```
# 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
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 gprocess(MainDir=None, lowerbound=None, upperbound=None, lbins=None,
ubins=None, nbins=None, VRval=None, nSP=None):
    """
    Load Geant response data, create DRC from FEP data and save to DRM
    Lower bound is in keV - total counts for energies above this value
```

```
    (default 1 keV)
    """

    #Read in the input tally results
    ftemp1 = os.listdir(MainDir)
    ftemp2 = []
    # Get Geant response filenames
    for names in ftemp1:
        if names.endswith(".o"):
            ftemp2.append(names)
    ftemp = sorted(ftemp2)
    # Must use 3 digits for theta and phi in naming to work (aka write 90 as
090)

    # Initialize variables for response theta and phi
    rtheta=-1*ones((1,np.size(ftemp, 0)))
    rphi=-1*ones((1,np.size(ftemp, 0)))
    DRM=zeros((1,1))
    for gg in range(0,np.size(ftemp, 0)):
        # Get the measurement angle from the filename
        rtheta[0,gg] = float(ftemp[gg][-18:-11])
        rphi[0,gg] = float(ftemp[gg][-9:-2])

        fname = ftemp[gg][:-2]
        #Get header info
        fid = open((''.join([MainDir, "/", fname, ".o"])), "r")
        # Read the data
        binstemp, valtemp, eDeptemp = getData(fid, lbins, ubins, int(nbins),
VRval, nSP)
        fid.close()

        # Track the unique measurement angles
        utheta = np.unique(rtheta[rtheta > -1], axis=1)
        uphi = np.unique(rphi[rtheta > -1], axis=1)

        # Apply desired energy cuts
        pickbins = binstemp > float(lowerbound)
        ind1 = np.nonzero(rtheta[0,gg] == utheta)
        ind2 = np.nonzero(rphi[0,gg] == uphi)
        # Save values to DRM matix.  Then remove extra entries in matrix
        if np.size(DRM,0) < np.size(utheta,1): DRM = np.insert(DRM,
np.size(utheta,1)-1, 0, axis=0)
        if np.size(DRM,1) < np.size(uphi,1): DRM = np.insert(DRM,
np.size(uphi,1)-1, 0, axis=1)
        DRM[ind1[1], ind2[1]] = sum(valtemp[pickbins[:-1]])

    # Plot the DRM if desired (defaults to plotting when running specific
design's analysis script)
    if np.size(DRM,1)>1 and sys.argv[18] == '1':
      # Plot DRM
      pdata =
[go.Surface(x=np.ravel(uphi),y=np.ravel(utheta),z=DRM,showscale=False)]
      layout = dict(autosize=True,scene=dict(
        xaxis=dict(title=u"\u03D5"),yaxis=dict(title=u"\
u03D1"),zaxis=dict(title='DRM',exponentformat='E'),
        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 + '/DRM.html',
include_mathjax='cdn')

      # Plot number of counts
      pdata =
[go.Surface(x=np.ravel(uphi),y=np.ravel(utheta),z=DRM*nSP/VRval,showscale=False)
```

```
]
      layout = dict(autosize=True,scene=dict(
        xaxis=dict(title=u"\u03D5"),yaxis=dict(title=u"\
u03D1"),zaxis=dict(title='Counts',exponentformat='E'),
        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 + '/Counts.html',
include_mathjax='cdn')

    # Return DRM and matching measurement angles
    return DRM, utheta, uphi

def getData(fid=None, lbins=None, ubins=None, nbins=None, VRval=None, nSP=None):
    """ Load Geant response data"""
    # Load Geant particle info
    temp = np.genfromtxt(fid,
           dtype={'names': ('initial','counts', 'runNum','eventid'),
           'formats': ('f8', 'f8', 'i8', 'i8')},invalid_raise = False)
    # Keep count info
    if np.size(temp) == 0:
        counts = [0.]
    elif np.size(temp) == 1:
        counts = [0, temp['counts']]
    else:
        counts = temp[:]['counts']
    # Create desired energy bins
    bins = np.linspace(float(lbins), float(ubins), num = nbins)

    # Initialize variables
    nParts = np.size(counts)
    val = zeros((nbins-1,1))
    eDep = zeros((nbins-1,1))
    for ii in range(0,nParts):
        if (counts[ii] > 0):
            # Put the particle contribution into the appropriate bin
            # Energy deposition
            eDep[:][(bins[:-1] <= counts[ii]) & (bins[1:] > counts[ii])] +=
counts[ii]*VRval/nSP
            # Counts
            val[:][(bins[:-1] <= counts[ii]) & (bins[1:] > counts[ii])] +=
VRval/nSP
    #Return bins, counts, and energy deposition
    return bins, val, eDep

def RSManalyze(MainDir, lowerbound, upperbound, lbins, ubins, nbins, VRval,
nSP):
    """
    Obtain the DRM then analyze the MAC for the non-imaging and imaging system
followed by the sensitivity
    and average number of counts per angle
    """
    # Obtain the DRM
    DRM, DRMtheta, DRMphi = gprocess(MainDir, lowerbound, upperbound, lbins,
ubins, nbins, VRval, nSP)

    # 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()
    avgMAC = np.mean(maxMat[maxMat > 0])
    # Calculate the sensitivity
    Sens = ((DRM).max(0) / (DRM).min(0)).min()
    # Record average number of counts per angle
    AvgCountsAngle = np.sum(np.sum(DRM)) / float(np.size(DRM,0) *
np.size(DRM,1))

      # Return DRMs and metrics
    return AutoPMAC, DRM, DRMred, DRMmean, DRMtheta, DRMphi, maxMat, maxMAC,
maxSingleMAC, Sens, AvgCountsAngle, avgMAC, avgSingleMAC

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,nSP,minCforGauss,sig,wallwidth,fin
```

```
width,deltatheta,s_subdiv,Geophi):
    """ 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]*nSP/VRval
        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
        # 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[18] == '1':
          gfit=go.Scatter(x=thetaval,y=gaussfit,
            mode='lines',name='Inverse Gaussian
Fit',line=dict(width=6))#dash='dash',width=6))

pdata=go.Scatter(x=thetaval,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))#dash='dash',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

        # 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*nSP/VRval,0))-savefit[:,6]
    # Make sure minimum has at least minCforGauss counts
    minSP1 = minCforGauss / np.min(MinCounts) * nSP
    # 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])))*nSP
    # Wall/fin to baseline required counts
    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)))*nSP
    # Width method: Half-power point required counts
    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])))*nSP
    # Width method: Half-power point required counts
    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])))*nSP#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

# ***********************************************************************
# 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])
nSP = float(sys.argv[10])
lcut = float(sys.argv[11])
hcut = float(sys.argv[12])
lbins = float(sys.argv[13])
ubins = float(sys.argv[14])
nbins = float(sys.argv[15])
wallwidth = int(sys.argv[16])
finwidth = int(sys.argv[17])
#plotfig = sys.argv[18]
GeoStart = float(sys.argv[19])
GeoFinal = float(sys.argv[20])

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

# Call code to get DRM, MAC values, and sensitivity
AutoPMAC, DRM, DRMred, DRMmean, DRMtheta, DRMphi, maxMat, maxMAC, maxSingleMAC,
Sens, AvgCountsAngle, avgMAC, avgSingleMAC =
RSManalyze(MainDir,lcut,hcut,lbins,ubins,nbins, VRval, nSP)

# ***********************************************************************
```

```
# 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,AvgNum
Part,savefit,savelabel,baseline,savestd =
CalcMinTime(DRM,np.transpose(DRMtheta),DRMphi,MainDir,VRval,nSP,minCforGauss,sig
,wallwidth,finwidth,deltatheta,s_subdiv,Geophi)

# 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])/VRval
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}
# 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()
```