```
# Created by Darren Holland 2/13/17
# Modified by Darren Holland 2020-11-02
#*********************************************************************
# File for creating the 2D mask using the design matrix
#*********************************************************************

# Make all numpy available via shorter 'num' prefix
import numpy as np
# Make all matlib functions accessible at the top level via M.func()
import numpy.matlib as M
# Make some matlib functions accessible directly at the top level via, e.g.
rand(3,3)
from numpy.matlib import rand,zeros,ones,empty,eye

import math as m
import subprocess
# Load functions for calculating mask inner wall
import intersectLinePlane
import intersectLineCylinder

# Define geometry creation function
def
CreateGeo(RSMmaxsize,th_angle,ph_angle,start_phi,end_phi,det_height,det_rad,slee
ve_inner_rad,sleeve_outer_rad,sleeve_height,sleeve_bottom,\
    start_cells,s_dist,Userfile,MainDir,Nodesfile,Elemfile,dir_settings,DR,DH):

    #  Initialize variables and coordinates
    theta_angle = th_angle
    phi_angle = ph_angle

    # Create a small gap between the mask and detector/sleeve so geometry does
not overlap
    edge_angle = m.degrees(m.atan(sleeve_height/sleeve_outer_rad))
    edge_dist = m.sqrt(sleeve_outer_rad ** 2 + sleeve_height ** 2)
    gap = max(np.asarray([0, edge_dist * m.cos(m.radians(edge_angle -
phi_angle / 2)) - sleeve_outer_rad, edge_dist * m.sin(m.radians(edge_angle +
phi_angle / 2)) - sleeve_height]))

    # Add 0.2 to gap to avoid overlapping elements
    gap = gap + 0.02
    RSM_outer_rad = sleeve_outer_rad + sleeve_outer_rad * (1 -
m.cos(m.radians(theta_angle / 2))) + gap

    # Initialize discretization
    theta = np.arange(theta_angle / 2.0, 361, theta_angle)
    phi = np.arange(start_phi+phi_angle/2.0, end_phi, phi_angle)
    n_theta = theta.shape[0]
    n_phi = phi.shape[0]

    # Create nodal mapping
    n_mapping = np.empty((n_theta,n_phi))
    n_mapping[range(n_theta),0] = np.transpose(np.arange(1,n_theta+1) +
start_cells)
    for kk in np.arange(0,n_phi-1):
        n_mapping[:, kk+1] = (n_mapping[:, kk] + n_theta)

    # Create matrices of all RSM theta and phi positions
    m_theta = np.kron(np.ones((1,n_phi)), np.transpose(theta[np.newaxis]))
    m_phi = np.kron(np.ones((n_theta,1)), (phi[np.newaxis]))

    # Initialize voxel corner positions (closest to detector)
    x_c1 = np.zeros((n_theta,n_phi))
    y_c1 = np.zeros((n_theta,n_phi))
    z_c1 = np.zeros((n_theta,n_phi))
```

```
    x_c2 = np.zeros((n_theta,n_phi))
    y_c2 = np.zeros((n_theta,n_phi))
    z_c2 = np.zeros((n_theta,n_phi))
    x_c3 = np.zeros((n_theta,n_phi))
    y_c3 = np.zeros((n_theta,n_phi))
    z_c3 = np.zeros((n_theta,n_phi))
    x_c4 = np.zeros((n_theta,n_phi))
    y_c4 = np.zeros((n_theta,n_phi))
    z_c4 = np.zeros((n_theta,n_phi))

    #Find points on the detector cylinder at each phi and theta value.
    # Side of cylinder
    p_cyl = np.asarray([0., 0., -500., 0., 0., sleeve_height, RSM_outer_rad])
    # Top of cylinder
    p_plane = np.asarray([0., 0., sleeve_height + gap, 1., 0., 0., 0., 1., 0.])

    # Angle voxels by small amount to avoid numerical overlap and parallel
planes
    overlapfactor=float(0.0001) # Value used to avoid Geant overlapping geometry
    overlapfactorzero=float(0.01) # Value used to avoid Geant overlapping
geometry at phi = 0

    # Calculate voxel corner positions closest to detector by finding
intersection of ray starting at detector center to cylinder face
    for kk in np.arange(1,6):
        # Calculate local voxel angle of all voxels with respect to voxel center
        if kk == 1:
            # corner 1
            temp_theta = m_theta - theta_angle / 2 + overlapfactor
            temp_phi = m_phi + phi_angle / 2 - overlapfactor
        if kk == 2:
            # corner 2 (may be degenerate case where phi = 0)
            temp_theta = m_theta - theta_angle / 2 + overlapfactor
            temp_phi = m_phi - phi_angle / 2.0
            temp_phi[temp_phi > 0] = temp_phi[temp_phi > 0] + overlapfactor
            temp_phi[temp_phi == 0] = overlapfactorzero
        if kk == 3:
            # corner 3
            temp_theta = m_theta + theta_angle / 2 - overlapfactor
            temp_phi = m_phi + phi_angle / 2 - overlapfactor
        if kk == 4:
            # corner 4 (may be degenerate case where phi = 0)
            temp_theta = m_theta + theta_angle / 2 - overlapfactor
            temp_phi = m_phi - phi_angle / 2.0
            temp_phi[temp_phi > 0] = temp_phi[temp_phi > 0] + overlapfactor
            temp_phi[temp_phi == 0] = overlapfactorzero
        if kk == 5:
            # voxel center
            temp_theta = m_theta
            temp_phi = m_phi

        # For each voxel find where it intersects the top or side of the
cylinder
        for jj in np.arange(n_phi):
            for ii in np.arange(n_theta):
                # Define line from detector center to direction of point
                p_line = np.asarray([0., 0., 0., m.sin(m.radians(temp_phi[ii,
jj])) * m.cos(m.radians(temp_theta[ii, jj])), m.sin(m.radians(temp_phi[ii, jj]))
* m.sin(m.radians(temp_theta[ii, jj])), m.cos(m.radians(temp_phi[ii, jj]))])

                # If intersects the top find intersection point with plane
                if RSM_outer_rad >=  sleeve_height *
m.tan(m.radians(temp_phi[ii, jj])) and temp_phi[ii, jj] < 90:
                    points = intersectLinePlane.intersectLinePlane(p_line,
```

```
p_plane)
                    points = points[np.newaxis]
                # Otherwise find intersection point with side of cylinder
                else:
                    points = intersectLineCylinder.intersectLineCylinder(p_line,
p_cyl)
                    if temp_phi[ii, jj] < 90:
                          pick_z = points[:, 2] >= 0
                          points = points[pick_z, :]

                    elif temp_phi[ii,jj] == 90: # Created to better handle the
boundary condition at phi = 90 deg.
                        if temp_theta[ii,jj] < 90 or temp_theta[ii,jj] > 270:
                                points = points[1,:]
                                points = points[np.newaxis]
                                points[0,0] = np.absolute(points[0,0])
                        else:
                                points = points[0,:]
                                points = points[np.newaxis]
                                points[0,0] = -np.absolute(points[0,0])

                        if 0 < temp_theta[ii,jj] < 180:
                                points[0,1] = np.absolute(points[0,1])
                        else:
                                points[0,1] = -np.absolute(points[0,1])

                    else:
                            pick_z = points[:, 2] <= 0
                            points = points[pick_z, :]

                # Save intersection point as closest mask coordinates
                if kk == 1:
                    # corner 1
                    x_c1[ii, jj] = points[0,0]
                    y_c1[ii, jj] = points[0,1]
                    z_c1[ii, jj] = points[0,2]
                elif kk == 2:
                    # corner 2
                    x_c2[ii, jj] = points[0,0]
                    y_c2[ii, jj] = points[0,1]
                    z_c2[ii, jj] = points[0,2]
                elif kk == 3:
                    # corner 3
                    x_c3[ii, jj] = points[0,0]
                    y_c3[ii, jj] = points[0,1]
                    z_c3[ii, jj] = points[0,2]
                elif kk == 4:
                    # corner 4
                    x_c4[ii, jj] = points[0,0]
                    y_c4[ii, jj] = points[0,1]
                    z_c4[ii, jj] = points[0,2]

    # Load design matrix
    EVcoords = np.genfromtxt(Userfile, dtype='float')

    # Calculate relative voxel corner positions (note angles slightly smaller to
avoid overlapping planes)
    xp1_RSM = EVcoords * np.sin(np.radians(m_phi + phi_angle / 2 -
overlapfactor)) * np.cos(np.radians(m_theta - theta_angle / 2 + overlapfactor))
    yp1_RSM = EVcoords * np.sin(np.radians(m_phi + phi_angle / 2 -
overlapfactor)) * np.sin(np.radians(m_theta - theta_angle / 2 + overlapfactor))
    zp1_RSM = EVcoords * np.cos(np.radians(m_phi + phi_angle / 2 -
overlapfactor))
    xp2_RSM = EVcoords * np.sin(np.radians(m_phi - phi_angle / 2 +
```

```
overlapfactor)) * np.cos(np.radians(m_theta - theta_angle / 2 + overlapfactor))
    yp2_RSM = EVcoords * np.sin(np.radians(m_phi - phi_angle / 2 +
overlapfactor)) * np.sin(np.radians(m_theta - theta_angle / 2 + overlapfactor))
    zp2_RSM = EVcoords * np.cos(np.radians(m_phi - phi_angle / 2 +
overlapfactor))
    xp3_RSM = EVcoords * np.sin(np.radians(m_phi + phi_angle / 2 -
overlapfactor)) * np.cos(np.radians(m_theta + theta_angle / 2 - overlapfactor))
    yp3_RSM = EVcoords * np.sin(np.radians(m_phi + phi_angle / 2 -
overlapfactor)) * np.sin(np.radians(m_theta + theta_angle / 2 - overlapfactor))
    zp3_RSM = EVcoords * np.cos(np.radians(m_phi + phi_angle / 2 -
overlapfactor))
    xp4_RSM = EVcoords * np.sin(np.radians(m_phi - phi_angle / 2 +
overlapfactor)) * np.cos(np.radians(m_theta + theta_angle / 2 - overlapfactor))
    yp4_RSM = EVcoords * np.sin(np.radians(m_phi - phi_angle / 2 +
overlapfactor)) * np.sin(np.radians(m_theta + theta_angle / 2 - overlapfactor))
    zp4_RSM = EVcoords * np.cos(np.radians(m_phi - phi_angle / 2 +
overlapfactor))

    # Add relative positions to closest corner positions to get position of far
voxel corners
    RSMcoords_x1 = x_c1 + xp1_RSM
    RSMcoords_y1 = y_c1 + yp1_RSM
    RSMcoords_z1 = z_c1 + zp1_RSM
    RSMcoords_x2 = x_c2 + xp2_RSM
    RSMcoords_y2 = y_c2 + yp2_RSM
    RSMcoords_z2 = z_c2 + zp2_RSM
    RSMcoords_x3 = x_c3 + xp3_RSM
    RSMcoords_y3 = y_c3 + yp3_RSM
    RSMcoords_z3 = z_c3 + zp3_RSM
    RSMcoords_x4 = x_c4 + xp4_RSM
    RSMcoords_y4 = y_c4 + yp4_RSM
    RSMcoords_z4 = z_c4 + zp4_RSM

    # Find the largest width of the mask (to automate the variance reduction).
DOES NOT CONSIDER RMC.
    maxpt = np.empty((1,4))
    maxpt[0,0] = (np.sqrt(RSMcoords_x1 ** 2 + RSMcoords_y1 ** 2 + RSMcoords_z1
** 2)).max(0).max(0)
    maxpt[0,1] = (np.sqrt(RSMcoords_x2 ** 2 + RSMcoords_y2 ** 2 + RSMcoords_z2
** 2)).max(0).max(0)
    maxpt[0,2] = (np.sqrt(RSMcoords_x3 ** 2 + RSMcoords_y3 ** 2 + RSMcoords_z3
** 2)).max(0).max(0)
    maxpt[0,3] = (np.sqrt(RSMcoords_x4 ** 2 + RSMcoords_y4 ** 2 + RSMcoords_z4
** 2)).max(0).max(0)

    # Write voxel nodes and positions save to Nodesfile
    w_file = open((MainDir + Nodesfile), 'w')

    tt = 1
    # Assume using Tessellated Solids (8 nodes per voxel, 4 nodes per face)
    n_nodes = n_theta * n_phi * 8
    NodeNums = np.zeros((n_nodes, 4))
    # Create RSM geometry
    for ii in np.arange(0,n_theta):
        for jj in np.arange(0,n_phi):
             # Create matrix NodeNums for checking node creation
             NodeNums[np.arange(tt-1,tt + 7), :] = np.asarray([[tt,
RSMcoords_x1[ii, jj], RSMcoords_y1[ii, jj], RSMcoords_z1[ii, jj]], \
             [tt + 1, RSMcoords_x2[ii, jj], RSMcoords_y2[ii, jj],
RSMcoords_z2[ii, jj]],\
             [tt + 2, RSMcoords_x3[ii, jj], RSMcoords_y3[ii, jj],
RSMcoords_z3[ii, jj]],\
             [tt + 3, RSMcoords_x4[ii, jj], RSMcoords_y4[ii, jj],
RSMcoords_z4[ii, jj]],\
```

```
            [tt + 4, x_c1[ii, jj], y_c1[ii, jj], z_c1[ii, jj]],\
            [tt + 5, x_c2[ii, jj], y_c2[ii, jj], z_c2[ii, jj]],\
            [tt + 6, x_c3[ii, jj], y_c3[ii, jj], z_c3[ii, jj]],\
            [tt + 7, x_c4[ii, jj], y_c4[ii, jj], z_c4[ii, jj]]])

            # Write node numbers and coordinates to file
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' % (tt,
RSMcoords_x1[ii,jj], RSMcoords_y1[ii,jj], RSMcoords_z1[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+1,RSMcoords_x2[ii,jj],RSMcoords_y2[ii,jj],RSMcoords_z2[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+2,RSMcoords_x3[ii,jj],RSMcoords_y3[ii,jj],RSMcoords_z3[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+3,RSMcoords_x4[ii,jj],RSMcoords_y4[ii,jj],RSMcoords_z4[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+4,x_c1[ii,jj],y_c1[ii,jj],z_c1[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+5,x_c2[ii,jj],y_c2[ii,jj],z_c2[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+6,x_c3[ii,jj],y_c3[ii,jj],z_c3[ii,jj]))
            w_file.write('     %u, %4.15f, %4.15f, %4.15f\n' %
(tt+7,x_c4[ii,jj],y_c4[ii,jj],z_c4[ii,jj]))
            tt = tt + 8
    w_file.close
    n_elements = n_theta * n_phi

    # Write element (node connection) file
    w_file = open((MainDir + Elemfile), 'w')
    Elem = np.zeros((n_elements, 9))
    for bb in np.arange(1,n_elements+1):
        w_file.write('%u, %u, %u, %u, %u, %u, %u, %u, %u \n'%  (bb, NodeNums[8 *
(bb - 1) ,0], NodeNums[8 * (bb - 1) + 1,0], NodeNums[8 * (bb - 1) + 3,0],
NodeNums[8 * (bb - 1) + 2,0], NodeNums[8 * (bb - 1) + 4,0], NodeNums[8 * (bb -
1) + 5,0], NodeNums[8 * (bb - 1) + 7,0], NodeNums[8 * (bb - 1) + 6,0]))
    w_file.close

    # Calculate VR for source cone for FEP based on detector size and source
distance
    red_cone = np.degrees(np.arctan((np.sqrt(DR**2 + DH**2) + 0.01) / (s_dist-
DR))) # Source variance reduction cone (degrees)

    # Save to Geant settings file
    sed_cone = 's?coneangle(17.5)?coneangle(' + str(red_cone) + ')?g'
    subprocess.call(['sed', '-i', '-e', sed_cone, dir_settings])
    w_file = open((MainDir + "coneangle.inp"), 'w')
    w_file.write('%4.15f'% (red_cone))
    w_file.close
```