```
# Created by Darren Holland
# Modified by Darren Holland 2020-11-02
#***********************************************************************
# File for analyzing the RSM's performance and returning it to Dakota
#***********************************************************************
'''
Can be rerun by command in AnalyzeCommand___.sh
(will generate plots unless input variable set to 0)
'''

# Load relevant modules
import matplotlib
matplotlib.use('TKAgg')
import numpy as np
import scipy.signal as sc
from scipy import optimize
from numpy.matlib import rand,zeros,ones,empty,eye
import os, sys, csv
import pandas as pd
import plotly as py
import plotly.graph_objs as go

# Desired Standard Deviation

import time

tic = time.perf_counter()
sig = 2

def MAC(Eigvec, p_fig):
    """Calculate the MAC criterion for the non-imaging RSM"""
    uu = 0
    vv = 0
    # Compare all theta=0 DRCs
    AutoPMAC = zeros((np.size(Eigvec, 1), np.size(Eigvec, 1)))
    for kk in range(0,np.size(Eigvec, 1)):
        for jj in range(0,np.size(Eigvec, 1)):
            # Calculate MAC value for the two vectors
            AutoPMAC[uu, vv] = ((((Eigvec[:, jj]).real).T).dot((Eigvec[:,
kk]).real)) ** 2 / ((((Eigvec[:, jj]).real).T).dot((Eigvec[:, jj]).real) *
(((Eigvec[:, kk]).real).T).dot((Eigvec[:, kk]).real))
            uu += 1
        uu = 0
        vv += 1
    return AutoPMAC

def MACloop(Eigvec, p_fig):
    """Calculate the MAC criterion for the imaging RSM"""
    uu = 0
    vv = 0
    # Only compare one DRC at a time
    AutoPMAC = zeros((np.size(Eigvec, 1), 1))
    for kk in range(0,1):
        for jj in range(0,np.size(Eigvec, 1)):
            # Calculate MAC value for the two vectors
            AutoPMAC[uu, vv] = ((((Eigvec[:, jj]).real).T).dot((Eigvec[:,
kk]).real)) ** 2 / ((((Eigvec[:, jj]).real).T).dot((Eigvec[:, jj]).real) *
(((Eigvec[:, kk]).real).T).dot((Eigvec[:, kk]).real))
            uu += 1
        uu = 0
        vv += 1
    return AutoPMAC

def det_atten(phi):
```

```
    """ Returns the bare detector curv-fit attenuation as a function of phi"""
    return (1.730363487313672e-25*phi**10 + -1.818130681131725e-22*phi**9 + \
      7.873995291136305e-20*phi**8 + -1.844235989319162e-17*phi**7 + \
      2.556499795059634e-15*phi**6 + -2.145909796941168e-13*phi**5 +  \
      1.067763006640183e-11*phi**4 + -2.965516933029438e-10*phi**3 + \
      4.236080358837612e-09*phi**2 + -2.856554887240579e-08*phi + \
      5.772953352062337e-06)

def
gprocess(MainDir=None,VRval=None,Geofile=None,minCforGauss=None,s_subdiv=None,DR
Mtheta=None,DRMphi=None,GeoStart=None,GeoFinal=None):
    # lower bound is in keV - total counts for energies above this value
    # (default 1 keV)

    #Read in the input geometry
    fid = open((''.join([Geofile])), "r")
    Geobegin = np.loadtxt(fid,ndmin=2)
    fid.close()

    Geotemp = Geobegin.copy()

    x_c = pd.read_csv(MainDir+'/x_c.txt', sep='\s+',header=None).to_numpy() #
line
    y_c = pd.read_csv(MainDir+'/y_c.txt', sep='\s+',header=None).to_numpy() #
line
    z_c = pd.read_csv(MainDir+'/z_c.txt', sep='\s+',header=None).to_numpy() #
line

    # Get the total distance
    mag=np.sqrt(x_c**2+y_c**2+z_c**2)

    Geotheta=zeros((np.size(x_c,0),np.size(x_c,1)))
    Geophi=zeros((np.size(x_c,0),np.size(x_c,1)))
    for jj in range(0, np.size(Geobegin,1)):
        for ii in range(0, np.size(Geobegin,0)):
            # Get the vector of each voxel center (from detector center)
            vec=[x_c[ii,jj],y_c[ii,jj],z_c[ii,jj]]
            vec_r=np.sqrt(vec[0]**2+vec[1]**2+vec[2]**2)
            vec_phi=np.arccos(vec[2]/vec_r)*180/np.pi
            vec_theta=np.arctan2(vec[1],vec[0])*180/np.pi
            if vec_theta<0:
                vec_theta+=360
            Geotheta[ii,jj]=vec_theta
            Geophi[ii,jj]=vec_phi
            # Project every other vector onto the vector
            dot_prod=x_c*vec[0]+y_c*vec[1]+z_c*vec[2]
            # Projection coefficient used
            proj_coef=dot_prod/(vec[0]**2+vec[1]**2+vec[2]**2)
            # Portion that is parallel
            para_x=proj_coef*vec[0]
            para_y=proj_coef*vec[1]
            para_z=proj_coef*vec[2]
            # Parallel distance
            para_dist=np.sqrt(para_x**2+para_y**2+para_z**2)
            # Calculate orthogonal piece
            ortho_x=x_c-para_x
            ortho_y=y_c-para_y
            ortho_z=z_c-para_z
            # Calculate the projection distance towards the voxel of interest
            ortho_dist=np.sqrt(ortho_x**2+ortho_y**2+ortho_z**2)
            # print(proj_dist)
            mask=Geobegin.copy()
            p_mask=Geobegin.copy()
            # Ignore backscatter (voxel is on other side) since only FEP
```

```
            mask[dot_prod<0]=np.nan
            p_mask[dot_prod<0]=-10
            # If "most" of voxel is NOT in the direction of interest, ignore it
            # aka voxel is near edge projection
            y1 = 0.095 # value greater than 90 degrees
            y2 = 0.205 # value at and before 90 degrees
            par_a = (y1-y2)/60**2
            # Create mask and average values (p_mask is for plotting the
contributing values)
            if Geophi[ii,jj]>90:
                mask[ortho_dist>(par_a*(Geophi[ii,jj]-
90)**2+y2)*para_dist]=np.nan
                p_mask[ortho_dist>(par_a*(Geophi[ii,jj]-90)**2+y2)*para_dist]=-
10
            else:
                mask[ortho_dist>y2*para_dist]=np.nan
                p_mask[ortho_dist>y2*para_dist]=-10
            Geotemp[ii,jj]=np.nanmean(mask)

    # Plot the current thicknesses (only if analysis is run after optimization)
    if sys.argv[14] == '1':
        pdata = [go.Surface(z=Geotemp,showscale=False)]
        layout = dict(autosize=True,scene=dict(
            yaxis=dict(title=u"\u03D1"),xaxis=dict(title=u"\
u03D5"),zaxis=dict(title='Geo Check'),
            camera=dict(eye=dict(x=2, y=2, z=1))),
        width=1000,height=1000,margin=dict(l=0,r=0,b=0,t=0))
        fig = go.Figure(data=pdata, layout=layout)
        py.offline.plot(fig, filename=MainDir + '/Check.html',
include_mathjax='cdn')

    # Initialize geometric attenuation matrices
    Geo = zeros((np.size(DRMtheta), np.size(DRMphi)))
    Geophimat = zeros((np.size(Geotheta,0), np.size(DRMphi)))
    Geotheta_phi_interp = zeros((np.size(Geotheta,0), np.size(DRMphi)))

    # Initialize interpolation vectors for off-center measurements
    phi_interp=np.concatenate((GeoStart*ones((np.size(Geophi,0),1)), Geophi,
GeoFinal*ones((np.size(Geophi,0),1))),axis=1)

Geotemp_interp=np.concatenate((ones((np.size(Geotheta,0),1))*np.mean(Geotemp[:,0
]), Geotemp, zeros((np.size(Geotheta,0),1))),axis=1)
    Geotheta_interp=np.concatenate((zeros((np.size(Geotheta,0),1)), Geotheta,
ones((np.size(Geotheta,0),1))*360),axis=1)

    #Interpolate over measured phi (and save to theta position)
    for jj in range(0,np.size(Geotheta,0)):
        Geophimat[jj,:] =
np.interp(DRMphi.flatten(),np.ravel(phi_interp[jj,:]),np.ravel(Geotemp_interp[jj
,:]))
        Geotheta_phi_interp[jj,:] =
np.interp(DRMphi.flatten(),np.ravel(phi_interp[jj,:]),np.ravel(Geotheta_interp[j
j,:]))

    theta_interp=np.concatenate((zeros((1,np.size(Geophimat,1))),
Geotheta_phi_interp, 360*ones((1,np.size(Geophimat,1)))),axis=0)
    Geo_ends=np.mean(np.concatenate((Geophimat[-1,:], Geophimat[0,:])),axis=0)
    Geo_interp=np.concatenate((Geo_ends, Geophimat, Geo_ends),axis=0)

    #Interpolate over measured theta
    for jj in range(0,np.size(DRMphi,1)):
        Geo[:,jj] =
np.reshape(np.interp(DRMtheta,np.ravel(theta_interp[:,jj]),np.ravel(Geo_interp[:
,jj])), (-1, 1))
```

```
    np.savetxt(sys.argv[2]+"Geo.txt",Geo,newline="\n")
    # Get assumed (constant) linear attentuation coefficient
    LinAtten =  sys.argv[13]

    # Covert to counts assuming exponential attentation (constant over energy).
Assume the minimum
    # number of counts is minCforGauss.
    Det=det_atten(DRMphi)/det_atten(DRMphi[0,0])# divide by 1st DRM value (looks
most like slab)#/2.314152e-02
    Resp=(np.exp(-float(LinAtten)*Geo))*np.diag(np.ravel(Det))
    print(np.size(Det,0),np.size(Det,1),np.size(Resp,0),np.size(Resp,1))
    DRM=minCforGauss*Resp/np.min(np.min(Resp))

    # Plot the DRM (only if analysis is run after optimization)
    if np.size(Geo,1)>1 and sys.argv[14] == '1':
        pdata =
[go.Surface(x=np.ravel(DRMphi),y=np.ravel(DRMtheta),z=DRM,showscale=False)]
        layout = dict(autosize=True,scene=dict(
            yaxis=dict(title=u"\u03D1"),xaxis=dict(title=u"\
u03D5"),zaxis=dict(title='Surrogate DRM'),
            camera=dict(eye=dict(x=2, y=2, z=1))),
        width=1000,height=1000,margin=dict(l=0,r=0,b=0,t=0))
        fig = go.Figure(data=pdata, layout=layout)
        py.offline.plot(fig, filename=MainDir + '/SurrDRM.html',
include_mathjax='cdn')

    # Return the DRM and final, interpolated thickness
    return DRM,Geo

def
RSManalyze(MainDir=None,VRval=None,Geofile=None,minCforGauss=None,s_subdiv=None,
DRMtheta=None,DRMphi=None,GeoStart=None,GeoFinal=None):
    DRM,Geo =
gprocess(MainDir,VRval,Geofile,minCforGauss,s_subdiv,DRMtheta,DRMphi,GeoStart,Ge
oFinal)

    # Force the DRM to be zero mean for each phi
    DRMred = DRM.copy()
    DRMmean = zeros((1, np.size(DRM,1)))
    for ii in range(0,np.size(DRM,1)):
        DRMmean[0,ii] = np.mean(DRM[:, ii])
        DRMred[:, ii] = DRMred[:, ii] - DRMmean[0,ii]

    # Calculate the MAC for the non-imaging design
    AutoPMAC = MAC(DRMred, 1)
    maxSingleMAC = (np.triu(AutoPMAC) - np.eye(np.size(AutoPMAC, 0))).max()
    avgSingleMAC = np.sum(np.sum(np.triu(AutoPMAC) - np.eye(np.size(AutoPMAC,
0)))) / np.sum(np.arange(1,np.size(DRM,1)))

    # Calculate the MAC for the imaging design
    maxMat = zeros(((np.size(DRMred,1)+np.size(DRMred,0)-1),np.size(DRMred,1)))
    PMAC=[np.inf]
    # Loop though all DRCs
    for pp in range(0,np.size(DRM,1)):
        DRMperm = DRMred[:, pp:]
        # Loop though all shifted DRCs
        for gg in range(0,np.size(DRM,0)):
             if gg == 0 and pp < np.size(DRMred,1)-1:
                 PMAC = MACloop(DRMperm, 0)
             else:
                 if gg == 0 and pp == np.size(DRMred,1)-1: pass
                     # Skip comparing with itself - just want to compare with
                     # permutations for final vector
                 else:
```

```
                     PMAC = MACloop(np.c_[DRMred[:, pp], np.roll(DRMperm,gg,
0)], 0)
             # Store largest value
             if np.size(PMAC) ==1 and np.isinf(PMAC[0])==1:
                   maxMat[gg + pp, pp] = np.inf
             else:
                   maxMat[gg + pp, pp] = (PMAC[1:]).max()

    # Store imaging MAC values
    maxMAC = ((maxMat).max()).max()
    print("maxMAC = ", maxMAC)
    avgMAC = np.mean(maxMat[maxMat > 0])
    print("avgSingleMAC = ",avgSingleMAC)
    # Calculate the sensitivity
    Sens = ((DRM).max(0) / (DRM).min(0)).min()
    print("Sens = ", Sens)

      # Record average number of counts per angle
    AvgCountsAngle = np.sum(np.sum(DRM)) / float(np.size(DRM,0) *
np.size(DRM,1))
    print("AvgCountsAngle = ", AvgCountsAngle)
      # Return DRMs and metrics
    return AutoPMAC, DRM, DRMred, DRMmean, maxMat, maxMAC, maxSingleMAC, Sens,
AvgCountsAngle, avgMAC, avgSingleMAC, Geo

def gauss(x,mu,sigma,A):
    """ Gaussian function for curve-fit"""
    return np.absolute(A*np.exp(-(x-mu)**2/(2*sigma**2)))

def step(x,mu,sigma,A):
    """ Middle step function for curve-fit"""
    return np.absolute(A*(np.heaviside(x-(mu-sigma),1) - np.heaviside(x-
(mu+sigma),1)))

def bimodal(x,mu1,sigma1_g,A1,mu2,sigma2_g,A2,sigma1_p,sigma2_p,c):
    """ Two valley curve-fit to DRCs.  Uses two gaussian functions with a middle
step function"""
    return gauss(x,mu1-sigma1_p,sigma1_g,A1)*(1-np.heaviside(x-(mu1-
sigma1_p),1))
+step(x,mu1,sigma1_p,A1)+gauss(x,mu1+sigma1_p,sigma1_g,A1)*(np.heaviside(x-
(mu1+sigma1_p),1)) \
           +gauss(x,mu2-sigma2_p,sigma2_g,A2)*(1-np.heaviside(x-(mu2-
sigma2_p),1))
+step(x,mu2,sigma2_p,A2)+gauss(x,mu2+sigma2_p,sigma2_g,A2)*(np.heaviside(x-
(mu2+sigma2_p),1))+c

def
CalcMinTime(DRM,DRMtheta,DRMphi,MainDir,VRval,deltatheta,minCforGauss,LinAtten,G
eo,sig,wallwidth,finwidth,s_subdiv):
    """ Based on the DRCs, determine the number of particles needed to ID the
source location/image"""
    # Initialize variables
    TPeak=np.inf        # Particles to ID using peak method
    TWidth=np.inf       # Particles to ID using width method
    Acc=0               # Accuracy
    RatioWallFinPeak = zeros([1,np.size(DRMphi,1)])
    RatioWallFinWidth = zeros([1,np.size(DRMphi,1)])
    DistPeak = 0        # Distinguishability of peak method
    DistWidth = 0       # Distinguishability of width method
    savefit = zeros([np.size(DRMphi,1),7])  # save curve-fit values
    savestd = zeros([np.size(DRMphi,1),7])  # save curve-fit standard deviation
values
    savelabel = pd.DataFrame(np.empty((np.size(DRMphi,1), 0), dtype = np.str))
# Label for fin/wall valleys
```

```
    MinCounts = zeros([1,np.size(DRMphi,1)])     # Minimum number of counts in
DRC

    # Introduce location ID based on curve-fit parameters
    # (distance from reference angle to valley peaks)

    # Load geometric fin and wall centers for accuracy calculation
    fid = open((''.join([MainDir, "/FinPos.inp"])), "r")
    fin = np.loadtxt(fid,ndmin=1)
    fid.close()
    fid = open((''.join([MainDir, "/WallPos.inp"])), "r")
    wall = np.loadtxt(fid,ndmin=1)
    fid.close()

    # Initialize variables
    thetastep=deltatheta/s_subdiv       # Theta substep
    deltaphi=float(sys.argv[5])         # Phi mask discretization
    fin_interp=np.concatenate(([0], fin*deltatheta))    # Spacing for
interpolating middle fin geometry
    wall_interp=np.concatenate(([0], wall*deltatheta))  # Spacing for
interpolating middle wall geometry
    fin_angle=fin*deltatheta            # Angle to middle fin geometry
    wall_angle=wall*deltatheta          # Angle to middle wall geometry
    ntheta = np.size(DRMtheta,0)        # Number of measurements
    NumPart = zeros([1,np.size(DRMphi,1)]) # total number of particles in DRC
    for ii in range(0,np.size(DRMphi,1)):
        # Convert to counts
        DRC=DRM[:,ii]
        NumPart[:,ii]=np.sum(DRC)

        #HAVE TO SHIFT THE CURVE SO PEAK NOT SPLIT AT DRC EDGE
        Wallshiftind = np.argmax(DRC)
        Wallshift = Wallshiftind*thetastep
        # Map phi values to geometry so know fin/wall positions for different
DRC phis
        print(DRMphi)
        # Have to assume initial source positions not at fin and wall center
        #     are linear interpolations of neighboring positions.
        phiPos=np.concatenate(([0], Geophi))
        if np.mod(DRMphi[0,ii],deltaphi)==0:
            FinPos = np.interp(DRMphi[0,ii],phiPos,fin_interp)
            WallPos = np.interp(DRMphi[0,ii],phiPos,wall_interp)
        else:
            FinPos=fin_angle[int(np.floor_divide(DRMphi[0,ii],deltaphi))]
            WallPos=wall_angle[int(np.floor_divide(DRMphi[0,ii],deltaphi))]
        # ---------- Invert DRC to make fitting easier then shift to move wall
peak----------------------------
        # If shifted less than the fin position, then the wall was moved to the
far right.  If shifted more than
        # the fin position or less than the wall position, then the wall remains
the
        # first valley and the fin is the second (both wall and fin moved to
end)
        if WallPos-Wallshift <0:
            shiftwallpos = WallPos-Wallshift + 360
        else:
            shiftwallpos = WallPos-Wallshift
        if FinPos-Wallshift <0:
            shiftfinpos = FinPos-Wallshift + 360
        else:
            shiftfinpos = FinPos-Wallshift
        # Check data near expected fin and wall locations for max value
        maxindwall=int(np.floor_divide(shiftwallpos,thetastep))
        maxindfin=int(np.floor_divide(shiftfinpos,thetastep))
```

```
        maxindwallnext = maxindwall + 1
        maxindfinnext = maxindfin + 1
        if maxindwallnext == ntheta:
            maxindwallnext = 0
        if maxindfinnext == ntheta:
            maxindfinnext = 0

        # Shift DRC and pick expected values for curve fit using max values,
wall width, etc.
        Inverse_DRC = np.ravel(np.roll((np.amax(DRC) - DRC), -Wallshiftind, 0))
        if Wallshift < FinPos and Wallshift > WallPos:
            Peaks_label=['Fin','Wall']

expected=(shiftfinpos,finwidth*thetastep/2,np.max([Inverse_DRC[maxindfin],Invers
e_DRC[maxindfinnext]]),\

shiftwallpos,wallwidth*thetastep/2,np.max([Inverse_DRC[maxindwall],Inverse_DRC[m
axindwallnext]]),0,0,0)
        else:
            Peaks_label=['Wall','Fin']

expected=(shiftwallpos,wallwidth*thetastep/2,np.max([Inverse_DRC[maxindwall],Inv
erse_DRC[maxindwallnext]]), \

shiftfinpos,finwidth*thetastep/2,np.max([Inverse_DRC[maxindfin],Inverse_DRC[maxi
ndfinnext]]),0,0,0)
        maxfev=100000  # Set maximum number of interations for curve fit

        try:
            # Fit curve to data to get ID parameters
            thetaval=np.arange(ntheta)*thetastep
            gpar_all, gcov_all =
optimize.curve_fit(bimodal,thetaval,Inverse_DRC,expected,maxfev=maxfev)
        except Exception:
            #print("Optimal parameters not found using {}
functions.".format(maxfev))
            gpar_all=np.ravel(np.empty((1,9)))
            gcov_all=np.empty((9,9))
            gpar_all[:]=np.inf
            gcov_all[:]=np.inf
            pass
        # Temporarily store curve-fit parameters and covariance
        #   gpar_all(mu1,sigma1_g,A1,mu2,sigma2_g,A2,sigma1_p,sigma2_p,c):
        gpar=np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
        gcov=np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
        gcovtemp=np.diag(gcov_all)
        # Center of first peak
        gpar[0]=gpar_all[0]
        gcov[0]=gcovtemp[0]
        # Note since sigma is squared, it can be negative and get an accurate
result
        #   gpar(mu1,sigma1,A1,mu2,sigma2,A2,c):
        # Total width of first peak
        gpar[1]=np.absolute(gpar_all[1])+np.absolute(gpar_all[6])
        gcov[1]=np.absolute(gcovtemp[1])+np.absolute(gcovtemp[6])
        # Amplitude of first peak and center of second peak
        gpar[2:4]=gpar_all[2:4]
        gcov[2:4]=gcovtemp[2:4]
        # Note since sigma is squared, it can be negative and get an accurate
result
        # Total width of second peak
        gpar[4]=np.absolute(gpar_all[4])+np.absolute(gpar_all[7])
        gcov[4]=np.absolute(gcovtemp[4])+np.absolute(gcovtemp[7])
        # Amplitude of second peak
```

```
        gpar[5]=gpar_all[5]
        gcov[5]=gcovtemp[5]
        # Baseline
        gpar[6]=gpar_all[8]
        gcov[6]=gcovtemp[8]

        # Calculate one standard deviation of parameters
        if np.all(np.isinf(gpar)) or np.any(gpar[0:5]<0):
            gstd=np.ravel(np.empty((1,len(gpar))))
            gstd[:] = np.inf
        else:
            gstd = sig*np.sqrt(gcov)
        savelabel.loc[ii,0] = Peaks_label[0]

        # Use parameters to create DRC (not inverted) curve fit
        gaussfit=np.ravel(bimodal(thetaval, *gpar_all))
        DRCgaussfit = np.ravel(np.max(DRC) - np.roll(bimodal(DRMtheta,
*gpar_all), Wallshiftind, 0))
        MinCounts[0,ii] = np.min(DRC)

        # Create plots of Inverse DRC and fits  (skipped during optimization,
useful for individual designs)
        if sys.argv[14] == '1':
              gfit=go.Scatter(x=DRMtheta,y=gaussfit,
                mode='lines',name='Inverse Gaussian
Fit',line=dict(width=6))#dash='dash',width=6))

pdata=go.Scatter(x=DRMtheta,y=np.ravel(Inverse_DRC),mode='markers',name='Inverse
DRC',marker=dict(size=10))
              line1=go.Scatter(x=[shiftwallpos,
shiftwallpos],y=[np.min(Inverse_DRC),
np.max(Inverse_DRC)],mode='lines',name='Wall Position',line=dict(width=4))
              line2=go.Scatter(x=[shiftfinpos, shiftfinpos],
y=[np.min(Inverse_DRC), np.max(Inverse_DRC)],mode='lines',name='Fin
Position',line=dict(width=4))
              layout = go.Layout(autosize=True,title="Inverse Detector Response
Curve (DRC): "+u"\u03D5" +" = " + str(DRMphi[0,ii]) + u"\u00B0",
                xaxis=dict(title="Azimuthal Angle = "+u"\
u03D1"),yaxis=dict(title="Counts",exponentformat='E'),
                width=1200,height=800,margin=dict(l=150,r=0,b=150,t=60))
              fig = go.Figure(data=[gfit,pdata,line1,line2], layout=layout)
              py.offline.plot(fig,
filename=str(MainDir)+'/InvDRCph'+str(DRMphi[0,ii])+'.html',
include_mathjax='cdn')

              # Create plots of DRC and fits (skipped during optimization,
useful for individual designs)
              gfit=go.Scatter(x=np.ravel(DRMtheta),y=DRCgaussfit,
                mode='lines',name='Gaussian Fit',line=dict(width=6))

pdata=go.Scatter(x=np.ravel(DRMtheta),y=np.ravel(DRC),mode='markers',name='DRC',
marker=dict(size=10))
              line1=go.Scatter(x=[WallPos, WallPos],y=[np.min(DRC),
np.max(DRC)],mode='lines',name='Wall Position',line=dict(width=4))
              line2=go.Scatter(x=[FinPos, FinPos], y=[np.min(DRC),
np.max(DRC)],mode='lines',name='Fin Position',line=dict(width=4))
              layout = go.Layout(autosize=True,title="Detector Response Curve
(DRC): "+u"\u03D5" +" = " + str(DRMphi[0,ii]) + u"\u00B0",
                xaxis=dict(title="Azimuthal Angle = "+u"\
u03D1"),yaxis=dict(title="Counts",exponentformat='E'),
                width=1200,height=800,margin=dict(l=150,r=0,b=150,t=60))
              fig = go.Figure(data=[gfit,pdata,line1,line2], layout=layout)
              py.offline.plot(fig,
filename=str(MainDir)+'/DRCph'+str(DRMphi[0,ii])+'.html', include_mathjax='cdn')
```

```
        # Calculate non-shifted Wall and Fin fit positions
        if gpar[0] >= 0 and gpar[0] <= 360:
            if gpar[0] + Wallshift <= 360:
               Gauss1Pos = gpar[0]+Wallshift
            else:
               Gauss1Pos = gpar[0]+Wallshift-360
        else:
            Gauss1Pos = np.inf
        if gpar[3] >= 0 and gpar[3] <= 360:
            if gpar[3] + Wallshift <= 360:
               Gauss2Pos = gpar[3]+Wallshift
            else:
               Gauss2Pos = gpar[3]+Wallshift-360
        else:
            Gauss2Pos = np.inf
        print("{} valley position (mu) = {}
degrees".format(Peaks_label[0],float(Gauss1Pos)))
        print("{} valley position (mu)  = {}
degrees".format(Peaks_label[1],float(Gauss2Pos)))
        print("{} valley sigma = {}".format(Peaks_label[0],gpar[1]))
        print("{} valley sigma  = {}".format(Peaks_label[1],gpar[4]))
        print("{} valley counts (amplitude) =
{}".format(Peaks_label[0],gpar[2]))
        print("{} valley counts (amplitude) =
{}".format(Peaks_label[1],gpar[5]))
        print("{} valley position (mu) sig * std = {}
degrees".format(Peaks_label[0],gstd[0]))
        print("{} valley position (mu) sig * std = {}
degrees".format(Peaks_label[1],gstd[3]))
        print("{} valley sigma sig * std = {}".format(Peaks_label[0],gstd[1]))
        print("{} valley sigma sig * std = {}".format(Peaks_label[1],gstd[4]))
        print("{} valley counts (amplitude) sig * std =
{}".format(Peaks_label[0],gstd[2]))
        print("{} valley counts (amplitude) sig * std =
{}".format(Peaks_label[1],gstd[5]))

        # Save fit info
        if Peaks_label[0]=='Wall':
            savefit[ii,0] = float(Gauss1Pos)
            savefit[ii,1] = gpar[1]
            savefit[ii,2] = gpar[2]
            savefit[ii,3] = float(Gauss2Pos)
            savefit[ii,4] = gpar[4]
            savefit[ii,5] = gpar[5]
            savefit[ii,6] = gpar[6]
            savestd[ii,:] = gstd
        else:
            savefit[ii,3] = float(Gauss1Pos)
            savefit[ii,4] = gpar[1]
            savefit[ii,5] = gpar[2]
            savefit[ii,0] = float(Gauss2Pos)
            savefit[ii,1] = gpar[4]
            savefit[ii,2] = gpar[5]
            savefit[ii,6] = gpar[6]
            savestd[ii,3:5] = gstd[0:2]
            savestd[ii,0:2] = gstd[3:5]
            savestd[ii,6] = gpar[6]

        # Calculate error in positioning
        if Peaks_label[0]=="Wall":
            theta_error=np.absolute(WallPos - Gauss1Pos)
            phi_error=np.absolute(FinPos - Gauss2Pos)
        else:
```

```
            theta_error=np.absolute(WallPos - Gauss2Pos)
            phi_error=np.absolute(FinPos - Gauss1Pos)
        # Calculate accuracy objective function
        Acc = Acc + theta_error + phi_error

        # Check to see if Wall peak/width is greater or less than Fin peak/width
for all phi
        # This condition guarantees that the wall and fin can be distinguished.
If they switch and
        # the relative peak distance are the same for two phis, then the
wall/fin can't be distinguished.
        # The phi=0 MAC does not catch this (but full MAC loop would).  Since
focused on Spartan design set
        # BestRatio to nan to ignore this design.  (This issue requires the fin
to pass 180 degrees from the wall)
        if gpar[2]>1e-8 and gpar[5]>1e-8 and gpar[1]>1e-8 and gpar[4]>1e-8:
            # Wall is negative if larger, fin is positive
            # If first peak is smaller
            if gpar[2] + gstd[2]  < gpar[5] - gstd[5]:
                if Peaks_label[0]=='Wall':
                    DistPeak = DistPeak + 1
                else:
                    DistPeak = DistPeak - 1
                RatioWallFinPeak[0,ii]=(gpar[2] + gstd[2])/(gpar[5] - gstd[5])
            # If first peak is larger
            elif gpar[2] - gstd[2] > gpar[5] + gstd[5]:
                if Peaks_label[0]=='Wall':
                    DistPeak = DistPeak - 1
                else:
                    DistPeak = DistPeak + 1
                RatioWallFinPeak[0,ii]=(gpar[5] + gstd[5])/(gpar[2] - gstd[2])
            else:
                # If indistinguishable, add zero
                # Don't calculate ratio
                print('Peak Indistinguishable')
                RatioWallFinPeak[0,ii]=np.inf
            # If first peak is less wide
            if gpar[1] + gstd[1] < gpar[4] - gstd[4]:
                if Peaks_label[0]=='Wall':
                    DistWidth = DistWidth + 1
                else:
                    DistWidth = DistWidth - 1
                RatioWallFinWidth[0,ii]=(gpar[1] + gstd[1])/(gpar[4] - gstd[4])
            # If first peak is wider
            elif gpar[1] - gstd[1] > gpar[4] + gstd[4]:
                if Peaks_label[0]=='Wall':
                    DistWidth = DistWidth - 1
                else:
                    DistWidth = DistWidth + 1
                RatioWallFinWidth[0,ii]=(gpar[4] + gstd[4])/(gpar[1] - gstd[1])
            else:
                # If indistinguishable, add zero
                # Don't calculate ratio
                print('Width Indistinguishable')
                RatioWallFinWidth[0,ii]=np.inf
        else:
            print('Non-physical curve fit (amplitude is zero or negative or no
peak width).')
            print('Return inf for Peak and Width.')
            RatioWallFinPeak[0,ii]=np.inf
            RatioWallFinWidth[0,ii]=np.inf
    print('Number of distiguishable peaks',DistPeak)
    print('Number of distiguishable widths',DistWidth)
    print('Peak ratios:',RatioWallFinPeak)
```

```
    print('Width ratios:',RatioWallFinWidth)
    # Get ratio for worst case of best distinguishable method
    # If couldn't perform ID a DRC, set all ratios to inf
    if np.any(np.isinf(RatioWallFinPeak)):
        PeakRatio = np.inf
    else:
        PeakRatio = np.max(RatioWallFinPeak)
    if np.any(np.isinf(RatioWallFinWidth)):
        WidthRatio = np.inf
    else:
        WidthRatio = np.max(RatioWallFinWidth)
    # If both methods were distinguishable, pick the most accurate one to be the
smallest ratio
    if np.absolute(DistPeak) == np.size(DRMphi,1) and np.absolute(DistWidth) ==
np.size(DRMphi,1):
        BestRatio = np.min([PeakRatio,WidthRatio])
        Dist=2
        if PeakRatio < WidthRatio:
            AccMethod="Peak"
        else:
            AccMethod="Width"
    elif np.absolute(DistPeak) == np.size(DRMphi,1):
        # If only the peak method was distinguishable, use the corresponding
values
        BestRatio = PeakRatio
        AccMethod="Peak"
        Dist=1
    elif np.absolute(DistWidth) == np.size(DRMphi,1):
        # If only the width method was distinguishable, use the corresponding
values
        BestRatio = WidthRatio
        AccMethod="Width"
        Dist=1
    else:
        # If neither methods was distinguishable, set the ratio to inf and don't
choose a best
        BestRatio = np.inf
        AccMethod="none"
        Dist=0

    # Calculated required time
    baseline = np.transpose(np.max(DRM,0))-savefit[:,6]
    # Make sure minimum has at least minCforGauss counts
    m_theta=np.min(DRM,axis=0)
    m_phi=np.min(DRM,axis=1)
    min_phi=np.argmin(m_theta)
    min_theta=np.argmin(m_phi)
    minSP1 =
minCforGauss*np.exp(float(LinAtten)*Geo[min_theta,min_phi])/(det_atten(DRMphi[0,
min_phi])/VRval)
    # For peak differentiation make sure enough counts for each phi position
    # Wall to fin or fin to wall required counts
    if np.any(np.isinf(baseline-savefit[:,2])) or np.any(np.isinf(baseline-
savefit[:,5])) or np.any(baseline-savefit[:,2]<0) or np.any(baseline-
savefit[:,5]<0):
        minSP2=np.inf
    else:
        minSP2 = np.max(sig**2 * np.divide(np.square(np.sqrt(baseline-
savefit[:,2]) + np.sqrt(baseline-savefit[:,5])),np.square(savefit[:,2] -
savefit[:,5])))*minSP1
    # Wall/fin to baseline
    if np.any(np.isinf([savefit[:,2], savefit[:,5]])) or
np.any(np.min([savefit[:,2], savefit[:,5]],0)<0):
        minSP3=np.inf
```

```
    else:
        Npeak = np.min([savefit[:,2], savefit[:,5]],0)
        minSP3 = np.max(sig**2 * np.divide(np.square(np.sqrt(baseline-Npeak) +
np.sqrt(baseline)),np.square(Npeak)))*minSP1
    if np.any(np.isinf(baseline-savefit[:,2])) or np.any(baseline-
savefit[:,2]<0):
        minSP4_1=np.inf
    else:
        minSP4_1 = 4 * np.max(sig**2 * np.divide(np.square(np.sqrt(baseline-
savefit[:,2]/2) + np.sqrt(baseline-
savefit[:,2])),np.square(savefit[:,2])))*minSP1
    if  np.any(np.isinf(baseline-savefit[:,5])) or np.any(baseline-
savefit[:,5]<0):
        minSP4_2=np.inf
    else:
        minSP4_2 = 4 * np.max(sig**2 * np.divide(np.square(np.sqrt(baseline-
savefit[:,5]/2) + np.sqrt(baseline-
savefit[:,5])),np.square(savefit[:,5])))*minSP1
    # Required counts for Peak and Width methods
    TPeak = max(minSP1, minSP2, minSP3)
    TWidth = max(minSP1, minSP3, max(minSP4_1,minSP4_2))
    # Method with the fewest required number of counts
    BestNSP=min(TPeak,TWidth)
    if TPeak < TWidth:
        TimeMethod = "Peak"
    else:
        TimeMethod = "Width"

    # Track average number of particles in DRM
    AvgNumPart=np.mean(NumPart)

      #Return accuracy, distingishability, and time objective functions and
other parameters of interest.
    return
Acc,Dist,PeakRatio,WidthRatio,BestRatio,AccMethod,TPeak,TWidth,BestNSP,TimeMetho
d,AvgNumPart,savefit,savelabel,baseline,savestd,minSP1,minSP2,minSP3,minSP4_1,mi
nSP4_2

# ***********************************************************************
# Start analysis
# ***********************************************************************
# Variables passed into Analysis code

#DakotaOutput=sys.argv[1]
#Outputmatrix=sys.argv[2]
#Sourcetype=sys.argv[3]
s_subdiv=float(sys.argv[4])
deltaphi=float(sys.argv[5])
#phifinal=float(sys.argv[6])
deltatheta=float(sys.argv[7])
MainDir = sys.argv[8]
VRval = float(sys.argv[9])
maskdensity = float(sys.argv[10])
#StartPhi = float(sys.argv[11])
Geofile = sys.argv[12]
#LinAtten =  sys.argv[13]
#plotfig = sys.argv[14]
wallwidth = int(sys.argv[15])
finwidth = int(sys.argv[16])
GeoStart = float(sys.argv[17])
GeoFinal = float(sys.argv[18])
#DR = float(sys.argv[19])
#DH = float(sys.argv[20])
#wallthick = float(sys.argv[21])
```

```
#finthick = float(sys.argv[22])
#Detmu = float(sys.argv[23])

# Minimum number of counts to approximate Poisson process (then Gaussian) at
peak
minCforGauss = 30


DRMtheta=np.transpose(np.arange(deltatheta*(0.5-np.floor(s_subdiv/2)/
s_subdiv),360,deltatheta/s_subdiv))
DRMphi=np.array(np.arange(float(sys.argv[11]),float(sys.argv[6]),deltaphi))
[np.newaxis]
print('PHI SET TO MATCH VOXEL EDGE IN MCNP CODE!!!!!!!!!!!!!!!')

# Call code to get DRM, MAC values, and sensitivity
AutoPMAC, DRM, DRMred, DRMmean, maxMat, maxMAC, maxSingleMAC, Sens,
AvgCountsAngle, \
    avgMAC, avgSingleMAC, Geo = RSManalyze(MainDir,VRval,Geofile,minCforGauss, \
    s_subdiv,DRMtheta,DRMphi,GeoStart,GeoFinal)
# *********************************************************************
# Get min time and ID accuracy
# *********************************************************************
# Map phi values to geometry so know fin/wall positions for different DRC phis
Geophi=np.arange(GeoStart+deltaphi/2,GeoFinal-deltaphi/2+0.00001,deltaphi)

# Calculate required number of particles, accuracy objective functions and other
parameters of interest
Acc,Dist,DPeak,DWidth,BestRatio,AccMethod,TPeak,TWidth,BestNSP,TimeMethod, \

AvgNumPart,savefit,savelabel,baseline,savestd,minSP1,minSP2,minSP3,minSP4_1,minS
P4_2 \
=
CalcMinTime(DRM,DRMtheta,DRMphi,MainDir,VRval,deltatheta,minCforGauss,sys.argv[1
3],Geo,sig,wallwidth,finwidth,s_subdiv)
#Pick the best combination of time and accuracy (note accuracy must be positive)
if Dist < 2:
    # Case where there is a clear winner
    if AccMethod == "Peak":
        Tmin=TPeak
        Amin=Acc
    elif AccMethod == "Width":
        Tmin=TWidth
        Amin=Acc
    else:
        Tmin=np.inf
        Amin=np.inf
        print("")
        print('WARNING: curves could not be distinguished using the peak values
or width. This could')
        print('be from bad design parameters or too few simulated particles.
Check the number')
        print('of simulated particles and consider increasing it if too few
particles are interacting.')
        print("")
else:
    # Case where both methods were distinguishable.  Pick the method requiring
the least counts.
    Amin=Acc
    Tmin=np.min([TWidth,TPeak])

# Calculate total time for one, constant velocity rotation
NAngles=np.size(DRMtheta)*np.size(DRMphi)
Ttotal=Tmin*np.size(DRMtheta)/np.min([wallwidth,finwidth])
Tavg=Tmin
```

```
AvgAcc=Amin/np.size(DRMphi)
# ***********************************************************************
# Write output
# ***********************************************************************
# Save DRM info
pd.set_option('display.max_colwidth', 1000)
Outputdata = {"DRM":DRM, "DRMred":DRMred, "DRMtheta":DRMtheta,
"DRMphi":DRMphi, \
              "savefit":savefit, "savelabel":savelabel, "minSP1":minSP1,
"minSP2":minSP2, \
              "minSP3":minSP3, "minSP4_1":minSP4_1, "minSP4_2":minSP4_2}
# Save output
f = open(sys.argv[2]+"OutVars.txt","w")
f.write( str(Outputdata) )
f.close()
np.savetxt(sys.argv[2]+"DRM.txt",DRM,newline="\n")
np.savetxt(sys.argv[2]+"Fit.txt",savefit,newline="\n")
np.savetxt(sys.argv[2]+"Baseline.txt",baseline,newline="\n")
np.savetxt(sys.argv[2]+"STD.txt",savestd,newline="\n")
# Pass output to Dakota via file
f1=open(sys.argv[1], 'w')
f1.write("%s %f\n%s %f\n%s %e\n%s %f\n%s %f\n%s %f\n%s %i\n%s %f\n%s %s\n%s %f\n
%s %f\n%s %f\n%s %s\n%s %e\n%s %e\n%s %f\n%s %f\n%s %f" % ("maxMAC", \
maxMAC,"maxSingleMAC", maxSingleMAC,"-AvgCountsAngle", -AvgCountsAngle,
"avgMAC", avgMAC, "avgSingleMAC", \
avgSingleMAC, "Acc", Acc,"Dist", Dist, "BestRatio", BestRatio, "AccMethod",
AccMethod,"TPeak", TPeak,"TWidth", \
TWidth,"BestNSP", BestNSP, "TimeMethod",TimeMethod, "Tavg", Tavg, "Ttotal",
Ttotal, "Amin", Amin, "AvgAcc", AvgAcc, \
"AvgNumPart",AvgNumPart))
f1.close()
C_df=pd.DataFrame(data=DRM,index=np.ravel(DRMtheta),columns=np.ravel(DRMphi))
C_df.to_pickle("Counts.pkl")

toc = time.perf_counter()
print("Ran in "+str(toc - tic)+" seconds")
```