JOURNAL OF APPLIED CRYSTALLOGRAPHY

Volume 53 (2020)

Supporting information for article:

Determination of the full deformation tensor by multi-Bragg fast scanning nano X-ray diffraction

Andreas Johannes, Jura Rensberg, Tilman A. Grünewald, Philipp Schöppe, Maurizio Ritzer, Martin Rosenthal, Carsten Ronning and Manfred Burghammer

Supporting information for

Determination of the full deformation tensor by multi-Bragg fast scanning nano X-ray diffraction

Authors

Andreas Johannes[a], Jura Rensberg[b], Tilman Grünewald[ca], Philipp Schöppe[b], Maurizio Ritzer[b], Martin Rosenthal[a], Carsten Ronning[b] and Manfred Burghammer[a]*

[a] European Synchrotron Radiation Facility, 71, avenue des Martyrs, Grenoble, 38043, France

[b]Institut für Festkörperphysik, Friedrich-Schiller-Universität Jena, Max-Wien-Platz 1, Jena, 07743, Germany

[c] Institut Fresnel, Avenue Escadrille Normandie Niemen, Marseille, 13013, France

Correspondence email: manfred.burghammer@esrf.fr

We provide supporting figures in the following sections.

1. Supporting Figures
2. Rotate Coordinates (jupyter notebook)
3. Deformation Tensor (jupyter notebook)
4. Count Projections (jupyter notebook)

1. Supporting Figures

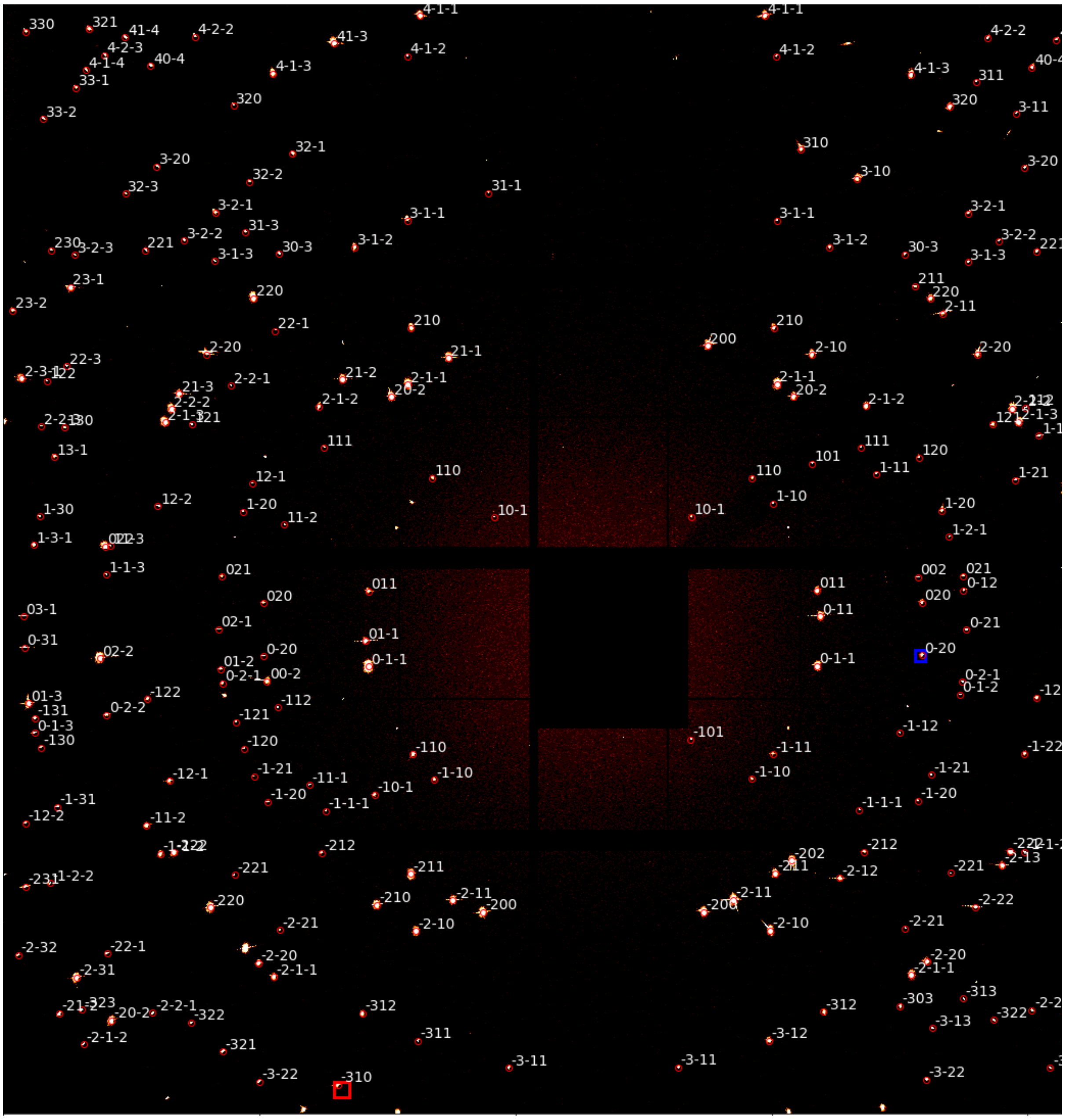


Figure S1 This figure shows the same data as presented in Figure 3 of the main manuscript with all indexed reflections being noted. The blue and the red squares show the regions of interest integrated for the calculation of the deformation tensor.

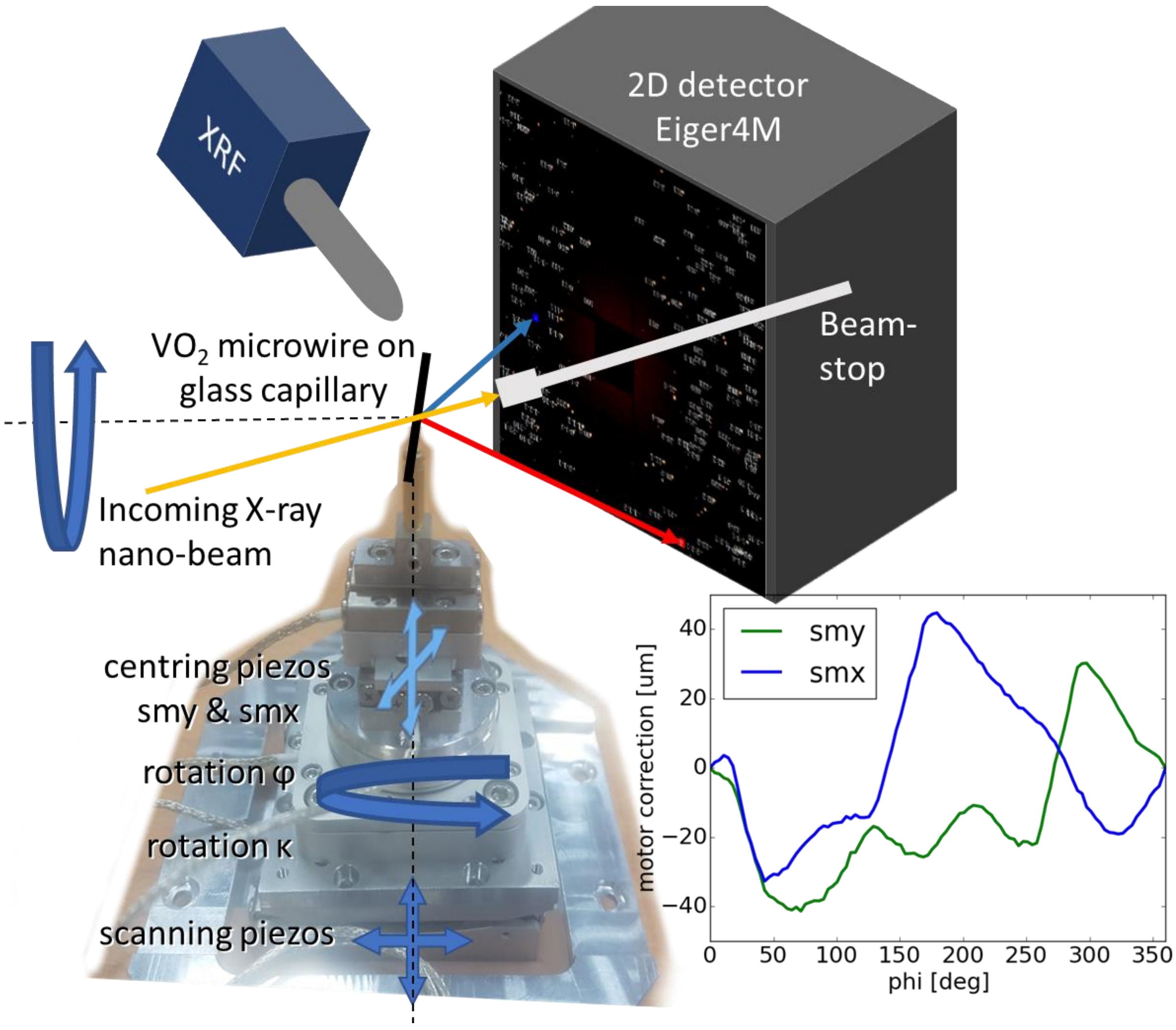
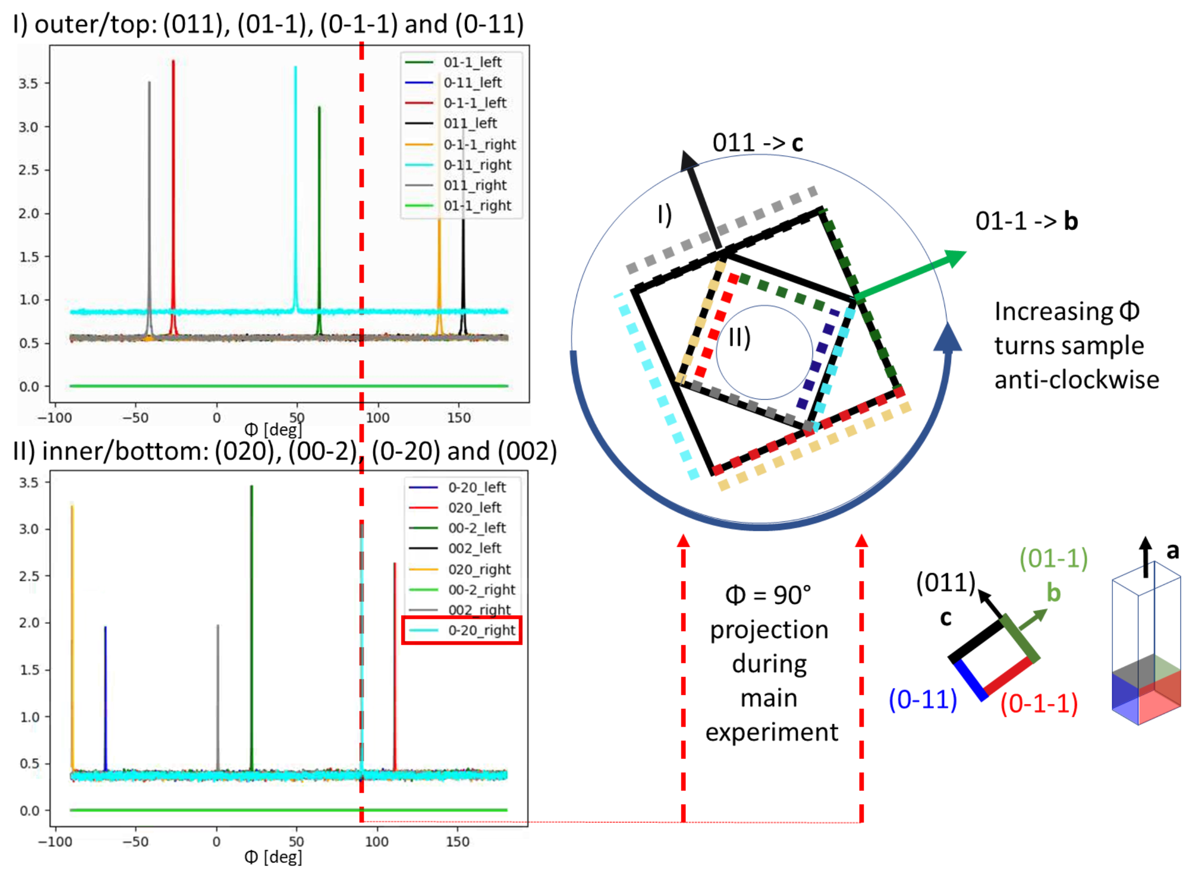


Figure S2 Sketch showing the measurement setup including a photo of the home assembled goniometer stage, the XRF detector and the 2D detector imaging two Bragg reflections. The goniometer is a stack starting with a xyz scanning piezo (PI-MARS) at the base; a cradle (SGO60.5) rotating the sample by κ against the vertical axis; a rotational stage (SR-2013) rotating around the vertical axis (when κ = 0); and ending with a set of crossed linear piezo motor stages (SLC-1720). The topmost piezos (smy and smx) are used to greatly reduce the sphere of confusion by following the trajectory plotted in the inset as a function of the rotation, which was obtained in an ex-situ microscopy setup (not shown).



The top left plot I) shows the integrated counts for the Bragg reflexes associated to the microwire facets as a function of the sample rotation angle φ. A schematic cross-section of the micro wire is shown to the right with the outer border accompanied by dashed lines according to the color used to plot the integrated counts for this respective reflex. Some reflexes appear twice as the sample is rotated, because the scattering geometry is mirror symmetric and the reflection can be imaged to the right or to the left on the detector as seen when following the incoming X-rays. The plot to the bottom left II) and the inner square in the cross-section to the right show the same plots for the reflexes corresponding to (020), (00-2), (0-20), (002). The scattering vector of these planes is equal to the sum of two facet scattering vectors. On the bottom right, a sketch shows a cross-section and the projection of the microwire volume with the facets colored accordingly. The dashed red arrows show projected view for the angle φ at which the full raster scan was made.

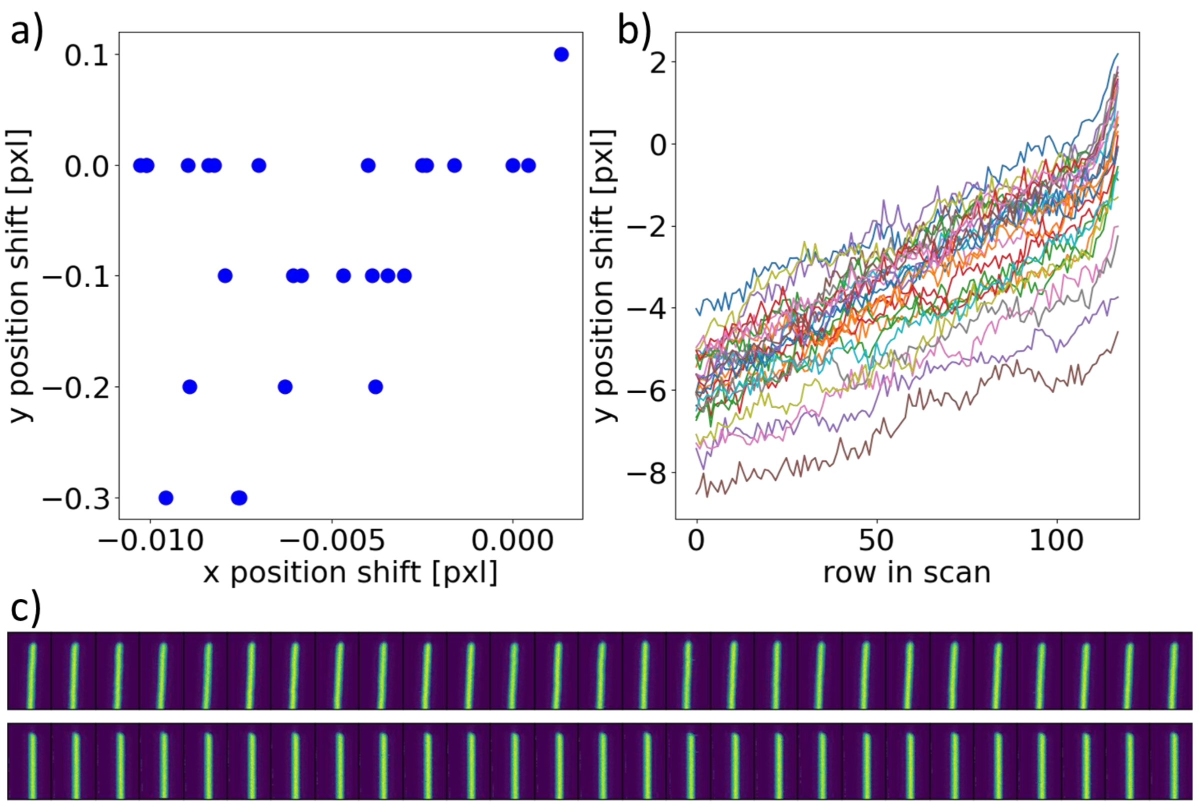


Figure S3 Plot a) is a scatter plot of the shift to the raster scan data, as applied to the whole set of raster scans. Plot b) shows the shift in y as applied to each individual row in the raster scan. The color plots in c) show the XRF intensity for all raster scans. The top row shows the original data as measured, while the bottom row represents the shifted data. All XRD data in the main text were shifted in this way.

2. Calculation of the transformation matrix from sample to laboratory coordinates

This script determines the transformation matrix $R = R_{xyzabc}$ to transform (rotate) the sample coordinates (abc) to laboratory (xyz) coordinates. I.e. the components of the vector expressed in laboratory coordinates (xyz) can be calculated from the components of the same vector expressed in sample (abc) coordinates:

$$a_i^{xyz} = (R_{xyzabc} a^{abc})_i$$

For tensors, this relation is expressed by:

$$\epsilon_{ij}^{xyz} = (R_{xyzabc}^T \epsilon^{abc} R_{xyzabc})_{ij}$$

In [1]:
```
# library imports
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
import h5py
import mpl_toolkits.mplot3d as m3d
import sys,os
import sympy
```

The following helper function determines all the vectors in reciprocal space that correspond to a given change in hkl. Using this, one gets more data points which correspond to a certain difference in hkl to get a better overall alignment.
i.e. (321) - (211) -> 110 The vector from the peak measured at (321) to the one at (211) can be used to also define the direction 110.

```
In [1]: def get_all_hkl_differences(hkl, hkl_dict, include_multiples=False,verbose=False):
    '''
    returns list of differences between q1 and q2 if (h1, k1, l1) - (h2, k2,l2)
=(h,k,l)
    '''
    return_list = []
    first = True
    running_sum=np.zeros(3,dtype=np.float32)
    hkl = np.asarray(hkl, dtype = np.float32)
    sacrificial_dict = dict(hkl_dict)
    for troiname in hkl_dict.keys():
        this_peak = sacrificial_dict.pop(troiname)
        for troiname, other_peak in sacrificial_dict.items():
            hkl_diff = np.asarray(other_peak['hkl_lr'][:3]-this_peak['hkl_lr'][:3],
dtype=np.float32)
            qxyz_diff = np.asarray(other_peak['qxyz'][:3]-this_peak['qxyz'][:3])

            # need to discard 0 indexes first:
            zero_index_list = np.arange(3)[np.where(hkl==0)]
            zero_check_passed = True
            for zindex in zero_index_list:
                if hkl_diff[zindex] != 0:
                    zero_check_passed = False

            if zero_check_passed:
                # check all indexes
                if include_multiples:
                    factor_list=range(-100,0)+range(1,100)
                else:
                    factor_list=[-1,1]
                checked_factor = [[qxyz_diff/factor, factor] for factor in factor_l
ist if list(hkl_diff/factor)==list(hkl)]

                if len(checked_factor)==1:
                    # catch outliers:
                    if not first:
                        if LA.norm(qxyz_diff-running_avg)<0.2:
                            result = checked_factor[0][0]
                            factor = checked_factor[0][1]
                            return_list.append(result)
                            running_sum += result
                            running_avg = running_sum/len(return_list)
                    else:
                        first = False
                        result = checked_factor[0][0]
                        factor = checked_factor[0][1]
                        return_list.append(result)
                        running_sum += result
                        running_avg = running_sum/len(return_list)

                    if verbose:
                        print('found q_diff {} at hkl: {} which is {} times {}'.for
mat(
                            '{:1.2f} {:1.2f} {:1.2f}'.format(*list(result)),
                            '{}{}{}'.format(*[int(x) for x in hkl_diff]),
                            factor,
                            '{}{}{}'.format(*[int(x) for x in hkl])))
                else:
                    pass
            else:
                pass
    return np.asarray(return_list)
```

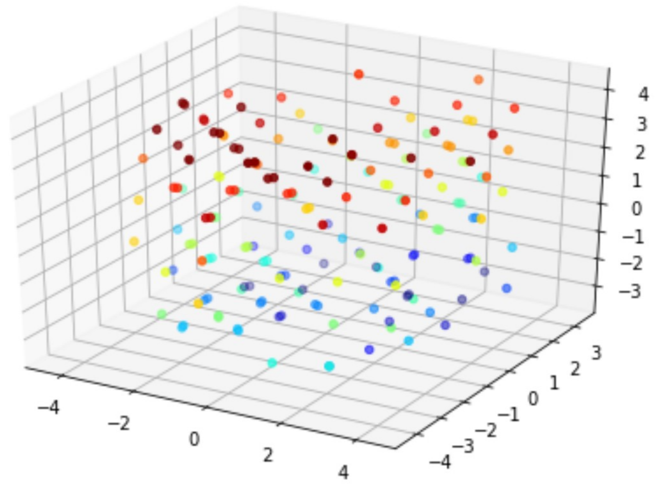
The input data hkl_dict was extracted from a rotation scan exposing the sample every 0.1 deg. We performed peak finding, indexing and transforming into the laboratory q_xyz coordinates. The data contains

- the hkl
- which side of the detector the peak was on
- q_xyz coordinates for all the detected peaks.

Next we plot all peaks in laboratory coordinates, scaled in inv. nm.

```
In [3]: dataset_todo = 'alignment'

        fname = '/data/id13/inhouse11/THEDATA_I11_1/d_2018-11-13_inh_ihma67_pre/PROCESS/hkl
        /fit_{}_hkl.h5'.format(dataset_todo)
        with h5py.File(fname,'r') as source_h5:
            hkl_g = source_h5['hkl_fit']
            peaks_dict={}
            qxyz_points = []
            hkl_points = []
            for troiname, troi_g in hkl_g.items():
                data_g = troi_g['data']
                fit_g = troi_g['fit']
                qxyz_point = np.asarray(fit_g['qxyz_data/fit3d_result'][:3])
                hkl_point = np.asarray(data_g['hkl_lr'])
                peaks_dict.update({troiname:{'troiname':troiname,
                                             'hkl_lr':hkl_point,
                                             'qxyz':qxyz_point}})
                qxyz_points.append(qxyz_point)
                hkl_points.append(hkl_point)

        hkl_points = np.asarray(hkl_points)
        qxyz_points = np.asarray(qxyz_points)
        ax = m3d.Axes3D(plt.figure())
        ax.scatter3D(*qxyz_points.T,c=2*hkl_points[:,0]-hkl_points[:,2], cmap='jet',vmax=8,
        vmin=-8)
```

```
Out[3]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f4746f3b290>
```

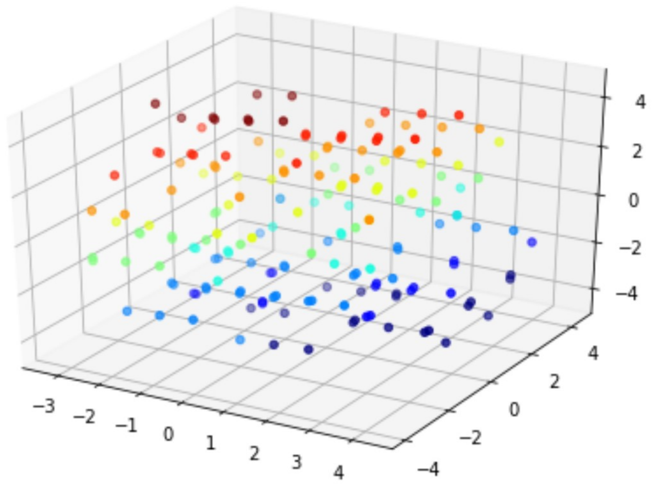


The function "get_rotation_matrix" is needed to calculate the rotation matrix that rotates a given vector **a** onto **b** in 3D.

In [4]:
```
def get_skew_symmetric(v):
    vx = np.zeros((3,3),dtype=np.float32)
    vx[1,0] = v[2]
    vx[0,1] = -v[2]
    vx[0,2] = v[1]
    vx[2,0] = -v[1]
    vx[1,2] = -v[0]
    vx[2,1] = v[0]
    return vx

def get_rotation_matrix(a,b):
    '''
    returns M so that Ma parallel to b
    https://math.stackexchange.com/questions/180418/calculate-rotation-matrix-to-al
ign-vector-a-to-vector-b-in-3d
    '''
    a = np.asarray(a)
    b = np.asarray(b)
    b_norm = b / LA.norm(b)
    a_norm = a / LA.norm(a)

    v = np.cross(a_norm,b_norm)
    c = np.dot(a_norm,b_norm)
    # cant have same or opposite directions
    assert(LA.norm(v)>0.0000001)
    vx = get_skew_symmetric(v)
    M = np.zeros((3,3),dtype=np.float32)
    M[0,0] = M[1,1] = M[2,2] = 1
    M += vx + np.dot(vx,vx)/(1+c)

    assert (LA.norm(np.dot(M,a_norm)-b_norm)<0.000001)
    return M
```

In [5]:
```
# find all vectors that are in line with 01-1 or 011
# use the mean to get a better R
b = [0,1,0]
c = [0,0,1]
qxyz_b = [0,1,-1]
qxyz_c = [0,1,1]
qxyz_a = [2,0,-1] # approx, not used!

qxyz_diff_c = get_all_hkl_differences(qxyz_c,peaks_dict, include_multiples=True, ve
rbose=False)
qxyz_diff_b = get_all_hkl_differences(qxyz_b,peaks_dict, include_multiples=True, ve
rbose=False)

q01m1_m = qxyz_diff_b.mean(axis=0)
q011_m = qxyz_diff_c.mean(axis=0)
# orthogonalize:
q011_m = q011_m - np.dot(q011_m,q01m1_m)/(LA.norm(q01m1_m)**2)*q01m1_m

# Mo rotates q01m1_m onto b
M0 = get_rotation_matrix(q01m1_m,b)
# M1 rotates np.dot(M0,q011_m) onto c.
# By construction (b orthogonal to c) and (q01m1_m orthogonal to q011_m)
# this rotation doesn't change np.dot(M0,q01m1_m) (now parallel to b)
M1 = get_rotation_matrix(np.dot(M0,q011_m),c)
# summarize the two rotations as R
R =  np.dot(M1,M0)

print('check result:')
print('a = R (dot) (q01m1_m x q011_m) : [{:2.4f}, {:2.4f}, {:2.4f}]'.format(*np.cro
ss(np.dot(R,q01m1_m), np.dot(R,q011_m))))
print('b = R (dot) q01m1_m:             [{:2.4f}, {:2.4f}, {:2.4f}]'.format(*np.dot
(R,q01m1_m)))
print('c = R (dot) q011_m:              [{:2.4f}, {:2.4f}, {:2.4f}]'.format(*np.dot
(R,q011_m)))
# check the resulting vectors are indeed orthogonal:
assert(LA.norm(np.dot(np.dot(R,q01m1_m), np.dot(R,q011_m)))<0.00001)

hkl_points = np.asarray(hkl_points)
abc_points = np.asarray([np.dot(R,qxyz_point) for qxyz_point in qxyz_points])
ax = m3d.Axes3D(plt.figure())
ax.scatter3D(*abc_points.T,c=hkl_points[:,1]+hkl_points[:,2], cmap='jet',vmax=4, vm
in=-4)
print('rotation matrix R aligns facets (abc) to laboritory coordinates (xyz):')
print(R)
```

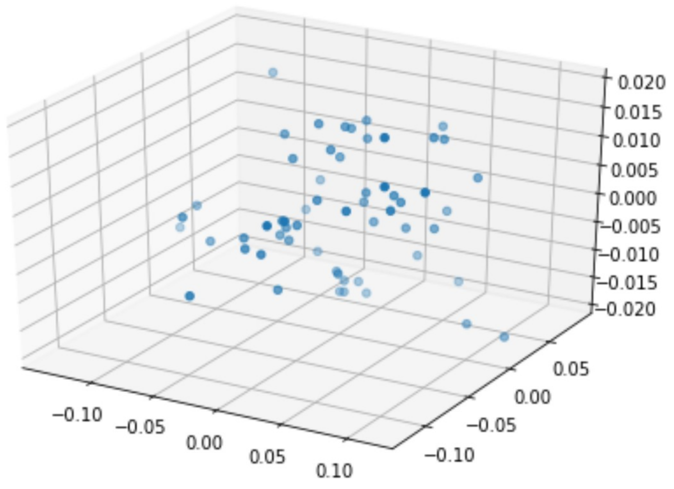
```
check result:
a = R (dot) (q01m1_m x q011_m) : [3.8748, -0.0000, -0.0000]
b = R (dot) q01m1_m:             [0.0000, 1.9761, -0.0000]
c = R (dot) q011_m:              [0.0000, -0.0000, 1.9609]
rotation matrix R aligns facets (abc) to laboritory coordinates (xyz):
[[-0.04893713 -0.16695799  0.9847488 ]
 [ 0.83148366 -0.55306786 -0.05244857]
 [ 0.55338955  0.81623584  0.16588841]]
```



Next, we plot the spread of the coordinates used to calculate the c_{mean} vector in xyz coordinates. This shows the accuracy of the diffractometer over the whole covered angular range. The axes are scaled in inv. nm.

```
In [6]: ax = m3d.Axes3D(plt.figure())
        mean_vec = qxyz_diff_c.mean(axis=0)
        ax.scatter3D(*(qxyz_diff_c-mean_vec).T)
        # ax.set_xlim(-0.02,0.02)
```

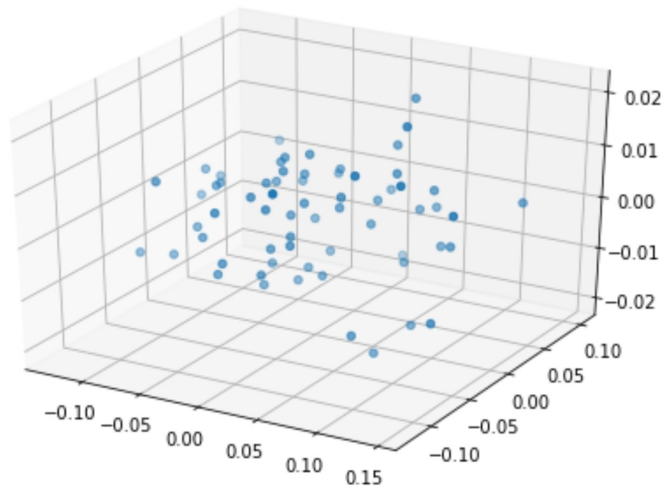
```
Out[6]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f4744d3d850>
```



We also plot the spread of the coordinates used to calculate the b_{mean} vector in xyz coordinates.

In [7]:
```
ax = m3d.Axes3D(plt.figure())
mean_vec = qxyz_diff_b.mean(axis=0)
ax.scatter3D(*(qxyz_diff_b-mean_vec).T)
```

Out[7]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f47443a6190>
```



3. Calculate the full deformation tensor from multiple Bragg reflections

This script determines a set of equations for the 6 unknown components of the deformation tensor. The starting point is analyzed data from scanned nano-diffraction data of at least two Bragg reflections measured simultaneously, i.e. in the same projection of the sample. An additional constraint is required when exactly two reflexes are measured. For this, we choose material incompressibility. Finally, the equations are solved generally and the solution applied iteratively to all measured data points.

In [1]:
```python
import numpy as np
import sympy as sympy
sympy.init_printing(use_unicode=True) # pretty printing sympy
import h5py # to read h5 data files
```

3.1 read data from processed .h5 files:

The required data was acquired by calculating the center of mass of the scattered diffraction vector in the laboratory reference frame. In spherical coordinates:
'qio' denotes the scattering vector $\mathbf{q} = (d, q_\phi, q_\theta)$ in the spherical coordinates of
d = lattice spacing along scattering vector;
ia = ϕ, in-xy-plane angle;
ia, oa = θ, out-of-plane angle.

Or cartesian coordinates:
'qxyz' is the scattering vector $\mathbf{q} = (q_x, q_y, q_z)$ in the lab coordinates x, y, z
'_r' for 'red' troi of the Bragg peak (3-10)
'_b' for 'blue' troi of the Bragg peak (020)

In [2]:
```python
rot_fname = '/data/id13/inhouse11/THEDATA_I11_1/d_2018-11-13_inh_ihma67_pre/PROCESS
/previews/alignment/rot_fit_bin3_q23_qxyz_kmap_rocking_merged.h5'
with h5py.File(rot_fname,'r') as source_h5:
    ori_g = source_h5['results/original_data']
    data_dict = {}
    for troiname, troi_g in ori_g.items():
        qxyz = np.asarray(troi_g['analytical/qxyz_mean'])
        dio = np.asarray(source_h5['results/strain/{}/dio_mean'.format(troiname)])
        data_dict.update({troiname: {'qxyz':qxyz,
                                     'dio':dio}})

ia_b = data_dict['blue']['dio'][1]
oa_b = data_dict['blue']['dio'][2]
ia_r = data_dict['red']['dio'][1]
oa_r = data_dict['red']['dio'][2]
print('ia_b = {:2.1f} deg'.format(ia_b*180/np.pi))
print('ia_r = {:2.1f} deg'.format(ia_r*180/np.pi))
print('oa_b = {:2.1f} deg'.format(oa_b*180/np.pi))
print('oa_r = {:2.1f} deg'.format(oa_r*180/np.pi))
```

```
ia_b = -100.5 deg
ia_r = 118.9 deg
oa_b = 5.4 deg
oa_r = 56.7 deg
```

We used an internally referenced deformation tensor, so that:

$$\epsilon_{dd} = (d_{ij} - d_{mean})/d_{mean},$$
$$\epsilon_{\phi\phi} = ia_{ij} - ia_{mean},$$
$$\epsilon_{\kappa\kappa} = oa_{ij} - oa_{mean},$$

where i,j indicate the indexes of the raster scanned map and ϵ is expressed in spherical coordinates. For small deformations, these values correspond to the respective cartesian components appropriately aligned with the Bragg reflection. We (arbitrarily) choose to align **y'** along the direction of the Bragg reflection, **x'** along **ia** (ϕ) and **z'** along **oa** (κ). Then:

$$\epsilon_{y'y'} = \epsilon_{dd},$$
$$\epsilon_{x'y'} = \epsilon_{\phi\phi},$$
$$\epsilon_{y'z'} = \epsilon_{\kappa\kappa}.$$

Remembering the construction for two linearly independent Bragg reflections, we can solve for the other 3 components $(e_{x'x'}, e_{z'z'}, e_{x'z'})$ by combing the two partially known deformation tensors if we first express them in a mutual coordinate system. We choose (again arbitrarily) the lab coordinates x, y, z. To arrive at these coordinates it is sensible to first rotate around the z-axis by ϕ (rotz) and then around the x-axis by κ (rotx). By construction, this rotation will ensure that respective angular components (in-plane and out-of-plane) are in fact aligned correctly.
As a reminder from the data-analysis code:
in_plane = np.arctan2(qx,qy); out_plane = np.arcsin(qz/q_3d)

3.2 calculate Rotation matrix R

Generally, if the components of vectors transform as $\mathbf{q'} = R\mathbf{q}$, tensor components transform as $\epsilon' = R^T \epsilon R$ when changing the coordinate base. As noted above, we can calculate the rotation matrix R directly for each Bragg vector:

In [3]:
```
az = sympy.Symbol('az')
ax = sympy.Symbol('ax')

rotz = sympy.Matrix([[sympy.cos(az), -sympy.sin(az), 0],
                     [sympy.sin(az),  sympy.cos(az), 0],
                     [0,              0             , 1]])
rotx = sympy.Matrix([[1, 0,             0],
                     [0, sympy.cos(ax), -sympy.sin(ax)],
                     [0, sympy.sin(ax),  sympy.cos(ax)]])

R = sympy.MatMul(rotx,rotz).doit()
assert(sympy.simplify(R.det())==1) # asserts R is a rotation matrix
R
```

Out[3]:

$$\begin{bmatrix} \cos(az) & -\sin(az) & 0 \\ \sin(az)\cos(ax) & \cos(ax)\cos(az) & -\sin(ax) \\ \sin(ax)\sin(az) & \sin(ax)\cos(az) & \cos(ax) \end{bmatrix}$$

We can double-check this result by applying it to the original scattering vector expressed in the laboratory coordinates. We find that the rotated vector is indeed aligned with the laboratory y-axis for both Bragg-peaks ('red' and 'blue').

In [4]:
```
qxyz_r = sympy.Matrix(data_dict['red']['qxyz'])
qrot_r = R*qxyz_r
qxyz_b = sympy.Matrix(data_dict['blue']['qxyz'])
qrot_b = R*qxyz_b
np.dot(qxyz_r.T,qxyz_b)


angle = np.arccos(float(np.dot(qxyz_r.T,qxyz_b)/(qxyz_r.norm()*qxyz_b.norm())))
print('Angle between the two scattering vectors = {:3.2f} deg'.format(angle*180/np.
pi))

print('Rotate both to the laboritory y axis:')
qrot_r.replace(az,ia_r).replace(ax,oa_r), qrot_b.replace(az,ia_b).replace(ax,oa_b)
```

```
Angle between the two scattering vectors = 110.14 deg
Rotate both to the laboritory y axis:
```

Out[4]:

$$\left(\left[\begin{matrix}-2.46030708028755 \cdot 10^{-8}\\4.12675848260819\\-6.5857470588071 \cdot 10^{-9}\end{matrix}\right], \ \left[\begin{matrix}3.98669780521388 \cdot 10^{-9}\\2.73494772985233\\-6.34500724183695 \cdot 10^{-9}\end{matrix}\right]\right)$$

3.3 Tensor coordinate transformation

Now one can setup the coordinates of the 3 ϵ tensors in their respective cartesian representation. The variable convention in sympy was chosen so that edr and edb read as ϵ'_{red} and ϵ'_{blue}, respectively.

In [5]:
```
e_xx, e_yy, e_zz, e_xy, e_xz, e_yz = sympy.symbols('e_xx e_yy e_zz e_xy e_xz e_yz')
edr_xx, edr_yy, edr_zz, edr_xy, edr_xz, edr_yz = sympy.symbols('edr_xx edr_yy edr_z
z edr_xy edr_xz edr_yz')
edb_xx, edb_yy, edb_zz, edb_xy, edb_xz, edb_yz = sympy.symbols('edb_xx edb_yy edb_z
z edb_xy edb_xz edb_yz')

eps = sympy.Matrix([[e_xx, e_xy, e_xz],
                    [e_xy, e_yy, e_yz],
                    [e_xz, e_yz, e_zz]])

edr = sympy.Matrix([[edr_xx, edr_xy, edr_xz],
                    [edr_xy, edr_yy, edr_yz],
                    [edr_xz, edr_yz, edr_zz]])
edb = sympy.Matrix([[edb_xx, edb_xy, edb_xz],
                    [edb_xy, edb_yy, edb_yz],
                    [edb_xz, edb_yz, edb_zz]])
eps, edr, edb
```

Out[5]:

$$\left(\left[\begin{matrix}e_{xx} & e_{xy} & e_{xz}\\e_{xy} & e_{yy} & e_{yz}\\e_{xz} & e_{yz} & e_{zz}\end{matrix}\right], \ \left[\begin{matrix}edr_{xx} & edr_{xy} & edr_{xz}\\edr_{xy} & edr_{yy} & edr_{yz}\\edr_{xz} & edr_{yz} & edr_{zz}\end{matrix}\right], \ \left[\begin{matrix}edb_{xx} & edb_{xy} & edb_{xz}\\edb_{xy} & edb_{yy} & edb_{yz}\\edb_{xz} & edb_{yz} & edb_{zz}\end{matrix}\right]\right)$$

Now perform the coordinate transforms of the Tensor components and equate:

$$\epsilon_{ij} = (R^T_{red} \cdot \epsilon'_{red} \cdot R_{red})_{ij} = (R^T_{blue} \cdot \epsilon'_{blue} \cdot R_{blue})_{ij}$$

The next three boxes perform this operation. We arrive at the equation zero_eq in [8].

In [6]:
```
eps_r = sympy.simplify(sympy.MatMul(R.T,(sympy.MatMul(edr,R))).doit())
eps_b = sympy.simplify(sympy.MatMul(R.T,(sympy.MatMul(edb,R))).doit())
```

We can verify that these are symmetric:

```
In [7]: eps_r.is_symmetric(), eps_b.is_symmetric()

Out[7]: (True, True)
```

Inserting the concrete values for the rotation angles:

```
In [8]: eps_br = eps_b.replace(az,ia_b).replace(ax,oa_b)
        eps_rr = eps_r.replace(az,ia_r).replace(ax,oa_r)
        zero_eq = sympy.simplify(eps_br-eps_rr)
```

3.4 calculating the full deformation tensor

The components of these two tensors in laboratory coordinates have to be equal (zero*eq): $$(\epsilon*{blue} - \epsilon*{red})*{ij} = 0$$ Note: non-dashed = expressed in the mutual laboratory coordinates. All deformation tensors are symmetric, so for the case of two peaks, we have 6 known components and 6 unknown components.

The known components are:

$$\begin{aligned}&\epsilon'_{b,yy}, \epsilon'_{b,xy}, \epsilon'_{b,yz}, \epsilon'_{r,yy}, \epsilon'_{r,xy}, \epsilon'_{r,yz}\\ &= \epsilon'_{b,qq}, \epsilon'_{b,\phi\phi}, \epsilon'_{b,\theta\theta}, \epsilon'_{r,qq}, \epsilon'_{r,\phi\phi}, \epsilon'_{r,\theta\theta}\end{aligned}$$

whereas the unknown components are:

$$\epsilon'_{b,xx}, \epsilon'_{b,zz}, \epsilon'_{b,xz}, \epsilon'_{r,xx}, \epsilon'_{r,zz}, \epsilon'_{r,xz}$$

However, one can show that apart from numerical errors, we only have five linearly independent equations. This can be understood by referring to the dashed coordinate systems. We lack information on the normal deformation in the direction perpendicular to the plane in which the two scattering vectors lie, while the four measured shear components reduce to three. This becomes most apparent in the case where the scattering vectors are perpendicular and two measured shear components are identical. Please see Figure 2 in the main text, which shows a sketch illustrating this point.

side note: number of equations

Here we will test the system of equations to show that it is under-determined.

```
In [9]: bx,bz = sympy.symbols('bx bz')
        eps_1 = eps_b.replace(az, bz).replace(ax,bx)
        eps_2 = eps_r

        T = eps_1 - eps_2
        eq_list = [T[0,0],T[1,0],T[2,0],T[1,1],T[2,1],T[2,2]]
        known_list = [edb_yy, edb_xy, edb_yz, edr_yy, edr_xy, edr_yz]
        unknown_list = [edb_xx, edb_zz, edb_xz, edr_xx, edr_xz, edr_zz]
        eq_A, eq_b = sympy.linear_eq_to_matrix(eq_list, unknown_list)
```

We have 6 equations for 6 unknown variables. The determinant of the Matrix corresponding to the system of equations, eq_A, is too cumbersome to show explicitly that it is zero for all angles. However, inserting a set of random values for the angles reveals it is zero for all tested combinations. Equivalent to this, solving the first 5 equations and solving manually for the 6th also returns a zero coefficient for the last unknown (bar rounding):

```
In [10]: NO_ITERATIONS = 3
         for i in range(NO_ITERATIONS):
             angle_list = [ax, az, bx, bz]
             angleval_array = 0.01 + np.random.random((NO_ITERATIONS,4))*(0.98*np.pi)
             eq_replace = list(eq_list)

             print('angles: {:2.2f}, {:2.2f}, {:2.2f}, {:2.2f}'.format(*angleval_array[i]))
             for angle, angleval in zip(angle_list, angleval_array[i]):
                 eq_replace = [eq.replace(angle,angleval) for eq in eq_replace]
             result = sympy.solve(eq_replace[:5], unknown_list[:])
             eq_A_replace, _ = sympy.linear_eq_to_matrix(eq_replace, unknown_list)
             eq6 = eq_replace[5]
             for key, val in result.items():
                 eq6 = eq6.replace(key, val)
             # eq6.subs([[x,y] for x,y in zip(known_list,[1]*6)])
             print(' -> det(eq_A_replace) = {}'.format(eq_A_replace.det()))
             print(' -> coefficient for {}: {}'.format(unknown_list[-1],eq6.coeff(unknown_li
         st[-1])))
```

```
angles: 1.41, 1.41, 2.95, 0.17
 -> det(eq_A_replace) = 7.82139544480764E-18
 -> coefficient for edr_zz: -2.91433543964104E-16
angles: 3.06, 2.48, 1.08, 1.94
 -> det(eq_A_replace) = -1.19837096338108E-16
 -> coefficient for edr_zz: 2.70894418008538E-14
angles: 2.36, 0.96, 2.94, 1.89
 -> det(eq_A_replace) = -6.23535066454246E-17
 -> coefficient for edr_zz: 0
```

As stated, this means we have only 5 linearly independent equations and will require additional constraints which can be imposed assuming incompressibility of the material. We can express this constraint in the 'red' and 'blue' components respectively:

$$\epsilon_{r,xx} + \epsilon_{r,yy} + \epsilon_{r,zz} = 0$$
$$\epsilon_{b,xx} + \epsilon_{b,yy} + \epsilon_{b,zz} = 0$$

Note, that these two equations are also not linearly independent, because incompressibility is obviously preserved under the rotation of coordinates. We could include only one of the two constraints. However, including both is a good additional constraint for the following fit.

We return to the equations with the experimental values for the angles and setup the system of equations to be solved:

```
In [11]: eq_list = [zero_eq[0,0],zero_eq[1,0],zero_eq[2,0],
                    zero_eq[1,1],zero_eq[2,1],zero_eq[2,2]]
         # incompressible expressed in 'red' coefficients
         eq_list.append(sympy.Eq(edr_xx + edr_yy + edr_zz))
         # incompressible expressed in 'blue' coefficients
         eq_list.append(sympy.Eq(edb_xx + edb_yy + edb_zz))
         eq_A, eq_b = sympy.linear_eq_to_matrix(eq_list, unknown_list)
```

For the case where more than two Bragg reflections were measured, the incompressibility constraint may be dropped. However, it often presents a valid approximation and it is advised to include it to further constrain the following fitting solution. If more than two reflections are available, the components of the respective deformation tensors can be equated pair-wise. Again, each Bragg reflection adds three knowns, three unkowns and three linearly independent equations. But, comparing the six components of each deformation tensor pair-wise quickly inflates the redundant number of equations.

3.5 Solving for unknown variables

We have for 6 unknows and 8 equations (more if more Bragg peaks were measured). This overdetermined system is incompatible, because it contains measurement errors. However, we can find the best solution by linear least squares using the QR decomposition [Trefethen, Lloyd; Bau, III, David (1997). Numerical Linear Algebra. ISBN 978-0898713619]:

$$Ax = b,$$
$$QRx = b,$$
$$x = R^{-1}Q^T b.$$

As we can trace the origin of each of the sets of 3 'unknowns' to 3 lineary independent equations of 3 measured terms (3 per Bragg peak), the colum space of A is equal to the number of unknown terms and a unique solution for this problem exists.

In [12]:
```
eq_x = eq_A.QRsolve(eq_b)
```

This solution is used to express the 'unknown' terms in 'known' terms:

In [13]:
```
eq_list2 = list(eq_x[i]-unknown_list[i] for i in range(len(unknown_list)))
eq_result = sympy.solve(eq_list2, unknown_list)
for key, val in eq_result.items():
    print('\n{} = '.format(key))
    print(val)
```

```
edb_xx =
-0.214815141087155*edb_xy - 0.940148908280475*edb_yy + 0.143640184979592*edb_yz
+ 0.22210890510513*edr_xy - 0.646846915334418*edr_yy + 0.847273581140151*edr_yz

edr_xz =
0.161781035685503*edb_xy - 1.06342841477801*edb_yy + 0.128722210513361*edb_yz -
0.396310468065141*edr_xy - 0.429614052466772*edr_yy + 0.383965540462569*edr_yz

edb_zz =
0.24256846940889*edb_xy - 0.032413105279403*edb_yy - 0.0741800351889031*edb_yz -
0.171826191672083*edr_xy + 0.619408928894296*edr_yy - 0.902650547215308*edr_yz

edr_xx =
-1.02093060206441*edb_xy - 0.145130469977867*edb_yy + 0.302290033867263*edb_yz +
0.156926633983404*edr_xy - 0.499175423885163*edr_yy + 0.370318090575313*edr_yz

edb_xz =
-0.278067513858006*edb_xy + 0.406409163503369*edb_yy + 0.0204914743239191*edb_yz
+ 0.854738716957513*edr_xy + 0.627244588699843*edr_yy + 0.20689775418145*edr_yz

edr_zz =
0.993177273742675*edb_xy + 0.117692483537746*edb_yy - 0.371750183657952*edb_yz -
0.20720934741645*edr_xy - 0.473386589674715*edr_yy - 0.314941124500156*edr_yz
```

We substitute all the unkowns by knows in the deformation tensor ϵ:

```
In [14]: eps_eq = sympy.Eq(eps,eps_rr,evaluate=False)
         eps_eq_dict = sympy.solve(eps_eq)
         eps_dict = {}
         eps_replace = eps
         for eps_symbol, eps_comp in eps_eq_dict.items():
             eps_comp_subs = eps_comp.subs(eq_result)
             eps_dict.update({eps_symbol: eps_comp_subs})
             print('\n{} = '.format(eps_symbol))
             print(eps_comp_subs)
```

```
e_yy =
-0.506787778872053*edb_xy - 0.843591616808998*edb_yy + 0.26222099637709*edb_yz +
0.271286800603049*edr_xy - 0.69298643782485*edr_yy + 0.718157204868533*edr_yz

e_yz =
-0.298050471717625*edb_xy + 0.485587164644313*edb_yy + 0.0204882228156621*edb_yz
+ 0.96823746923152*edr_xy + 0.533393817518826*edr_yy + 0.0763823015440065*edr_yz

e_zz =
0.299881028528728*edb_xy + 0.035536196756093*edb_yy - 0.112246655635794*edb_yz -
0.0625650162028488*edr_xy + 0.555124047223651*edr_yy - 1.01329335740492*edr_yz

e_xx =
0.17915342202159*edb_xy + 0.780617433612784*edb_yy - 0.219434490531986*edb_yz -
0.259004497833247*edr_xy + 0.165300377041321*edr_yy + 0.350513118611545*edr_yz

e_xy =
-0.797047649967784*edb_xy + 0.377917865631124*edb_yy + 0.180198285378315*edb_yz
+ 0.0110230032240591*edr_xy - 0.00753469290138578*edr_yy - 0.309879908172543*edr
_yz

e_xz =
0.356335576105857*edb_xy + 0.329520508132902*edb_yy - 0.18360724611488*edb_yz +
0.425370902934432*edr_xy - 0.478308915584187*edr_yy - 0.575365188507923*edr_yz
```

Now we one can iterate over all the positions in the raster scan and calculate the full deformation tensor in laboratory coordinates at each point. Finally, we want to rotate the coordinate system in which we express the deformation tensor to align with the crystal facets (abc) by applying

$$\epsilon_{ij}^{abc} = (R_{abcxyz}^T \cdot \epsilon^{xyz} \cdot R_{abcxyz})_{ij}.$$

The superscript denotes the respective coordinate system the components are expressed in. In chapter 2 of this supplementary information one can find the calculation of R_{xyzabc}. We are looking to rotate the other way around xyz -> abc, which is done by $R_{abcxyz} = R_{xyzabc}.T$:

```
In [15]: R_xyzabc = sympy.Matrix([[-0.04893713, -0.16695799,  0.9847488 ],
                                  [ 0.83148366, -0.55306786, -0.05244857],
                                  [ 0.55338955,  0.81623584,  0.16588841]])
         R_abcxyz = R_xyzabc.T
         R_abcxyz
```

Out[15]:

$$\begin{bmatrix} -0.04893713 & 0.83148366 & 0.55338955 \\ -0.16695799 & -0.55306786 & 0.81623584 \\ 0.9847488 & -0.05244857 & 0.16588841 \end{bmatrix}$$

The expressions for ϵ_{ij}^{abc} are then:

In [16]:
```
eps_abc = sympy.MatMul(R_abcxyz.T,sympy.MatMul(eps, R_abcxyz)).doit()
eps_abc[0]
```

Out[16]:
$$\begin{array}{l}0.0023948426926369e_{xx} + 0.0163408897223374e_{xy} - 0.096381560085888e_{xz} + 0.0278749704248 \\ + 0.96973019910144e_{zz}\end{array}$$

3.6 error estimation

The experimental accuracy for the 'known' components was estimated to be 1e-4, as described in the main text of the manuscript.

In [17]:
```
measurement_se = dict([(known, 1e-4) for known in known_list])
```

For the 'unknown' components, we can directly calculate the standard error for the fit [S. Sheather *A Modern Approach to Regression with R (Springer Texts in Statistics)* ISBN-10: 0387096078]:

In [18]:
```
eq_Q, eq_R = eq_A.QRdecomposition()
df = len(eq_b) - len(eq_x)
sigma2 = sympy.Add(*[j*j
                     for j in (eq_b-sympy.MatMul(eq_A,eq_x))[:]])/df
varbeta = sigma2*sympy.MatMul(eq_R.T,eq_R).inv().doit()
fit_se = dict([(unknown_list[i],sympy.sqrt(varbeta[i,i]))
               for i in range(varbeta.shape[0])])
fit_se[edb_xx]
```

Out[18]:
$$\sqrt{\begin{array}{l}0.583444831411397(-0.162056813649001edb_{xy} - 0.160215474118407edb_{yy} - 0.405590652773 \\ + 0.160215474118406edr_{yy} + 0.323356340099236edr_{yz})^2 \\ + 0.583444831411397(-0.0277533283217355edb_{xy} - 0.0274379864401222edb_{yy} - 0.06946014 \\ + 0.0274379864401216edr_{yy} + 0.0553769660751574edr_{yz})^2 \\ + 0.583444831411397(0.0277533283217352edb_{xy} + 0.0274379864401216edb_{yy} + 0.0694601497 \\ - 0.0274379864401217edr_{yy} - 0.0553769660751572edr_{yz})^2 \\ + 0.583444831411397(0.0470853851721249edb_{xy} + 0.0465503864943246edb_{yy} + 0.1178438084 \\ - 0.0465503864943248edr_{yy} - 0.0939507415862051edr_{yz})^2 \\ + 0.583444831411397(0.0594647718334059edb_{xy} + 0.0587891147438384edb_{yy} + 0.1488265447 \\ - 0.0587891147438381edr_{yy} - 0.118651666362717edr_{yz})^2 \\ + 0.583444831411397(0.0696337481522183edb_{xy} + 0.0688425480153704edb_{yy} + 0.1742771360 \\ - 0.0688425480153704edr_{yy} - 0.138942099643287edr_{yz})^2 \\ + 0.583444831411397(0.070936921307535edb_{xy} + 0.0701309141151118edb_{yy} + 0.17753867653 \\ - 0.0701309141151118edr_{yy} - 0.141542356260275edr_{yz})^2 \\ + 0.583444831411397(0.311327108333445edb_{xy} + 0.307789713646901edb_{yy} + 0.779179611470 \\ - 0.307789713646902edr_{yy} - 0.621199393333879edr_{yz})^2\end{array}}$$

"fit_se" is a dictionary for all 'unknown' variables and the corresponding (lengthy) terms of the standard error of the fit for each, as a function of the known components. This estimation ignores the fact that the known components are also error-laden, however, as these values are all equally accurate we find it instructive to keep these two contributions separate.

One can separately add up the contribution of each error source in terms in the final expression for ϵ:

```
In [19]: eps_eq = sympy.Eq(eps, eps_rr, evaluate=False)
         eps_eq_dict = sympy.solve(eps_eq)
         eps_error_dict = {'fit':{},'measurement':{}}



         for eps_symbol, eps_comp in eps_eq_dict.items():
             # the fit error
             # the contibution of 'unknown', a function of known_list:
             error_terms = [sympy.Abs(sympy.diff(eps_comp, unknown)*fit_se[unknown])
                            for unknown in unknown_list]
             eps_fit_error = sympy.Add(*error_terms)
             #in case there are unknown terms left:
             eps_fit_error.subs(eq_result)
             eps_error_dict['fit'].update({eps_symbol: eps_fit_error})

             # measurement error
             # eliminate the 'unknown' variables:
             eps_comp_replace = eps_comp.subs(eq_result)
             error_terms = [sympy.Abs(sympy.diff(eps_comp_replace, known)*measurement_se[kno
         wn])
                            for known in known_list]
             eps_measurement_error = sympy.Add(*error_terms)
             eps_error_dict['measurement'].update({eps_symbol: eps_measurement_error})

         #for eps_symbol, eps_comp in eps_error_dict['fit'].items():
         #    print(eps_symbol, eps_comp)
         #for eps_symbol, eps_comp in eps_error_dict['measurement'].items():
         #    print(eps_symbol, eps_comp)
```

As with the deformation tensor results, we iterate over all the points in a measured map, inserting the measured values into these equations. The results are plotted in the main publication.

4 Count number of projections for multi-Bragg XRD

In the following, we confirm that for a given crystal lattice there are many pairs of (hkl) planes that can be simultaneously brought into diffraction conditions.

Initializations

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d as m3d
import matplotlib.patches as mpatches

import numpy.linalg as LA
import sympy.utilities.iterables as sympy_iterables
```

Function definitions

The following function is used to construct a reciprocal lattice, saved in dictionary form.

In [2]:
```
def get_reciprocal_lattice(a_qvector, b_qvector, c_qvector,q_max=10, lattice=None):
    '''
    calculate 3D q_space coordinates for Bragg reflexes from given unit-cell vector
s
    up to |(qx, qy, qz| = qmax [in inverse nm]
    returns dict((h,k,l):(qx,qy,qz))
    follows selection rules: lattice can be None,'fcc','bcc','diamond'
    '''
    # quickly find some lower bound for max(h,k,l)
    # for very skew lattices this may fail to get all reflexes!
    a = np.asarray(a_qvector, dtype=np.float32)
    b = np.asarray(b_qvector, dtype=np.float32)
    c = np.asarray(c_qvector, dtype=np.float32)
    factors = np.array([-1,0,1])
    permutations = sympy_iterables.permutations(factors)
    min_length = 0.5*min(*[LA.norm(a*p1+b*p2+c*p3)/(abs(p1)+abs(p2)+abs(p3))
                           for [p1,p2,p3] in permutations])
    max_hkl = int(np.ceil(q_max/min_length))


    if lattice=='fcc':
        print('Applied {} selection rules'.format(lattice))
        def check_lattice(h,k,l):
            if h%2 + k%2 + l%2 == 0: # all even
                return True
            elif h%2 + k%2 + l%2 == 3: # all odd
                return True
            return False

    elif lattice=='bcc':
        print('Applied {} selection rules'.format(lattice))
        def check_lattice(h,k,l):
            if (h + k + l)%2 == 0: # sum even
                return True
            return False

    elif lattice=='diamond':
        print('Applied {} selection rules'.format(lattice))
        def check_lattice(h,k,l):
            if h%2 + k%2 + l%2 == 3: # all odd
                return True
            elif h%2 + k%2 + l%2 == 0: # all even
                if (h + k + l)%4 == 0: # diamond special case
                    return True
            return False
    else:
        # further selection rules can be inserted here
        print('No selection rules')
        def check_lattice(h,k,l):
            return True

    hkl_dict = {}

    for h in range(-max_hkl,max_hkl):
        for k in range(-max_hkl,max_hkl):
            for l in range(-max_hkl,max_hkl):
                q_vector = h*a + k*b + l*c
                if LA.norm(q_vector)<q_max:
                    if check_lattice(h,k,l):
                        hkl_dict.update({(h,k,l):q_vector})

    # we don't include the origin
    hkl_dict.pop((0,0,0))
    print('found {} reflections'.format(len(hkl_dict.keys())))

    return hkl_dict
```

Function to get rotation matrix, needed later:

```
In [3]: def get_skew_symmetric(v):
            vx = np.zeros((3,3),dtype=np.float32)
            vx[1,0] = v[2]
            vx[0,1] = -v[2]
            vx[0,2] = v[1]
            vx[2,0] = -v[1]
            vx[1,2] = -v[0]
            vx[2,1] = v[0]
            return vx

        def get_rotation_matrix(a,b):
            '''
            returns M so that Ma parallel to b
            https://math.stackexchange.com/questions/
            180418/calculate-rotation-matrix-to-align-vector-a-to-vector-b-in-3d
            '''
            a = np.asarray(a)
            b = np.asarray(b)
            b_norm = b / LA.norm(b)
            a_norm = a / LA.norm(a)

            v = np.cross(a_norm,b_norm)
            c = np.dot(a_norm,b_norm)
            # cant have same or opposite directions
            if LA.norm(v)<0.0000001:
                M = np.zeros((3,3),dtype=np.float32)
                M[0,0] = M[1,1] = M[2,2] = 1
                return M * np.sign(c)
            vx = get_skew_symmetric(v)
            M = np.zeros((3,3),dtype=np.float32)
            M[0,0] = M[1,1] = M[2,2] = 1
            M += vx + np.dot(vx,vx)/(1+c)

            assert (LA.norm(np.dot(M,a_norm)-b_norm)<0.00001)

            return M
```

4.1 Defining the experiment

A restriction on which reflections can be imaged is given in the laboratory geometry and layout. Let us assume that we can image all reflections in the experimental setup up to $q_{max} = 2\pi sin(\Theta_{max})/\lambda$. This restriction is defined by the maximal scattering angle $2\Theta_{max} < 180^\circ$ to which the detector can be moved. A relatively harsh restriction is $2\Theta_{max} < 15^\circ$.

The number of Bragg reflections that will appear within this detectable cone is critically dependent on the X-ray energy. We choose the experimental value $\lambda = 0.082nm$ (15 keV).

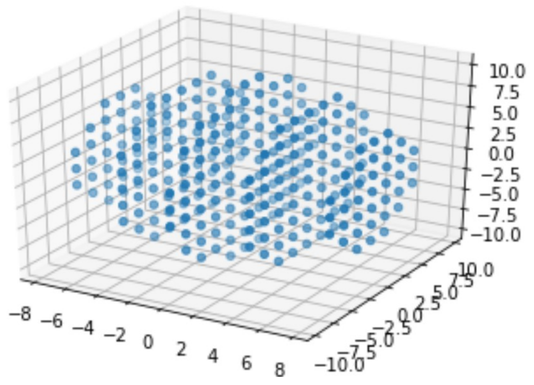
```
In [4]: THETA_MAX = 0.5*15./180*np.pi
        LAMBDA = 8.2e-2
        Q_IN = 2.0*np.pi/LAMBDA
        Q_MAX = 2.0*np.pi*np.sin(THETA_MAX)/LAMBDA
```

We will continue with the cubic approximation of the VO_2 lattice. Note again that the lattice symmetry, spacing, wavelength and maximum scattering angle all contribute very strongly to the number of reflections that can be imaged. Here we present a conservative estimation which represents the experimental setup, as described in the main text of the manuscript.

```
In [5]: # in HKL_DICT we collect all Bragg reflections up to q_max.
        HKL_DICT = get_reciprocal_lattice([3.87,0,0],
                                          [0,1.92,0],
                                          [0,0,1.96],
                                          q_max=Q_MAX, lattice='')
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        ax = fig.gca(projection='3d')
        qxyz_points = np.asarray(list(HKL_DICT.values()))
        ax.scatter3D(*qxyz_points.T)
```

```
No selection rules
found 300 reflections
```

```
Out[5]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1d7705e2978>
```



4.2 Definition of check criteria

The check for simultaneous scattering works as an Ewald sphere construction with a sphere radius = $q_{in} = 2\pi/\lambda$. This radius is strictly larger than q_{max} from before so that one can image all peaks in hkl_dict that was already defined.

For a given Bragg reflection\, the scattering vector $q_{\Delta} = q_{in} - q_{out}$ can be used to restrict the position of the center of the Ewald's sphere in reciprocal space. We have one axis of rotational freedom, because sample rotations around the scattering vector preserve the diffraction condition. Experimentally, this rotation corresponds to rotating the sample in such a way that the reflection remains visible at all times. During the rotation the scattered beam will trace a cone around the direct beam, while the center of the Ewald's sphere traces a circle in 3D reciprocal space.

To check whether a partner reflection can be simultaneously imaged, we need to verify that it will also lie on the Ewald's sphere at some point along the rotation. The partner peak needs to cross the surface of the Ewald's sphere to be on it at any time. This is equivalent to checking whether the partner peak enters and leaves the Ewald's sphere as the Ewald's sphere is rotated around the scattering vector. In other words, the partner peak must be in the union volume of all Ewald's spheres and NOT in their conjunction.

- check whether partner peak is in the volume created by rotating the Ewald's sphere around the scattering vector (union) .
- check that partner peak is NOT in the inner volume (conjunction). Here it never leaves the Ewald's sphere as the sphere rotates.

We will perform the calculation by rotating the coordinates in hkl_dict, so that the scattering vector (identified by the popped hkl) points along the x-axis. Next, the remaining partner peaks are in turn rotated around the x-axis into the xy-plane. This reduces the 3D problem to 2D. Now, the partner peak is valid:

- union
  - If the rotated coordinates ($q'_x, q'_y, q'_z = 0$) are in the Ewald's sphere (now circle).
- conjunction
  - It is not in the rotated volume of the negative circular segment (where y<0) (->never leaves Ewald's sphere).
  - This corresponds to qy' not being in the Ewald's sphere mirrored along the x-axis (rotated by 180°).

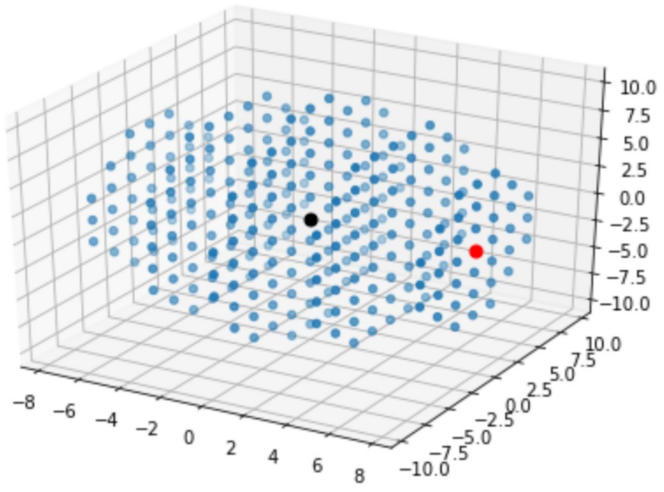
4.3 Example for one pair:

The following selects a reference and partner peak at random and plots the selection process:

- rotate all reflexes so that the reference is parallel to the x-axis
- rotate partner reflex into xy-plane
- check which area the partner peak is in

In [6]:
```
# choose random example reference peak and rotate coordinates to x-axis
# example that fails if partner_hkl = (1,0,0) is reference_hkl = (2,0,0)

pairs = []
hkl_dict = dict(HKL_DICT)
i = np.random.randint(len(hkl_dict.keys()))
reference_hkl = list(hkl_dict.keys())[i]
reference_hkl = (2,0,0)
reference = hkl_dict.pop(reference_hkl)
R = get_rotation_matrix(reference,[1,0,0])
reference = np.dot(R, reference)

# plot all peaks in the rotated coordinates
qxyz_points = np.asarray(list(hkl_dict.values()))
abc_points = np.asarray([np.dot(R,qxyz_point)
                         for qxyz_point in qxyz_points])
ax = m3d.Axes3D(plt.figure())
ax.scatter3D(*abc_points.T)
ax.scatter3D(*reference,c='r',s=50)
ax.scatter3D(0,0,0,c='black',s=50)
```

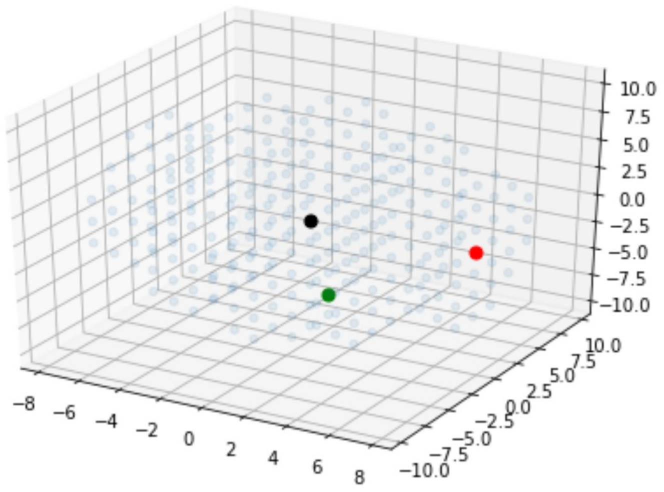
Out[6]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1d7706bad68>
```

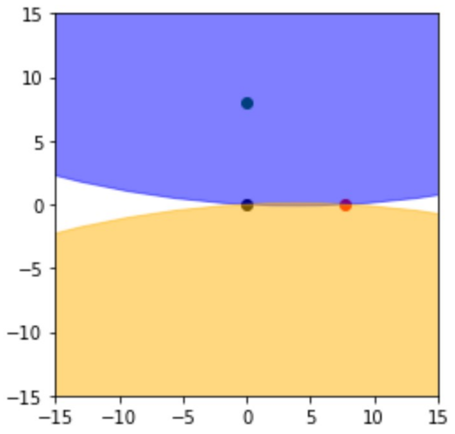


We rotate around the x-axis and plot in the xy-plane.

In [7]:
```python
# select random partner example
# example that fails if reference_hkl = (2,0,0) is partner_hkl = (1,0,0)
i = np.random.randint(len(hkl_dict.keys()))
partner_hkl = list(hkl_dict.keys())[i]
# rotate partner coordinates to x-axis
partner = np.dot(R,hkl_dict[partner_hkl])

# plot all reflexes in reference coordinates
ax = m3d.Axes3D(plt.figure())
ax.scatter3D(*abc_points.T,alpha=0.1)
ax.scatter3D(*reference,c='r',s=50)
ax.scatter3D(0,0,0,c='black',s=50)
ax.scatter3D(*partner,c='green',s=50)

# rotated partner to xy-plane
phi = np.arctan2(partner[2],partner[1])
R2 = [[1,0,0],[0,np.cos(phi),np.sin(phi)],[0,-np.sin(phi),np.cos(phi)]]
partner = np.dot(R2,partner)

# plot reflexes in xy-plane
fig, ax = plt.subplots(1)
ax.set_aspect('equal')
ax.scatter(0,0,color='black',marker='o')
ax.scatter(*reference[:2],color='red',marker='o')
ax.scatter(*partner[:2],color='green',marker='o')

# calculates Ewald's spheres parameters
ewald_x = 0.5*reference[0]
ewald_y = np.cos(ewald_x/Q_IN)*Q_IN
ewald_coord = [ewald_x, ewald_y]
ewald_mirror = [ewald_x, -ewald_y]

# add Ewald's spheres to plot
circle_in = mpatches.Circle(ewald_coord, Q_IN, color =[0,0,1,0.5])
ax.add_patch(circle_in)
circle_out = mpatches.Circle(ewald_mirror, Q_IN, color =[1,0.7,0,0.5])
ax.add_patch(circle_out)
ax.set_xlim((-15,15))
ax.set_ylim((-15,15))

# define function to check whether partner peak is in the valid area of the plot
def cuts_ewald(reference, q_in, partner):
    ewald_x = 0.5*reference[0]
    ewald_y = np.cos(0.5*reference[0]/q_in)*q_in
    ewald_coord = [ewald_x, ewald_y]
    ewald_mirror = [ewald_x, -ewald_y]
    check = False
    if LA.norm(partner-ewald_coord)<q_in:
        check=True
    if LA.norm(partner-ewald_mirror)<q_in:
        check=False
    return check

# result for this pair
cuts_ewald(reference[:2],Q_IN, partner[:2])
```

```
Out[7]: True
```





The origin, reference and partner points are plotted above in black, red and green. In the second plot above the initial slice of the Ewald's sphere is shown in blue. The Ewald's sphere mirrored along the x-axis is shown in orange. Thus the the conjunction of all Ewald's spheres is where these two circles overlap.

The pair of reflexes can be brought into reflection condition at the same time if the green dot is in the blue area and not in the orange area.

4.4 All pairs

We iterate over all pairs of reflections and call the cuts_ewald function to keep the ones which can be simultaneously excited:

```
In [8]: hkl_dict = dict(HKL_DICT)
        pairs_list = []
        not_valid = []
        hkl_list = list(hkl_dict.keys())
        N_PAIRS = (len(HKL_DICT.keys())*(len(HKL_DICT.keys())+1))/2

        i=0
        for reference_hkl in hkl_list:

            reference = hkl_dict.pop(reference_hkl)
            R = get_rotation_matrix(reference,[1,0,0])
            # rotate the reference to x-axis
            reference = np.dot(R, reference)

            for partner_hkl in hkl_dict.keys():
                i+=1
                # rotate partner with reference
                partner = np.copy(np.dot(R,hkl_dict[partner_hkl]))
                # rotate partner around reference into xy-plane
                phi = np.arctan2(partner[2],partner[1])
                R2 = [[1,0,0],
                      [0,np.cos(phi),np.sin(phi)],
                      [0,-np.sin(phi),np.cos(phi)]]
                partner = np.dot(R2,partner)

                # find ewald sphere coordinates
                ewald_x = 0.5*reference[0]
                ewald_y = np.cos(ewald_x/Q_IN)*Q_IN
                ewald_coord = [ewald_x, ewald_y]
                ewald_mirror = [ewald_x, -ewald_y]

                # performn check and save original coordinates
                if cuts_ewald(reference[:2], Q_IN, partner[:2]):
                    pairs_list.append(((reference_hkl, HKL_DICT[reference_hkl]),
                                       (partner_hkl, HKL_DICT[partner_hkl])))
                    not_valid.append(((reference_hkl, HKL_DICT[reference_hkl]),
                                       (partner_hkl, HKL_DICT[partner_hkl])))


        print('Iterated over {} pairs.'.format(i))
        print('Found {} simulatneously reflecting pairs = {:2.1f}% of all pairs.'.format(le
        n(pairs_list),
                                                                                         10
        0.0*len(pairs_list)/i))
```

```
Iterated over 44850 pairs.
Found 44520 simulatneously reflecting pairs = 99.3% of all pairs.
```

4.5 Conclusion

For large X-ray energies, the Ewald's sphere is so large that most reflexes can be combined. The curvature of the sphere so low, so that almost all the reflections are accessible. Only those pairs pointed along a mutual axis are excluded. In practice not all combinations will be achievable, because it may be impossible to rotate the sample into position and the intersection of the partner reflection with the Ewald's sphere may be too shallow.

However, each reflection crosses the Ewald's sphere twice, doubling the number of suitable projections. For small samples and suitable goniometers, it is possible to arrive at many sample orientations which show more than one Bragg reflection simultaneously. Therefore, it is possible to measure sufficient projections to allow the full 3D reconstruction of the deformation tensor in the sample.