1. Registration

Rigid registration function from SimpleITK

```
# to do: define helper script to wrap the main preprocessing steps for demons registration of two volumetric images.
def SITK_multiscale_affine_registration(vol1, vol2,
                                        imtype=16,
                                        p12=None,
                                        rescale_intensity=True,
                                        centre_tfm_model='geometry',
                                        tfm_type = 'rigid',
                                        metric='Matte',
                                        metric_numberOfHistogramBins=16,
                                        sampling_intensity_method='random',
                                        smooth_displacement_field = True,
                                        MetricSamplingPercentage=0.1,
                                        shrink_factors = [1],
                                        smoothing_sigmas = [0],
                                        optimizer='gradient',
                                        optimizer_params=None,
                                        eps=1e-12):

    import SimpleITK as sitk

    def imadjust(vol, p1, p2):
        import numpy as np
        from skimage.exposure import rescale_intensity
        # this is based on contrast stretching and is used by many of the biological image processing algorithms.
        p1_, p2_ = np.percentile(vol, (p1,p2))
        vol_rescale = rescale_intensity(vol, in_range=(p1_,p2_))
        return vol_rescale

    import numpy as np

    if p12 is None:
        im1_ = vol1.copy()
        im2_ = vol2.copy()
        # no contrast stretching
        if rescale_intensity == False:
            im1_ = im1_ / (2**imtype - 1)
            im2_ = im2_ / (2**imtype - 1)
        else:
            im1_ = (im1_ - im1_.min()) / (im1_.max() - im1_.min() + eps)
            im2_ = (im2_ - im2_.min()) / (im2_.max() - im2_.min() + eps)
    else:
        # contrast stretching
        im1_ = imadjust(vol1, p12[0], p12[1])
        im2_ = imadjust(vol2, p12[0], p12[1])

        if rescale_intensity == False:
            im1_ = im1_ / (2**imtype - 1)
            im2_ = im2_ / (2**imtype - 1)
        else:
            im1_ = (im1_ - im1_.min()) / (im1_.max() - im1_.min() + eps)
            im2_ = (im2_ - im2_.min()) / (im2_.max() - im2_.min() + eps)

    ### main analysis scripts.
    v1 = sitk.GetImageFromArray(im1_, isVector=False)
    v2 = sitk.GetImageFromArray(im2_, isVector=False)

    # a) initial transform
    # translation.
    if centre_tfm_model=='geometry':
        translation_mode = sitk.CenteredTransformInitializerFilter.GEOMETRY
    if centre_tfm_model=='moments':
        translation_mode = sitk.CenteredTransformInitializerFilter.MOMENTS
```

```python
# if tfm_type == 'translation':
#     transform_mode = sitk.TranslationTransform(3)
# these are the built in transforms without additional modification.
if tfm_type == 'rigid':
    transform_mode = sitk.Euler3DTransform()
if tfm_type == 'iso_similarity':
    transform_mode = sitk.Similarity3DTransform()
if tfm_type == 'aniso_similarity':
    transform_mode = sitk.ScaleVersor3DTransform() # this version allows anisotropic scaling
if tfm_type == 'affine':
    # transform_mode = sitk.AffineTransform(3) # this is a generic that requires setting.
    transform_mode = sitk.ScaleSkewVersor3DTransform() # this is anisotropic + skew

initial_transform = sitk.CenteredTransformInitializer(v1,
                                                      v2,
                                                      transform_mode, # this is where we change the transform
                                                      translation_mode)

# a) Affine registration transform
# Select a different type of affine registration
# multiscale rigid.
registration_method = sitk.ImageRegistrationMethod()

# Similarity metric settings.
registration_method.SetMetricAsMattesMutualInformation(numberOfHistogramBins=metric_numberOfHistogramBins)
# key for making it fast. (subsampling, don't use all the pixels)
if sampling_intensity_method == 'random':
    registration_method.SetMetricSamplingStrategy(registration_method.RANDOM)
registration_method.SetMetricSamplingPercentage(MetricSamplingPercentage)
registration_method.SetInterpolator(sitk.sitkLinear) # use this to help resampling the intensity at each iteration.

# Optimizer settings. # put these settings in a dedicated params file?
if optimizer == 'gradient':
    registration_method.SetOptimizerAsGradientDescent(learningRate=optimizer_params['learningRate'],
                                                      numberOfIterations=optimizer_params['numberOfIterations'], # increase this.
                                                      convergenceMinimumValue=optimizer_params['convergenceMinimumValue'],
                                                      convergenceWindowSize=optimizer_params['convergenceWindowSize'],
                                                      estimateLearningRate=registration_method.EachIteration)
if optimizer == '1+1_evolutionary':
    registration_method.SetOptimizerAsOnePlusOneEvolutionary(numberOfIterations=optimizer_params['numberOfIterations'],
                                                             epsilon=optimizer_params['epsilon'],
                                                             initialRadius= optimizer_params['initialRadius'],
                                                             growthFactor= optimizer_params['growthFactor'],
                                                             shrinkFactor= optimizer_params['shrinkFactor']
                                                             )
registration_method.SetOptimizerScalesFromIndexShift()

# set the multiscale registration parameters here.
registration_method.SetShrinkFactorsPerLevel(shrinkFactors = shrink_factors) # use just the one scale.
registration_method.SetSmoothingSigmasPerLevel(smoothingSigmas=smoothing_sigmas) # don't filter.
registration_method.SetInitialTransform(initial_transform, inPlace=False)

tfm = registration_method.Execute(sitk.Cast(v1, sitk.sitkFloat32),
                                  sitk.Cast(v2, sitk.sitkFloat32))
```

```python
return tfm
```

```python
for ii in tqdm(np.arange(len(imfiles)-1)[:]):

    if ii == 0:
        imfile1 = imfiles[ii]
        # im1 = skio.imread(imfile1, multifile=False) # multifile is required to prevent loading of all associated files.
        im1 = read_tiff(imfile1)
        im1 = im1[x1:x2,y1:y2,z1:z2].copy()
        im1 = np.uint16(ndimage.zoom( im1, [z_scale, 1, 1], order=1, mode='reflect')) # pad the image symmetrically.

        #save this out in the output folder.
        savefile = os.path.join(saveoutfolder, 'rigid_'+os.path.split(imfiles[ii])[-1])
        skio.imsave(savefile, np.uint16(im1))
        # no need to save the transform here.
    else:
        # if np.mod(ii, n_ref) == 0:
        print(ii, ii//n_ref*n_ref)
        # read the registered.
        file_ind_ii = ii//n_ref*n_ref
        im1 = read_tiff(os.path.join(saveoutfolder, 'rigid_'+os.path.split(imfiles[file_ind_ii])[-1])) # reinitialise from the already registered.

    im1_raw = im1.copy() # visualisation only.

    imfile2 = imfiles[ii+1]
    # im2 = skio.imread(imfile2, multifile=False) # multifile is required to prevent loading of all associated files.
    im2 = read_tiff(imfile2)
    im2 = im2[x1:x2,y1:y2,z1:z2].copy()
    im2 = np.uint16(ndimage.zoom( im2, [z_scale, 1, 1], order=1, mode='reflect'))
    im2_raw = im2.copy() # make a copy for transformation later.
    # =============================================================================
    #   Rigid register routine implemented from library.
    # =============================================================================

    # replace all 0 intensity pixels by the average of the volume.
    im1[im1 == 0] = np.nanmean(im1)
    im2[im2 == 0] = np.nanmean(im2)

    rigid_optimizer_params = params.gradient_descent_affine_reg()
    rigid_tfm = registration.SITK_multiscale_affine_registration(im1,
                                                                  im2,
                                                                 imtype=16,
                                                                 p12=(2,99.8), # percentage scaling to enhance contrast.
                                                                 rescale_intensity=True,
                                                                 centre_tfm_model='geometry',
                                                                 tfm_type = 'rigid', # how to do affine?
                                                                 metric='Matte',
                                                                 metric_numberOfHistogramBins=16,
                                                                 sampling_intensity_method='random',
                                                                 smooth_displacement_field = True, # remove this.
                                                                 MetricSamplingPercentage=0.1,
                                                                 shrink_factors = [1],
                                                                 smoothing_sigmas = [0],
                                                                 optimizer='gradient',
                                                                 optimizer_params=rigid_optimizer_params,
                                                                 eps=1e-12)

    # save this transform
    savetformfile = os.path.join(saveoutfolder, 'Euler3D_'+os.path.split(imfiles[ii+1])[-1].replace('.tif', '.tfm'))
    sitk.WriteTransform(rigid_tfm, savetformfile)


    # apply the transform to im2
    im2_tfm = registration.transform_img_sitk(im2_raw, rigid_tfm) # apply to image 2.
    savefile = os.path.join(saveoutfolder, 'rigid_'+os.path.split(imfiles[ii+1])[-1])
    skio.imsave(savefile, np.uint16(im2_tfm)) # save the transformed.
```

Non-rigid demons registration function modified from SimpleITK

```python
# to do: define helper script to wrap the main preprocessing steps for demons registration of two volumetric images.
def SITK_multiscale_demons_registration(vol1, vol2,
                                        imtype=16,
                                        p12=None,
                                        rescale_intensity=True,
                                        centre_tfm_model='geometry',
                                        demons_type='diffeomorphic',
                                        n_iters = 25,
                                        smooth_displacement_field = True,
                                        smooth_alpha=.8,
                                        shrink_factors = [2.,1.],
                                        smoothing_sigmas = [1.,1.],
                                        eps=1e-12,
                                        multiscale_iters=None,
                                        intensity_diff_thresh=None):

    import SimpleITK as sitk

    def imadjust(vol, p1, p2):
        import numpy as np
        from skimage.exposure import rescale_intensity
        # this is based on contrast stretching and is used by many of the biological image processing algorithms.
        p1_, p2_ = np.percentile(vol, (p1,p2))
        vol_rescale = rescale_intensity(vol, in_range=(p1_,p2_))
        return vol_rescale

    import numpy as np

    if p12 is None:
        im1_ = vol1.copy()
        im2_ = vol2.copy()
        # no contrast stretching
        if rescale_intensity == False:
            im1_ = im1_ / (2**imtype - 1)
            im2_ = im2_ / (2**imtype - 1)
        else:
            im1_ = (im1_ - im1_.min()) / (im1_.max() - im1_.min() + eps)
            im2_ = (im2_ - im2_.min()) / (im2_.max() - im2_.min() + eps)
    else:
        # contrast stretching
        im1_ = imadjust(vol1, p12[0], p12[1])
        im2_ = imadjust(vol2, p12[0], p12[1])

        if rescale_intensity == False:
            im1_ = im1_ / (2**imtype - 1)
            im2_ = im2_ / (2**imtype - 1)
        else:
            im1_ = (im1_ - im1_.min()) / (im1_.max() - im1_.min() + eps)
            im2_ = (im2_ - im2_.min()) / (im2_.max() - im2_.min() + eps)

    ### main analysis scripts.
    v1 = sitk.GetImageFromArray(im1_, isVector=False)
    v2 = sitk.GetImageFromArray(im2_, isVector=False)

    # a) initial transform
    # translation.
    if centre_tfm_model=='geometry':
        translation_mode = sitk.CenteredTransformInitializerFilter.GEOMETRY
    if centre_tfm_model=='moments':
        translation_mode = sitk.CenteredTransformInitializerFilter.MOMENTS
    initial_transform = sitk.CenteredTransformInitializer(v1,
                                                          v2,
                                                          sitk.Euler3DTransform(),
                                                          translation_mode)
```

```python
# a) demons transform (best to have corrected out any rigid transforms a priori)
 # Select a Demons filter and configure it.
if demons_type == 'diffeomorphic':
    demons_filter = sitk.DiffeomorphicDemonsRegistrationFilter() # we should use this version.
if demons_type == 'symmetric':
    demons_filter = sitk.FastSymmetricForcesDemonsRegistrationFilter()
if demons_type == 'demons':
    demons_filter = sitk.DemonsRegistrationFilter()
# set the number of iterations
demons_filter.SetNumberOfIterations(n_iters) # 5 for less. # long time for 20?
# Regularization (update field - viscous, total field - elastic).
demons_filter.SetSmoothDisplacementField(smooth_displacement_field)
demons_filter.SetStandardDeviations(smooth_alpha)

# run the registration and return the final transform parameters
final_tfm = multiscale_demons(registration_algorithm=demons_filter,
                              fixed_image = v1,
                              moving_image = v2,
                              initial_transform = initial_transform,
                              shrink_factors = shrink_factors, # did have 2 here. -> test, can we separate the  # do at the same scale.
                              smoothing_sigmas = smoothing_sigmas,
                              multiscale_iters = multiscale_iters,
                              intensity_diff_threshold=intensity_diff_thresh) # set smoothing very low, since we want it to zone in on interesting features.
# check again how this is parsed .
return final_tfm
```

```python
"""
Using simpleitk for rigid and non-rigid registration (Python/Anaconda route without external languages that is faster than dipy)
"""

def smooth_and_resample(image, shrink_factors, smoothing_sigmas):
    """
    Args:
        image: The image we want to resample.
        shrink_factor(s): Number(s) greater than one, such that the new image's size is original_size/shrink_factor.
        smoothing_sigma(s): Sigma(s) for Gaussian smoothing, this is in physical units, not pixels.
    Return:
        Image which is a result of smoothing the input and then resampling it using the given sigma(s) and shrink factor(s).
    """
    import SimpleITK as sitk
    import numpy as np

    if np.isscalar(shrink_factors):
        shrink_factors = [shrink_factors]*image.GetDimension()
    if np.isscalar(smoothing_sigmas):
        smoothing_sigmas = [smoothing_sigmas]*image.GetDimension()

    smoothed_image = sitk.SmoothingRecursiveGaussian(image, smoothing_sigmas)

    original_spacing = image.GetSpacing()
    original_size = image.GetSize()
    new_size = [int(sz/float(sf) + 0.5) for sf,sz in zip(shrink_factors,original_size)]
    new_spacing = [((original_sz-1)*original_spc)/(new_sz-1)
                   for original_sz, original_spc, new_sz in zip(original_size, original_spacing, new_size)]
    return sitk.Resample(smoothed_image, new_size, sitk.Transform(),
                         sitk.sitkLinear, image.GetOrigin(),
                         new_spacing, image.GetDirection(), 0.0,
                         image.GetPixelID())
```

```python
def multiscale_demons(registration_algorithm,
                      fixed_image, moving_image, initial_transform = None,
                      shrink_factors=None, smoothing_sigmas=None,
                      multiscale_iters=None,
                      intensity_diff_threshold=None):
    """
    Run the given registration algorithm in a multiscale fashion. The original scale should not be given as input as the
    original images are implicitly incorporated as the base of the pyramid.
    Args:
        registration_algorithm: Any registration algorithm that has an Execute(fixed_image, moving_image, displacement_field_image)
                                method.
        fixed_image: Resulting transformation maps points from this image's spatial domain to the moving image spatial domain.
        moving_image: Resulting transformation maps points from the fixed_image's spatial domain to this image's spatial domain.
        initial_transform: Any SimpleITK transform, used to initialize the displacement field.
        shrink_factors (list of lists or scalars): Shrink factors relative to the original image's size. When the list entry,
                                                   shrink_factors[i], is a scalar the same factor is applied to all axes.
                                                   When the list entry is a list, shrink_factors[i][j] is applied to axis j.
                                                   This allows us to specify different shrink factors per axis. This is useful
                                                   in the context of microscopy images where it is not uncommon to have
                                                   unbalanced sampling such as a 512x512x8 image. In this case we would only want to
                                                   sample in the x,y axes and leave the z axis as is: [[[8,8,1],[4,4,1],[2,2,1]].
        smoothing_sigmas (list of lists or scalars): Amount of smoothing which is done prior to resmapling the image using the given shrink factor. These
                          are in physical (image spacing) units.
    Returns:
        SimpleITK.DisplacementFieldTransform
    """
    import SimpleITK as sitk
    import numpy as np

    if intensity_diff_threshold is not None:
        registration_algorithm.SetIntensityDifferenceThreshold(intensity_diff_threshold)

    # Create image pyramid in a memory efficient manner using a generator function.
    # The whole pyramid never exists in memory, each level is created when iterating over
    # the generator.
    def image_pair_generator(fixed_image, moving_image, shrink_factors, smoothing_sigmas):
        end_level = 0
        start_level = 0
        if shrink_factors is not None:
            end_level = len(shrink_factors)
        for level in range(start_level, end_level):
            f_image = smooth_and_resample(fixed_image, shrink_factors[level], smoothing_sigmas[level])
            m_image = smooth_and_resample(moving_image, shrink_factors[level], smoothing_sigmas[level])
            yield(f_image, m_image)
        yield(fixed_image, moving_image)

    # Create initial displacement field at lowest resolution.
    # Currently, the pixel type is required to be sitkVectorFloat64 because
    # of a constraint imposed by the Demons filters.
    if shrink_factors is not None:
        original_size = fixed_image.GetSize()
        original_spacing = fixed_image.GetSpacing()
        s_factors =  [shrink_factors[0]]*len(original_size) if np.isscalar(shrink_factors[0]) else shrink_factors[0]
        df_size = [int(sz/float(sf) + 0.5) for sf,sz in zip(s_factors,original_size)]
        df_spacing = [((original_sz-1)*original_spc)/(new_sz-1)
                      for original_sz, original_spc, new_sz in zip(original_size, original_spacing, df_size)]
    else:
        df_size = fixed_image.GetSize()
        df_spacing = fixed_image.GetSpacing()

    if initial_transform:
        initial_displacement_field = sitk.TransformToDisplacementField(initial_transform,
                                                                       sitk.sitkVectorFloat64,
                                                                       df_size,
                                                                       fixed_image.GetOrigin(),
                                                                       df_spacing,
```

```python
                                                                fixed_image.GetDirection())
    else:
        initial_displacement_field = sitk.Image(df_size, sitk.sitkVectorFloat64, fixed_image.GetDimension())
        initial_displacement_field.SetSpacing(df_spacing)
        initial_displacement_field.SetOrigin(fixed_image.GetOrigin())

    # Run the registration.
    # Start at the top of the pyramid and work our way down.
    count = 0

    if multiscale_iters is not None:
        multiscale_iters_ = list(multiscale_iters)
        multiscale_iters_.append(0)

    for f_image, m_image in image_pair_generator(fixed_image, moving_image, shrink_factors, smoothing_sigmas):
        initial_displacement_field = sitk.Resample (initial_displacement_field, f_image)

        if multiscale_iters is not None:
            registration_algorithm.SetNumberOfIterations(multiscale_iters_[count])
        initial_displacement_field = registration_algorithm.Execute(f_image, m_image, initial_displacement_field)
        count+=1

    return sitk.DisplacementFieldTransform(initial_displacement_field)
```

The function for warping given a transform (rigid or non-rigid)

```python
# define helper scripts to transform new volumes given a deformation.
def transform_img_sitk(vol, tfm):

    import SimpleITK as sitk

    v = sitk.GetImageFromArray(vol, isVector=False)
    v_transformed = sitk.Resample(v,
                                  v,
                                  tfm, # this should work with all types of transforms.
                                  sitk.sitkLinear,
                                  0.0,
                                  v.GetPixelID())
    v_transformed = sitk.GetArrayFromImage(v_transformed) # back to numpy format.

    return v_transformed
```

```python
im1_raw = im1.copy() # visualisation only.

imfile2 = imfiles[ii+1]
# im2 = skio.imread(imfile2, multifile=False) # multifile is required to prevent loading of all associated files.
im2 = read_tiff(imfile2)
# if demons_pad > 0:
#     im2 = im2[demons_pad:-demons_pad,
#               demons_pad:-demons_pad,
#               demons_pad:-demons_pad]
im2 = im2[x1:x2,y1:y2,z1:z2].copy()
# im2 = np.uint16(ndimage.zoom( im2, [z_scale, 1, 1], order=1, mode='reflect'))
im2_raw = im2.copy() # make a copy for transformation later.
# ===========================================================================
#   Rigid register routine implemented from library.
# ===========================================================================

# replace all 0 intensity pixels by the average of the volume.
im1[im1 == 0] = np.nanmean(im1)
im2[im2 == 0] = np.nanmean(im2)

# this should be the same settings as for the 8 bit now.,....
demons_tfm = registration.SITK_multiscale_demons_registration(im1,
                                                              im2,
                                                              imtype=16,
                                                              p12=(2,98), # percentage scaling
                                                              rescale_intensity=False, # if rescale does not work
                                                              centre_tfm_model='geometry',
                                                              demons_type='diffeomorphic',
                                                              n_iters = 50, # try just a small number of frames first. # this does indeed seem to work !. -> can we try 25? and save the output of this?
                                                              smooth_displacement_field = True,
                                                              smooth_alpha= .8,
                                                              shrink_factors = [2.,1.],
                                                              smoothing_sigmas = [1.,1.],
                                                              eps=1e-12)

# save this transform
savetformfile = os.path.join(saveoutfolder, 'Demons3D_'+os.path.split(imfiles[ii+1])[-1].replace('.tif', '.mat'))
# sitk.WriteTransform(demons_tfm, savetformfile)
WriteDemonsTransformEfficient(demons_tfm, savetformfile)


# apply the transform to im2
im2_tfm = registration.transform_img_sitk(im2_raw, demons_tfm) # apply to image 2.
savefile = os.path.join(saveoutfolder, 'demons_'+os.path.split(imfiles[ii+1])[-1])
skio.imsave(savefile, np.uint16(im2_tfm)) # save the transformed.
```

Reconstruction of the vertices from the demons displacement fields for each time point

```python
for ii in np.arange(len(imfiles))[file_start:file_end+1]: #

    if ii == 0:

        v1 = v.copy()

    else:

        # deformation is required.
        tformfile = os.path.join(saveoutfolder,
                                 'Demons3D_rigid_1_CH00_%s.mat' %(str(ii).zfill(6)))
        print(ii, tformfile)
        # print()
        # # slow?
        # tform = read_demons_transform(tformfile, im0.shape)

        # =============================================================================
        #   Load the deformation field and use just euler step. (optionally we can do mesh based propagation (c.f. active contour))
        # =============================================================================
        # # get the fimage.
        # tform_field = sitk.GetArrayFromImage(tform.GetDisplacementField()) # ok so we need to figure out what is the size relative to the image size. !
        tform_field, fixed = read_demons_field(tformfile, im0.shape)

        # smooth the tform_field a bit
        # tform_field = ndimage.gaussian_filter(tform_field, sigma=1) # this is the correct way to mimic the effect of extracting the original surface with a Gaussian pre-filter.

        # linear interpolate at the exact coordinate point ( but this might not be accurate... )
        dx = uzip.map_intensity_interp3(v[...,[2,1,0]], im0.shape, tform_field[...,0]) # this should be 2... was 0 # doing the other way seems to abide much better!.
        dy = uzip.map_intensity_interp3(v[...,[2,1,0]], im0.shape, tform_field[...,1]) # this should be 1... was 1
        dz = uzip.map_intensity_interp3(v[...,[2,1,0]], im0.shape, tform_field[...,2]) # this should be 0... was 2
        dxyz = np.vstack([dx,dy,dz]).T

        # # for stability we might want to instead do this with curvature regularised updating?
        v1 = (v + dxyz).copy() # i think this is correct due to the nature of the points sampling ... oh wait... shouldn't this be dz,dy,dx.... o crap?
        """
```

2. Binary Segmentation and mesh creation

```python
import skimage.exposure as skexposure

# patch_size_3D_factor = 4 # think this is fine # this is the main culprit!
# # # patch_size_3D_factor = 16
# # check the kernel size.... ---> smaller will give more precise... ?
# kernel_size = np.hstack([im_crop_norm.shape[0]//patch_size_3D_factor,
#                          im_crop_norm.shape[1]//patch_size_3D_factor,
#                          im_crop_norm.shape[2]//patch_size_3D_factor])
# im_crop_norm = skexposure.equalize_adapthist(im_crop_norm,
#                                              kernel_size = kernel_size,
#                                              clip_limit=0.01) # was boost 0.01? boost to 0.02?
# # # # boost  and smooth?
# im_crop_norm = image_fn.normalize(im_crop_norm, pmin=2, pmax=99.8, clip=True)
im_crop_norm = skexposure.adjust_gamma(im_crop_norm, gamma=.5) #.5 is prob enough? # this seems to work better. for datasets2 and 3
# im_crop_norm = skexposure.adjust_gamma(im_crop_norm, gamma=.8) # this seems to work better. for datasets2 and 3
# im_crop_norm = ndimage.gaussian_filter(im_crop_norm, sigma=1, mode='reflect')
# im_crop_norm = ndimage.median_filter(im_crop_norm, size=5)

import prox_tv as ptv
im_crop_norm = ptv.tvgen(im_crop_norm, [.05,.05,.05],[1,2,3], [2,2,2], max_iters=100, n_threads=16) # L1 enhance ridge! # can we use this to get rid of those volume ridge artifacts?

im_crop_norm = image_fn.normalize(im_crop_norm, pmin=2, pmax=99.8, clip=True)
# # im_crop_norm = (im_crop_norm-im_crop_norm.min()) / (im_crop_norm.max() - im_crop_norm.min())

"""
transpose did not seem to make much difference.
"""
im_crop_deconv = restoration.wiener(im_crop_norm, PSF, balance=1.) # use a smaller balance to retain sharper features.
im_crop_deconv = np.clip(im_crop_deconv, 0, 1)
im_crop_deconv = (im_crop_deconv - im_crop_deconv.min()) / (im_crop_deconv.max() - im_crop_deconv.min()+1e-8)

# rescale
im_crop_deconv = image_fn.normalize(im_crop_deconv, pmin=2, pmax=99.8, clip=True)

# im_crop_deconv = skexposure.equalize_adapthist(im_crop_deconv,
#                                                kernel_size = kernel_size,
#                                                clip_limit=0.01) # should only do this.

im_crop_deconv_  = image_fn.std_norm(im_crop_deconv, factor=1.) # 1.3 # we will anyway get rid of this....

# get the higher frequency detail
dog = im_crop_deconv - ndimage.gaussian_filter(im_crop_deconv, sigma=2) # this doesn't work ? was 3? --- match ushape3D? still think 3 is more reliable...

==============================================================================
2. export the segmentation at the end.
==============================================================================
# attempt to segment now using just threshold.
binary_thresh = skfilters.threshold_otsu(im_crop_deconv_)

im_binary = im_crop_deconv_ > binary_thresh ;
im_binary = skmorph.binary_closing(im_binary, skmorph.ball(3))
im_binary = segmentation.largest_component_vol(im_binary)
im_binary = ndimage.morphology.binary_fill_holes(im_binary)

# hm seems we still need to grab the higher frequency signals?
comb = np.maximum(im_binary*1, (dog-dog.mean())/(4*np.std(dog))) # this can be tuned?
final_binary = comb >= 1
# do some clean up ?
final_binary = segmentation.largest_component_vol(final_binary)
```

Restoration is skimage.restoration library. Restoration.wiener was extended to allow 3-d deconvolution. PSF was that of meSPIF made available in uShape3D publication.

```
binary_surf_mesh = meshtools.marching_cubes_mesh_binary(final_binary,
                                                        presmooth=1,
                                                        contourlevel=0.5,
                                                        remesh=True, # make sure to remesh to make better triangles.
                                                        remesh_method='pyacvd', # pyacvd # pyavcd.... seems to deform...
                                                        remesh_samples=.98, # upgrad this resolution..
                                                        predecimate=True, # predecimate still helps for sure.
                                                        remesh_params=optimesh_remesh_params) # how to best pass the parameters object?
```

Modified marching cubes which runs skimage marching cubes after Gaussian smoothing with sigma=1, and then uses pyacvd to remesh equitriangly to 98% of the original vertices

3. Curvature / Intensity measurement

Code for measuring mesh curvature for vertex

```
principal_curvatures = [igl.principal_curvature(vv[...,::-1], f)[:] for vv in v]
#  # separate into the directional fields.
principal_curvature_components_vectors = np.array( [cc[:2] for cc in principal_curvatures], dtype=np.float32)
principal_curvatures = np.array([ cc[2:] for cc in principal_curvatures], dtype=np.float32)

# principal_curvatures_r2 = np.array([igl.principal_curvature(vv[...,::-1], f, radius=2)[2:] for vv in v]) too noisy....
# visualize the curvatures on the surface now in comparison.
prin_kappa_min = (np.max(principal_curvatures, axis=1)) / px_res
prin_kappa_max = np.min(principal_curvatures, axis=1) / px_res

mean_curvatures = .5*(prin_kappa_min+prin_kappa_max)
```

Code for steepest gradient descent into cell i.e. implicit Euler integration

```python
def parametric_mesh_constant_img_flow(mesh, external_img_gradient,
                                      niters=1,
                                      deltaL=5e-4,
                                      step_size=1,
                                      method='implicit',
                                      robust_L=False,
                                      mollify_factor=1e-5,
                                      conformalize=True, gamma=1, alpha=0.2, beta=0.1, eps=1e-12):

    """
    input is a mesh, this just solves the update.
    enable robust_laplacian
    """
    import igl
    import numpy as np
    import scipy.sparse as spsparse
    from tqdm import tqdm
    from ..Unzipping import unzip_new as uzip

    vol_shape = external_img_gradient.shape[:-1]
    v = np.array(mesh.vertices.copy())
    f = np.array(mesh.faces.copy())

    if robust_L:
        import robust_laplacian
        L, M = robust_laplacian.mesh_laplacian(np.array(v), np.array(f), mollify_factor=mollify_factor)
        L = -L # need to invert sign to be same convention as igl.
    else:
        L = igl.cotmatrix(v,f)

    """
    initialise
    """
    # initialise the save array.
    Usteps = np.zeros(np.hstack([v.shape, niters+1]))
    Usteps[...,0] = v.copy()
    U = v.copy()

    """
    propagation
    """
    for ii in tqdm(range(niters)):

        U_prev = U.copy(); # make a copy.
        # get the next image gradient
        U_grad = np.array([uzip.map_intensity_interp3(U_prev,
                                                      grid_shape=vol_shape,
                                                      I_ref=external_img_gradient[...,ch]) for ch in np.arange(v.shape[-1])])
        U_grad = U_grad.T

        if method == 'explicit':
            U_grad = U_grad / (np.linalg.norm(U_grad, axis=-1)[:,None]**2 + eps) # square this.
            U = U_prev + U_grad * step_size #update.
            Usteps[...,ii+1] = U.copy()

        if method == 'implicit':
            U_grad = U_grad / (np.linalg.norm(U_grad, axis=-1)[:,None] + eps)

            if conformalize:
                # if ii ==0:
                if robust_L:
                    import robust_laplacian
                    _, M = robust_laplacian.mesh_laplacian(U_prev, f, mollify_factor=mollify_factor)
                else:
                    M = igl.massmatrix(U_prev, f, igl.MASSMATRIX_TYPE_BARYCENTRIC) # -> this is the only matrix that doesn't degenerate.
                # # implicit solve.
                S = (M - deltaL*L) # what happens when we invert?
                b = M.dot(U_prev + U_grad * step_size)
            else:
                # construct the active contour version.
                S = gamma * spsparse.eye(len(v),len(v)) - alpha * L  + beta * L.dot(L)
                b = U_prev + U_grad * step_size

            # get the next coordinate by solving
            U = spsparse.linalg.spsolve(S,b)
            Usteps[...,ii+1] = U.copy()

    # return all the intermediate steps.
    return Usteps
```

For external edge gradient computation use gradient of signed distance transform

```python
def sdf_distance_transform(binary, rev_sign=True, method='edt'):

    import numpy as np

    pos_binary = binary.copy()
    neg_binary = np.logical_not(pos_binary)

    if method == 'edt':
        from scipy.ndimage import distance_transform_edt
        res = distance_transform_edt(neg_binary) * neg_binary - (distance_transform_edt(pos_binary) - 1) * pos_binary

    if method =='fmm':
        import skmm
        res = skmm.distance(neg_binary) * neg_binary - (skmm.distance(pos_binary) - 1) * pos_binary

    if rev_sign:
        res = res * -1

    return res
```

```python
def surf_normal_sdf(binary, return_sdf=True, smooth_gradient=None, eps=1e-12, norm_vectors=True):

    import numpy as np
    import scipy.ndimage as ndimage

    sdf_vol = sdf_distance_transform(binary, rev_sign=True) # so that we have it pointing outwards!.

    # compute surface normal of the signed distance function.
    sdf_vol_normal = np.array(np.gradient(sdf_vol))
    # smooth gradient
    if smooth_gradient is not None: # smoothing needs to be done before normalization of magnitude.
        sdf_vol_normal = np.array([ndimage.gaussian_filter(sdf, sigma=smooth_gradient) for sdf in sdf_vol_normal])

    if norm_vectors:
        sdf_vol_normal = sdf_vol_normal / (np.linalg.norm(sdf_vol_normal, axis=0)[None,:]+eps)

    return sdf_vol_normal, sdf_vol
```

Intensity measurement (actual computation of raw intensity values)

Binary voxelisation of the mesh into a binary and then taking the mean for the volumetric intensity for normalization.

```python
# for voxelisation we do need to transpose.
vv_vol_binary = meshtools.voxelize_image_mesh_pts(trimesh.Trimesh(vv[...,[2,1,0]],
                                                                  ff,
                                                                  process=False, validate=False),
                                                  dilate_ksize=dilate_voxel,
                                                  erode_ksize=dilate_voxel,
                                                  vol_shape=im0.shape,
                                                  upsample_iters_max=10,
                                                  pitch=2)
```

```python
volume_intensity_all.append(np.nanmean(rigid_im[vv_vol_binary>0])) # important for normalization!.
```

Setting the 1um distance

```python
sample_radius = 1.
# get the barycenter and construct a circle.
sphere_radius = sample_radius/pix_res # 1 um radius as using the same mesh.
```

```python
def percentile_mean( vals, percent=90) :
    thresh = np.percentile(vals, percent)
    return np.nanmean(vals[vals>=thresh])
```

```python
sphere_radius = 1./.104    # first depth = 1um
v_depth = meshtools.parametric_mesh_constant_img_flow(mesh,
                                                       external_img_gradient = sdf_vol_normal.transpose(1,2,3,0),
                                                       niters=int(sphere_radius * 2.),  # as we are taking half steps!.
                                                       deltaL=5e-5,
                                                       step_size=.5, # more stable.  # o crap..... so i should be working ....
                                                       method='implicit',
                                                       conformalize=True,
                                                       gamma=1,
                                                       alpha=0.2,
                                                       beta=0.1,
                                                       eps=1e-12)

##### then we simply use this to look up in the image....
##### accumulate the distance along the trajectory and derive a mask to avoid taking deeper than a threshold....
# pulldown intensities
v_depth_I = uzip.map_intensity_interp3(v_depth.transpose(0,2,1).reshape(-1,3),
                                       rigid_im.shape,
                                       I_ref=rigid_im)
v_depth_I = v_depth_I.reshape(-1,v_depth.shape[-1])

dist_v_depth0 = np.linalg.norm(v_depth - v_depth[...,0][...,None], axis=1)
valid_I = dist_v_depth0<=sphere_radius
v_depth_I[ valid_I == 0 ] = np.nan # replace with nans

# we can also set to nan for...
inmesh_face_pts_I_mean_ = []
for sig in v_depth_I:
    thresh = np.percentile( sig[~np.isnan(sig)], 95)
    inmesh_face_pts_I_mean_.append(np.nanmean(sig[sig>thresh]))
inmesh_face_pts_I_mean = np.hstack(inmesh_face_pts_I_mean_)

# inmesh_face_pts_I_mean = np.nanmax(v_depth_I, axis=1)
# # inmesh_face_pts_I_mean = np.hstack([np.nanmean(xyz_I[ind_ii]) for ind_ii in ind]) # get the mean intensity
inmesh_face_pts_I_mean[np.isnan(inmesh_face_pts_I_mean)] = 0
septin_all_depth1.append(inmesh_face_pts_I_mean)


if ii == 0:
    I_min_depth1 = np.percentile(inmesh_face_pts_I_mean, 1)
    I_max_depth1 = np.percentile(inmesh_face_pts_I_mean, 99)

# apply coloring percentile? # this is fluctuating a lot?
intensity_colors = vol_colors.get_colors(inmesh_face_pts_I_mean,
                                         colormap=cm.RdYlBu_r,
                                         vmin=I_min_depth1,
                                         vmax=I_max_depth1) # was 275,400 for 180522, Cell5
                                         # vmin=np.percentile(inmesh_face_pts_I_mean,0),  # set this to be absolute scale.
                                         # vmax=np.percentile(inmesh_face_pts_I_mean,99))

# create a new mesh withe added color schema. # this works?
binary_surf_mesh = trimesh.Trimesh(vertices = vv ,
                                   faces = ff,
                                   # vertex_colors= np.uint8(255*H_colors[...,:3]),
                                   vertex_colors=np.uint8(255*intensity_colors[...,:3]), #face_colors
                                   process=False,
                                   validate=False)
```

Actual sampling for a single timepoint.

4. Bleach correction

```python
# correct and express as a relative series.
septin_faces_all_correct = septin_faces_all/(septin_vol_intensity_all[:,None] + 1e-12) # to prevent 0 division.
```

5. Spatial mesh smoothing of timeseries

```python
def smooth_scalar_function_mesh(mesh, scalar_fn, exact=False, n_iters=10, weights=None, return_weights=True, alpha=0):

    """
    If exact we use M-1^L else we use the cotangent laplacian which is super fast with no inversion required. 
    allow optional usage of weights. 
    """
    import igl 
    import time
    import numpy as np 
    import scipy.sparse as spsparse
    
    v = mesh.vertices.copy()
    f = mesh.faces.copy()

    scalar0 = scalar_fn.copy()

    # # if ii ==0:
    L = igl.cotmatrix(v,f)
    M = igl.massmatrix(v, 
                       f, 
                       igl.MASSMATRIX_TYPE_BARYCENTRIC) # -> this is the only matrix that doesn't degenerate.            
    
    if exact:
        if weights is None:
            # t1 = time.time()
           weights = spsparse.linalg.spsolve(M, L) # m_inv_l # this inversion is slow.....  the results for me is very similar....  but much slower!. 
           # print('matrix inversion for weights in: ', time.time() - t1,'s') # 65s? 
    else:
        weights = L
    
    for iteration in np.arange(n_iters):
        
        scalar0_next =  (np.squeeze((np.abs(weights)).dot(scalar0)) / (np.abs(weights).sum(axis=1))) 
        scalar0_next = np.array(scalar0_next).ravel()
        scalar0_next = scalar0_next.reshape(scalar_fn.shape)
        scalar0 = scalar0_next.copy()
        # print(kappa0_next.shape)
        
    # case to numpy array and return
    if return_weights:
        return weights, np.array(scalar0)
    else:
        return np.array(scalar0)
```

```python
septin_faces_all_smooth = []
dseptin_faces_all_smooth = []
curvature_faces_all_smooth = []


for tt in tqdm(np.arange(len(septin_faces_all))):

    # this was previously a mistake!.
    mesh = trimesh.Trimesh(vertices=v[tt],
                           faces=f,
                           process=False,
                           validate=False)

    # septin
    # modify for exact, we only need to compute the matrix once?
    smooth_weights, septin_faces_all0 = meshtools.smooth_scalar_function_mesh(mesh,
                                                                             scalar_fn = I_smooth_ma[tt][:,None],
                                                                             exact=True,
                                                                             n_iters=int(np.ceil(np.log2(spatial_size))), #-1,
                                                                             return_weights=True)
    # smooth_scalar_function_mesh(mesh, scalar_fn, exact=False, n_iters=10, weights=None, return_weights=True)
    septin_faces_all0 = np.squeeze(septin_faces_all0)
    septin_faces_all_smooth.append(septin_faces_all0)

    # curvature.
    kappa_faces_all0 = meshtools.smooth_scalar_function_mesh(mesh,
                                                             scalar_fn = Kappa_smooth_ma[tt][:,None],
                                                             exact=True,
                                                             n_iters=int(np.ceil(np.log2(spatial_size))),
                                                             weights=smooth_weights,
                                                             return_weights=False)
    kappa_faces_all0 = np.squeeze(kappa_faces_all0)
    curvature_faces_all_smooth.append(kappa_faces_all0)


    # we need to design a faster and more effective smoother!.
    dseptin_faces_all_smooth0 = meshtools.smooth_scalar_function_mesh(mesh,
                                                                      scalar_fn = I_smooth_ma_lincoef[tt][:,None],
                                                                      exact=True,
                                                                      n_iters=int(np.ceil(np.log2(spatial_size))),
                                                                      weights=smooth_weights,
                                                                      return_weights=False)
    dseptin_faces_all_smooth.append(np.squeeze(dseptin_faces_all_smooth0))


septin_faces_all_smooth = np.array(septin_faces_all_smooth)   ######### we would use this for analysis mainly.
septin_faces_all_smooth_faces = igl.average_onto_faces(f, septin_faces_all_smooth.T).T

dseptin_faces_all_smooth = np.array(dseptin_faces_all_smooth)
dseptin_faces_all_smooth_faces = igl.average_onto_faces(f, dseptin_faces_all_smooth.T).T

curvature_faces_all_smooth = np.array(curvature_faces_all_smooth)
# average onto faces.
curvature_faces_all_smooth_faces = igl.average_onto_faces(f, curvature_faces_all_smooth.T).T   ####### we use this for analysis.
```

6. Autocorrelation determination

```python
"""
Autocorr
"""
autocorr_kappa =  tsa.autocorr_timeseries_set_1d(curvature_faces_all_um_mesh[:,include_vertex].T, norm=True, eps=1e-12)
autocorr_septin = tsa.autocorr_timeseries_set_1d(septin_vertex_all[:,include_vertex].T, norm=True, eps=1e-12)

mean_bleb_autocorr_kappa = np.nanmean(autocorr_kappa, axis=0)
std_bleb_autocorr_kappa = np.nanstd(autocorr_kappa, axis=0)
mean_bleb_autocorr_septin = np.nanmean(autocorr_septin, axis=0)
std_bleb_autocorr_septin = np.nanstd(autocorr_septin, axis=0)

# find peaks and use this to estimate the periodicity
from scipy.signal import find_peaks

# peaks_mean_bleb_autocorr_kappa = find_peaks(mean_bleb_autocorr_kappa, prominence=0.0001, height=0, distance=5) # must have height 0
# peaks_mean_bleb_autocorr_septin = find_peaks(mean_bleb_autocorr_septin, prominence=0.0001, height=0, distance=5)
# peaks_mean_bleb_autocorr_kappa = find_peaks(mean_bleb_autocorr_kappa,  height=0, distance=5) # must have height 0
# peaks_mean_bleb_autocorr_septin = find_peaks(mean_bleb_autocorr_septin,  height=0, distance=5)
peaks_mean_bleb_autocorr_kappa = find_peaks(mean_bleb_autocorr_kappa,  distance=5) # must have height 0
peaks_mean_bleb_autocorr_septin = find_peaks(mean_bleb_autocorr_septin,  distance=5)

xplot = np.arange(len(mean_bleb_autocorr_kappa)) * Ts
plt.figure()
plt.plot(xplot,mean_bleb_autocorr_kappa, label=r'$\kappa [1/um]')
plt.fill_between(xplot,
                 mean_bleb_autocorr_kappa-std_bleb_autocorr_kappa,
                 mean_bleb_autocorr_kappa+std_bleb_autocorr_kappa, color='lightgrey')
plt.plot(xplot[peaks_mean_bleb_autocorr_kappa[0][0]],
         mean_bleb_autocorr_kappa[peaks_mean_bleb_autocorr_kappa[0][0]], 'ko')
plt.vlines(xplot[peaks_mean_bleb_autocorr_kappa[0][0]],
           -0.3, 1, colors='k', linestyles='dashed')

plt.plot(xplot, mean_bleb_autocorr_septin, label='Norm. Septin')
plt.fill_between(xplot,
                 mean_bleb_autocorr_septin-std_bleb_autocorr_septin,
                 mean_bleb_autocorr_septin+std_bleb_autocorr_septin, color='grey')
plt.plot(xplot[peaks_mean_bleb_autocorr_septin[0][0]],
         mean_bleb_autocorr_septin[peaks_mean_bleb_autocorr_septin[0][0]], 'ro')
plt.vlines(xplot[peaks_mean_bleb_autocorr_septin[0][0]],
           -0.3, 1, colors='r', linestyles='dashed')
plt.ylabel('Normalized Autocorrelation')
plt.xlabel('Lag [s]')
plt.legend(loc='best')
plt.tick_params(length=5, right=True)
"""
save this -> this establishes the timescales
"""
plt.savefig(os.path.join(analysis_savefolder,
                         'Autocorrelation_septin_curvature_raw.png'), dpi=300, bbox_inches='tight')
plt.show()
```

7. Spatial correlation determination

```python
"""
Spatialcorr To determine the size coupling.
"""
spatial_corr_mesh = trimesh.Trimesh(vertices=v[0],
                                    faces = f,
                                    process=False,
                                    validate=False)

import time
t1 = time.time()
k_rings = meshtools.get_k_neighbor_ring(spatial_corr_mesh, K=15, stateful=True)
t2 = time.time()
print('k-ring constructions ', t2-t1)

# exclusion k_rings.. to limit the distance... computation....
exclude_k_rings = [[np.setdiff1d(k_rings[ii+1][rr], k_rings[ii][rr]) for rr in np.arange(len(k_rings[0]))] for ii in np.arange(len(k_rings)-1)]
exclude_k_rings = [[np.setdiff1d(k_rings[0][ii], ii) for ii in np.arange(len(k_rings[0]))]] + exclude_k_rings   #### this can not be used to establish distance... . k_rings 0 should exclude self... .


"""
create a version that is nan-masked for excluded vertices.
"""
curvature_faces_all_um_mesh_masked = curvature_faces_all_um_mesh.copy()
curvature_faces_all_um_mesh_masked[:,exclude_vertex] = np.nan

septin_vertex_all_masked = septin_vertex_all.copy()
septin_vertex_all_masked[:,exclude_vertex] = np.nan

"""
spatialcorr of curvature and septin
"""
t1 = time.time()

all_ring_corrs_curvature = []
all_ring_corrs_septin = []

# this needs paralellisation!.
for ring_ii in np.arange(len(exclude_k_rings))[:]:
    vertex_means_pearsonr_curvature = tsa.spatialcorr_k_neighbors(curvature_faces_all_um_mesh_masked.T,
                                                                  exclude_k_rings[ring_ii],
                                                                  norm=True, eps=1e-12)
    all_ring_corrs_curvature.append(vertex_means_pearsonr_curvature)

    vertex_means_pearsonr_septin = tsa.spatialcorr_k_neighbors(septin_vertex_all_masked.T,
                                                               exclude_k_rings[ring_ii],
                                                               norm=True, eps=1e-12)
    all_ring_corrs_septin.append(vertex_means_pearsonr_septin)

print(time.time()-t1)

all_ring_corrs_means_curvature = np.hstack([np.nanmean(cc) for cc in all_ring_corrs_curvature])
all_ring_corrs_std_curvature = np.hstack([np.nanstd(cc) for cc in all_ring_corrs_curvature])
all_ring_corrs_means_septin = np.hstack([np.nanmean(cc) for cc in all_ring_corrs_septin])
all_ring_corrs_std_septin = np.hstack([np.nanstd(cc) for cc in all_ring_corrs_septin])

# for all of this append 0 and 0
all_ring_corrs_means_curvature = np.hstack([1, all_ring_corrs_means_curvature])
all_ring_corrs_std_curvature = np.hstack([0, all_ring_corrs_std_curvature])
all_ring_corrs_means_septin = np.hstack([1, all_ring_corrs_means_septin])
all_ring_corrs_std_septin = np.hstack([0, all_ring_corrs_std_septin])


all_lengths = [igl.edge_lengths(vv, f)*.104 for vv in v]
all_edge_lengths = np.hstack([np.nanmean(igl.edge_lengths(vv, f)) for vv in v] )
```

```python
##### plotting
plt.figure()
plt.plot(np.arange(len(all_ring_corrs_means_curvature)) * .104 * all_edge_lengths.mean(),
         all_ring_corrs_means_curvature, label=r'$\kappa [1/um]$')
plt.fill_between(np.arange(len(all_ring_corrs_means_curvature)) * .104 * all_edge_lengths.mean(),
                 all_ring_corrs_means_curvature -all_ring_corrs_std_curvature,
                 all_ring_corrs_means_curvature +all_ring_corrs_std_curvature, color='lightgrey')
# # plt.plot(peaks_mean_bleb_autocorr_kappa[0],
#             # mean_bleb_autocorr_kappa[peaks_mean_bleb_autocorr_kappa[0]], 'ko')
# # plt.vlines(peaks_mean_bleb_autocorr_kappa[0],
#              # -0.3, 1, colors='k', linestyles='dashed')
plt.plot(np.arange(len(all_ring_corrs_means_curvature)) * .104 * all_edge_lengths.mean(),
         all_ring_corrs_means_septin, label='Norm. Septin')
plt.fill_between(np.arange(len(all_ring_corrs_means_septin))*.104 * all_edge_lengths.mean(),
                 all_ring_corrs_means_septin-all_ring_corrs_std_septin,
                 all_ring_corrs_means_septin+all_ring_corrs_std_septin, color='grey')
plt.ylabel('Norm. Spatial Correlation')
plt.xlabel('Lag [um]')
plt.legend(loc='best')
plt.tick_params(length=5, right=True)
"""
save this -> this establishes the timescales
"""
plt.vlines((np.arange(len(all_ring_corrs_means_curvature)) * .104 * all_edge_lengths.mean())[spatial_size],
           -0.3, 1, colors='k', linestyles='dashed')
plt.savefig(os.path.join(analysis_savefolder,
                         'Spatialcorrelation_septin_curvature_raw.png'), dpi=300, bbox_inches='tight')
plt.show()
```

Utility functions:

Converting vertex timeseries to face timeseries (igl.average_onto_faces)