Refinement of centre of mass and strain from boxscan data

JPW and SEH

Autumn 2011

Abstract

Try to put together a documented refinement routine for fitting the centre of mass and strain of grains from a series of 3dxrd scans when the sample was translated to cover a larger volume. Learn from the experience in previous ImageD11 code to try to make something a bit better.

Contents

1 Introduction

The makemap (and behind it refinegrains) codes are not very extensible. Jette's FitAllB does a better job, but is not obvious to modify for the boxscan case. Combining the indexing with grain refinement would seem to help. Something we've learned is that an iterative procedure can help, with human guidance along the way, a bit like structure refinement. Could say this is grain map refinement.

Would like to have a program, or collection of programs, that sit on top of a data structure that holds all the relevant information in a way that is easy to update and transform.

Peaks, output from the peaksearch (or peaksearches) won't change during refinement, so can sit in one big static table.

Grains will change as we progress in our analysis.

The assignment of peaks to grains is a fundamental problem. We need to allow for twinning, where some peaks come from multiple grains. Also we should check each grain has one and only one observed peak as coming from a particular lattice vector in a particular scan.

There is no need for a great deal of backwards compatibility or special cases that do not apply here. Also the ImageD11 code will be heavily used to get going faster.

2 Overall structure of this program

Hard to predict.

Perhaps have three objects that are backed by files; one for peaks, one for grains and one that matches the grains to peaks assignments.

The peaks would be a columnfile, perhaps later in hdf format to load and save faster. It will hold the instrument parameters corresponding to a single detector.

Need also a diffractometer thing to deal with the orientation of the sample.

The grains holds orientations, centres of mass and perhaps volumes.

The assignments link grains to peak files.

Thinking of another case - refinement against multiple detectors with a single set of grains, it sort of makes sense to divide things up like this.

3 Example: refinement of the sand data

To make sure the program is useful, we will directly attack the sand data problem. The data were collected as a series of rotations around +/- 90 degrees going through the bars of the stress rig. To cover the area of a large sample translations along samx were used to map out about 10 mm diameter volume, with a beamsize of 1.5 mm horizontally. Then 4 layers in z were included.

3.1 Combining the peaksearch files together

For boxscanning we need to put together the peaksearch outputs from these multiple scans into a single file. To process everything together we use a trick of adding the sample translations to the peak position translations, so each peak will be described by its xl, yl and zl position in 3D space. This will not be the true position on the detector, but the position corrected for the sample shift.

"sandcombine.py" 3a≡

```
import glob, os
def sand_special_case(fname):
    """
    Create an ascii file holding fltname, x, y, z in microns
    one file per line
    """
    rootdir = "/data/id11/inhouse/jon/sand0910/newpeaksearches/peaksearches"
    stem = "sand_097_0910"
    scans = "a_","b_"
    xr = range(7)
    zr = range(1,2)
    fltnames = [ ("%s%s%d_%d__all.flt"%(stem, s, xp, zp),xp,zp)
                 for s in scans
                 for zp in zr
                 for xp in xr ]
    #for f,x,z in fltnames:
    #    print f, x, z, os.path.exists(os.path.join(rootdir, f))
    assert len(fltnames) == len(glob.glob(os.path.join(rootdir,"*_all.flt")))
    f = open(fname,"w")
    y = 0
    for flt, x,z in fltnames:
        f.write("%s %f %f %f\n"%(
            os.path.join(rootdir, flt),
            x * 1500 - 4500,
            y,
            z * 500 - 750
            ))
    f.close()
    # print open(fname).read()
    print "Hardwired sand special case!"
    print "using default file with fltname samtx samty samtz"

if __name__=="__main__":
    from combine_flt import *
    sand_special_case( "sand.xyz")
    combine_flt_from_file( "sand.xyz", "sand.hdf", "sand0910" )

◇
```

3.2 Testing and running the codes

We want to keep things running all the time, so we define test cases.

⟨ *jobs* 3b ⟩ ≡

```
python26 sandcombine.py
◇
```

Fragment never referenced.

"Makefile" 4≡

```
all: boxref.pdf

boxref.pdf: boxref.dvi xlylzl.eps
        dvipdfm boxref.dvi

boxref.dvi: boxref.tex
        latex boxref.tex
        nuweb boxref.w
        latex boxref.tex
        latex boxref.tex

boxref.tex: boxref.w
        nuweb boxref.w

◇
```

4 More general program

Here we will try to have to code that is not specific to the sand example that can later be recycled more easily

4.1 Combining flt files

Input to the program will be a series of flt files and x,y,z sample translations. To be compatible with the future we will label them samtx, samty and samtz. In principle they could be input as vectors to be more general, but that is not needed for now.

An object will hold the current list of flt files and we will append to it.

The data are written to an output hdf file which will have a data group with the name supplied.

⟨ *combine_flt* 5 ⟩ ≡

```
import numpy, columnfile, sys

class combine_flt(object):
    def __init__(self, fltfile, samtx=0, samty=0, samtz=0):
        """
        First flt file (defines columns, should be same for all)
        Translations to be supplied if not zero
        """
        print "reading", fltfile
        sys.stdout.flush()
        self.fltfile = columnfile.columnfile( fltfile )
        onearray = numpy.ones(self.fltfile.nrows, numpy.float32)
        self.fltfile.addcolumn( onearray*samtx, "samtx" )
        self.fltfile.addcolumn( onearray*samty, "samty" )
        self.fltfile.addcolumn( onearray*samtz, "samtz" )
        # Convert to list for easier appending later
        self.datalist = [ list(col) for col in self.fltfile.bigarray ]
        print len(self.datalist),len(self.datalist[0])
    def addfltfile(self, fltfile, samtx=0, samty=0, samtz=0):
        """
        Add another flt file, probably with different translations
        """
        c = columnfile.columnfile(fltfile)
        onearray = numpy.ones(c.nrows, numpy.float32)
        c.addcolumn( onearray*samtx, "samtx" )
        c.addcolumn( onearray*samty, "samty" )
        c.addcolumn( onearray*samtz, "samtz" )
        # Check columnfiles have equivalent titles
        for i,j in zip(c.titles, self.fltfile.titles):
            assert i == j
        for i in range(len(self.datalist)):
            self.datalist[i] += list(c.bigarray[i])
        print len(c.bigarray[0]), len(self.datalist[0])
    def write_hdf(self, name, datagroup):
        """ Save the output, the hdf code is in columnfile.py """
        self.datalist = numpy.array(self.datalist)
        self.fltfile.bigarray = self.datalist
        self.fltfile.ncols, self.fltfile.nrows = self.fltfile.bigarray.shape
        self.fltfile.set_attributes()
        columnfile.colfile_to_hdf(self.fltfile, name, datagroup)
```

◇

Fragment referenced in 6b.

Driver method which reads an ascii file holding the name of the flt file then the samtx, samty and samtz translations, one per line:

⟨ *combinefltfromfile* 6a ⟩ ≡

```
def combine_flt_from_file( fname, hdfname, hdfgroup ):
    """ Read one flt per line with x,y,z translations """
    o = None
    for line in open(fname).readlines():
        name = line.split()[0]
        x,y,z = [float(v) for v in line.split()[1:]]
        print name, x, y, z
        if o is None:
            o = combine_flt( name, x, y, z )
        else:
            o.addfltfile( name, x, y, z)
    o.write_hdf( hdfname, hdfgroup )
◇
```

Fragment referenced in 6b.

Finally, the command line interface

"combine_flt.py" 6b≡

```
⟨ copyright 20 ⟩

⟨ combine_flt 5 ⟩

⟨ combinefltfromfile 6a ⟩

if __name__=="__main__":
    try:
        xyzfile = sys.argv[1]
        hdfile =  sys.argv[2]
        hdfgroup = sys.argv[3]
    except:
        print "Usage: %s xyzfile hdffile hdfgroup"%( sys.argv[0] )
        print """ ... where:
   xyzfile = ascii file with lines containing:
        fltfilename   samtx   samty   samtz
   hdffile = output hdffile with combined data in it
   hdfgroup = group name (directory) in the hdf file """

    combine_file_from_file( sys.argv[1], sys.argv[2], sys.argv[3] )
◇
```

4.2 Utilities to skip malloc

In the course of putting together the code there were a couple of things that seem to be missing from numpy, which is the main python module we use. In optimising python code for large arrays it can help to avoid doing lots of malloc and free creating temporary arrays. Many numpy functions have the option to receive an output array as argument so you skip this, but not all, so those needed are added here.

Length of the 3 vector to use for normalisation. Candidate for a C function to just normalise the vectors in place.

⟨ *vec3mod* 7a ⟩ ≡

```
import numpy
def vec3mod( t, result, tmp):
    """
    Make the sqrt( i*i+j*j+k*k ) without temporaries (no malloc)
    Surely this would be faster in c
    Modifies result in place
    """
    assert len(t) == 3
    assert len(t[0]) == len(result)
    assert len(result) == len(tmp)
    if 0: # slower
        numpy.add.reduce( t*t, axis=0, out=result )
        numpy.sqrt( result, result )
    else:
        numpy.multiply( t[0], t[0], tmp )
        numpy.multiply( t[1], t[1], result )
        numpy.add( result, tmp, result )
        numpy.multiply( t[2], t[2], tmp )
        numpy.add( result, tmp, result )
        numpy.sqrt( result, result ) # Write output in same array
    # Note that the input argument is modified in this function call
    # so nothing is returned
```

◇

Fragment referenced in 7c.

Computes the outerproduct as in numpy but without allocating a new result array.

⟨ *outerprod_res* 7b ⟩ ≡

```
import numpy
def outerprod_res( a, b, result):
    """
    This is numpy.outer expanded to take a result
    """
    numpy.multiply( a.ravel()[:,numpy.newaxis],
                    b.ravel()[numpy.newaxis,:],
                    result )
```

◇

Fragment referenced in 7c.

Put these together:

"utils_numpy.py" 7c≡

⟨ *vec3mod* 7a ⟩
⟨ *outerprod_res* 7b ⟩
◇

4.3 Rotation axes

This takes the gv_general code from ImageD11 and modifies it to do what we want here. The idea is to be ready for moving multiple rotation axes on the new diffractometer, so we go for an axis/angle description

⟨ *axis* 8 ⟩ ≡

```
from utils_numpy import outerprod_res, vec3mod
class axis(object):
    def __init__(self, n):
        """
        n is the axis normal, eg, direction of the rotation axis
        """
        arn = numpy.asarray( n )
        assert arn.shape == (3,)
        # check normalised
        assert abs(numpy.dot( n, n ) - 1) < 1e-6
        self.n = arn
⟨ rotate_one_v 9 ⟩
⟨ rotate_vs 10 ⟩
◇
```

Fragment referenced in 15b.

For the grain origin we have one single grain location and a multitude of rotation angles which are different for each diffraction spot. We want to compute the position of the grain origin as a function of the omega angle.

Angles are passed in as cosine and sin which are expected to be cached. The idea is to move everything out of the function which does not need to be in the function, so you need to do the convert to radians and cos or sin elsewhere.

⟨ *rotate_one_v* 9 ⟩ ≡

```
    def rotate_one_v( self, r , cost, sint, result=None, tmp=None, back=False):
        """
        r = vector, one to many angles
        cost, sint = cosine and sin of angles
        rotates a single vector to multiple angles
        if back then invert sign (sin(-x)==-sin(x), cos(-x)==cos(x))
        http://mathworld.wolfram.com/RotationFormula.html
        r' = r cos(t) + n(n.r)(1-cos(t)) + rxn sin(t)
        """
        ar = numpy.asarray( r )
        assert ar.shape == (3,)
        if result is None:
            result = numpy.zeros( (3, len(cost) ), numpy.float32)
        if tmp is None:
            tmp =  numpy.zeros( (3, len(cost) ), numpy.float32)
        # 3x1     3x1      1         3x1   3x1
        nndotr = self.n * numpy.dot(self.n, ar)
        # 3x1             3x1     3x1
        rxn = numpy.cross( ar, self.n )
        # inplace to skip mallocs
        outerprod_res( ar, cost, result )
        outerprod_res( nndotr, (1-cost), tmp)
        numpy.add( result,  tmp, result )
        outerprod_res( rxn, sint, tmp )
        # reverse rotation, sin-> -sin
        if back:
            numpy.subtract( result,  tmp, result )
        else:
            numpy.add( result, tmp, result )
        return result
```

◇

Fragment referenced in 8.

Now the case of rotating many vectors to many angles. This should almost certainly be a C function for the general case. For now we hard wire for an axis along z (sorry!)

⟨ *rotate_vs* 10 ⟩ ≡

```
    def rotate_vs( self, vs, cost, sint, result, tmp, back=False):
        """
        rotate many vectors to many different angles

        R = I3cos(t) + ww^T (1-cos(t)) - W sin(t)
        t = angle
        W = vector with hat operator = [ 0  -wz  wy
                                         wz  0  -wx
                                        -wy  wx  0  ]
        """
        if 1:
            # Rotate the vectors into sample frame
            assert (self.n == (0,0,1)).all()
            numpy.multiply(  peakdata.cosomega , vs[0], result[0] )
            numpy.multiply(  peakdata.sinomega , vs[1], tmp )
            if back:
                numpy.subtract( result[0], tmp, result[0] )
            else:
                numpy.add( result[0], tmp, result[0] )
            numpy.multiply(  peakdata.sinomega, vs[0], tmp )
            numpy.multiply(  peakdata.cosomega, vs[1], result[1] )
            if back:
                numpy.add(  result[1] , tmp, result[1] )
            else:
                numpy.subtract(  result[1] , tmp, result[1] )
            result[2]=  vs[2]
        else:
            dx, dy, dz = self.direction
            e = n.array([self.direction]*3)
            w = n.transpose(n.array( [ [  0, -dz,  dy]  ,
                                       [ dz,   0, -dx]  ,
                                       [-dy,  dx,   0] ], n.float))
            # self.matrix = n.identity(3, n.float)*ct - st * w  + \
            #               (1 - ct)*e*n.transpose(e)
            #
            # identity term:
            result[i] = cost
            # 1-cost terms:
            omc = 1 - cost
            for i in range(3):
                for j in range(3):
                    numpy.add(w[i,j]*w[i,j]*omc, result[i], result[i] )
                for j in range(i+1,3):
                    numpy.add(w[i,j]*w[i,j]*omc, result[i], result[i] )

            raise Exception("FIXME")
```

◇

Fragment referenced in 8.

4.4 Detector parameters

Based on the peak pixel coordinates in the image and also the we want to compute the x,y,z coordnates of the spots in the lab frame in 3D. We will add on the sample translations, not yet using our rotation axis thing above but hard wiring an axis. Shame on me.

⟨ *update_det_pars* 11a ⟩ ≡

```
def update_det_pars( peakdata, pars ):
    """
    Update the xl, yl, zl columns
    peakdata = columnfile
    pars = ImageD11 parameters object
    """
    print "Update xl, yl, zl for current parameters"
    for label in ["xl","yl","zl"]:
        if label not in peakdata.titles:
            peakdata.titles.append(label)
    peakdata.xl,peakdata.yl,peakdata.zl = transform.compute_xyz_lab(
        [peakdata.sc, peakdata.fc],
        **pars.get_parameters() )
    # Imagine sample translates along +x. Detector is at x=+distance
    # To get the difference vector right we add samtx to xl for omega=0
    # Same for samty, yl and samtz, zl
    # when omega rotates to + 90 degrees (right handed)
    #   samtx is along +yl
    #   samty is along -xl
    rad = float(pars.parameters['omegasign'])*numpy.pi/180.0
    peakdata.sinomega = numpy.sin(peakdata.omega * rad) # == 0 at omega = 0
    peakdata.cosomega = numpy.cos(peakdata.omega * rad) # == 1 at omega = 0
    peakdata.xl += peakdata.samtx * peakdata.cosomega - \
                   peakdata.samty * peakdata.sinomega
    peakdata.yl += peakdata.samtx * peakdata.sinomega + \
                   peakdata.samty * peakdata.cosomega
    peakdata.zl += peakdata.samtz
    print "lab x shape",peakdata.xl.shape
    return peakdata
```

◇

Fragment referenced in 15b.

4.5 Potential peaks

Figure out which peaks in peakdata might belong to a grain, that is they are indexed within tol.

REFACTOR: Make a class which holds the statically allocated data and make class methods to break up the different parts of the computations.

⟨ *potentialpeaks* 11b ⟩ ≡

```
def potentialpeaks( grainlist, peakdata, pars, tol ):
    """
    For each grain, see which peaks are potentially interesting
```

```
grainlist = python list of grain objects
peakdata  = columnfile object with xl,yl,zl,cosomega,sinomega
pars      = ImageD11 parameters object
tol       = Indexing integer tolerance
"""
assign = {} # Dict to hold output
print "potential peak finding"
# Pre-allocate necessary work arrays
# Logically these could be part of peakdata, but that is bad
# for doing something in parallel
# ... note ... npks is of the order 1e6
npks = len(peakdata.omega)
indices = numpy.arange( npks, dtype = numpy.intp )
fac = numpy.zeros( npks, numpy.float32)
tmp = numpy.zeros( npks , numpy.float32)
g = numpy.zeros( (3, npks), numpy.float32)
t = numpy.zeros( (3, npks), numpy.float32)
hkli = numpy.zeros( g.shape, numpy.int32)
hklr = numpy.zeros( g.shape, numpy.float32)
drlv = numpy.zeros( g.shape, numpy.float32)
i = 0
for gc in grainlist:
    # Find the scattering vectors in lab frame:
    tx, ty, tz = gc.translation
    omega_axis.rotate_one_v( [tx,ty,tz],
                             peakdata.cosomega,
                             peakdata.sinomega,
                             result = t,
                             tmp = g,
                             back=True )
    numpy.subtract( peakdata.xl , t[0], t[0] )
    numpy.subtract( peakdata.yl , t[1], t[1] )
    numpy.subtract( peakdata.zl , t[2], t[2] )
    # unit vectors
    vec3mod( t, fac, tmp)
    numpy.divide( t, fac, t )
    # t is now the k vector
    beam_direction = (1,0,0)
    for j in range(3):
        if beam_direction[j] > 0:
            numpy.subtract( t[j,:], beam_direction[j], t[j,:] )
    #
    # Rotate the vectors into sample frame
    omega_axis.rotate_vs( t,
                          peakdata.cosomega,
                          peakdata.sinomega,
                          g, tmp, back=False)
    # Index the vectors using h,k,l indices
    ubioy = gc.ubi / float(pars.parameters['wavelength'])
    hklr = numpy.dot( ubioy, g )
    # Score to match up how good they are
    numpy.floor( hklr + 0.5, hkli )
    drlv = (hklr - hkli)
    drlv = numpy.sqrt( (drlv*drlv).sum(axis=0))
    inds = numpy.compress( drlv < tol, indices )
```

```
            assign[i] = inds, numpy.take(hkli, inds, axis=1), numpy.take(drlv,
                                                                         inds)
            print i, len(inds)
            i+=1
        peakdata.drlv = drlv
        return assign
```
◇

Fragment referenced in 15b.

4.6 Pick peaks

Given coarse assignments to grains (eg, peaks within some tol) refine this to choose specific peaks per hkl per grain. This is where people can hopefully contribute ideas for how to assign which grain to which peak in better ways, depending on the sample etc.

⟨ *pick_peaks* 13 ⟩ ≡

```
def pick_peaks( peakdata, inds, priority="drlv"):
    """
    Given drlv and whatever you like from peakdata, choose your
    favourites.
    We will go in the order of drlv, the getting uniq omega,
    samtx, samty and samtz

    Another option would be to pick the more intense peaks
    """
    priority_array = numpy.take( getattr( peakdata, priority), inds)
    order = numpy.argsort( priority_array )
    assert len(priority_array) == len(inds)
    j = order[0]
    uniq = [ inds[j] ]
    for i in order[1:]:
        j = inds[i]
        # Decide if we have already see this peak in uniq
        attrs = ['samtx', 'samty', 'samtz', 'omega' ]
        tols  = numpy.array([ 0.1,     0.1,     0.1,     10 ])
        new = True
        for p in uniq:
            diffs = [abs(getattr(peakdata,a)[p] - getattr(peakdata,a)[j])
                     for a,t  in zip(attrs, tols)]
            if ((tols < diffs) == False).all():
                new = False
#            print diffs, new,
        if new:
            uniq.append( inds[i] )
    return uniq
```
◇

Fragment referenced in 15b.

4.7 Selection of peaks for a single grain

⟨ *grain_choose_single_peaks* 14 ⟩ ≡

```
def grain_choose_single_peaks( peakdata, agrain, assign, magic = 32,
                               proirity = 'sum_intensity' ):
    """
    peakdata = all the data
    agrain = one single grain
    assign = current list of indices and hkl indexing

    Choose the best peak for each possible hkl of this grain
    Assuming the grain has got a list of potential peaks, we want to
    whittle this down so it only has one observation per calculated
    point. It is for the case of a grain that indexes into a cloud
    of data
    """
    inds, hkls, drlv = assign
    numpy.put(peakdata.drlv, inds, drlv)
    # print inds.shape, hkls.shape
    assert magic > abs(hkls.ravel()).max()
    key = (hkls[0]*magic + hkls[1])*magic + hkls[2]
    order = numpy.argsort(key)
    # Several cases:
    #   Single observation of this hkl, keep it
    #   Multiple observations of this hkl:
    #       group into "equivalence", eg: samtx, omega etc
    #       pick best in equivalent group

    uniq = []
    j = 0
    doubles = []

    while j < len(drlv)-1:
        i = j + 1
        pks = [ order[j] ] # local indexing for this grain
        kj = key[ order[j] ]
        while i<len(drlv) and key[order[i]] == kj:
            pks.append( order[i])
            i = i + 1
        j = i + 1
        if len(pks) == 1:
            # Single observation for this hkl
            uniq.append( pks[0] )
            continue
        else:
            allpks = inds[pks] # global selection into dataset
            #uniq += pick_peaks( peakdata , allpks, priority = 'sum_intensity')
            uniq += pick_peaks( peakdata , allpks, priority = 'drlv')
    # print
    # s = set(key)
    print len(uniq)
    return     inds[uniq], hkls[uniq], drlv[uniq]
```

◇

Fragment referenced in 15b.

4.8 Refinement

⟨ *refine_sparse* 15a ⟩ ≡

```
def refine_sparse( peakdata, gr, indices, hkls ):
    hcalc = gr.ubi
    pass
```

◇

Fragment referenced in 15b.

4.9 The box fit script

"boxfit.py" 15b≡

```
import numpy, columnfile
from ImageD11 import grain, parameters, transform
⟨ axis 8 ⟩

⟨ update_det_pars 11a ⟩

⟨ potentialpeaks 11b ⟩



⟨ pick_peaks 13 ⟩

⟨ grain_choose_single_peaks 14 ⟩

⟨ refine_sparse 15a ⟩

if __name__=="__main__":
    import sys
    if 0:
     print """fuckit - you need a proper grain object that
     just copies all the data for the peak it is interested in"""
     sys.exit()

    omega_axis = axis( [ 0, 0, 1] ) # z
    chi_axis   = axis( [ 1, 0, 0] ) # x
    wedge_axis = axis( [ 0, 1, 0] ) # y



    # Data object
    peakdata = columnfile.colfile_from_hdf( sys.argv[1] )
    if not hasattr( peakdata, 'sum_intensity'):
        peakdata.sum_intensity = peakdata.npixels * peakdata.avg_intensity

    # Grain collection object
    grainlist = grain.read_grain_file( sys.argv[2] )

    if sys.argv[2][0:2] == "x_":
```

```
        items = sys.argv[2].split("_")
        tx = float( items[1] )*1500 - 4500
        ty = float( items[3] )*1500 - 4500
        tz = float( items[5].split(".")[0] )*500 -750
        for g in grainlist:
            #print tx,ty,tz,g.translation
            g.translation += [tx,ty,tz]
            #print tx,ty,tz,g.translation
        print "Sand translations added"
    # Detector parameters
    pars = parameters.parameters()
    pars.loadparameters( sys.argv[3] )

    peakdata = update_det_pars( peakdata, pars )


    tol = 0.05

    assignments = potentialpeaks( grainlist, peakdata, pars, tol )

    print "Choose uniq within potentials"
    narrowassign = {}
    for i in range(len(grainlist)):
        narrowassign[i] =  grain_choose_single_peaks( peakdata,
                                                      grainlist[i],
                                                      assignments[i])
```

◇

5 Peaks - observed data

We have a series of scans with a large beamsize where the sample was translated. For each peak we will expect the usual peaksearch output columns. We will assume, for now, that multiple threshold levels have been merged using the merge_flt.py script in ImageD11. Those to be used:

- sc - centre of mass in slow pixel direction
- fc - centre of mass in fast pixel direction
- omega - centre of mass in rotation direction (image stack)
- sum intensity
- samtx, samty, samtz sample positions, default to zero
- a unique scan number in the case of repeated scans of same thing

Based on these parameters there are some derived quantities we can make that do not depend on the grain indexing. The xlab, ylab and zlab co-ordinates of the diffraction spot centre of mass can

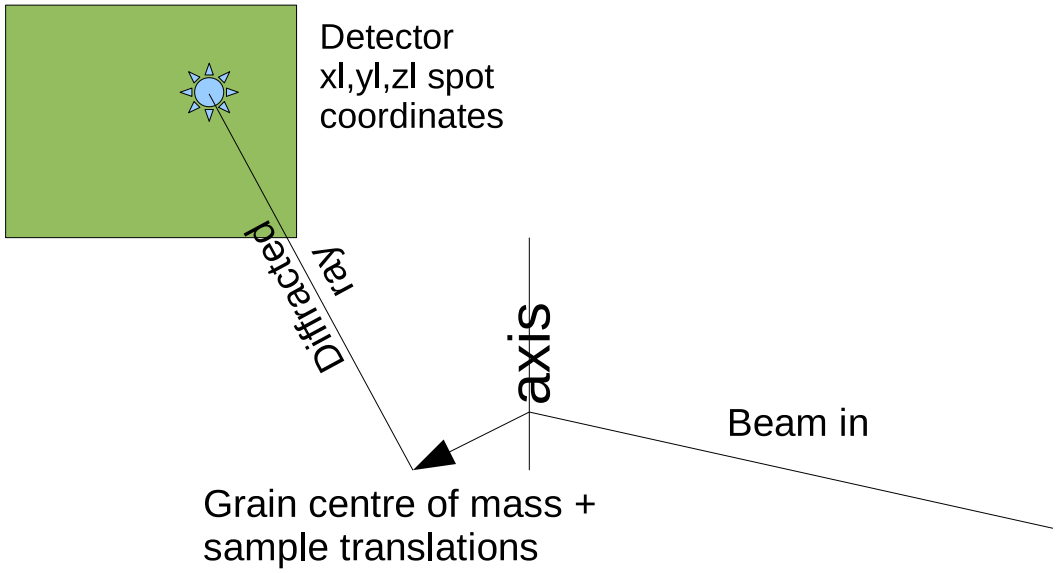

Figure 1: Geometry: Sample translations can actually be added to the spot position, which is going to make the coding easier

be computed from sc and fc coordinates and instrument parameters. So some instrument parameters "belong" strictly to the diffraction spots:

- sample to detector distance
- beam centre and pixel size
- distortion correction (if not fixed)
- detector orientation and tilts
- diffractometer angles and axis definitions

The last one, the diffractometer angle and axis definitions reflects that we have a new diffractometer with lots of axes and will need to deal with that soon. The old "wedge" plus "chi" was a bit of a nightmare anyway.

5.1 Taking account of the boxscan translations

For each diffraction spot there may be sample translations, which is the box scan issue. We could compute the position of the grain centre of mass with these translations added but then the calculation will depend on the grain. It is easier if we put the sample translations onto the detector co-ordinates so that they are independent of the grain assignment (this part of refinegrains.py is really screwed up).

So, we will take a series of columnfiles and make a new object which has all the usual columnfile columns, plus some instrument parameters, plus the xlab, ylab and zlab translations.

For refinement we might want $d/dpar$ of xlab, ylab and zlab.

⟨ *peaksobject* 18 ⟩ ≡

```
from ImageD11.columnfile import columnfile

class detectorpeaks(object):
    def __init__(self, parfile):
        """
        Fill up parameters needed
        """
    def addcolfile(self, filename, tx=0, ty=0, tz=0):
        """
        read the data, add to master data
        """
    def compute_lab(self, selection=None):
        """
        Update the xl, yl, zl arrays based on current info
        for peaks in selection
        """
    def do_derivative(self, parlist, selection=None):
        """
        make (or update) dxldpar, dyldpar, dzldpar foreach par
        for peaks in selection
        """
    def change_pars(self, pardict):
        """
        Update parameters according to dict
        """
    def save(self, filename):
        """
        Dump myself on disk in a colfile with my current parameters
        """
    def load(self, filename):
        """
        Read myself from disk in a colfile and get current parameters
        """
```

◇

Fragment never referenced.

6 Grains - the model

For each grain we have the unit cell and orientation and centre of mass.

There are a variety of formats. The "UBI" formalism gives a 3x3 matrix that are the real space lattice vectors and JPW likes it for various reasons. Practically, we probably should plan for splitting into U and B and putting symmetry constraints on the unit cell. Or, eg, fixing the cell but refining the orientation.

Perhaps the underlying representation is not the key issue. Would be nice to have both.

Recycle the ImageD11.grain object. Get it to write and read gff files are just columnfiles with necessary columns.

Grain volume for intensity information will require structure factors and crystallography.

Each grain will have computed ubi and translation columns. Given a diffractometer, compute grain

centres of mass.

7 Peak assignments

Given peaks only we can try to guess at grains in various ways (the indexing problem, a lot of fun in itself). Given grains too we need to decide which peaks belong to which grains. Search is npeaks x ngrains and rather tedious.

Grain centric view - grain can potentially index many peaks. Has a list of which peaks it is "interested" in indexing based on various criteria. Might have some level of confidence about how probable it thinks a peak is owned (intensity matches others, position fits, other peak is preferred to this one).

Peak centric view - peak would like to be assigned to a grain. Indeed it actually wants to link up with a group of other peaks who agree about the orientation matrix and translation. It might belong to several cliques in the case of a twinned crystal.

Grains might link to some kind of crystallography to use space group absences when they are known. Twinning operations should also be available.

In terms of coding this is the more fiddly area. Need dynamic arrays and to avoid slow algorithms.

7.1 Matching criteria

A lot of debate about what is best.

From the data we will have detector coordinates, xl, yl and zl and could have derivatives and weights (sigs, sigf and sigomega). Given the grain translation centre of mass and beam direction we can compute the scattering vectors, the k (omega rotated) and g (omega=0) vectors. From the grains (and diffractometer omega) we have grain origins as a function of tx, ty and tz.

Given a list of g-vectors we can simply average them to get the UBI matrix. We could equally get into refinement with B fixed and only U varying. We can compute scattering vectors in crystal coordinates (g-vectors) given hkl indices and UBI matrix, also dg/dpar.

We should probably have a detector object and a diffractometer object. One to figure out the pixel geometry, another to do the rotations. Then we could refine one diffractometer with multiple detectors.

Up until now we did the matching figure of merit in crystal space, so applied the UBI to observed g-vectors and compared these to integers.

Equally we could project this into some other coordinates where various authors suggest life is better. For example, the detector face is popular (project into $d/d(fc), d/d(sc)$).

How to put it together? We will have a grain with ubi and translation. Also a table of all peaks. To be faster with many grains we should only do computations on the interesting subset of peaks.

We want a figure of merit and derivatives of the figure of merit with respect to the parameters in the model. Also we want an idea of the contribution of each peak to the fit.

$$h = UBI.g$$

$$gcalc = UB.h$$

Derivatives of gcalc w.r.t UB or even u and B should be easy.

Observations:

$$k = s1 - s0$$

$$s0 = incidentbeam = -1/\lambda, 0, 0$$

$$s1 = diffractedbeam = 1/\lambda(labvector)/|labveector|$$

$$labvector = xl - grainx, yl - grainy, zl - grainz$$

This matches equation 119 from the matrix cookbook, which is reproduced: http://orion.uwaterloo.ca/ hwolkow

$$\frac{\partial}{\partial x}\frac{x-a}{||x-a||_2} = \frac{I}{||x-a||_2} - \frac{(x-a)(x-a)^T}{||x-a||_2^3}$$

That last line looks to be a 3x3 output quantity, derivative of element i of result with respect to element j of input. So dk(xyz) by d(xyz)l.

So we have lab vectors fixed. We need to compute k and g relatively efficiently.

Given xl,yl,zl, omega (axis angle), ubi, translation. apply omega to translation compute k vectors apply omega to compute g vectors round to nearest hkls and get FOM score Perhaps normalise this (project) into detector coordinates

8 License etc

This code was written for the benefit of the ID11 user community. Anyone can use it and see what it does and how it works. If you modify the program, I ask that you make your code available as well so that I can see what you have done and so that a scientist using the program can still see how it works.

⟨ *copyright* 20 ⟩ ≡

```
# ImageD11 Software for beamline ID11
# Copyright (C) 2011  Jon Wright
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

◇

Fragment referenced in 6b.