**omeEtaGridToNVecs**(*tTh*, *omegas*, *etas*)

hexrd.xrd.xrdutil.**roty90**(*v*)

hexrd.xrd.xrdutil.**makeMeasuredScatteringVectors**(*tTh*, *eta*, *ome*, *convention='hexrd'*, *frame='sample'*)

Construct scattering vectors from angles (2theta, eta, ome) will do HEXRD/APS and Fable frames, sample or lab.

for fable frame geomtery, see:

http://sourceforge.net/apps/trac/fable/attachment/wiki/WikiStart/Geometry_version_1.0.8.pdf

hexrd.xrd.xrdutil.**doItMP**(*nSkip*)

hexrd.xrd.xrdutil.**readFrameStack_multiproc**(*reader*, *func*, *nPerChunk=None*, *nCPUs=None*, *debug=False*)

read the full frame stack using multiprocessing, applying fund to each frame to obtain the result;

use makeNew for each chunk; if reader.dark is a shared memory array then it remains so

hexrd.xrd.xrdutil.**thresholdStackDisplay**(*data*, *threshold*, *cmap=None*, *pw=None*, *detectorGeom=None*, *planeData=None*, *displayKWArgs={}*, *drawRingsKWArgs={}*)

passes sumImg=num.maximum to dataToFrame so that if data is a reader then frame ends up being the maximum over the image stack

hexrd.xrd.xrdutil.**grainPolesGUI**(*omeEtaPfigs*)

GUI with sliders for rotating a grain's spots

execfile('examples/feSynthSpotsPfig.py') gui = grainPolesGUI([pwSB])

hexrd.xrd.xrdutil.**darkFromStack**(*reader*, *nFrames=0*, *nChunk=4*, *medianSize=None*, *medianRange=(-15, 15)*, *cutMinFactor=None*, *checkIntensityResolution=False*)

If nFrames == 0, read all frames.

If medianSize is specified then a median filter of the given size is used to find dead pixels, with pixels outside of medianRange marked as dead.

hexrd.xrd.xrdutil.**tryFromShelf**(*shelfFileName*, *thingName*)

try to pull the thing from the shelf and return None if it does not work

hexrd.xrd.xrdutil.**putInShelf**(*shelfFileName*, *thingName*, *thing*)

hexrd.xrd.xrdutil.**pullFromStack**(*reader*, *detectorGeom*, *tThMM*, *angWidth*, *angCen*, *threshold=20*, *distInAng=False*, *padSpot=True*, *mask3D=None*, *exitOnFail=False*)

angWidth is distance from angCen, so actually half-width

do not yet deal with wrap-around (eg, reader that spans 2*pi)

hexrd.xrd.xrdutil.**grainSpotsFromStack**(*g*, *reader*, *detectorGeom*, *angWidth*, *threshold*, ***kwargs*)

wrapper around spotFromStack; takes a grain and returns a dictionary of spots for the grain, with keys (hklID, iThisHKL)

angWidth is for orientation spread; for 2-theta, g.planeData 2-theta ranges are used

can specify hklIDs to look for just a subset of spots

set distInAng=False if want the spots to have to contain the predicted angles, otherwise, the closest spots in the bounding boxes will be returned

hexrd.xrd.xrdutil.**spotFromStack**(*reader*, *detectorGeom*, *tThMM*, *angWidth*, *angCen*, *threshold*, *fullBackground=False*, *asFrame=False*, *exitOnFail=True*, *distInAng=False*, *debug=True*)

if asFrame, then omegas come out as frame indices; note that angCen should still be specified as omega, not a frame index

hexrd.xrd.xrdutil.**collapse**(*vAll*, *eta*, *ome*, *nOme=None*, *nEta=None*, *weightedList=[]*, *averagedList=[]*, *auxList=[]*, *debug=False*)

Returns a sparse matrix, with zeros where no intensity falls;

pass nEta and nOme to control the number of eta and omega bins, otherwise they are determined automatically, with the omega binning assuming that omegas fall on frames at regular intervals with no gaps

for each entry in weightedList, also returns that data collapsed and weighted by vAll; similarly for averagedList

hexrd.xrd.xrdutil.**displaySparse**(*a*, *vmin=None*, *vmax=None*, *cmap=None*, *fmt=None*, *markersize=3*, *colorUnder=None*, *ijNZ=None*, ***kwargs*)

hexrd.xrd.xrdutil.**displayEtaOme**(*eta*, *ome*, *vAll*, *nEta=None*, *nOme=None*, ***kwargs*)

hexrd.xrd.xrdutil.**getLorenz**(*detectorGeom*, **args*)

hexrd.xrd.xrdutil.**getEtaResolution**(*detectorGeom*, *tTh*)

angular resolution along eta

hexrd.xrd.xrdutil.**getTThResolution**(*detectorGeom*, *tTh*)

hexrd.xrd.xrdutil.**drEtaOme**(*angCen*, *dEta*, *dOme*)

compute true angular changes and dA corresponding to steps in eta and omega

hexrd.xrd.xrdutil.**omeEtaGridToDA**(*tThNominal*, *etaEdges*, *omeEdges*)

get grid patch areas, in the sense of solid angle (pole figure) coverage;

hexrd.xrd.xrdutil.**bboxIntersect3D**(*b1*, *b2*)
0-tolerance bounding box intersection in 3d

hexrd.xrd.xrdutil.**pfigFromSpots**(*spots*, *iHKL*, *phaseID=None*, *nOme=None*, *nEta=None*, *tThTol=None*, *etaExcl=0.1*, *plot=False*, *plotPrefix=None*, *debug=False*)
probably want to have collected spots without discarding those at boundaries (discardAtBounds=False)

depending on the context, may need to have done something like iHKL = hklIDs.index(hklID)

if nEta is negative, it is treated as the target lumping of pixels

etaExcl is in radians – etas within this range of +-pi/2 degrees are left out; can set to None to turn this behavior off

can use tThTol to tighten down the two-theta tolerance

hexrd.xrd.xrdutil.**mapAngCen**(*ang*, *angCen*)
map angle ang into equivalent value that is closest to angCen

hexrd.xrd.xrdutil.**makeSynthFrames**(*spotParamsList*, *detectorGeom*, *omegas*, *intensityFunc=<hexrd.xrd.spotfinder.IntensityFuncGauss3D object at 0x7f70fc4d7550>*, *asSparse=None*, *output=None*, *cutoffMult=4.0*, *debug=1*)
intensityFunc is an instance of a class that works as an intensity fuction.

spotParamsList should be a list with each entry being a list of arguments appropriate to the intensityFunc.constructParams function. For intensityFunc=spotfinder.IntensityFuncGauss3D(), each spotParamsList entry should be (center, fwhm, A), with center being the 3D spot center in angular coordinates (radians), fwhm being the (2-theta, eta, omega) widths in 3D, and A being an intensity scaling.

If output is specified as a string, then the frames with the given prefix are dumped to files instead of being accumulated in memory. If output is callable then frames are passed to output().

If asSparse is true then sparse matrices are used to reduce memory footprint. The asSparse option is currently not coded for the case of output having been specied.

cutoffMult is the multiplier on the FWHM to determine the angular cutoff range for evaluating intensity for each spot.

hexrd.xrd.xrdutil.**validateAngleRanges**(*angList*, *startAngs*, *stopAngs*, *ccw=True*)
A better way to go. find out if an angle is in the range CCW or CW from start to stop

There is, of course an ambigutiy if the start and stop angle are the same; we treat them as implying 2*pi

hexrd.xrd.xrdutil.**tVec_d_from_old_parfile**(*old_par*, *detOrigin*)

hexrd.xrd.xrdutil.**objFun_tVec_d**(*tvd_xy*, *rMat_d*, *beamXYD*, *detOrigin*, *bHat_l*)

hexrd.xrd.xrdutil.**beamXYD_from_tVec_d**(*rMat_d*, *tVec_d*, *bVec_ref*, *detOrigin*)

hexrd.xrd.xrdutil.**write_old_parfile**(*filename*, *results*)

```
simulateOmeEtaMaps(omeEdges, etaEdges, planeData, expMaps, chi=0.0, etaTol=None, omeTol=No
[-0.],
[-1.]]), eVec=array([[ 1.],
[ 0.],
[ 0.]]), vInv=array([[ 1.],
[ 1.],
[ 1.],
[ 0.],
[ 0.],
```

[0.]]))

all angular info is entered in degrees

quats are (4, n)

...might want to creat module-level angluar unit flag ...might want to allow resvers delta omega

hexrd.xrd.xrdutil.**simulateGVecs**(*pd, detector_params, grain_params, ome_range=[(-3.141592653589793, 3.141592653589793)], ome_period=(-3.141592653589793, 3.141592653589793), eta_range=[(-3.141592653589793, 3.141592653589793)], panel_dims=[(-204.8, -204.8), (204.8, 204.8)], pixel_pitch=(0.2, 0.2), distortion=(<function dummy at 0x7f7100cd5e60>, [])*)

simulate the monochormatic scattering for a specified

- •space group
- •wavelength
- •orientation
- •strain
- •position
- •detector parameters
- •oscillation axis tilt (chi)

subject to

- •omega (oscillation) ranges (list of (min, max) tuples)
- •eta (azimuth) ranges

pd.....a hexrd.xrd.crystallography.PlaneData instance detector_params...a (10,) ndarray containing the tilt angles (3), translation (3),

chi (1), and sample frame translation (3) parameters

grain_params.....a (12,) ndarray containing the exponential map (3), translation (3), and inverse stretch tensor compnents in Mandel-Voigt notation (6).

- •currently only one panel is supported, but this will likely change very soon

hexrd.xrd.xrdutil.**angularPixelSize**(*xy_det, xy_pixelPitch, rMat_d, rMat_s, tVec_d, tVec_s, tVec_c, distortion=(<function dummy at 0x7f7100cd5e60>*[ ]*,)*)

- •choices to beam vector and eta vector specs have been supressed
- •assumes xy_det in UNWARPED configuration

hexrd.xrd.xrdutil.**pullSpots**(*pd, detector_params, grain_params, reader, ome_period=(-3.141592653589793, 3.141592653589793), eta_range=[(-3.141592653589793, 3.141592653589793)], panel_dims=[(-204.8, -204.8), (204.8, 204.8)], panel_buff=[20, 20], pixel_pitch=(0.2, 0.2), distortion=(<function dummy at 0x7f7100cd5e60>, []), tth_tol=0.15, eta_tol=1.0, ome_tol=1.0, npdiv=1, threshold=10, doClipping=False, filename=None, save_spot_list=False, use_closest=False*)

hexrd.xrd.xrdutil.**validateQVecAngles**(**args*, ***kwargs*)

h

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X