Electronic Supplementary Material (ESI) for Biomaterials Science.

Supplementary Information

FORMATION AND DYNAMICS OF ENDOPLASMIC RETICULUM-LIKE LIPID NANOTUBE NETWORKS

Taylan Bilal and Irep Gözen

IMAGE ANALYSIS PYTHON SCRIPT

```
from __future__ import division
from skimage import filters, color, exposure, io, measure, img_as_ubyte #
collection of the image processing algorithms for image loading, filtering,
morphology, segmentation, color conversions, and transformations
from thinning import guo_hall_thinning #skeletonizes the image
from collections import defaultdict
from itertools import izip
import matplotlib.pyplot as plt  # for plotting
import numpy as np # fundamental package for scientific computing
import networkx as nx # for the creation, manipulation, and study of the
structure, dynamics, and functions of complex networks
import pickle as pkl
import gc
import json
import os
import sys
import cv2 as cv
from scipy.spatial import ConvexHull # calculates and outlines the convex hull
of the points in the network

__author__ = 'taylanbil'

# keyword arguments for image processing

KWARGS = {
    'gaussian_filter': {'sigma': .8},
    'threshold': {'method': 'li'},
    'high_contrast': {'in_range': (0, 255)},
    'merge_blobs': {'min_distance': 4},
    'depth_first_edge_detection': {'min_len': 10, 'min_angle': 30}
}
YJUNCTION = 'yjunction'  # Y-junction:three-way junctions connecting 3 tubes
ENDPT = 'endpt'  # endpoint:the terminal point of lipid tubes

BLOB = 'blob'  # blob:flat membrane area which is retracting over time

TURNING = 'turning'  # turning: points along the tubes through which the tubes
make sharp, V-shaped turn

# different types of points will be assigned to the colors specified below:

NODE_COLORS = {YJUNCTION: (255, 0, 0), # red
               TURNING: (255, 0, 255), # magenta
               BLOB: (0, 255, 0), # green
               ENDPT: (255, 255, 0)} # yellow
```

```
def load_image(filename):
    return img_as_ubyte(color.rgb2gray(io.imread(filename)))


def save_image(img, filename, **kwargs):
    kwargs['cmap'] = kwargs.get('cmap', 'gray')
    kwargs['interpolation'] = kwargs.get('interpolation', 'nearest')
    fig, ax = plt.subplots()
    ax.imshow(img, **kwargs)
    fig.savefig(filename)
```

DETECTING TUBES (EDGES), POINTS (NODES) and AREAS (BLOBS)

```
def draw_graph(img, graph, edge_trace=None):
    tmp = color.gray2rgb(img.copy())
    if edge_trace is not None:
        assert all(i == j for i, j in zip(img.shape, edge_trace.shape))
        tmp[edge_trace > 0, :] = [0, 0, 255]  # draws edges (tubes) in dark
blue

    node_size = int(np.ceil((max(img.shape) / 750.)))
    tmp = draw_nodes(tmp, graph, max(node_size, 2))  # draws the different
types of nodes in associated colors specified above
    return tmp
def draw_convex_hull(img, graph, edge_trace=None):
    tmp = draw_graph(img, graph, edge_trace=edge_trace)
    # plotting convex hull
    points = np.array(
        [node for node, nodedat in graph.nodes_iter(data=True)
         if nodedat['type'] == ENDPT])
    ch = ConvexHull(points)
    for simplex in ch.simplices:
        start = tuple(reversed(points[simplex[0]]))
        end = tuple(reversed(points[simplex[1]]))
        cv.line(tmp, start, end, (0, 255, 0), 1)  # draws a green outline of a
convex hull (tubular area)

    return tmp


def draw_nodes(img, graph, radius=3):
    for (x, y), dat in graph.nodes_iter(data=True):
        color = NODE_COLORS[dat['type']]
        cv.rectangle(img, (y-radius, x-radius), (y+radius, x+radius),
                     color, -1)
    color = NODE_COLORS[TURNING]
    for n1, n2, dat in graph.edges_iter(data=True):
        for x, y in dat.get('turning_points', []):
            cv.rectangle(img, (y-radius, x-radius), (y+radius, x+radius),
                         color, -1)
    return img
```

NOISE REMOVAL AND THINNING

```
def high_contrast(img, in_range=(0, 255)):
    return exposure.rescale_intensity(img, in_range=in_range)
```

```
def gaussian_filter(img, sigma=0.8):
    if sigma is None:
        return img
    return color.rgb2gray(filters.gaussian(img, sigma))


def threshold(img, method='li', adjust=1):
    method = method if method.startswith('threshold_') else \
        'threshold_%s' % method
    try:
        t = getattr(filters, method)(img)
        t *= adjust
        return img_as_ubyte(img > t)
    except Exception:
        raise


def thin(bw_img):
    img = bw_img.copy()
    skel = guo_hall_thinning(img_as_ubyte(img))
    return skel
```

DETECTION OF POINTS ('NODES')

```
def check_pixel_neighborhood(x, y, skel):
    # checks the number of components around a pixel; if it is either 1 or
more than 3, it is detected as a node (point)

    item = skel.item
    p1 = (x, y)
    p2 = item(x - 1, y)
    p3 = item(x - 1, y + 1)
    p4 = item(x, y + 1)
    p5 = item(x + 1, y + 1)
    p6 = item(x + 1, y)
    p7 = item(x + 1, y - 1)
    p8 = item(x, y - 1)
    p9 = item(x - 1, y - 1)


    A = (p2 == 0 and p3 > 0) + (p3 == 0 and p4 > 0) + \
        (p4 == 0 and p5 > 0) + (p5 == 0 and p6 > 0) + \
        (p6 == 0 and p7 > 0) + (p7 == 0 and p8 > 0) + \
        (p8 == 0 and p9 > 0) + (p9 == 0 and p2 > 0)
    if A >= 3:
        return YJUNCTION, A  # detecting as a Y-junction
    if A == 1:
        return ENDPT, A  # detecting as an endpoint
    else:
        return False, A

def zhang_suen_node_detection(skel):

    # From 'Network Extraction From Images' (NEFI)[1]:
    """
```

```
    Node detection has been performed as described by Zhang and Suen[2].
Briefly, pixels(p) of the skeleton are categorized as nodes/non-nodes based on
the value of a function A(p) and the pixel neighborhood of p.
    A(p1) == 1: The pixel p1 sits at the end of a skeleton line, thus a node
of degree 1 has been found. (endpoint)
    A(p1) == 2: The pixel p1 sits in the middle of a skeleton line but not at
a branching point, thus a node of degree 2 has been found. Such nodes are
ignored and not introduced to the graph. (tube)
    A(p1) >= 3: The pixel p1 belongs to a branching point of a skeleton line,
thus a node of degree >=3 has been found. (Y-junction)
    Args:skel: Skeletonised source image. The skeleton must be exactly 1 pixel
wide.Returns: Graph object with detected nodes.

    """
    # Turning points are going to be identified separately, in edge detection.

    graph = nx.Graph()
    w, h = skel.shape
    item = skel.item
    for x in xrange(1, w - 1):
        for y in xrange(1, h - 1):
            node_type, numlegs = check_pixel_neighborhood(x, y, skel)
            if item(x, y) != 0 and node_type is not False:
                graph.add_node((x, y), {'type': node_type, 'numlegs':
numlegs})
    return graph

def neighbors_(x, y, skel, graph, tie_it_up, oldnode=None):
    width, height = skel.shape
    nbs = []
    for dy in [-1, 0, 1]:
        for dx in [-1, 0, 1]:
            if (dx != 0 or dy != 0) and 0 <= x + dx < width and 0 <= y + dy <
height and skel.item(x + dx, y + dy) != 0:
                nbs.append((x + dx, y + dy))
        if oldnode:
        nbs = [n for n in nbs if n != oldnode]
        diff = np.subtract((x, y), oldnode)
        diffs = map(lambda nb: np.subtract(nb, (x, y)), nbs)
        diffs = [d-diff for d in diffs]
        diffs = [sum(map(abs, d)) for d in diffs]
        nbs = sorted(zip(nbs, diffs), key=lambda r: r[1], reverse=True)
stack.pop
        nbs = [nb for nb, d in nbs]
    if tie_it_up:
        node_nbs = [pixel for pixel in nbs if graph.has_node(pixel) if pixel
!= oldnode]
        if node_nbs:
            return node_nbs[:1]
    return nbs

def distance_transform_diameter(edge_trace, segmented):
    dt = cv.distanceTransform(segmented, distanceType=2, maskSize=0)
    edge_pixels = np.nonzero(edge_trace)
    diameters = defaultdict(list)
    for label, diam in izip(edge_trace[edge_pixels], 2.0 * dt[edge_pixels]):
        diameters[label].append(diam)
```

```
    return diameters


def get_edge_trace(edge_pixels, skel):
    edge_trace = np.zeros(skel.shape, np.uint32)
    for label, pixels in edge_pixels.iteritems():
        for pixel in pixels[1:-1]:
            edge_trace[pixel] = label
                return edge_trace
```

DETECTION OF TUBES ('EDGES')

```
def depth_first_edge_detection(skel, graph_, min_len=10, min_angle=30):

    g, label = graph_.copy(), 1
    label_length = defaultdict(int)
    seen = set()
    edge_pixels = defaultdict(list)
    for source_x, source_y in graph_.nodes_iter():
        x, y = source_x, source_y
        stack = [
            ((a, b), (x, y))
            for a, b in neighbors_(x, y, skel, graph_, tie_it_up=True)
            if (a, b) not in seen]
        while stack:
            (x, y), (xold, yold) = stack.pop()

            if (x, y) in seen:
                continue
            edge_pixels[label].append((x, y))
            diag_advance = abs(x-xold) == 1 and abs(y-yold) == 1
            label_length[label] += 1.414214 if diag_advance else 1
            if graph_.has_node((x, y)):
                edge_pixels[label] = [(source_x, source_y)] +
edge_pixels[label]
                g.add_edge(
                    (source_x, source_y), (x, y),
                    label=label,
                    turning_points=identify_turning_pts(
edge_pixels[label], min_len=min_len, min_angle=min_angle),
                    numpixels=len(edge_pixels[label]),
                    length=label_length[label]
                                    )
                label += 1
            else:
                # edge still continues
                newstack = [
                    ((a, b), (x, y))
                    for a, b in neighbors_(
                        x, y, skel, graph_, tie_it_up=True, oldnode=(xold,
yold))

                    if (a, b) not in seen and (a, b) != (xold, yold)]
                if len(newstack) != 1:
                    newstack = (
                        (n1, n) for n1, n in newstack
```

```
                        if not any(_[0] == n1 for _ in stack))
                stack.extend(newstack)
                seen.add((x, y))
        popthese = set()
    for label, pl in edge_pixels.iteritems():
        if len(pl) < 2:
            import pdb; pdb.set_trace()  # XXX BREAKPOINT
        try:
            assert len(pl) >= 2, 'pixel list length is %s' % str(len(pl))
            assert g.has_node(pl[0]), ('initial pixel is not a node: ' +
str(pl[0]))
            assert g.has_node(pl[-1]), ('last pixel is not a node: ' +
str(pl[-1]))
            assert all(not g.has_node(i) for i in pl[1:-1]), 'there are nodes
in the edge?'
        except Exception as e:
            print e.message
            print 'edge deleted, %s pixels' % len(pl)
            popthese.add(label)

    for k in popthese:
        edge_pixels.pop(k)
    return g, edge_pixels


def keep_largest(graph_):
    graph = graph_.copy()
    connected_components = sorted(
        list(nx.connected_component_subgraphs(graph)),
        key=lambda graph: graph.number_of_nodes(),
        reverse=True)
    to_be_removed = connected_components[1:]

    for subgraph in to_be_removed:
        graph.remove_nodes_from(subgraph)
    return graph

def tube_length(pixels):

    # To get the length of a tube spawning from the blob:
    ans = 0
    j = 1
    while j < len(pixels):
        (x, y), (a, b) = pixels[j-1], pixels[j]
        ans += 1.414214 if abs(x - a) == 1 and abs(y - b) == 1 else 1
        j += 1
    return ans
```

DETECTION OF FLAT MEMBRANE AREA ('BLOB')

```
# Detection of the flat intact membrane area, the so called 'BLOB. Thinning
creates a collection of nodes and edges throughout the original blob area.
This network structure is not really there; it is a consequence of the
thinning algorithm applied to a region of white pixels. Therefore, this part
of the network needs to be eliminated.

def merge_nodes(graph, bloblabel, nodes, dist_transf_conn_comp, edge_pixels):
```

```
    G = graph.copy()
    new_node = tuple(map(int, np.mean(np.array(list(nodes)), axis=0)))
    for n in nodes:
        G.remove_node(n)
    G.add_node(new_node, {'blob': bloblabel, 'type': BLOB})
    for n1, n2, dat in graph.edges_iter(nbunch=nodes, data=True):
        if n1 in nodes and n2 in nodes:
            continue
        keepnode = n1 if n2 in nodes else n2
        label = dat['label']
        edge_pixels[label] = [pixel for pixel in edge_pixels[label]
                              if dist_transf_conn_comp.item(pixel) == 0]
        turning_points = [pixel for pixel in dat['turning_points']
                          if dist_transf_conn_comp.item(pixel) == 0]
        G.add_edge(
            new_node, keepnode,
            blob=True,
            label=label,
            turning_points=turning_points,
            numpixels=len(edge_pixels[label]),
            length=tube_length(edge_pixels[label])

    return G


def merge_blobs(bw_image, graph, edge_pixels, min_distance=4):
        dt = cv.distanceTransform(bw_image.copy(), distanceType=2, maskSize=0)
    conn_comp = measure.label(dt > min_distance)
    labels = {i: set() for i in np.unique(conn_comp)}
    G = graph.copy()
    for x, y in G.nodes_iter():
        label = conn_comp[x, y]
        if label > 0:
            labels[label].add((x, y))
    for label, nodes in labels.iteritems():
        if not nodes:
            continue
        G = merge_nodes(G, label, nodes, conn_comp, edge_pixels)
    big_blob_area, big_blob_index = max(
        ((np.sum(conn_comp == i), i) for i in np.unique(conn_comp) if i),
        key=lambda r: r[0]
    )
    return G, big_blob_area, big_blob_index, conn_comp


def identify_turning_pts(pixel_list, min_len=10, min_angle=30):

    # This identifies turning points, by keeping track of the current
location,`Min_len` pixels ahead and `min_len` pixels behind. If those 3 points
make an angle larger than `min_angle`, then the current location has been
determined as a turning point
        if len(pixel_list) < 2*min_len + 1:
        return []
    pixels = pixel_list
    turn_list, current_run = [], []
    for i, (p0, p1, p2) in enumerate(izip(pixels, pixels[min_len:],
pixels[min_len*2:]), min_len):
```

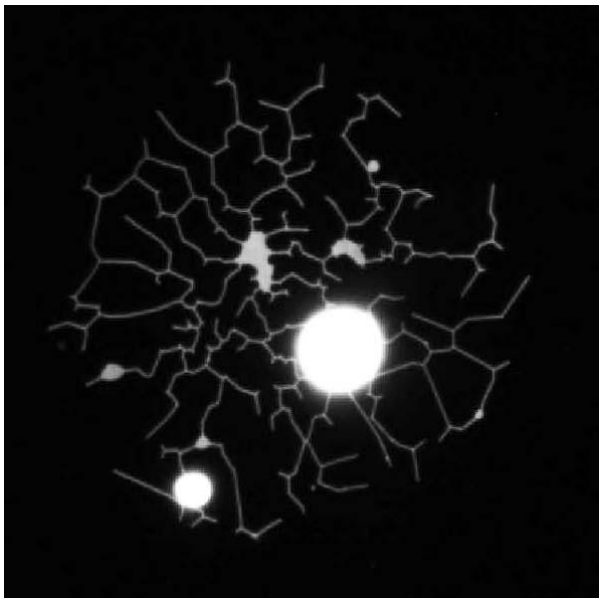
```
        v1, v2 = np.array(p1)-np.array(p0), np.array(p2)-np.array(p1)
        angle = np.dot(v1, v2)/(np.dot(v1, v1)**0.5)/(np.dot(v2, v2)**0.5)
        angle = np.rad2deg(np.arccos(angle))
        if angle >= min_angle:
            current_run.append(i)
        elif current_run:
            # current run is done
            turn_list.append(pixel_list[int(np.median(current_run))])
            current_run = []
    if current_run:
        turn_list.append(pixel_list[int(np.median(current_run))])
    return turn_list


class ImageGraphExtractor(object):

    # names of the functions used:

    THINGS = ['depth_first_edge_detection',
              'breadth_first_edge_detection', 'check_pixel_neighborhood',
              'merge_blobs', 'distance_transform_diameter',  # 'rgb2gray',
              'gaussian_filter',
              'threshold', 'guo_hall_thinning', 'zhang_suen_node_detection',
              'keep_largest', 'load_image', 'merge_nodes', 'neighbors',
              'save_image', 'thin', 'high_contrast']

    # to display selected frames with the following order

    TITLES = ['original image',  # original (raw)
              'high contrast',  # increased contrast
              'gaussian filter',  # Gaussian filter
              'thresholded',  # binary image
              'thinned - overlaid',  # skeletized image
              'graph - overlaid',  # different types of points with associated
color codes and the skeleton of the image are superimposed on the raw image
              'largest connected component graph - overlaid',
              'graph - FINAL',  # same as 'graph - overlaid' but additionally
shows the flat membrane area and the MLV
              'FINAL 2']
```

EXTRACTION AND OVERLAY OF PROCESSED FRAMES

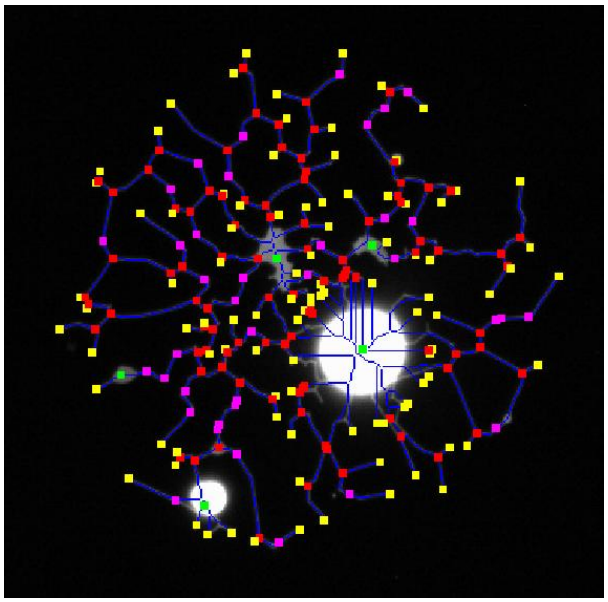
```
# The following extracts the points&tubes from the video (sequence of images),
using the specified selection of parameters per image range, and subsequently
exports the frames of interest where the detected points and tubes are
superimposed on the original one.

# VIDEO 1 (Movie 1)

kwargs_1 = {

     '1-830': {
      'high_contrast': {'in_range': (0, 255)},
      'gaussian_filter': {'sigma': 0.5},
      'threshold': {'method': 'li', 'adjust': 0.5},
      'depth_first_edge_detection': {'min_len': 10, 'min_angle': 30},
      'merge_blobs': {'min_distance': 5}},
```

```
 }

kwargs_1 = {

video_name = '1'
#show_these = []
force = []
for key, kwargs in kwargs_1.iteritems():
    img_range = map(int, key.split('-'))
    video_main(kwargs, video_name, img_range=img_range, force=force,
show_these=show_these)
```

 # Example of a raw frame from Mov1 and its corresponding processed frame, have been presented below:



Original image



Detected points and tubes are superimposed on the original image. Blue: Tubes, Red: Y-junctions, Magenta: Turning Points(V-junctions), Yellow: Endpoints

The rest of the script is about exporting the number of points, the length of tubes or the area of the flat membrane; all of which have been detected with the code above. This part is relatively generic and has not been included.

REFERENCES

1. M. Dirnberger, T. Kehl and A. Neumann, *Scientific Reports*, 2015, 5.
2. T. Y. Zhang and C. Y. Suen, *Communications of the ACM*, 1984, 27, 236-239.