Appendix A

List of Parameters

Table A.1: This table gives the parameters for UO_2 generating the energy function.

Array number	Parameter name	Parameter value
1	Energy Scaling Factor (e_{RGB})	1.6012 J/m^2
2	$\langle 100 \rangle$ Max Distance	0.405
3	$\langle 110 \rangle$ Max Distance	0.739
4	$\langle 111 \rangle$ Max Distance	0.352
5	$\langle 100 \rangle$ Weight	50.5
6	$\langle 110 \rangle$ Weight	4.55
7	$\langle 111 \rangle$ Weight	0.08
8	$\langle 100 \rangle$ Tilt/Twist Mix Power Law (1)	0.03325
9	$\langle 100 \rangle$ Tilt/Twist Mix Power Law (2)	0.00053125
10	Maximum $\langle 100 \rangle$ Twist Energy	0.60903
11	$\langle 100 \rangle$ Twist Shape Factor	1.4486
12	$\langle 100 \rangle$ Asymmetric Tilt Interpolation Power	35.8
13	$\langle 100 \rangle$ Symmetric Tilt First Peak Energy	1.0058
14	$\langle 100 \rangle$ Symmetric Tilt First $\Sigma 5$ Energy	0.84456
15	$\langle 100 \rangle$ Symmetric Tilt Second Peak Energy	0.97259
16	$\langle 100 \rangle$ Symmetric Tilt Second $\Sigma 5$ Energy	0.9379
17	$\langle 100 \rangle$ Symmetric Tilt $\Sigma 17$ Energy	0.96881
18	$\langle 100 \rangle$ Symmetric Tilt First Peak Angle	0.31569
19	$\langle 100 \rangle$ Symmetric Tilt Second Peak Angle	0.88538

Continued on next page.

Table A.1 – *Continued from previous page*

Array number	Parameter name	Parameter value
20	$\langle 110\rangle$ Tilt/Twist Mix Power Law (1)	3.1573
21	$\langle 110\rangle$ Tilt/Twist Mix Power Law (2)	1.9784
22	$\langle 110\rangle$ Twist Peak Angle	0.46145
23	$\langle 110\rangle$ Twist Peak Energy	1.1444
24	$\langle 110\rangle$ Twist $\Sigma 3$ Energy	1.0931
25	$\langle 110\rangle$ Twist 90° Energy	1.152
26	$\langle 110\rangle$ Asymmetric Tilt Shape Factor	3.1843
27	$\langle 110\rangle$ Symmetric Tilt Third Peak Energy	1.0514
28	$\langle 110\rangle$ Symmetric Tilt $\Sigma 3$ Energy	0.61703
29	$\langle 110\rangle$ Symmetric Tilt Second Peak Energy	1.0902
30	$\langle 110\rangle$ Symmetric Tilt $\Sigma 11$ Energy	0.56686
31	$\langle 110\rangle$ Symmetric Tilt First Peak Energy	1.1024
32	$\langle 110\rangle$ Symmetric Tilt Third Peak Angle	0.88736
33	$\langle 110\rangle$ Symmetric Tilt Second Peak Angle	1.8711
34	$\langle 110\rangle$ Symmetric Tilt First Peak Angle	2.731
35	$\langle 111\rangle$ Tilt-Twist Linear Interpolation	38.201
36	$\langle 111\rangle$ Twist Shape Factor	1.2414
37	$\langle 111\rangle$ Twist Peak Angle	0.49979
38	$\langle 111\rangle$ Twist Peak Energy	0.7971
39	$\langle 111\rangle$ Symmetric Tilt Peak Angle	0.25966
40	$\langle 111\rangle$ Symmetric Tilt Max Energy	1.0288
41	$\langle 111\rangle$ Symmetric Tilt $\Sigma 3$ Energy	1.1311
42	$\langle 111\rangle$ Asymmetric Tilt Symmetry Point Energy	3.7674
43	$\langle 111\rangle$ Asymmetric Tilt Scale Factor	0.053417

Appendix B

Grain Boundary Representations

Visual representations of the GB space helped Bulatov *et al.* develop their 5D function. However, the size of the five-space in which GBs reside makes representing them difficult. Researchers have developed different methods to represent them, each with their advantages and disadvantages. Three of these methods are the axis-angle representation, the Rodrigues representation, and the fundamental zone representation. These methods, though described separately, can be used together to form a better picture of what the GB space looks like (see for example **??** which combines the Rodrigues representation and the fundamental zone representation).

B.1 Axis-Angle Representation

Of the three described, the axis-angle representation most simplistically describes GB space. The axis of rotation of the GB specifies the point in axis-angle space, and the angle of misorientation between the two grains at the GB specifies the magnitude of the vector. Thus, the axis ($\boldsymbol{a}$, where $\boldsymbol{a}$ has components a_x, a_y, and a_z) and the angle (θ) mathematically represent an axis-angle vector as:

$$\boldsymbol{A} = \boldsymbol{a}\,\theta \tag{B.1}$$

The axis-angle space can only take into account three degrees of freedom: the two angles specifying the axis, and the angle rotated through. Thus, axis-angle space cannot fully visualize all of the necessary information contained in the full 5D space.[?] This representation suffers from the difficulties of understanding an infinite space because it maps an axis and an angle onto a Cartesian coordinate system. Without the help of additional methods, this infinite space remains difficuly to understand. The best uses of this representation focus on using it as a starting point to move to other, more robust representations, and to represent the misorientation between two grains.[?]

B.2 Rodrigues Representation

The Rodrigues representation (sometimes called the "Rodrigues-Frank" representation) uses Rodrigues vectors to represent rotations in Rodrigues space. This representation takes ideas from the axis-angle space, but makes a few changes allowing crystal symmetries to be taken into account. The orientation of the GB normal still specifies the point in space, but the tangent of half the angle represents the magnitude of the vector. Thus, a Rodrigues vector can be represented as:[?, ?, ?, ?, ?]

$$\boldsymbol{R} = \boldsymbol{a}\tan\left(\frac{\theta}{2}\right) \tag{B.2}$$

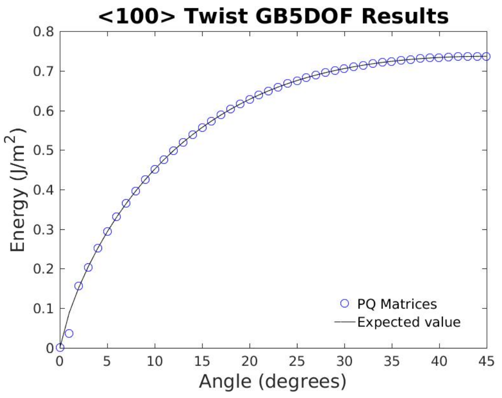
Some researchers favor this representation over others because of the lack of curvature such a mapping entails.[?, ?] However, it still only specifies three of the five degrees of freedom. Bulatov *et al.* attached a unit vector at the points along the axis to represent the other two DoFs in **??**. A parallel vector represents a twist boundary, and a perpendicular vector represents a tilt boundary. Anything else represents a mix of twist and tilt (or a mixed boundary). One limitation of Rodrigues space lies in that it also maps to an infinite space.[?, ?].

B.3 Fundamental Zone Representation

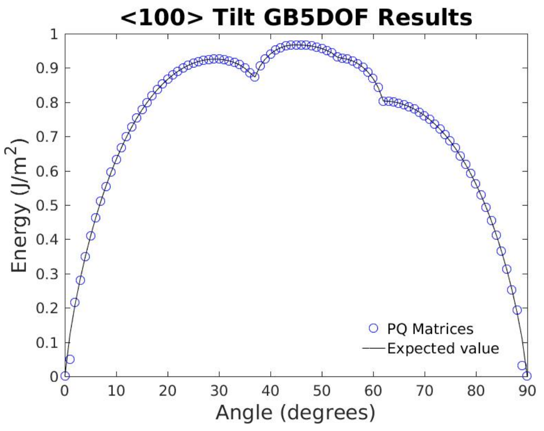
The fundamental zone best graphically represents the full 5D GB. This representation takes advantage of the symmetries inherent in crystals[?] to simplify an infinite space into a compact, finite area called the fundamental zone.[?, ?, ?, ?, ?] Every point within the space represents a unique orientation, and every point outside the space can be represented as a point inside the space through symmetry operations.[**?**, **?**, **?**] Bulatov *et al.* used this idea in connection with Rodrigues space to create **??**. In Rodrigues space, the crystal symmetries of the material determine the shape of the fundamental zone.[**?**, **?**] For fcc crystals, the fundamental zone takes the form of a truncated tetrahedron.[**?**] The edges of the fundamental zone in Rodrigues space represent the high-symmetry rotation axes, and points on one face can represent another point on a different face of the fundamental zone.

Appendix C

Graphs

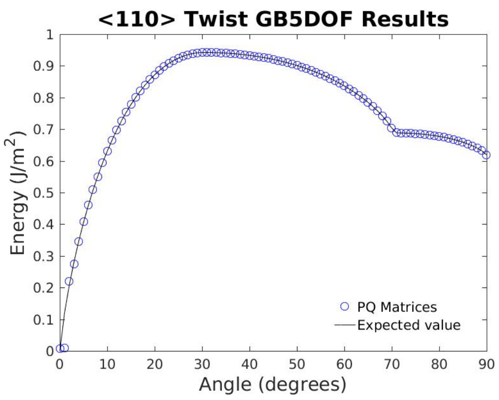


(a)

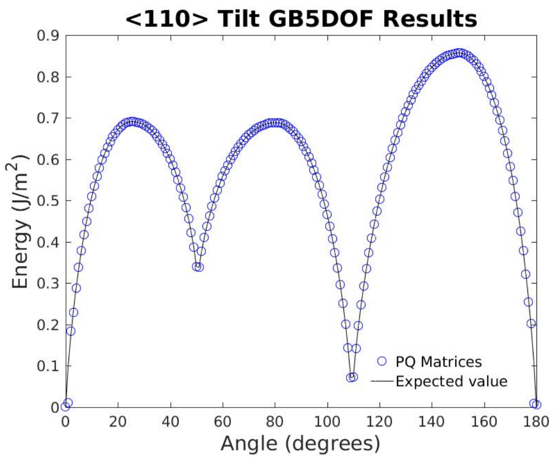


(b)

Figure C.1: The ⟨100⟩ twist (a) and tilt (b) results for the P and Q matrices as compared to Bulatov *et al.*'s energy profiles. Bulatov *et al.*'s GB5DOF .m MATLAB® script calculated the expected values by using the default parameters. The GB5DOF.m script calculated the values using the generated matrices. With the exception of the data points at 1° in both (a) and (b) and 89° in (b), the energies calculated from the matrices matches the expected curves exactly.

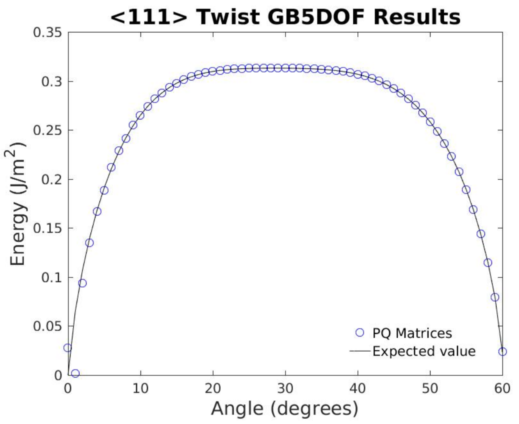


(a)

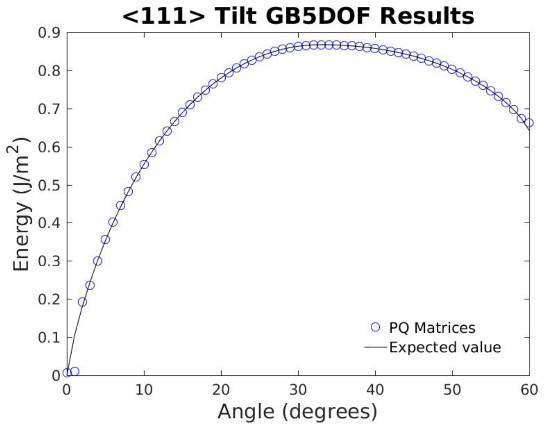


(b)

Figure C.2: The $\langle 110 \rangle$ twist (a) and tilt (b) results for the P and Q matrices as compared to Bulatov *et al.*'s energy profiles. Bulatov *et al.*'s GB5DOF .m MATLAB® script calculated the expected values by using the default parameters. The GB5DOF.m script calculated the values using the generated matrices. With the exception of the data points at 1° in both (a) and (b) and 179° in (b), the energies calculated from the matrices matches the expected curves exactly.

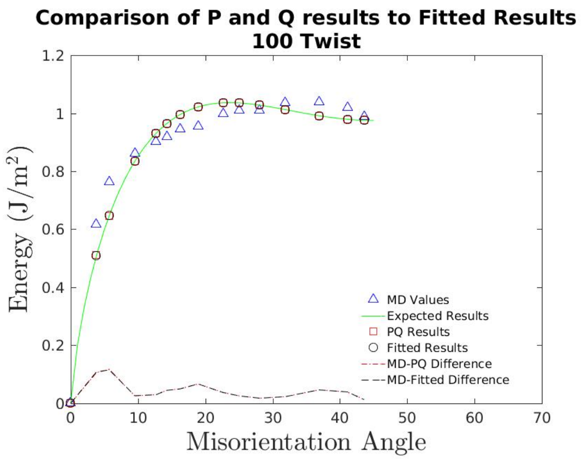


(a)

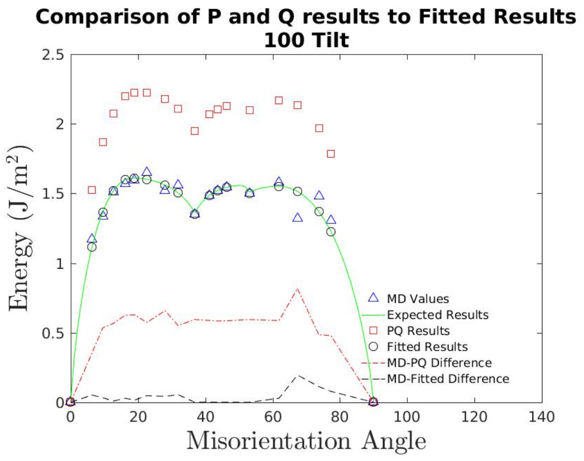


(b)

Figure C.3: The $\langle 111 \rangle$ twist (a) and tilt (b) results for the P and Q matrices as compared to Bulatov *et al.*'s energy profiles. Bulatov *et al.*'s GB5DOF .m MATLAB® script calculated the expected values by using the default parameters. The GB5DOF.m script calculated the values using the generated matrices. With the exception of the data points at 1° in both (a) and (b) and 60° in (b), the energies calculated from the matrices matches the expected curves exactly.

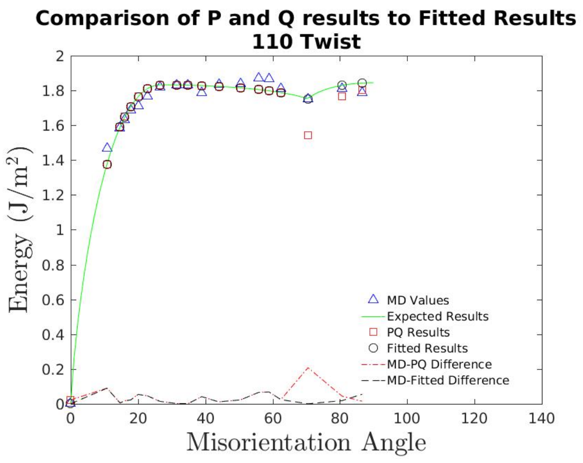


(a)

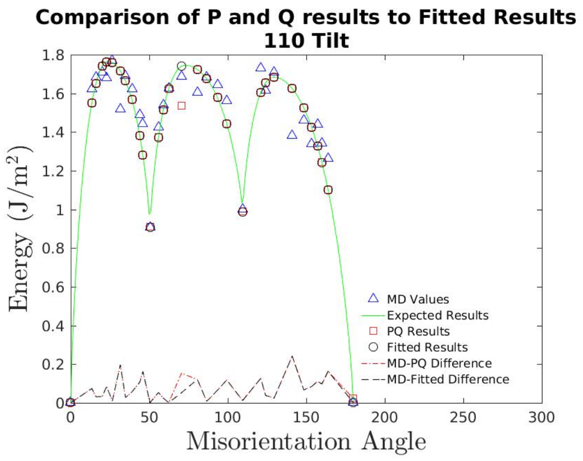


(b)

Figure C.4: A comparison of the expected value of the fitted function with the values calculated using the P and Q matrices for the ⟨100⟩ 1D subsets, with the MD values shown for reference. (a) PQ results follow exactly the fitted curve. (b) has a scaling issue yet to be fixed. The cause of the scaling issue remains unknown.

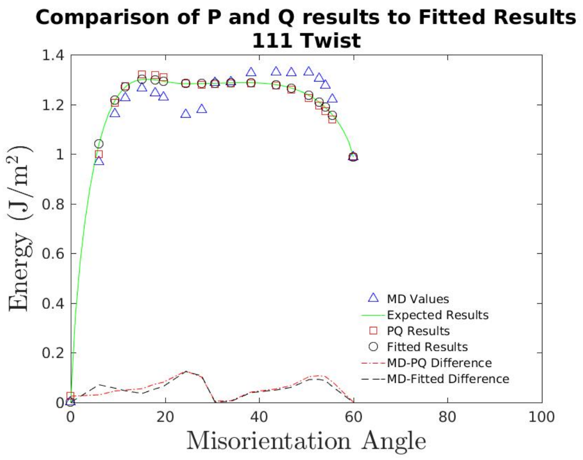


(a)

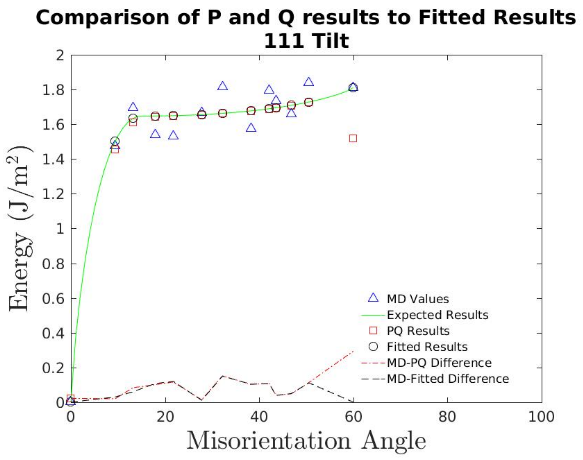


(b)

Figure C.5: A comparison of the expected value of the fitted function with the values calculated using the P and Q matrices for the $\langle 110 \rangle$ 1D subsets, with the MD values shown for reference. (a) follows the fitted result until the cusp, at which point some anomalies appear. The results from the PQ matrices dip well below the expected value at the cusp, and never make it back to the original fitted line. (b) has a similar issue on a lesser scale. Only two of the calculated points do not follow the fitted curve. At the endpoint the expected value is zero, but the PQ matrices calculated a value slightly higher. Also, an unexpected cusp from the PQ matrices appears in the middle of the second hump. All other data points follow the fitted curve exactly.



(a)



(b)

Figure C.6: A comparison of the expected value of the fitted function with the values calculated using the P and Q matrices for the $\langle 111 \rangle$ 1D subsets, with the MD values shown for reference. (a) closely follows the expected fitted values, but has a slight error throughout. (b) follows the expected values exactly in the center of the fitting, but misses slightly for lower angle boundaries, and misses completely at the end.

Appendix D

Orientation Matrix Generator

This code generates the orientation matrices (known as the P and Q matrices in Bulatov *et al.*'s code). Provision for calculating the matrices one of two ways appears in-code through the use of command-line options.

```
from __future__ import division,print_function # To
    avoid numerical problems with division, and for ease
      of printing
from sys import argv # for CLI arguments
from math import cos, sin, pi, atan2, sqrt # Trig
    functions
from os.path import exists # For checking existence of
    a file
from numpy import array, linalg
from myModules import * # imports my functions from the
      file myModules.py

# Helper functions
def displayHelp():
    print('''
    This script will calculate the orientation matrices
        for any given misorientation
    for any of the high-symmetry axes.
    Arguments:

        _axis: The axis of orientation (type: int)
    _misorientation: The angle of misorientation (
        type: float)
         ——————OR——————
    (with option -e or --euler)
    _z1: The first rotation angle  (Z ) (type:
        float)
    _x:  The second rotation angle (X') (type:
        float)
    _z2: The third rotation angle  (Z") (type:
        float)

If the option -e or --euler-angles is entered, the
    calculation skips to simply
output the orientation matrices.  Otherwise, the
    Euler angles are calculated from
the axis, orientation, and grain boundary normal,
    and then the orientation matrix is
created through the use of the Rodrigues Rotation
    Formula, which is:
R = I + sin(theta) * K + (1 - cos(theta))*K^2
where I is the identity matrix, theta is the
    misorientation angle, and K is
the skew-symmetric matrix formed by the axis of
    rotation:
K = 0  -kz   ky
     kz   0  -kx
   -ky   kx   0
where the vector k is the unit vector defining the
    axis of rotation, or using
a set of predefined rotations for each axis (
    default is the predefined rotations).
The Euler angles are calculated in this case simply
     for the file to be written
to.  If the user does not specify to save, then the
     angles are not used for
anything.

Options:
```

```
-e --euler <_z1> <_x> <_z2>                  Returns the
   Bunge orientation matrix
                                             based on the
                                                euler angles
                                                 provided.

-f --file <filename>                         Reads the file
   filename and uses the
                                             Euler angles
                                                from them to
                                                 calculate
                                                the
                                             orientation
                                                matrix.

--rrf                                        Calculates the
   matrices using the Rodrigues
                                             Rotation
                                                Formula

-a --angles                                  Displays the
   Euler angles.  Can be used
                                             in conjunction
                                                with -q or
                                                --quiet to
                                             display only
                                                the Euler
                                                angles.

-s --save                                    Saves the
   resultant orientation matrix to
                                             a database (
                                                orientation_matrix_database
                                                .m)
                                             with the
                                                accompanying
                                                 Euler
                                                angles.
```

```
    -q --quiet                              Suppresses
        output of the orientation matrices
                                            to the terminal

    --help                                  Displays this
        help info

    Output:
    For an Euler angle set, the ouput is simply its
        orientation matrix.
    For the misorientations, the first matrix is the 'P
        ' orientation matrix, and
    the second matrix is the 'Q' orientation matrix (
        see Bulatov et al., Acta Mater
    65 (2014) 161-175).
    ''')
    return

def displayAngles(z1, x, z2): # Displays an Euler angle
    set (Bunge convention)
    print("Euler angles:")
    # This is the "new" way to format strings.  The 16
        indicates the padding to
    # be done before the next character.  The '<'
        character below says which side
    # to pad (the right side).
    print("{:16}{:16}{:16}".format('Z', 'X', 'Z'))
    print("----------------------------------------------")
    print("{:<16}{:<16}{:<16}\n".format(rad2deg(z1),
        rad2deg(x), rad2deg(z2)))
    return

def check4RRF(args): # Check the args for the rrf
   command
    if "--rrf" in args:
        index = args.index("--rrf")
        del args[index]
        return True, args
    else:
        return False, args

def check4Euler(args): # Check the args for the -a or
   --angles command
    if "-a" in args or "--angles" in args:
        try:
            index = args.index("-a")
        except:
            index = args.index("--angles")
        del args[index]
        return True, args
    else:
        return False, args

# Write the matrix and angles to a file
def writeMat(m, _z1, _x, _z2, grain, axis):
    # This is to avoid issues with duplicates
    if _z1 == 0:
        _z1 = abs(_z1)
    if _x == 0:
        _x = abs(_x)
    if _z2 == 0:
        _z2 = abs(_z2)

    lastVal = 1
    # This is the default filename to be used.
    # TODO: make provisions to provide the database
        file via command line
    tex_filename = "orientation_matrix_database.m"
    var_name = "%s%d"%(grain, axis) # Will generally
        look like P100 or Q100
    if not exists(tex_filename):
        tex_file = open(tex_filename, "a")
        tex_file.write("%Database␣for␣orientation␣
            matrices␣for␣specified␣Euler␣Angles\n")
        tex_file.write("
          %————————————————————————————————————————————————————————
          n")
        tex_file.write("%Orientation␣Matrix␣␣␣␣␣␣␣␣␣␣␣␣
          ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Euler␣Angles\n")
        tex_file.write("%s(:,:,%d)=[%2.6f␣␣%2.6f␣␣%2.6f
          ␣␣␣␣␣␣␣␣␣␣␣%%%2.4f␣␣%2.4f␣␣%2.4f\n"%(
          var_name, lastVal, m[0][0], m[0][1], m
          [0][2], _z1, _x, _z2))
        tex_file.write("%2.6f␣␣%2.6f␣␣%2.6f\n"%(m
          [1][0], m[1][1], m[1][2]))
        tex_file.write("%2.6f␣␣%2.6f␣␣%2.6f];\n"%(m
          [2][0], m[2][1], m[2][2]))
        tex_file.write("
          %————————————————————————————————————————————————————————
          n")
        tex_file.close()
    else:
        f = open(tex_filename,"r")
        while True:
            data = f.readline().split()
            if not data:
                break
            elif len(data) != 6:
                continue
            else:
                assert data[0][0] in {'P', 'Q'}, "
                   Unknown␣orientation␣matrix␣type␣(
                   should␣be␣\'P\'␣or␣\'Q\')."
                if not "%d"%(axis) in data[0][1:4]:
                    lastVal = 0
                elif "%d"%(axis) in data[0][1:4]:
                    try:
                        try:
                            if data[0][0] == 'P': #
                               Handles anything 3
                               digits long
                lastVal = int(data
                    [0][9:12]) - 1
            else:
                lastVal = int(data
                    [0][9:12])
        except:
            if data[0][0] == 'P': #
               Handles anything 2
               digits long
                lastVal = int(data
                    [0][9:11]) - 1
            else: # data[0][0] == 'Q'
                lastVal = int(data
                    [0][9:11])
    except:
        if data[0][0] == 'P': # One
           digit case
            lastVal = int(data[0][9]) -
                1
        else: # data[0][0] == 'Q'
            lastVal = int(data[0][9])
else:
    print("Error: Unknown last index.")
    exit()

# Checks to see if the Euler angles
   have already been used before
# If so, the calculated matrix is not
   saved (assumed to already
# be in the database)
if data[0][0] == grain and data[3] == (
   '%' + "%2.4f"% z1) and data[4] == "
   %2.4f"% x and data[5] == "%2.4f"% z2
   :
    unique = False
    break
else:
    unique = True
```

```
            if unique:
                tex_file = open(tex_filename, "a")
                tex_file.write("%s(:,:,%d)=[%2.6f  %2.6f  
                   %2.6f           %%%2.4f  %2.4f  %2.4f\n"
                   %(var_name, lastVal + 1, m[0][0], m
                   [0][1], m[0][2], _z1, _x, _z2))
                tex_file.write("%2.6f  %2.6f  %2.6f\n"%(m
                   [1][0], m[1][1], m[1][2]))
                tex_file.write("%2.6f  %2.6f  %2.6f];\n"%(m
                   [2][0], m[2][1], m[2][2]))
                tex_file.write("
                   %-----------------------------------------------------------------------------
                   n")
                tex_file.close()
        return

if "--help" in argv: # Help info
        displayHelp()
        exit()

orientation_matrix = []
save, argv = check4Save(argv) # Save the file?  Delete
    the save argument
quiet, argv = check4Quiet(argv) # Checks for
    suppressing output. Delete the quiet argument.
useRRF, argv = check4RRF(argv) # Checks for using the
    RRF method. Delete the rrf argument.
dispEuler, argv = check4Euler(argv) # Checks for
    displaying the Euler angles.  Delete the angle
    argument

# If the arguments come from a file...
if "-f" in argv or "--file" in argv: #input arguments
    come from file
        try:
            try:
                index = argv.index("-f")
            except:
```

```
        index = argv.index("--file")
except:
    print("ERROR: Unable to find filename.")
    exit()
filename = argv[index + 1]
try:
    f1 = open(filename, 'r')
except:
    print("ERROR: Unable to read file.", filename)

while True: # Read the file line by line.
    line = f1.readline()
    # break if we don't read anything.  If there
       are blank lines in the
    # file, this will evaluate to TRUE!
    if not line:
        break;
    data = line.split()
    if len(data) != 4: # If there are less than 4
       parts to the data, move along (format of
       file MUST be _z1 _x _z2 1.00)
        continue
    else:
        # Convert the data to stuff we can use
        _z1 = float(data[0])
        _x  = float(data[1])
        _z2 = float(data[2])

        _z1 = deg2rad(_z1)
        _x  = deg2rad(_x)
        _z2 = deg2rad(_z2)
        orientation_matrix = calcRotMat(_z1, _x,
           _z2)
        if not quiet:
            displayMat(orientation_matrix)
        if save:
            writeMat(orientation_matrix, _z1, _x,
               _z2, 'P', _axis)
```

```
# Input is a set of euler angles
elif "-e" in argv or "--euler-angles" in argv:
    try:
        try:
            index = argv.index("-e")
        except:
            index = argv.index("--euler-angles")
    except:
        print("ERROR: Unable to read Euler angles.")
        exit()
    _z1 = float(argv[index + 1])
    _x =  float(argv[index + 2])
    _z2 = float(argv[index + 3])
    _z1 = deg2rad(_z1)
    _x  = deg2rad(_x)
    _z2 = deg2rad(_z2)

    orientation_matrix = calcRotMat(_z1, _x, _z2)

    if not quiet:
        displayMat(orientation_matrix)
    if save:
        writeMat(orientation_matrix, _z1, _x, _z2, 'P',
            _axis)

else:
    if len(argv) < 3:
        print("ERROR: Not enough command line arguments
           .")
        print("Input either an axis, and a
           misorientation, or a ZXZ Euler angle set
           with the option -e or --euler-angles.")
        displayHelp()
        exit()
    try:
        _axis = int(argv[1])
        _misorientation = float(argv[2])
    except:
        print('''
        ERROR: Command line argument(s) is (are) not of
            correct type.
        Please enter an int for argument 1, a float for
            argument 2, and an int for argument 3.
          ''')
        exit()

    if not len(str(_axis)) == 3: # axis length greater
      than 3
        print("ERROR: Argument 1 must by a 3 digit
          number like \'100\'.")
        exit()

    _misorientation = deg2rad(_misorientation) # Change
        input to radians
    axis = [None]*3
    _z1 = [None]*2
    _x = [None]*2
    _z2 = [None]*2
    q = [None]*2
    for i in range(0, len(str(_axis))):
        axis[i] = int(str(_axis)[i])

#
  ------------------------------------------------------------------------
#---------------------------------The Actual Calculations
  --------------------------#
#
  ------------------------------------------------------------------------

    # First convert to a quaternion
    # These functions are from a myModules.py.
    q[0] = axis2quat(axis, _misorientation / 2)
    q[1] = axis2quat(axis, -_misorientation / 2)
    # Convert the quaternion to Euler Angles
    for i in range(0, len(_z1)):
        _z1[i], _x[i], _z2[i] = quat2euler(q[i])

#
 ------------------------------------------------------------

    # Using the Rodrigues Rotation Formula, defined as
       R = I + sin(theta) * K + (1 - cos(theta))*K^2
    # with K = [0 -k_z, k_y; k_z, 0, -k_x; -k_y, k_x,
       0], and the components of
    # k coming from the vector being rotated about.
       Theta is specified by the misorientation.
    if useRRF:
        orientation_matrix1, orientation_matrix2 =
           calcRotMatRRF(axis, _misorientation)

        # Normalize the matrices using their
           determinants
        orientation_matrix1 = orientation_matrix1 /
           linalg.det(orientation_matrix1)
        orientation_matrix2 = orientation_matrix2 /
           linalg.det(orientation_matrix2)

        if not quiet:
            displayMat(orientation_matrix1)
            displayMat(orientation_matrix2)

        for i in range(0, len(_z1)):
            if dispEuler:
                displayAngles(_z1[i], _x[i], _z2[i])

            if save:
                assert i < 2, "ERROR: Too many Euler 
                   angles."
                if i == 0:
                    writeMat(orientation_matrix1, _z1[i
                       ], _x[i], _z2[i], 'P', _axis)
            else:
                writeMat(orientation_matrix2, _z1[i
                    ], _x[i], _z2[i], 'Q', _axis)
#
    ------------------------------------------------------------------------

    else:
        for i in range(0,len(_z1)):
            orientation_matrix = calcRotMat(_z1[i], _x[
                i], _z2[i])

            # Normalize the matrix using the
                determinant
            orientation_matrix = orientation_matrix /
                linalg.det(orientation_matrix)

            if not quiet: # Display the results
                displayMat(orientation_matrix)

            if dispEuler: # Display the Euler Angles
                displayAngles(_z1[i], _x[i], _z2[i])

            if save: # We only calculate 2 angles at a
                time. If there are more, that's a
                problem.
                assert i < 2, "ERROR: Too many Euler
                    angles."
                if i == 0:
                    writeMat(orientation_matrix, _z1[i
                        ], _x[i], _z2[i], 'P', _axis)
                else:
                    writeMat(orientation_matrix, _z1[i
                        ], _x[i], _z2[i], 'Q', _axis)
```

Appendix E

Rotation Matrix Generator

This code generates the rotation matrices used to rotate the axes to the [100] direction as required by Bulatov *et al.*'s script.

```
from __future__ import division,print_function #
    Automatically divides as floats, and considers print
    () a function
from sys import argv # for CLI arguments
from numpy import array,linalg # for matrix operations
from os.path import exists # For checking existence of
    a file
from myModules import * # For using my user-defined
    functions

# Helper functions
def displayHelp():
     print('''
     This script will generate the rotation matrix for
        the given misorientation axis
     Input:
          _rotation_axis       This specifies the axis
             around which the grains are
                               rotated. (type: int (100)
                                  or string ('100'))

          _gbnormal            This specifies the boundary
              plane normal.  (type:
```

```
                              int (100)  or string
                                 ('100'))
    Options:
        -s --save             Saves the resultant
           rotation matrix to a database
                              (rotation_matrix_database.m
                                 ) with the accompanying
                              rotation axis and
                                 misorientation type.

        -q --quiet            Suppresses output of the
           rotation matrix

        --help                Display this help info.
    Output:
    The output displayed will be the resultant rotation
        matrix for the given
    misorientation.
    ''')

# This function is an adaptation from MOOSE
    RotationMatrix class.
def rotVecToZ(vec): # Creates the rotation matrix to
    rotate vec to the z direction
    # REALLY make sure vec is normalized
    vec = vec / linalg.norm(vec)

    # Initialize our vectors
    v1 = array([[0.,0.,0.]])
    v0 = array([[0.,0.,0.]])

    # Temp vector that gives a prototype of v1 by
        looking at the smallest component of vec
    w = array([[abs(vec[0][0]), abs(vec[0][1]), abs(vec
        [0][2])]])
    if ( (w[0][2] >= w[0][1] and w[0][1] >= w[0][0]) or
         (w[0][1] >= w[0][2] and w[0][2] >= w[0][0]) ):
         v1[0][0] = 1.0
    elif ( (w[0][2] >= w[0][0] and w[0][0] >= w[0][1])
        or (w[0][0] >= w[0][2] and w[0][2] >= w[0][1]) )
        :
          v1[0][1] = 1.0
    else:
          v1[0][2] = 1.0

    # Gram-Schmidt method to find v1
    v1 = v1 - ((v1.dot(vec.T))*vec)
    v1 = v1 / linalg.norm(v1)

    # v0 = v1 x vec
    v0[0][0] = v1[0][1]*vec[0][2] - v1[0][2]*vec[0][1]
    v0[0][1] = v1[0][2]*vec[0][0] - v1[0][0]*vec[0][2]
    v0[0][2] = v1[0][0]*vec[0][1] - v1[0][1]*vec[0][0]

    # Rotation matrix is just:
    rot = array([[v0[0][0], v0[0][1], v0[0][2]],
                 [v1[0][0], v1[0][1], v1[0][2]],
                 [vec[0][0],vec[0][1],vec[0][2]]])
    return rot

def rotVec1ToVec2(vec1, vec2):
    rot1_to_z = rotVecToZ(vec1)
    rot2_to_z = rotVecToZ(vec2)
    return (rot2_to_z.T).dot(rot1_to_z)

def writeMat(m, _axis, gbnormal): # Write the matrix
   and angles to a file
    tex_filename = "rotation_matrix_database.m"
    normName = _axis + '_' + gbnormal
    var_name = "rot%snorm"%(normName)
    if not exists(tex_filename):
        tex_file = open(tex_filename, "a")
        tex_file.write("%Database␣for␣rotation␣matrices
            ␣for␣specified␣misorientation␣axes\n")
        tex_file.write("
            %
          n")
        tex_file.write("%Rotation␣Matrix\n")
        tex_file.write("%s=[%2.4f␣␣%2.4f␣␣%2.4f\n"%(
           var_name, m[0][0], m[0][1], m[0][2]))
        tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f\n"%(m
           [1][0], m[1][1], m[1][2]))
        tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f];\n"%(m
           [2][0], m[2][1], m[2][2]))
        tex_file.write("
           %------------------------------------------------------------
           n")
        tex_file.close()
    else:
          # Check for already written
          numlines = countFileLines(tex_filename)
        if numlines <= 4:
        tex_file = open(tex_filename, "a")
        tex_file.write("%s=[%2.4f␣␣%2.4f␣␣%2.4f\n"
           %(var_name, m[0][0], m[0][1], m[0][2]))
        tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f\n"%(m
           [1][0], m[1][1], m[1][2]))
        tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f];\n"%(m
           [2][0], m[2][1], m[2][2]))
        tex_file.write("
           %------------------------------------------------------------
           n")
        tex_file.close()
        else:
            f = open(tex_filename,"r")
            while True:
           data = f.readline().split()
           if not data:
              break
           elif len(data[0]) > 14:
              if not data[0][14] == '=':
                 continue
              else:
                 if data[0][0:10] == var_name:
```

```
                        unique = False
                    else:
                        unique = True
            if unique:
                tex_file = open(tex_filename, "a")
                tex_file.write("%s=[%2.4f␣␣%2.4f␣␣%2.4f
                    \n"%(var_name, m[0][0], m[0][1], m
                    [0][2]))
                tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f\n"
                    %(m[1][0], m[1][1], m[1][2]))
                tex_file.write("%2.4f␣␣%2.4f␣␣%2.4f];\n
                    "%(m[2][0], m[2][1], m[2][2]))
                tex_file.write("
                    %----------------------------------------------------------------
                    n")
                tex_file.close()

# Check what we were given...
if "--help" in argv: # Help info
    displayHelp()
    exit()

save, argv = check4Save(argv)
quiet, argv = check4Quiet(argv) # Checks for
    suppressing output

if len(argv) != 3: # if not both values given
    print("ERROR:␣Incorrect␣number␣of␣command␣line␣
        arguments.␣Line␣150")
    displayHelp()
    exit()
else: # len(argv) == 3
    script, _rotation_axis, _gbnormal = argv

if not type(_gbnormal) in {int, str}:
    print("ERROR:␣Grain␣boundary␣normal␣type␣is␣
        incorrect.␣␣Please␣enter␣an␣int␣or␣a␣string.␣
        Line␣157")
    print("You entered %s with type %s"%(str(_gbnormal)
        , type(_gbnormal)))
    exit()
else:
    if type(_gbnormal) == int:
        _gbnormal = '0' + '0' + '0' + str(_gbnormal)
        _gbnormal =_gbnormal[-3:] # gets the last three
            characters
        assert _gbnormal != '000', "ERROR: invalid 
           boundary normal. Line 173"

    assert(len(_rotation_axis) == 3), "ERROR: Something
       went wrong converting  gbnormal into a string. 
       Line 166"

if not type(_rotation_axis) in {int, str}:
    print("ERROR: Grain boundary rotation axis type is 
       incorrect.  Please enter an int or a string. 
       Line 169")
    print("You entered %s with type %s"%(str(
       _rotation_axis), type(_rotation_axis)))
    exit()
else: # Convert anything besides a string into a string
    if type(_rotation_axis) == int:
        _rotation_axis = '0' + '0' + '0' + str(
           _rotation_axis)
        _rotation_axis = _rotation_axis[-3:] # Get the
           last three characters
        assert _rotation_axis != '000', "ERROR: invalid
           rotation axis. Line 176"

    assert(len(_rotation_axis) == 3), "ERROR: Something
       went wrong converting  rotation_axis into a 
       string. Line 178"

# Now that the input is taken care of, do the work
axis = array([[1, 0, 0]]) # This is the axis that we
   rotate the grain boundary normal to
```

```
# This part converts _gbnormal to an array for use in
    the rotation functions
gbnormal = array([[None]*3])
j = 0
indices = []
for i in range(0,len(gbnormal[0])):
    try:
        gbnormal[0][i] = int(_gbnormal[j])
        j = j+1
    except:
        #print(_gbnormal[i:i+2])
        gbnormal[0][i] = int(_gbnormal[j:j + 2])
        j = j + 2
        indices.append(i)


# So much work...
gbnorm = ''
for i in range(0,len(gbnormal[0])):
    if gbnormal[0][i] < 0:
        gbnorm += str(abs(gbnormal[0][i])) + 'bar'
    else:
        gbnorm += str(gbnormal[0][i])
gbnormal = gbnormal / linalg.norm(gbnormal) # Normalize
    the gbnormal vector.
axis = axis / linalg.norm(axis) # Just to be sure...
rotation_matrix = rotVec1ToVec2(gbnormal, axis)



if not quiet:
    displayMat(rotation_matrix)
if save:
    writeMat(rotation_matrix, _rotation_axis, gbnorm)
```

Appendix F

genOrientationMatrix.sh Bash Script

This bash script reads a CSV file containing misorientation angles data, and uses those angles to generate the P and Q matrices. This script calls the script orientation_matrix.py.

```
#! /bin/bash

# This script will generate the orientation matrices
    through python by looping
# through the CSV values given in the input files.
# Argument(s):
#     $1         Should be a filename that specifies the
    angles and relative
#                energies for the 100, 110, and 111
    symmetric tilt and twist
#                boundaries

# Command-line argument counter that checks for the
    correct number of arguments.
# Does not check for correct syntax.
if [ "$#" -ne 1 ]; then
  echo "Illegal number of parameters"
  exit 1
fi
```

```
# This takes the first argument from the command line -
    this is assumed to be a
# filename of the format 100Tilt.
FN=$1

echo "Determining the axis..."
# Pulls out the axis from the input file name.  This
    uses regex syntax to find
# a series of numbers that match either 100, 110, or
    111.  This also has an issue
# where it will find a match for 101, but as long as
    the files are named correctly
# it shouldn't be an issue.
AXIS=`echo $FN | grep -o "1[01][01]"`

echo "Reading the file..."
IFS="," # separation character is the comma
# Exit with error code 99 if unable to read the file
[ ! -f $FN ] && { echo "$FN file not found"; exit 99; }

# This makes the assumption that the file
    orientation_matrix.py has executable
# rights.
echo "Running the command: ~/projects/scripts/
    orientation_matrix.py $AXIS <angle> -s -q"
while read -r angle en; do # read the file with comma
    separated values

  ~/projects/scripts/orientation_matrix.py $AXIS $angle
      -s -q
done < "$FN" # the "$FN" is required if it's going to
    run properly!

IFS=$OLDIFS # go back to the old separation character
    based on the system value.
```