#BGK CNT

self.minPixelStabilityCNT = self.ui.SBMinPS.value()

self.maxPixelStabilityCNT = self.ui.SBMaxPS.value()

self.useHistoryCNT = self.ui.CBHistory.isChecked()

self.isParallelCNT = self.ui.CBIsParallel.isChecked()

#BGK GMG

self.initializationFramesGMG = self.ui.SBInitialF.value()

self.decisionThresholdGMG = self.ui.DSBThreshold.value()

#BGK GSOC

self.nSamplesGSOC = self.ui.SBNSamples.value()

self.replaceRateGSOC = self.ui.DSBReplace.value()

self.propagationRateGSOC = self.ui.DSBPropagation.value()

self.hitsThresholdGSOC = self.ui.SBNThreshold.value()

self.alphaGSOC = self.ui.DSBalpha.value()

self.betaGSOC = self.ui.DSBBeta.value()

#BGK KNN

self.historyKNN = self.ui.SBHistory.value()

self.dist2ThresholdKNN = self.ui.DSBThreshold\_2.value()

self.detectShadowsKNN = self.ui.CBDShadowsKNN.isChecked()

#BKG LSBP

self.nSamplesLSBP = self.ui.SBNSamplesLSBP.value()

self.LSBPRadius = self.ui.SBLRadius.value()

self.TupperLSBP = self.ui.DSBTupper.value()

self.RscaleLSBP = self.ui.DSRScale.value()

self.LSBPthreshold = self.ui.DSBLThreshold.value()

self.minCountLSBP = self.ui.DSBPmCount.value()

#BKG MOG

self.historyMOG = self.ui.SBHistoryMOG.value()

self.nmixturesMOG = self.ui.SBMixtures.value()

self.backgroundRatioMOG = self.ui.DSBRatio.value()

self.noiseSigmaMOG = self.ui.SBNoise.value()

#BKG MOG2

self.historyMOG2 = self.ui.SBHistoryMOG2.value()

self.varThresholdMOG2 = self.ui.SBThresholdMOG2.value()

self.detectShadowsMOG2 = self.ui.CBDshadowsMOG2.isChecked()

#BKG RNB

self.alphaRNB = self.ui.DSBAlpha.value()

self.ThuRNB = self.ui.SBThu.value()

self.ThsRNB = self.ui.SBThs.value()

#BKG FZBL

self.alphaFZBL = self.ui.DSBAlphaFL.value()

self.ThsFZBL = self.ui.SBThSFL.value()

self.ThfsFZBL= self.ui.DSBThfsFL.value()

#BKG FZB

self.alphaFZB = self.ui.DSBAlphaFZB.value()

self.alphaMinFZB = self.ui.DSBAlphaFZB\_2.value()

self.ThsFZB = self.ui.SBThSFZB.value()

self.ThfsFZB = self.ui.DSBThfsFZB.value()

#Image Img Subtractor

self.alphaSUB = self.ui.DSBAlphaI.value()

self.bethaSUB = self.ui.DSBBetaI.value()

self.constantSUB = self.ui.DSBConstant.value()

#Image Lapse

self.alphaLapse = self.ui.DSBAlphaL.value()

self.bethaLapse = self.ui.DSBBetaL.value()

self.constantLapse = self.ui.DSBConstantL.value()

#BGK CNT

self.minPixelStabilityCNT

self.maxPixelStabilityCNT

self.useHistoryCNT

self.isParallelCNT

#BGK GMG

self.initializationFramesGMG

self.decisionThresholdGMG

#BGK GSOC

self.nSamplesGSOC

self.replaceRateGSOC

self.propagationRateGSOC

self.hitsThresholdGSOC

self.alphaGSOC

self.betaGSOC

#BGK KNN

self.historyKNN

self.dist2ThresholdKNN

self.detectShadowsKNN

#BKG LSBP

self.nSamplesLSBP

self.LSBPRadius

self.TupperLSBP

self.RscaleLSBP

self.LSBPthreshold

self.minCountLSBP

#BKG MOG

self.historyMOG

self.nmixturesMOG

self.backgroundRatioMOG

self.noiseSigmaMOG

#BKG MOG2

self.historyMOG2

self.varThresholdMOG2

self.detectShadowsMOG2

#BKG RNB

self.alphaRNB

self.ThuRNB

self.ThsRNB

#BKG FZBL

self.alphaFZBL

self.ThsFZBL

self.ThfsFZBL

#BKG FZB

self.alphaFZB

self.alphaMinFZB

self.ThsFZB

self.ThfsFZB

#Image Img Subtractor

self.alphaSUB

self.bethaSUB

self.constantSUB

#Image Lapse

self.alphaLapse

self.bethaLapse

self.constantLapse

# BGK CNT  
'minPixelStabilityCNT': self.minPixelStabilityCNT,  
'maxPixelStabilityCNT': self.maxPixelStabilityCNT,  
'useHistoryCNT': self.useHistoryCNT,  
'isParallelCNT': self.isParallelCNT,  
# BGK GMG  
'initializationFramesGMG': self.initializationFramesGMG,  
'decisionThresholdGMG': self.decisionThresholdGMG,  
# BGK GSOC  
'nSamplesGSOC': self.nSamplesGSOC,  
'replaceRateGSOC': self.replaceRateGSOC,  
'propagationRateGSOC': self.propagationRateGSOC,  
'hitsThresholdGSOC': self.hitsThresholdGSOC,  
'alphaGSOC': self.alphaGSOC,  
'betaGSOC': self.betaGSOC,  
# BGK KNN  
'historyKNN': self.historyKNN,  
'dist2ThresholdKNN': self.dist2ThresholdKNN,  
'detectShadowsKNN': self.detectShadowsKNN,  
# BKG LSBP  
'nSamplesLSBP': self.nSamplesLSBP,  
'LSBPRadius': self.LSBPRadius,  
'TupperLSBP': self.TupperLSBP,  
'RscaleLSBP': self.RscaleLSBP,  
'LSBPthreshold': self.LSBPthreshold,  
'minCountLSBP': self.minCountLSBP,  
# BKG MOG  
'historyMOG': self.historyMOG,  
'nmixturesMOG': self.nmixturesMOG,  
'backgroundRatioMOG': self.backgroundRatioMOG,  
'noiseSigmaMOG': self.noiseSigmaMOG,  
# BKG MOG2  
'historyMOG2': self.historyMOG2,  
'varThresholdMOG2': self.varThresholdMOG2,  
'detectShadowsMOG2': self.detectShadowsMOG2,  
# BKG RNB  
'alphaRNB': self.alphaRNB,  
'ThuRNB': self.ThuRNB,  
'ThsRNB': self.ThsRNB,  
# BKG FZBL  
'alphaFZBL': self.alphaFZBL,  
'ThsFZBL': self.ThsFZBL,  
'ThfsFZBL': self.ThfsFZBL,  
# BKG FZB  
'alphaFZB': self.alphaFZB,  
'alphaMinFZB': self.alphaMinFZB,  
'ThsFZB': self.ThsFZB,  
'ThfsFZB': self.ThfsFZB,  
# Image Img Subtractor  
'alphaSUB': self.alphaSUB,  
'bethaSUB': self.bethaSUB,  
'constantSUB': self.constantSUB,  
# Image Lapse  
'alphaLapse': self.alphaLapse,  
'bethaLapse': self.bethaLapse,  
'constantLapse': self.constantLapse

Thu = 30  
#Ths = 30  
#Thfs = 0.4  
Ths = self.ThsFZBL  
Thfs = self.ThfsFZBL  
alphamin = 0.9

img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, img, 0)), self.bethaSUB, self.constantSUB)

Thu = 30  
#Ths = 30  
#Thfs = 0.4  
#alphamin = 0.9  
Ths = self.ThsFZB  
Thfs = self.ThfsFZB  
alphamin = self.alphaMinFZB

img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, img, 0)), self.bethaSUB, self.constantSUB)

#alpha = 0.9  
#Thu = 30  
#Ths = 30  
alpha = self.alphaRNB  
Thu = self.ThuRNB  
Ths = self.ThsRNB

if self.ui.TWImageV.currentIndex() == 0:  
 self.bgkSubtractor()  
 if (self.subtractor <= 8):  
 if (self.subtractor == 3):  
 if (self.subtractor == 3 and self.imagedisplay ==3 ):  
 if first\_imgGMG:  
 self.ForegroundGMG = img  
 first\_imgGMG = False  
 else:  
 self.ForegroundGMG = self.ImageBackground(img, self.subtractor, self.imagedisplay)  
 img = self.ForegroundGMG  
 else:  
 first\_imgGMG = True  
 img = self.ImageBackground(img, self.subtractor, self.imagedisplay)  
 if (self.subtractor == 7):  
 if (self.subtractor == 7 and self.imagedisplay ==3 ):  
 if first\_imgMOG:  
 self.ForegroundMOG = img  
 first\_imgMOG = False  
 else:  
 self.ForegroundMOG = self.ImageBackground(img, self.subtractor, self.imagedisplay)  
 img = self.ForegroundMOG  
 else:  
 first\_imgMOG = True  
 img = self.ImageBackground(img, self.subtractor, self.imagedisplay)  
 if (self.subtractor != 3 and self.subtractor != 7 ):  
 first\_imgGMG = True  
 first\_imgMOG = True  
 img = self.ImageBackground(img, self.subtractor, self.imagedisplay)  
 #  
 # Running fuzzy lineal  
 if (self.subtractor == 10):  
 Thu = 30  
 #Ths = 30  
 #Thfs = 0.4  
 Ths = self.ThsFZBL  
 Thfs = self.ThfsFZBL  
 alphamin = 0.9  
 Height, Width = img.shape  
 InputImage = np.asarray(img, dtype=np.float32)  
 if first\_imgFL:  
 Background\_Image = InputImage  
 Background\_Updated = np.zeros((Height, Width))  
 Background\_Updated = np.asarray(Background\_Updated,  
 dtype=np.float32)  
 Result\_Comparison = np.zeros((Height, Width))  
 Result\_Comparison = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 Background\_Substraction = np.zeros((Height, Width))  
 Background\_Substraction = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 alpha = np.zeros((Height, Width))  
 alpha = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 Saturating\_Limiter = np.zeros((Height, Width))  
 Saturating\_Limiter = np.asarray(Saturating\_Limiter, dtype=np.float32)  
 Fuzzy\_Background\_Substraction = np.zeros((Height, Width))  
 Fuzzy\_Background\_Substraction = np.asarray(Saturating\_Limiter, dtype=np.float32)  
 first\_imgFL = False  
  
 else:  
 Result\_Comparison = np.absolute(np.subtract(InputImage,  
 Background\_Image))  
 Saturating\_Limiter = np.divide(Result\_Comparison, Ths)  
 Fuzzy\_Background\_Substraction = np.where(Result\_Comparison > Ths,  
 np.ones((Height, Width)),  
 Saturating\_Limiter)  
 Abs\_LPF\_Fuzzy\_Background\_Substraction = np.absolute(cv2.blur(  
 Fuzzy\_Background\_Substraction, (3, 3)))  
  
 Background\_Substraction = np.where(Abs\_LPF\_Fuzzy\_Background\_Substraction > Thfs,  
 np.ones((Height, Width)), 0)  
 alpha = 1 + np.multiply(0.1, (np.subtract(Fuzzy\_Background\_Substraction, 1)))  
 Background\_Updated = np.add(np.multiply(alpha, Background\_Image), np.multiply(  
 (1 - alpha), InputImage))  
 Background\_Image = Background\_Updated  
 if (self.imagedisplay == 1):  
 img = img  
 if (self.imagedisplay == 2):  
 img = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 if (self.imagedisplay == 3):  
 img = np.asarray(Background\_Image, dtype=np.uint8)  
 if (self.imagedisplay == 4):  
 bmg = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 fmg = np.asarray(Background\_Image, dtype=np.uint8)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 #img = cv2.addWeighted(fmg, 0.25,  
 # (np.where(bmg > fmg, img, 0)), 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, img, 0)), self.bethaSUB, self.constantSUB)  
  
 # imb = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 # img = cv2.addWeighted(imb, 0.25,img, 0.75, 0)  
 else:  
 first\_imgFL = True  
  
 # Running Fuzzy logic  
 if (self.subtractor == 11):  
 Thu = 30  
 #Ths = 30  
 #Thfs = 0.4  
 #alphamin = 0.9  
 Ths = self.ThsFZB  
 Thfs = self.ThfsFZB  
 alphamin = self.alphaMinFZB  
 Height, Width = img.shape  
 InputImage = np.asarray(img, dtype=np.float32)  
 if first\_imgF:  
 Background\_Image = InputImage  
 Background\_Updated = np.zeros((Height, Width))  
 Background\_Updated = np.asarray(Background\_Updated,  
 dtype=np.float32)  
 Result\_Comparison = np.zeros((Height, Width))  
 Result\_Comparison = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 Background\_Substraction = np.zeros((Height, Width))  
 Background\_Substraction = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 alpha = np.zeros((Height, Width))  
 alpha = np.asarray(Result\_Comparison,  
 dtype=np.float32)  
 Saturating\_Limiter = np.zeros((Height, Width))  
 Saturating\_Limiter = np.asarray(Saturating\_Limiter, dtype=np.float32)  
 Fuzzy\_Background\_Substraction = np.zeros((Height, Width))  
 Fuzzy\_Background\_Substraction = np.asarray(Saturating\_Limiter, dtype=np.float32)  
 first\_imgF = False  
 else:  
 Result\_Comparison = np.absolute(np.subtract(InputImage,  
 Background\_Image))  
 Saturating\_Limiter = np.divide(Result\_Comparison, Ths)  
 Fuzzy\_Background\_Substraction = np.where(Result\_Comparison > Ths,  
 np.ones((Height, Width)),  
 Saturating\_Limiter)  
 Abs\_LPF\_Fuzzy\_Background\_Substraction = np.absolute(cv2.blur(  
 Fuzzy\_Background\_Substraction, (3, 3)))  
 Background\_Substraction = np.where(Abs\_LPF\_Fuzzy\_Background\_Substraction > Thfs,  
 np.ones((Height, Width)), 0)  
 alpha = np.subtract(1, np.multiply(  
 (1 - alphamin), np.exp(np.multiply(-5, Fuzzy\_Background\_Substraction))))  
 Background\_Updated = np.add(np.multiply(alpha, Background\_Image), np.multiply(  
 (1 - alpha), InputImage))  
 Background\_Image = Background\_Updated  
 # img = np.asarray(Background\_Image, dtype=np.uint8)  
 # img = np.multiply(Background\_Substraction, 255)  
 if (self.imagedisplay == 1):  
 img = img  
 if (self.imagedisplay == 2):  
 img = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 if (self.imagedisplay == 3):  
 img = np.asarray(Background\_Image, dtype=np.uint8)  
 if (self.imagedisplay == 4):  
 bmg = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 fmg = np.asarray(Background\_Image, dtype=np.uint8)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 #img = cv2.addWeighted(fmg, 0.25,  
 # (np.where(bmg > fmg, img, 0)), 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, img, 0)), self.bethaSUB, self.constantSUB)  
  
 # imb = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 # img = cv2.addWeighted(imb, 0.25,img, 0.75, 0)  
 else:  
 first\_imgF = True  
  
 # Running average  
 if self.subtractor == 9:  
 #alpha = 0.9  
 #Thu = 30  
 #Ths = 30  
 alpha = self.alphaRNB  
 Thu = self.ThuRNB  
 Ths = self.ThsRNB  
 Frame = img  
 if first\_imgR:  
 Background\_Image = Frame.astype(np.float32)  
 Background\_Substraction = np.zeros((Frame.shape[0], Frame.shape[1]))  
 first\_imgR = False  
 else:  
 Result\_Comparison = cv2.absdiff(Frame.astype(np.float32), Background\_Image)  
 Background\_Substraction = np.where(Result\_Comparison > Ths,  
 np.ones((Frame.shape[0], Frame.shape[1])), 0)  
 Background\_Updated = cv2.add(np.multiply(alpha, Background\_Image),  
 np.multiply((1 - alpha), Frame.astype(np.float32)))  
 Running\_Average\_Method = np.where(Result\_Comparison < Thu,  
 Background\_Updated, Background\_Image)  
 Background\_Image = Running\_Average\_Method  
 # img = np.multiply(Background\_Substraction,255)  
 if (self.imagedisplay == 1):  
 img = img  
 if (self.imagedisplay == 2):  
 img = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 if (self.imagedisplay == 3):  
 img = np.asarray(Background\_Image, dtype=np.uint8)  
 if (self.imagedisplay == 4):  
 bmg = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 fmg = np.asarray(Background\_Image, dtype=np.uint8)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 #img = cv2.addWeighted(fmg, 0.25,  
 # (np.where(bmg > fmg, img, 0)), 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, img, 0)), self.bethaSUB, self.constantSUB)  
  
 # imb = np.asarray(np.multiply(Background\_Substraction, 255), dtype=np.uint8)  
 # img = cv2.addWeighted(imb, 0.25, img, 0.75, 0)  
 # img = np.asarray(Background\_Image, dtype=np.uint8)  
 else:  
 first\_imgR = True  
  
 if (self.lapse == True):  
 if (cont == 0):  
 img0 = img  
 cont = 1  
 else:  
 #img0 = cv2.addWeighted(img0, 0.95, img, 0.05, 0)  
 #NEW VALUE OF LAPSE VIDEO  
 img0 = cv2.addWeighted(img0, self.alphaLapse, img, self.bethaLapse, self.constantLapse)  
 img = img0  
 # self.iv.setImage(img, autoRange=False, autoLevels=False, autoHistogramRange=False)  
 else:  
 cont = 0

def ImageBackground(self, inputImage, BGKimage, Dimage):  
 # ImageDefault  
 if (BGKimage == 1):  
 img = inputImage  
 if (Dimage == 1):  
 img = inputImage  
  
 if (BGKimage == 2):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbg.apply(inputImage)  
 if (Dimage == 3):  
 img = self.fgbg.apply(inputImage)  
 img = self.fgbg.getBackgroundImage()  
 if (Dimage == 4):  
 # fimg = self.fgbg.getBackgroundImage()  
 # bimg = img  
 bmg = self.fgbg.apply(inputImage)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(self.fgbg.getBackgroundImage(), self.alphaSUB,  
 (np.where(bmg > self.fgbg.getBackgroundImage(), inputImage, 0)),self.bethaSUB, self.constantSUB)  
 if (BGKimage == 3):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgGMG.apply(inputImage)  
 if (Dimage == 3):  
 # self.ForegroundGMG  
 # img = self.fgbgGMG.apply(inputImage)  
 # img = self.fgbgGMG.getBackgroundImage()  
 product = np.multiply(np.asarray(inputImage, dtype=np.float32),  
 np.asarray(self.fgbgGMG.apply(inputImage), dtype=np.float32))  
 img = np.where(product == 0, inputImage, self.ForegroundGMG)  
 # img = self.ForegroundGMG  
 if (Dimage == 4):  
 # fimg = self.fgbgGMG.getBackgroundImage()  
 # bimg = img  
 bmg = self.fgbgGMG.apply(inputImage)  
 product = np.multiply(np.asarray(inputImage, dtype=np.float32),  
 np.asarray(self.fgbgGMG.apply(inputImage), dtype=np.float32))  
 fmg = np.where(product == 0, inputImage, self.ForegroundGMG)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, inputImage, 0)),self.bethaSUB, self.constantSUB)  
  
 if (BGKimage == 4):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgGSOC.apply(inputImage)  
 if (Dimage == 3):  
 img = self.fgbgGSOC.apply(inputImage)  
 img = self.fgbgGSOC.getBackgroundImage()  
 if (Dimage == 4):  
 bmg = self.fgbgGSOC.apply(inputImage)  
 fmg = cv2.cvtColor(self.fgbgGSOC.getBackgroundImage(), cv2.COLOR\_BGR2GRAY)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, inputImage, 0)), self.bethaSUB, self.constantSUB)  
  
 if (BGKimage == 5):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgKNN.apply(inputImage)  
 if (Dimage == 3):  
 img = self.fgbgKNN.apply(inputImage)  
 img = self.fgbgKNN.getBackgroundImage()  
 if (Dimage == 4):  
 bmg = self.fgbgKNN.apply(inputImage)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(self.fgbgKNN.getBackgroundImage(), self.alphaSUB,  
 (np.where(bmg > self.fgbgKNN.getBackgroundImage(), inputImage, 0)), self.bethaSUB, self.constantSUB)  
  
 if (BGKimage == 6):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgLSBP.apply(inputImage)  
 if (Dimage == 3):  
 img = self.fgbgLSBP.apply(inputImage)  
 img = self.fgbgLSBP.getBackgroundImage()  
 if (Dimage == 4):  
 bmg = self.fgbgLSBP.apply(inputImage)  
 fmg = cv2.cvtColor(self.fgbgLSBP.getBackgroundImage(), cv2.COLOR\_BGR2GRAY)  
 # bmg = self.fgbgLSBP.apply(inputImage)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, inputImage, 0)), self.bethaSUB, self.constantSUB)  
  
 if (BGKimage == 7):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgMOG.apply(inputImage)  
 if (Dimage == 3):  
 # img = self.fgbgMOG.apply(inputImage)  
 # img = self.fgbgMOG.getBackgroundImage()  
 product = np.multiply(np.asarray(inputImage, dtype=np.float32),  
 np.asarray(self.fgbgMOG.apply(inputImage), dtype=np.float32))  
 img = np.where(product == 0, inputImage, self.ForegroundMOG)  
 # img = self.ForegroundMOG  
 if (Dimage == 4):  
 bmg = self.fgbgMOG.apply(inputImage)  
 product = np.multiply(np.asarray(inputImage, dtype=np.float32),  
 np.asarray(self.fgbgMOG.apply(inputImage), dtype=np.float32))  
 fmg = np.where(product == 0, inputImage, self.ForegroundMOG)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(fmg, self.alphaSUB,  
 (np.where(bmg > fmg, inputImage, 0)), self.bethaSUB, self.constantSUB)  
  
 if (BGKimage == 8):  
 if (Dimage == 1):  
 img = inputImage  
 if (Dimage == 2):  
 img = self.fgbgMOG2.apply(inputImage)  
 if (Dimage == 3):  
 img = self.fgbgMOG2.apply(inputImage)  
 img = self.fgbgMOG2.getBackgroundImage()  
 if (Dimage == 4):  
 bmg = self.fgbgMOG2.apply(inputImage)  
 # img = cv2.addWeighted(self.fgbg.apply(inputImage), 0.25,inputImage, 0.75, 0)  
 img = cv2.addWeighted(self.fgbgMOG2.getBackgroundImage(), self.alphaSUB,  
 (np.where(bmg > self.fgbgMOG2.getBackgroundImage(), inputImage, 0)), self.bethaSUB, self.constantSUB)  
 return img