```
# -*- coding: utf-8 -*-
"""
Created on Sat Jul 16 22:45:11 2016
@author: david

Volume class
custom FUNcitons using SimpleITK
https://itk.org/Wiki/SimpleITK/GettingStarted#Generic_Distribution

works only with cropped CT and MRI images (showing only one rod),
both Volumes should have the same PixelSpacing,
 and x and y PixelSpacing shoult be equal
sitk_write() creates .mha file with pixel values corresponding
 to distortion in pixel distance * PixelSpacing (mm)

important to remember:
    sitk.Image saves Volume like this (x,y,z)
    array returned by sitk.GetArrayFromImage(Image)
    is transposed: (z,y,x)

based on:
https://pyscience.wordpress.com/2014/10/19/image-segmentation-with-python-and-SimpleITK/

"""

import numpy as np
from scipy import ndimage
import SimpleITK as sitk
import matplotlib.pyplot as plt
import os
from skimage.draw import circle


class Volume:
    '''
    Create a Volume (SimpleITK.Image with convenient properties and functions)
    recommended use:
    create new Volume (optional use denoise=True)
    Volume.getThresholds()

    Parameters
    ----------
    path : string_like
        directory containing DICOM data
    method : string_like, recommended
        either "CT or "MR", used for automatic calculations
    denoise : bool, optional
        If true, the imported data will be denoised using
        SimpleITK.CurvatureFlow(image1=self.img,
                                timeStep=0.125,
                                numberOfIterations=5)
    ref : int, optional
        slice used to make calculations (idealy isocenter) e.g. thresholds
        all plots show this slice
        by default it is set to be in the middle of the image (z-axis)
    resample : int, optional
        resample rate, becomes part of title
    seeds : array_like (int,int,int), optional
        coordinates (pixel) of points inside rod, used for segmentation
        by default list of brightest pixel in each slice
    radius: double, optional
        overrides radius value (default CT:4mm, MR:2mm)
    spacing: double, optional
        by default SitpleITK.img.GetSpacing is used to find relation of pixels
        to real length (in mm)
    skip: int, optional
        neglecting first 'skip' number of slices
    leave: int, optional
        neglects last 'leave' number of slices
    rotate: bool, optional
        if True: mirrors x- & z-axis, effectively rotating the image by 180°
        (looked at from above), this is applied after skip&size
```

```
    '''
    def __init__(self, path=None, method=None, denoise=False, ref=None,
                 resample=False, seeds='auto', radius=0, spacing=0, skip=0,
                 leave=False, rotate=False):
        if(path is None):
            print("Error: no path given!")
        else:
            self.path = path
            self.method = method
            self.denoise = denoise
            self.resample = resample
            self.centroid = False
            self.mask = False
            self.masked = False
            self.title = method
            self.radius = radius
            self.bestRadius = 0
            self.lower = False
            self.upper = False

            file_no = len([name for name in os.listdir(path) if os.path.isfile(os.path.join(path,
name))])
            size = file_no - skip - leave
            if size <= 0:
                print("There nothing left to load after skipping {} file(s) and ignoring the last {}
files.".format(skip,leave))
                print("The directory only contains {} files!".format(file_no))
            else:

                print("Import {} DICOM Files from: {}\n".format(size, path))
                shortened_img = sitk_read(path, denoise)[:, :, skip:(file_no + skip - leave)]
                if rotate is True:
                    self.img = shortened_img[::-1,:,::-1]
                else:
                    self.img = shortened_img

                if (self.img and self.denoise):
                    a = self.title
                    self.title = a + " denoised"

                if resample:
                    a = self.title
                    self.title = a + ", x" + str(resample)

                self.xSize, self.ySize, self.zSize = self.img.GetSize()
                if spacing == 0:
                    self.xSpace, self.ySpace, self.zSpace = self.img.GetSpacing()

                if type(ref) == int:
                    self.ref = ref
                else:
                    self.ref = int(self.zSize / 2)

                # niceSlice used to remember which slices show irregularities such
                # as parts of plastic pane (CT)
                # and should therefore not be used to calculate COM, dice, etc.
                self.niceSlice = np.ones((self.zSize, 1), dtype=bool)
                self.maxBrightness = np.zeros((self.zSize, 1))
                self.meanBrightness = np.zeros((self.zSize, 1))
                arr = sitk.GetArrayFromImage(self.img)
                average = np.average(arr[ref])
    #             print("\nAverage @ ref: ", average)
                for index in range(self.zSize):
                    # save value of brightest pixel in each slice
                    self.maxBrightness[index] = arr[index].max()
                    self.meanBrightness[index] = np.average(arr[index])
                    # if average value of slice differs too much -> badSlice
                    # difference between ref-Slice and current chosen arbitratry
                    # seems to be big enough not to detect air bubble in MRI
                    # entire air block (no liquid) should be recognised, though.
                    # small enough to notice plastic pane
                    if np.absolute(self.meanBrightness[index] - average) > 40:
```

```
                        print("Irregularities detected in slice {}".format(index))
                        self.niceSlice[index] = False
                        # maybe also set slice prior and after current slice as
                        # self.niceSlice[index+1] = self.niceSlice[index+1] = False
                        # because small changes happening around irregularities
                        # might not have been big enough for detection, but already
                        # leading to false calculations?

                if type(seeds) == list:
                    self.seeds = seeds
                elif seeds == 'auto':
                    self.seeds = []
                    for index in range(self.zSize):
                        yMax = int(arr[index].argmax() / self.xSize)
                        xMax = arr[index].argmax() - yMax*self.xSize
                        if self.niceSlice[index] == True:
                            self.seeds.append((xMax, yMax, index))
    #                     print("{}: found max at ({},{})".format(index, xMax, yMax))

    def show(self, pixel=False, interpolation=None, ref=None, save=False):
        '''
        plots ref slice of Volume

        Parameters
        ----------
        pixel: bool, optional
            if True, changes axis from mm to pixels
        interpolation: "string", optional, default: 'nearest'
            using build-in interpolation of matplotlib.pyplot.imshow
            Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic',
            'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
            'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc',
            'lanczos'
        ref: int, optional
            slice to be plotted instead of self.ref (default: 0)
        '''

        if ref is None:
            ref = self.ref

        if interpolation is None:
            a = 'nearest'

        extent = None
        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)
        # The location, in data-coordinates, of the lower-left and upper-right corners
        # (left, right, bottom, top)

        sitk_show(img=self.img, ref=ref, extent=extent, title=self.title, interpolation=a, save=save)

    def showSeed(self, pixel=False, interpolation='nearest', ref=None, save=False):
        '''
        plots slice containing seed

        Parameters
        ----------
        pixel: bool, optional
            if True, changes axis from mm to pixels
        interpolation: "string", optional, default: 'nearest'
            using build-in interpolation of matplotlib.pyplot.imshow
            Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic',
            'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
            'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc',
            'lanczos'
        ref: int, optional
            slice of seed to be plotted instead of self.ref (default: zSize/2)
        '''
        if ref is None:
            ref = self.ref
```

```
        if type(self.seeds[ref]) != tuple:
            print("No seed found @ slice {}".format(ref))
            return None

        x, y = -1,-1
        extent = None
        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)
            x = (self.seeds[ref][0] * self.xSpace)
            y = (self.seeds[ref][1] * self.xSpace)
        else:
            x, y, z = self.seeds[ref]

        arr = sitk.GetArrayFromImage(self.img)
        fig = plt.figure()
        plt.set_cmap("gray")
        plt.title(self.title + ", seed @ {}".format(self.seeds[ref]))

        plt.imshow(arr[ref, :, :], extent=extent, interpolation=interpolation)
        plt.scatter(x, y)
        plt.show()
        if save != False:
            fig.savefig(str(save) + ".png")

    def getThresholds(self, pixelNumber=0, scale=1):
        '''
        Calculates threshold based on number of pixels representing rod.
        If no pixelNumber is given, self.radius is used to get estimated
        pixelNumber. If self.raduis == 0: use method to get raduis
         All calculations based on ref-slice.

        approx. number of pixels being part of rod:
        pn = realRadius^2 * pi / pixelSpacing^2

        Parameters
        ----------
        pixelNumber: int, optional
            if 0, uses self.radis to calculate pixelnumber
            if self.radius also 0, uses self.method instead (CT: 4mm, MR: 2mm)
        scale: double, optional
            factor altering pixelNumber

        Returns
        -------
        lower and upper threshold value: (double, double)
        '''

        if pixelNumber == 0:
            if self.radius != 0:
                realRadius = self.radius
            else:
                if self.method == "CT":
                    realRadius = 4
                if self.method == "MR":
                    realRadius = 2
                if self.method != "MR" and self.method != "CT":
                    print("method is unknown, please set pixelNumber!")
                    return None
            pixelNumber = np.power(realRadius, 2)*np.pi/np.power(self.xSpace, 2)*scale

        pn = int(pixelNumber)
        arr = sitk.GetArrayFromImage(self.img)
        self.upper = np.double(arr.max())

    #    hist, bins = np.histogram(arr[self.ref, :, :].ravel(), bins=100)
    # alternatively, increase number of bins for images with many pixels
        hist, bins = np.histogram(arr[self.ref, :, :].ravel(), bins=int(pn*2))
        self.lower = np.double(bins[np.argmax((np.cumsum(hist[::-1]) < pn)[::-1])])
        print("number of pixels (pn): {}\n lower: {}\n upper: {}".format(pn, self.lower, self.upper))

        return (self.lower, self.upper)
```

```
    def getCentroid(self, threshold='default', pixelNumber=0, scale=1,
                    percentLimit=False, iterations=5, top = 1,
                    plot=False, save=False):
        '''
        Calculates centroid, either by setting threshold or percentLimit

        Parameters
        ----------
        threshold: float or 'auto', default='auto'
            if 'auto': uses getThreshold(pixelnumber, scale) and then
                sitk_centroid(threshold=self.lower)
                sets self.lower and self.upper
        percentLimit: float from 0 to 1 (or "auto" =experimental)
            if percentLimit is True: used instead of threshold method
            if 'auto': makes 5 iterations by default, uses getThreshold()
                and getDice(), but does NOT set self.mask
                sets self.lower and self.upper
        plot, save: bool, optional
            plot and save iteration (percentLimit='auto')

        Returns
        -------
        self.centroid: numpy.ndarray
        '''

        if (threshold is False and percentLimit is False) or (percentLimit == "auto" and threshold is
not False and threshold != 'default'):
            print("Please use percentLimit or threshold! (default setting: threshold = 'auto')")
            return None

        if (percentLimit == "auto" and threshold is False) or (percentLimit == "auto" and threshold ==
'default'):
            # EXPERIMENTAL!!!
            # looks at whole range of possible percentLimits
            # reduces range by finding out which half yields higher result
            # starts at A=25% and B=75% of all pixels
            # if DC(A) > DC(B): next values come from lower half (0-50%)
            # else: upper half (50-100%)
            # calculates 5 centroids with different percentLimits
            # gets dice coefficient for each centroid percentLimit combination
            # returns best result

            print("\n\n")
            arr = sitk.GetArrayFromImage(self.img)
            direction = np.zeros(iterations)
            left = np.zeros(iterations)
            right = np.zeros(iterations)
            left[0] = 0
            right[0] = top
            guess = np.zeros(iterations)
            guess[0] = (left[0]+right[0])/2
            thresholdsA = np.zeros((iterations,2))
            thresholdsB = np.zeros((iterations,2))
            centroidScoreA = np.zeros(iterations)
            centroidScoreB = np.zeros(iterations)
            centroidsA = np.zeros((iterations, self.zSize, 2))
            centroidsB = np.zeros((iterations, self.zSize, 2))
            diceA = np.zeros((iterations, self.zSize, 1))
            diceB = np.zeros((iterations, self.zSize, 1))
            for index in range(iterations):
                print("    ITERATION #{}, current guess: ~{:.4f}\nA @ ~{:.4f}%".format(index, guess
[index]*100, (guess[index]+left[index])/2*100))
                thresholdsA[index] = self.getThresholds(pixelNumber=self.xSize*self.ySize*(guess[index]
+left[index])/2)
                # create mask including all pixels relevant for guess
                maskA = sitk.ConnectedThreshold(image1=self.img,
                                                seedList=self.seeds,
                                                lower=self.lower,
                                                upper=self.upper,
                                                replaceValue=1)
                # shift values so that they're all positive and apply mask
```

```
                maskedA2 = sitk_applyMask(self.img - arr.min(), maskA)
                # now shift values back, this results in all masked pixels to be assigned the minimum
value
                maskedA = maskedA2 + arr.min()
                # use all pixels above minimum value for centroid:
                centroidsA[index] = self.xSpace*sitk_centroid(maskedA,
                                                              ref=self.ref,
                                                              threshold=arr.min()+1)
                diceA[index] = self.getDice(centroidsA[index], maskA)
                # all irregular Slices result in DC of -1:
                diceA[index][np.where(self.niceSlice==False)] = -1
                # for the final DC score it will look only at the niceSlices:
                centroidScoreA[index] = np.average(diceA[index, self.niceSlice==True])
#                 centroidScoreA[index] = np.average(diceA[index,diceA[index]>-1])
#                 centroidScoreA[index] = np.average(diceA[index])


                print("\nB @ ~{:.4f}%".format((guess[index]+right[index])/2*100))
                thresholdsB[index] = self.getThresholds(pixelNumber=self.xSize*self.ySize*(guess[index]
+right[index])/2)
                maskB = sitk.ConnectedThreshold(image1=self.img,
                                                seedList=self.seeds,
                                                lower=self.lower,
                                                upper=self.upper,
                                                replaceValue=1)
                maskedB2 = sitk_applyMask(self.img - arr.min(), maskB)
                maskedB = maskedB2 + arr.min()
                centroidsB[index] = self.xSpace*sitk_centroid(maskedB,
                                                              ref=self.ref,
                                                              threshold=arr.min()+1)
                diceB[index] = self.getDice(centroidsB[index], maskB)
                # all irregular Slices result in DC of -1:
                diceB[index][np.where(self.niceSlice==False)] = -1
                # for the final DC score it will look only at the niceSlices:
                centroidScoreB[index] = np.average(diceB[index, self.niceSlice==True])
#                 centroidScoreB[index] = np.average(diceB[index,diceB[index]>-1])
#                 centroidScoreB[index] = np.average(diceB[index])


                if centroidScoreA[index] < centroidScoreB[index] and index < iterations-1:
                    left[index+1] = guess[index]
                    right[index+1] = right[index]
                    guess[index+1] = (left[index+1] + right[index+1]) / 2
                    direction[index] = 1
                elif centroidScoreA[index] > centroidScoreB[index] and index < iterations-1:
                    right[index+1] = guess[index]
                    left[index+1] = left[index]
                    guess[index+1] = (left[index+1] + right[index+1]) / 2
                    direction[index] = -1
                elif centroidScoreA[index] == centroidScoreB[index] and index < iterations-1:
                    right[index+1] = (guess[index] + right[index]) / 2
                    left[index+1] = (guess[index] + left[index]) / 2
                    guess[index+1] = guess[index]
                else:
                    break

                print("---------------------------")
                print("next guess (#{}): ~{:.4f}% \n\n\n\n".format(index+1, guess[index+1]*100))


            if centroidScoreA.max() > centroidScoreB.max():
                self.centroid = centroidsA[centroidScoreA.argmax()]
                self.lower, self.upper = thresholdsA[centroidScoreA.argmax()]
                self.dice = diceA[centroidScoreA.argmax()]
                self.diceAverage = centroidScoreA.max()
                print("\nmax dice-coefficient obtained during iteration #{}: ~{:.4f}".format
(centroidScoreA.argmax(), centroidScoreA.max()))
            elif (centroidScoreA.max() <= centroidScoreB.max() and centroidScoreB.max() != 0):
                self.centroid = centroidsB[centroidScoreB.argmax()]
                self.lower, self.upper = thresholdsB[centroidScoreB.argmax()]
                self.dice = diceB[centroidScoreB.argmax()]
                self.diceAverage = centroidScoreB.max()
```

```
                print("\nmax dice-coefficient obtained during iteration #{}: ~{:.4f}".format
(centroidScoreB.argmax(), centroidScoreB.max()))
            else:
                return None

            print("\n\n-o-o-o-o-- Summary: --o-o-o-o-\n")
            for index in range(np.size(guess)):
                print("\n  Iteration #{}: range({},{})".format(index, left[index]*100, right
[index]*100))
                if centroidScoreA[index] > centroidScoreB[index]:
                    print("A @ {}%, Score: {}  <---".format((guess[index]+left[index])/2*100,
centroidScoreA[index]))
                    print("B @ {}%, Score: {}".format((guess[index]+right[index])/2*100, centroidScoreB
[index]))
                if centroidScoreA[index] < centroidScoreB[index]:
                    print("A @ {}%, Score: {}".format((guess[index]+left[index])/2*100, centroidScoreA
[index]))
                    print("B @ {}%, Score: {}  <---".format((guess[index]+right[index])/2*100,
centroidScoreB[index]))
                if centroidScoreA[index] == centroidScoreB[index]:
                    print("A @ {}% same as for B @ {}%, Score: = {}".format((guess[index]+left
[index])/2*100, (guess[index]+right[index])/2*100, centroidScoreA[index]))


            if plot == True:
                fig = plt.figure()
                for index in range(iterations):
                    if guess[index] > 0 and centroidScoreA[index] > 0:
                        plt.plot((guess[index]+left[index])/2*100, centroidScoreA[index], 'bo')
                    if guess[index] > 0 and centroidScoreB[index] > 0:
                        plt.plot((guess[index]+right[index])/2*100, centroidScoreB[index],
'go')
                plt.show()
                if save != False:
                    fig.savefig(str(save) + ".png")


        if percentLimit != "auto" and percentLimit is not False:
            self.centroid = self.xSpace * sitk_centroid(self.img, ref=self.ref,
                                                        percentLimit=percentLimit)

        if (threshold == 'auto' or threshold == 'default') and percentLimit is False:
            self.getThresholds(pixelNumber=pixelNumber, scale=scale)
            self.centroid = self.xSpace * sitk_centroid(self.img, ref=self.ref,
                                                        threshold=self.lower)

        if (threshold != "auto" and threshold != 'default') and threshold is not False and
percentLimit is False:
            self.centroid = self.xSpace * sitk_centroid(self.img, ref=self.ref,
                                                        threshold=threshold)

        for index in range(self.zSize):
            if not self.niceSlice[index]:
                self.centroid[index] = -1, -1
            if self.centroid[index,0] < 0 or self.centroid[index,1] < 0 :
                self.centroid[index] = -1, -1
        print("\n\n")
        return self.centroid


    def showCentroid(self, img=None, com2=0, title=None, pixel=False,
                     interpolation='nearest', ref=None, save=False):
        '''
        shows slice with centroid coordinates

        Parameters
        ----------
        img: SimpleITK.img, optional
            slice of this volume will be shown
            default: self.img
        com2: numpy.ndarray
            supposed to be of same length as img
```

```
            will also be shown in plot alongside self.centroid
            helps creating nice plot for comparing COM-shift
        pixel: bool, optional
            if True, changes axis from mm to pixels
        interpolation: "string", optional, default: 'nearest'
            using build-in interpolation of matplotlib.pyplot.imshow
            Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic',
            'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
            'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc',
            'lanczos'
        ref: int, optional
            slice to be plotted instead of self.ref (default: 0)
        save: string, optional
            save plot as save + ".png"
        '''

        if self.centroid is False:
            print("Volume has no centroid yet. use Volume.getCentroid() first!")
            return None

        if title is None:
            title = self.title
        if ref is None:
            ref = self.ref
        if img is None:
            img = self.img

        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)
            sitk_centroid_show(img=img, com=self.centroid, com2=com2,
                               extent=extent, save=save, title=title,
                               interpolation=interpolation, ref=ref)
        else:
            sitk_centroid_show(img=img, com=self.centroid/self.xSpace,
                               com2=com2/self.xSpace, save=save, title=title,
                               interpolation=interpolation, ref=ref)

    def getMask(self, lower=False, upper=False):

        if lower is False and self.lower is not False:
            lower = self.lower
        if upper is False and self.upper is not False:
            upper = self.upper

        if lower is False:
            print("Lower threshold missing!")
            return None
        if upper is False:
            print("Upper threshold missing!")
            return None

        self.mask = sitk_getMask(self.img, self.seeds, upper, lower)
        return self.mask

    def applyMask(self, mask=0, replaceArray=False, scale=1000):
        if mask == 0:
            if self.mask:
                mask = self.mask
            else:
                print("Volume has no mask yet. use Volume.getMask() first!")
                return None

        self.masked = sitk_applyMask(self.img, mask, replaceArray=replaceArray,
                                     scale=scale)

        return self.masked

    def showMask(self, interpolation=None, ref=None, save=False, pixel=False):
        if self.mask is False:
            print("Volume has no mask yet. use Volume.getMask() first!")
            return None
```

```
        if ref is None:
            ref = self.ref

        if interpolation is None:
            interpolation = 'nearest'

        title = self.title + ", mask"

        extent = None
        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)

        sitk_show(img=self.mask, ref=ref, title=title, extent=extent,
                      interpolation=interpolation, save=save)

    def showMasked(self, interpolation=None, ref=None, save=False, pixel=False):
        if self.masked is False:
            print("Volume has not been masked yet. use Volume.applyMask() first!")
            return None
        if ref is None:
            ref = self.ref

        if interpolation is None:
            interpolation = 'nearest'

        title = self.title + ", masked"

        extent = None
        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)

        sitk_show(img=self.masked, ref=ref, title=title, extent=extent,
                  interpolation=interpolation, save=save)

    def getDice(self, centroid=None, mask=None, iterations=15,
                CT_guess=(3.5,5.5), MR_guess=(1.5,4.5),
                show=False, showAll=False, plot=False, save=False, pixel=False):
        '''
        Calculates dice coefficient ('DC') and average DC of the volume
        if iterations > 0: varies radius and finds DC with best average DC
        else: if self.raduis == 0: use method to get raduis for DC calculation
        average DC is mean value of all slices, except those with DC of -1

        slice DC is set to -1 if centroid lies outside image or reference
        circle exceeds image

        Parameters
        ----------
        centroid: numpy.ndarray, optional
            centroid to place circles in instead of self.centroid
        mask: SimpleITK image, optional
            binary image to calculate DC of instead of self.mask
        iterations: int, optional
        show: int, optional
            shows circle used to compare mask to in slice nr. "show"
        showAll: bool, optional
            shows all circles tried during iteration
        plot, save: bool, optional
            plot and save iteration

        Returns
        -------
        self.dice: numpy.ndarray
        '''
        if centroid is None:
            centroid = self.centroid
        # to get from mm to pixel coordinates:
        com = centroid / self.xSpace
        if mask is None:
```

```
            if self.mask is False:
                self.getMask()
            mask = self.mask

        extent = None
        if pixel is False:
            extent = (-self.xSpace/2, self.xSize*self.xSpace - self.xSpace/2, self.ySize*self.ySpace -
self.ySpace/2, -self.ySpace/2)

        if self.radius != 0:
            print("{}_x{}.radius is {} and will therefore be used to calculate DC.".format
(self.method, self.resample, self.radius))
            self.dice = sitk_dice_circle(img=mask, centroid=com, extent=extent,
                                         radius=self.radius/self.xSpace, show=show)

#        print("\n{}_x{}:".format(self.method, self.resample))

        if self.radius == 0 and iterations == 0:
            if self.method == "CT":
                self.dice = sitk_dice_circle(img=mask, centroid=com, extent=extent,
                                             radius=4/self.xSpace, show=show)
            if self.method == "MR":
                self.dice = sitk_dice_circle(img=mask, centroid=com, extent=extent,
                                             radius=2/self.xSpace, show=show)
            if self.method != "CT" and self.method != "MR":
                print("Unknown method!")
                return None

        if self.radius == 0 and iterations > 0:
            low, up = 0, 0
            if self.method == "CT":
                low, up = CT_guess
                radii = np.linspace(low, up, num=iterations)/self.xSpace
            if self.method == "MR":
                low, up = MR_guess
                radii = np.linspace(low, up, num=iterations)/self.xSpace
            if self.method != "CT" and self.method != "MR":
                # radii = np.linspace(1.5, 4.5, num = 11)
                print("Unknown method!")
                return None

            DCs = np.zeros(len(radii))
            for index, r in enumerate(radii, start=0):
                dice = sitk_dice_circle(img=mask, centroid=com, radius=r,
                                        show=showAll, extent=extent)
#                DCs[index] = np.average(dice)
#                DCs[index] = np.average(dice[dice>-1])
                DCs[index] = np.average(dice[self.niceSlice==True])


            if plot == True:
#                fig = plt.figure()
#                plt.ylim(ymin=0.6, ymax=1)
#                plt.xlim(xmin=(low-.1), xmax=(up+.1))
                plt.plot(radii*self.xSpace, DCs, '+-')
                if save is not False:
                    fig.savefig(str(save) + ".png")

            self.dice = sitk_dice_circle(img=mask, centroid=com, show=show,
                                         extent=extent, radius=radii[DCs.argmax()])
            self.bestRadius = radii[DCs.argmax()]*self.xSpace
            print("max dice-coefficient obtained for {} when compared to circle with radius =
{}".format(self.method, self.bestRadius))

#        self.diceAverage = np.average(self.dice[self.dice>-1])
        self.diceAverage = np.average(self.dice)
        print("dice-coefficient average for the whole volume is: {:.4f}".format(self.diceAverage))
        return self.dice



def sitk_read(directory, denoise=False):
```

```
    '''
    returns DICOM files as "SimpleITK.Image" data type (3D)
    if denoise is True: uses SimpleITK to denoise data
    '''
    reader = sitk.ImageSeriesReader()
    filenames = reader.GetGDCMSeriesFileNames(directory)
    reader.SetFileNames(filenames)
    if denoise:
        print("\n...denoising...")
        imgOriginal = reader.Execute()
        return sitk.CurvatureFlow(image1=imgOriginal,
                                  timeStep=0.125,
                                  numberOfIterations=5)
    else:
        return reader.Execute()


def sitk_write(image, output_dir='', filename='3DImage.mha'):
    '''
    saves image as .mha file
    '''
    output_file_name_3D = os.path.join(output_dir, filename)
    sitk.WriteImage(image, output_file_name_3D)


def sitk_show(img, ref=0, extent=None, title=None, interpolation='nearest', save=False):
    """
    shows plot of img at z=ref
    """
    arr = sitk.GetArrayFromImage(img)
    fig = plt.figure()
    plt.set_cmap("gray")
    if title:
        plt.title(title)

    plt.imshow(arr[ref], extent=extent, interpolation=interpolation)
    plt.show()
    if save != False:
        fig.savefig(str(save) + ".png")

def sitk_centroid(img, ref=False, percentLimit=False, threshold=False):
    '''
    returns array with y&x coordinate of centroid for every slice of img
    centroid[slice, y&x-coordinate]
    if no pixel has value > threshold:
        centroid x&y-coordinate of that slice = -1,-1
    '''
    if (threshold is False and percentLimit is False) or (threshold is True and percentLimit is True):
        print("Please set either percentLimit or threshold!")
        return None

    arr = sitk.GetArrayFromImage(img)
    z, y, x = np.shape(arr)
    # create array with centroid coordinates of rod in each slice
    com = np.zeros((z, 2))

    if ref is False:
        ref = int(z/2)

    if threshold is False:
#        hist, bins = np.histogram(arr[ref, :, :].ravel(), density=True, bins=100)
     #  alternatively, increase number of bins for images with many pixels
        hist, bins = np.histogram(arr[ref, :, :].ravel(), density=True, bins=int(y*x))
        threshold = bins[np.concatenate((np.array([0]), np.cumsum(hist))) *
                         (bins[1] - bins[0]) > percentLimit][0]

    for index in range(z):
        if arr[index].max() > threshold:
            # structuring_element=[[1,1,1],[1,1,1],[1,1,1]]
            segmentation, segments = ndimage.label(arr[index] > threshold)
            # print("segments: {}".format(segments))
            # add ', structuring_element' to label() for recognising
```

```
            # diagonal pixels as part of object
            com[index, ::-1] = ndimage.center_of_mass(arr[index, :, :]-threshold,
                                                      segmentation)
            # add ', range(1,segments)' to center_of_mass for list of centroids
            # in each slice (multiple rods!)
        else:
            com[index] = (-1,-1)

    return com

def sitk_centroid_show(img, com, com2=0, extent=None, title=None,
                       save=False, interpolation='nearest', ref=0):

        arr = sitk.GetArrayFromImage(img)
        fig = plt.figure()
        plt.set_cmap("gray")
        if title:
            plt.title(title + ", centroid")
        x = y = 0
        plt.imshow(arr[ref], extent=extent, interpolation=interpolation)
        if type(com2) == np.ndarray:
            x = [com[ref,0],com2[ref,0]]
            y = [com[ref,1],com2[ref,1]]
        else:
            x, y = com[ref]
        plt.scatter(x, y, c=['b','r'])
        plt.show()
        if save != False:
            fig.savefig(str(save) + ".png")

def sitk_coordShift(first, second):
    '''
    returns array with difference of y&x coordinates for every
    centroid[slice, y&x-coordinate]
    '''
    if (np.shape(first) == np.shape(second) and
            np.shape((np.shape(first))) == (2,)):
        z, xy = np.shape(first)
        diff = np.zeros((z, 2))
        for slice in range(z):
            if first[slice,0]==-1 or first[slice,1]==-1 or second[slice,0]==-1 or second[slice,0]==-1:
                diff[slice, 0] = diff[slice, 1] = -1
            else:
                diff[slice, 0] = first[slice, 0] - second[slice, 0]
                diff[slice, 1] = first[slice, 1] - second[slice, 1]
        return diff
    else:
        print("Wrong shape! sitk_coordShift returned 'False'")
        return False


def sitk_coordDist(shift):
    '''
    calculates norm for each entry of array
    returns array with list of calculated values
    '''
    if np.shape(shift)[1] != 2:
        print("shift has wrong shape!")
        return False

    dist = np.zeros((len(shift), 1))
    for slice in range(len(shift)):
        if shift[slice,0] == -1 or shift[slice,1] == -1:
            dist[slice,:] =-1
        else:
            dist[slice, :] = np.linalg.norm(shift[slice, :])
    return dist


def sitk_getMask(img, seedList, upper, lower):
    '''
    creates new SimpleITK.img using a SimpleITK segmentation function
```

```
    which is made up by all pixels with values between upper and lower and
    connected to a seed from seedList.
    Returns binary image (SimpleITK.img)
    '''

    if seedList is False:
        print("no seeds given!")
        return None

    return sitk.ConnectedThreshold(image1=img, seedList=seedList,
                                   lower=lower, upper=upper,
                                   replaceValue=1)


def sitk_applyMask(img, mask, replaceArray=False, scale=1000, errorValue=-1):
    '''
    masks img (SimpleITK.Image) using mask (SimpleITK.Image)
    if a replaceArray is given, the values*scale (default scale=1000) of the
    array will be used as pixel intensity for an entire slice each
    '''
    if img.GetSize() != mask.GetSize():
        print(mask.GetSize())
        print(img.GetSize())

        print("mask and image are not the same size!")
        return False

    arr = sitk.GetArrayFromImage(img)
    maskA = sitk.GetArrayFromImage(mask)
    xSize, ySize, zSize = img.GetSize()

    imgMaskedA = (arr - arr.min() + 1)*maskA

    if np.shape(replaceArray) == (img.GetDepth(), 1) or np.shape(replaceArray) == (img.GetDepth(),):
        for slice in range(zSize):
            imgMaskedA[slice][imgMaskedA[slice] != 0] = replaceArray[slice]*scale
            imgMaskedA[slice][imgMaskedA[slice] < 0] = errorValue


    return sitk.GetImageFromArray(imgMaskedA)


def sitk_dice_circle(img, centroid, radius=2.1, show=False, extent=None,
                     interpolation='nearest', save=False):
    """
    Dice coefficient, inspired by
     Medpy (http://pythonhosted.org/MedPy/_modules/medpy/metric/binary.html)

    Computes the Dice coefficient (akas Sorensen index) between a binary
    object in an image and a circle.

    The metric is defined as:

        DC=\frac{2|A\cap B|}{|A|+|B|}

    where A is the first and B the second set of samples (here: binary objects)

    Parameters
    ----------
    input_umg : SimpleITK.Image
        Input data containing objects. Can be any type but will be converted
        into binary: background where 0, object everywhere else.
    centroid : array_like
        array with coordinates for circle centre
    radius : float
        radius for creating reference circles

    Returns
    -------
    DC : array_like
        The Dice coefficient between the object(s) in ```input``` and the
        created circles. It ranges from 0 (no overlap) to 1 (perfect overlap).
```

```
        if centroid coordinates + radius would create circle exceeding image
        size: DC of this slice = -1
        Other errors occuring during the calculation should also result in -1
    """

    xSize, ySize, zSize = img.GetSize()
    xSpace, ySpace, zSpace = img.GetSpacing()
    profile = np.zeros((zSize, ySize, xSize), dtype=np.uint8)
    DC = np.zeros((zSize, 1))
    for slice in range(zSize):
        if centroid[slice,0]+radius < xSize and centroid[slice, 1]+radius < ySize and centroid
[slice,0]-radius > 0 and centroid[slice, 1]-radius > 0:

            rr, cc = circle(centroid[slice, 0], centroid[slice, 1], radius, (xSize,ySize))
            profile[slice, cc, rr] = 1
        else:
            # print("something's fishy!")
            DC[slice]= -1

    input = sitk.GetArrayFromImage(img)

    input = np.atleast_1d(input.astype(np.bool))
    reference = np.atleast_1d(profile.astype(np.bool))

    intersection = np.zeros((zSize, 1))
    size_input = np.zeros((zSize, 1))
    size_reference = np.zeros((zSize, 1))
    for slice in range(zSize):
        intersection[slice] = np.count_nonzero(input[slice, :, :] & reference[slice, :, :])
        size_input[slice] = np.count_nonzero(input[slice, :, :])
        size_reference[slice] = np.count_nonzero(reference[slice, :, :])

        try:
            if (DC[slice] == 0) and (float(size_input[slice] + size_reference[slice]) != 0):
                DC[slice] = 2. * intersection[slice] / float(size_input[slice] + size_reference[slice])

        except ZeroDivisionError:
            DC[slice] = -1

    if show != False:
        profile_img = sitk.GetImageFromArray(profile)
        sitk_centroid_show(profile_img, centroid*xSpace, extent=extent,
                           title="profile, radius: {:03.2f}".format(radius*xSpace),
                           ref=show, save=save)

    return DC


# to view in 3D Slicer, type this in IPython console or in jupyter notebook:
# %env SITK_SHOW_COMMAND /home/david/Downloads/Slicer-4.5.0-1-linux-amd64/Slicer
# sitk.Show(imgFillingCT)
```