```
import numpy as np
from matplotlib import pyplot as plt
from scipy.interpolate import interp1d
from scipy.signal import fftconvolve
import sys
#from scipy.misc import imread,factorial
from glob import glob
import sys,os
import numpy as np
import hashlib
from matplotlib import pyplot as plt
from scipy.ndimage import morphology
from scipy.ndimage import filters
from scipy import ndimage
from scipy import signal
from mpl_toolkits.axes_grid1 import make_axes_locatable
from scipy.optimize import curve_fit
from numpy.fft import fft,ifft,fftshift
from time import time
import itertools

class Clock:

    def tick(self):
        self.t0 = time()

    def tock(self,label=''):
        print '%s: %0.5f'%(label,time()-self.t0)


def scaleshow(im,dpi=50,clim=None):
    if clim is None:
        clim = (np.min(im),np.max(im))

    sy,sx = im.shape
    iy = float(sy)/float(dpi)
    ix = float(sx)/float(dpi)*1.25
    plt.figure(figsize=(ix,iy))
    plt.axes([0,0,.8,1])
    plt.imshow(im,interpolation='none',cmap='gray',clim=clim)
    plt.xticks([])
    plt.yticks([])
    plt.colorbar(fraction=.05,pad=.05)
    plt.pause(.1)
    return clim

def hash(self,input_string,max_val=2**32):
    md5 = hashlib.md5()
    md5.update(input_string)
    return int(md5.hexdigest(),16)%max_val
```

```
def polyfit2d(x, y, z, order=3):
    ncols = (order + 1)**2
    G = np.zeros((x.size, ncols))
    ij = itertools.product(range(order+1), range(order+1))
    for k, (i,j) in enumerate(ij):
        G[:,k] = x**i * y**j
    m, _, _, _ = np.linalg.lstsq(G, z)
    return m

def polyval2d(x, y, m):
    order = int(np.sqrt(len(m))) - 1
    ij = itertools.product(range(order+1), range(order+1))
    z = np.zeros_like(x).astype(np.float64)
    for a, (i,j) in zip(m, ij):
        z += a * x**i * y**j
    return z


def lateral_smooth_3d(volume,kernel_radius):

    if kernel_radius<=1:
        out = volume
    else:
        fvol1 = np.fft.fft2(volume,axes=(1,2))
        fvol2 = np.zeros(fvol1.shape,dtype='complex64')
        for k in range(fvol2.shape[0]):
            fvol2[k,:,:] = np.fft.fft2(volume[k,:,:])

        fvol = fvol2

        sz,sy,sx = fvol.shape
        n = np.ceil(kernel_radius*2)
        XX,YY = np.meshgrid(np.arange(sx),np.arange(sy))
        XX = XX - sx/2.0
        YY = YY - sy/2.0
        d = np.sqrt(XX**2+YY**2)
        kernel = np.zeros(d.shape)
        kernel[np.where(d<=kernel_radius)] = 1.0
        kernel = kernel/np.sum(kernel)
        fkernel = np.fft.fft2(kernel)
        fout = fvol*fkernel
        out = np.abs(np.fft.ifft2(fout))
    return out


def get_z_sampling(lambda_1,lambda_2,n=1.38):
    return 1.0/((2*n)/lambda_1 - (2*n)/lambda_2)

def autotrim_volume(vol,depth):
```

```
    rad = depth/2
    prof = np.mean(np.mean(vol,axis=2),axis=0)
    max_idx = np.argmax(prof)
    z1 = max_idx - rad
    z2 = max_idx + rad
    if z2>len(prof):
        z2 = len(prof)
        z1 = z2-2*rad
    if z1<0:
        z1 = 0
        z2 = 2*rad
    return vol[:,z1:z2,:]


def autotrim_bscan(b):
    ab = np.abs(b)
    ab[-20:,:] = 0.0
    noise_rms = np.std(np.ravel(ab[:20,:]))
    ax_prof = np.mean(ab,axis=1)
    thresh = np.min(ax_prof)+3*noise_rms
    valid = np.where(ax_prof>thresh)[0]
    z1 = valid[0]
    z2 = valid[-1]
    return b[z1:z2,:],z1,z2

def translation(im0in, im1in, xlims=None, ylims=None, debug=False):
    """Return translation vector to register two images
    of equal size. Returns a 3-tuple (translation_x,translation_y,correlation)."""

    # if either image is blank, return 0, 0, 0.0 and stop
    if np.max(im0in)==np.min(im0in) or np.max(im1in)==np.min(im1in):
        return (0.0,0.0,0.0)

    im0 = (im0in-np.mean(im0in))/np.std(im0in)
    im1 = (im1in-np.mean(im1in))/np.std(im1in)

    # if the images are identical, return 0, 0, 1.0 and stop
    if np.array_equal(im0,im1):
        return (0.0,0.0,1.0)

    shape = im0.shape

    f0 = np.fft.fft2(im0)
    f1 = np.fft.fft2(im1)

    # original line:
    #ir = abs(np.fft.ifft2((f0 * f1.conjugate()) / (abs(f0) * abs(f1))))


    # break it down for checking:
```

```
    f1c = f1.conjugate()
    num = f0 * np.conj(f1)
    denom = abs(f0) * abs(f1)

    # to handle some stupid corner cases, esp. where test images are used:
    # put 1's into the denominator where both numerator and denominator are 0
    denom[np.where(np.logical_and(num==0,denom==0))] = 1.0
    frac = num/denom

    ir = np.abs(np.fft.ifft2(frac))

    goodness = np.max(ir)
    ty, tx = np.unravel_index(np.argmax(ir), shape)
    if debug:
        plt.subplot(3,2,1)
        plt.cla()
        plt.imshow(im0,interpolation='none',aspect='auto')
        plt.subplot(3,2,2)
        plt.cla()
        plt.imshow(im1,interpolation='none',aspect='auto')
        plt.subplot(3,2,3)
        plt.cla()
        plt.imshow(np.abs(f0),interpolation='none',aspect='auto')
        plt.subplot(3,2,4)
        plt.cla()
        plt.imshow(np.abs(f1),interpolation='none',aspect='auto')
        plt.subplot(3,2,5)
        plt.cla()
        plt.imshow(ir,interpolation='none',aspect='auto')
        plt.autoscale(False)
        plt.plot(tx,ty,'ws')
        plt.pause(.0001)
    if ty > shape[0] // 2:
        ty -= shape[0]
    if tx > shape[1] // 2:
        tx -= shape[1]
    return (tx, ty, goodness)

def translation1(vec0in, vec1in, xlims=None, equalize=False, debug=False):
    """Return translation vector to register two vectors
    of equal size. Returns a 2-tuple (translation, goodness).
    Translation is the amount to shift vec1in to align with
    vec0in."""
    if equalize:
        if len(vec0in)>len(vec1in):
            newvec1 = np.zeros(vec0in.shape)
            newvec1[:len(vec1in)] = vec1in
            vec1in = newvec1
        elif len(vec1in)>len(vec0in):
            newvec0 = np.zeros(vec1in.shape)
```

```
            newvec0[:len(vec0in)] = vec0in
            vec0in = newvec0

    # if either vector is blank, return 0, 0, 0.0 and stop
    if np.max(vec0in)==np.min(vec0in) or np.max(vec1in)==np.min(vec1in):
        return (0.0,0.0)

    vec0 = (vec0in-np.mean(vec0in))/np.std(vec0in)
    vec1 = (vec1in-np.mean(vec1in))/np.std(vec1in)

    # if the vectors are identical, return 0, 0, 1.0 and stop
    if np.array_equal(vec0,vec1):
        return (0.0,1.0)

    shape = len(vec0)

    f0 = np.fft.fft(vec0,axis=0)
    f1 = np.fft.fft(vec1,axis=0)

    # original line:
    # ir = abs(np.fft.ifft2((f0 * f1.conjugate()) / (abs(f0) * abs(f1))))

    f1c = f1.conjugate()
    num = f0*f1c
    denom = abs(f0) * abs(f1)
    denom[np.where(np.logical_and(num==0,denom==0))] = 1.0
    frac = num/denom
    ir = np.abs(np.fft.ifft(frac,axis=0))

    if xlims is not None:
        ir[xlims:-xlims] = 0.0

    goodness = np.max(ir)
    tx = np.argmax(ir)
    if debug:
        plt.subplot(1,2,1)
        plt.cla()
        plt.plot(vec0,'k-')
        plt.plot(vec1,'r--')
        plt.subplot(1,2,2)
        plt.cla()
        plt.plot(ir)
        plt.pause(.1)
    if tx > shape // 2:
        tx -= shape
    return (tx, goodness)

def find_peaks(prof,intensity_threshold=-np.inf,gradient_threshold=-np.inf):
    left = prof[:-2]
    center = prof[1:-1]
```

```
    right = prof[2:]
    peaks = np.where(np.logical_and(center>right,center>left))[0]+1
    peak_vals = prof[peaks]
    all_gradients = np.abs(np.diff(prof))
    l_gradients = all_gradients[:-1]
    r_gradients = all_gradients[1:]
    gradients = np.max([l_gradients,r_gradients],axis=0)
    peak_vals = np.array(peak_vals)
    gradient_vals = np.array(gradients[peaks-1])
    valid = np.where(np.logical_and(peak_vals>=intensity_threshold,gradient_vals>=gradient_threshold))[0]

    return peaks[valid]

def gaussian(x,x0,sigma):
    return np.exp(-(x-x0)**2/2.0/sigma**2)

def show_projections(vol):
    """Averages a volume in each dimension and shows
    resulting averages in 3 figures."""

    for axis in range(3):
        plt.figure()
        plt.imshow(np.mean(vol,axis=axis),interpolation='none')
        plt.set_cmap('gray')
        plt.colorbar()
    plt.show()

def odd(n):
    if n%1:
        sys.exit('utils.odd requires integer input, not %f.'%n)
    return n%2==1

def even(n):
    if n%1:
        sys.exit('utils.even requires integer input, not %f.'%n)
    return n%2==0

def power_spectrum(im):
    f = np.fft.fftshift(np.fft.fft2(im))
    return np.real(f)**2+np.imag(f)**2

def raps(im,N=1024,kind='linear'):
    imf = power_spectrum(im)
    sy,sx = imf.shape
    freqy,freqx = np.fft.fftfreq(sy),np.fft.fftfreq(sx)
    freqy,freqx = np.fft.fftshift(freqy),np.fft.fftshift(freqx)

    XX,YY = np.meshgrid(freqx,freqy)

    freqr = np.sqrt(XX**2+YY**2).ravel()
```

```
    imf = imf.ravel()

    sidx = np.argsort(freqr)
    freqr = freqr[sidx]
    imf = imf[sidx]

    freq_out = np.linspace(freqr[0],freqr[-1],N)
    interpolator = interp1d(freqr,imf,kind=kind)
    im_out = interpolator(freq_out)

    return im_out

def gaussian_convolve(im,sigma,mode='same'):

    kernel_width = np.ceil(sigma*8) # 4 standard deviations gets pretty close to zero
    vec = np.arange(kernel_width)-kernel_width/2.0
    XX,YY = np.meshgrid(vec,vec)

    g = np.exp(-(XX**2+YY**2)/2.0/sigma**2)
    return fftconvolve(im,g,mode=mode)/np.sum(g)


def map_rasters(target,reference,strip_width=1.0,reference_width=None,collapse_strip=False):

    sy,sx = target.shape
    sy2,sx2 = reference.shape

    assert sy==sy2 and sx==sx2

    reference = np.abs(reference)
    target = np.abs(target)



    # basic idea:
    # 1. window the reference in order to constrain the fit to portions of the reference near the target
    # 2. Window the target, usually with a very smal fwhm, in order to find a specific match in the reference
    # 3. average the target and match the two
    # 4. shift the window position on the target and go to step 2
    # for the time being let's keep things simple by not offering any initial upsampling options, and istead
    # upsample (or interpolate) the resulting traces by smoothing

    for ix in range(sx):
        x = np.arange(sx)-float(ix)
        if not reference_width is None:
            g = np.exp((-x**2)/(2*float(reference_width)**2))
            g = g/np.max(g) # normalize by maximum value so that the most likely match has amplitude equal to strip
            temp_ref = reference*g
        else:
            temp_ref = reference
```

```
        temp_ref = (temp_ref - np.mean(temp_ref))/np.std(temp_ref)
        target = (target - np.mean(target))/np.std(target)

        if collapse_strip:
            # speed things up by doing a 2x1 cross-correlation
            # better option is below, 2d correlation between two windowed images

            f1 = np.fft.fft(temp_ref,axis=0)
            f1c = f1.conjugate()

            g = np.exp((-x**2)/(2*float(strip_width)**2))
            g = g/np.sum(g) # normalize so that the sum of the windowed target has the same amplitude as the whole tar
get
            line = np.sum(target*g,axis=1)

            line = (line - np.mean(line))/np.std(line)

            f0 = np.fft.fft(line)
            num = (f0*f1c.T).T
            denom = (np.abs(f0)*np.abs(f1).T).T

            denom[np.where(np.logical_and(num==0,denom==0))] = 1.0
            frac = num/denom
            ir = np.abs(np.fft.ifft(frac,axis=0))

            goodness = np.max(ir)
            peakcoords = np.where(ir==goodness)
            peaky = peakcoords[0][0]
            peakx = peakcoords[1][0]

            if peaky > sy // 2:
                peaky -= sy

            peakx = ix-peakx

        else:
            f1 = np.fft.fft2(temp_ref)
            f1c = f1.conjugate()

            g = np.exp((-x**2)/(2*float(strip_width)**2))/np.sqrt(2*strip_width**2*np.pi)
            #g = g/np.sum(g) # normalize so that the sum of the windowed target has the same amplitude as the whole ta
rget
            temp_tar = target*g

            f0 = np.fft.fft2(temp_tar)
            num = f0*f1c
            denom = np.abs(f0)*np.abs(f1)

            denom[np.where(np.logical_and(num==0,denom==0))] = 1.0
```

```
            frac = num/denom
            ir = np.abs(np.fft.ifft2(frac))

            goodness = np.max(ir)
            peakcoords = np.where(ir==goodness)
            peaky = peakcoords[0][0]
            peakx = peakcoords[1][0]

            if peaky > sy // 2:
                peaky -= sy

            if peakx > sx // 2:
                peakx -= sx



        return peaky,peakx,goodness


        print peaky,peakx,goodness
        y_peaks.append(peaky)
        x_peaks.append(peakx)
        goodnesses.append(goodness)




        # sanity check to make sure things are scaled correctly
        # single_line = target[:,ix]
        # plt.plot(line)
        # plt.plot(single_line)
        # plt.show()

        # def print_stats(arr):
        #     print np.min(arr),np.mean(arr),np.max(arr)







def dewarp(im,shifts=None,upsample_factor=5,shift_limit=5):

    sy,sx = im.shape

    if shifts is None:
        shifts = [0]
```

```
        for y in range(1,sy):
            ref = im[y-1,:]
            tar = im[y,:]
            fref = np.fft.fft(ref,n=int(round(sx*upsample_factor)))
            ftar = np.fft.fft(tar,n=int(round(sx*upsample_factor)))
            xc = np.abs(np.fft.fftshift(np.fft.ifft(fref*np.conj(ftar))))
            shift = np.argmax(xc)-sx/2*upsample_factor
            if len(ref)%2==1:
                shift = shift - 1
            if np.abs(shift)>shift_limit:
                shift = 0.0
            shifts.append(shift)

        shifts = np.cumsum(shifts)
        shifts = shifts-np.min(shifts)

    out_sx = sx + np.max(shifts)
    out_sy = sy

    out = np.ones((out_sy,out_sx))*np.mean(im)

    x_in_0 = np.arange(sx)
    for y in range(sy):
        x_in = x_in_0 + shifts[y]
        y_in = im[y,:]
        x_out = np.arange(out_sx)
        interpolator = interp1d(x_in,y_in,bounds_error=False,fill_value=0.0)
        y_out = interpolator(x_out)
        out[y,:] = y_out

    return out,shifts


def gaussian_projection(vol,depth,sigma):
    zprof = np.mean(np.mean(vol,axis=1),axis=0)
    g = gaussian(np.arange(len(zprof)),x0=depth,sigma=sigma)
    return np.mean(vol*g,axis=2)


def nxcorr2(im1,im2,plot=False):
    sy1,sx1 = im1.shape
    sy2,sx2 = im2.shape

    im1 = (im1 - np.mean(im1))/np.std(im1)
    im2 = (im2 - np.mean(im2))/np.std(im2)

    sy = max(sy1,sy2)
    sx = max(sx1,sx2)

    bigy = sy1+sy2-1
```

```
    bigx = sx1+sx2-1
    temp1 = np.zeros((bigy,bigx))
    temp2 = np.zeros((bigy,bigx))

    temp1[:sy1,:sx1] = im1
    temp2[:sy2,:sx2] = im2

    nxcval = np.real(np.fft.fftshift(np.fft.ifft2(np.fft.fft2(temp1)*np.conj(np.fft.fft2(temp2)))))

    if plot:
        plt.subplot(3,2,2)
        plt.cla()
        plt.imshow(im1)
        plt.subplot(3,2,4)
        plt.cla()
        plt.imshow(im2)
        plt.subplot(3,2,5)
        plt.cla()
        plt.imshow(nxcval)
        plt.subplot(3,2,6)
        plt.cla()
        maxprof = nxcval.max(axis=0)
        plt.plot(maxprof)
        maxidx = np.argmax(maxprof)
        plt.plot(maxidx,maxprof[maxidx],'ks')
        plt.show()()

    peakVal = np.max(nxcval)
    peakIdx = np.where(nxcval==peakVal)
    yoff,xoff = peakIdx[0][0],peakIdx[1][0]

    if nxcval.shape[0]%2:
        yshift = (nxcval.shape[0]-1)/2.0 - yoff
    else:
        yshift = nxcval.shape[0]/2.0 - yoff

    if nxcval.shape[1]%2:
        xshift = (nxcval.shape[1]-1)/2.0 - xoff
    else:
        xshift = nxcval.shape[1]/2.0 - xoff

    peakVal = peakVal/float(sy)/float(sx)
    return yshift,xshift,peakVal,nxcval

def nxcorr2same(im1,im2,plot=False,ymax=None,xmax=None):
    # ymax and xmax are the absolute values of allowable shift in y and x directions
    sy1,sx1 = im1.shape
    sy2,sx2 = im2.shape

    if sy1!=sy2 or sx1!=sx2:
```

```
        sys.exit('nxcorr2same requires im1,im2 to have same shape')

    im1 = (im1 - np.mean(im1))/np.std(im1)
    im2 = (im2 - np.mean(im2))/np.std(im2)

    nxcval = np.real(np.fft.fftshift(np.fft.ifft2(np.fft.fft2(im1)*np.conj(np.fft.fft2(im2)))))

    if ymax is not None and xmax is not None:
        newnxcval = np.zeros(nxcval.shape)
        newnxcval[sy2/2-ymax:sy2/2+ymax,sx2/2-xmax:sx2/2+xmax] = nxcval[sy2/2-ymax:sy2/2+ymax,sx2/2-xmax:sx2/2+xmax]
        nxcval = newnxcval
        del newnxcval


    if plot:
        plt.subplot(3,2,2)
        plt.cla()
        plt.imshow(im1)
        plt.subplot(3,2,4)
        plt.cla()
        plt.imshow(im2)
        plt.subplot(3,2,5)
        plt.cla()
        plt.imshow(nxcval)
        plt.subplot(3,2,6)
        plt.cla()
        maxprof = nxcval.max(axis=0)
        plt.plot(maxprof)
        maxidx = np.argmax(maxprof)
        plt.plot(maxidx,maxprof[maxidx],'ks')
        plt.draw()

    peakVal = np.max(nxcval)
    peakIdx = np.where(nxcval==peakVal)
    yoff,xoff = peakIdx[0][0],peakIdx[1][0]

    if nxcval.shape[0]%2:
        yshift = (nxcval.shape[0]-1)/2.0 - yoff
    else:
        yshift = nxcval.shape[0]/2.0 - yoff

    if nxcval.shape[1]%2:
        xshift = (nxcval.shape[1]-1)/2.0 - xoff
    else:
        xshift = nxcval.shape[1]/2.0 - xoff

    peakVal = peakVal/float(sy1)/float(sx1)
    return yshift,xshift,peakVal,nxcval

#### Below are functions cut and pasted from old utils; make sure to check
```

```
def now():
    return datetime.datetime.now()

def getFwhm(x,fx):

    try:
        fxrange = fx.max() - fx.min()
        hm = fxrange/2.0 + fx.min()

        highidx = np.where(fx>hm)[0]


        # need to find xhm such that f(xhm)=hm / rising edge
        idx1 = highidx[0]-1
        idx2 = highidx[0]


        x1 = x[idx1]
        x2 = x[idx2]
        y1 = fx[idx1]
        y2 = fx[idx2]

        m = (float(y2)-float(y1))/(float(x2)-float(x1))
        b = y1 - m*x1

        xhm1 = (hm-b)/m

        # need to find xhm such that f(xhm)=hm / falling edge

        idx1 = highidx[-1]
        idx2 = highidx[-1]+1

        x1 = x[idx1]
        x2 = x[idx2]
        y1 = fx[idx1]
        y2 = fx[idx2]

        m = (float(y2)-float(y1))/(float(x2)-float(x1))
        b = y1 - m*x1
        xhm2 = (hm-b)/m

        return xhm2 - xhm1
    except Exception as e:
        print e
        print 'getFwhm: function may be ill-formed'
        plt.figure()
        plt.plot(x,fx)
        plt.show()
        sys.exit()
```

```
def normal(x,sigma=1.0,x0=0,normalize=True):
    sigma = np.float(sigma)
    x = x.astype(np.float64)
    x0 = np.float(x0)
    #return 1/(sigma*np.sqrt(2*np.pi)) * exp(-(x-x0)**2/(2*sigma**2))
    g = np.exp(-(x-x0)**2/(2*sigma**2))/(sigma*np.sqrt(2*np.pi))
    if normalize:
        g = g/g.sum()
    return g

def cdf(x):
    return (1 + erf(x/sqrt(2))) / 2

def skewNormal(x,sigma=1.0,x0=0,normalize=True,a=0):

    t = (x-x0)/sigma
    cdf = (1.0 + erf((a*t)/np.sqrt(2.0))) / 2.0

    sn = normal(x,sigma,x0) * cdf
    if normalize:
        sn = sn/sn.sum()
    return sn

# x = np.linspace(-100,100,1024)
# snx = skewNormal(x,sigma=13,a=5)
# plt.plot(x,snx)
# plt.show()
# sys.exit()



def findPeaks2d(im,axis=2,vSlopeThreshold=0,hSlopeThreshold=0):

    imdv = np.diff(im,axis=0)

    top = imdv[:-1,:]
    bottom = imdv[1:,:]


    tbDirectionMask = np.zeros(top.shape)
    tbDirectionMask[np.where(top>bottom)] = 1


    imdh = np.diff(im,axis=1)
    left = imdh[:,:-1]
    right = imdh[:,1:]
    lrDirectionMask = np.zeros(left.shape)
    lrDirectionMask[np.where(left>right)] = 1
```

```
    htemp = left*right*(-1)
    htemp[np.where(htemp<hSlopeThreshold**2)] = 0
    vtemp = top*bottom*(-1)
    vtemp[np.where(vtemp<vSlopeThreshold**2)] = 0


    hmap = htemp.clip(0,1)*lrDirectionMask
    vmap = vtemp.clip(0,1)*tbDirectionMask
    hmapfull = np.zeros(im.shape)
    vmapfull = np.zeros(im.shape)
    hmapfull[:,1:-1] = hmap
    vmapfull[1:-1,:] = vmap
    hmap = hmapfull
    vmap = vmapfull
    cmap = hmap*vmap
    summap = hmap+vmap

    twos = np.where(vmap)

    # plt.figure()
    # plt.imshow(im)
    # plt.plot(twos[1],twos[0],'b.')
    # plt.axis('tight')

    if axis==3:
        return summap
    elif axis==2:
        return cmap
    elif axis==1:
        return hmap
    elif axis==0:
        return vmap
    else:
        sys.exit('findPeaks2d: bad value for axis parameter; use 0, 1, or 2.')


def nxcorr1(vec1,vec2,doPlots=False):
    '''Returns shift,xc:
    shift is the number of pixels that vec2 must be
      shifted in order to align with vec1
    xc is the cross correlation peak'''

    l1 = len(vec1)
    l2 = len(vec2)

    vec1 = (vec1 - np.mean(vec1))/np.std(vec1)
    vec2 = (vec2 - np.mean(vec2))/np.std(vec2)

    temp1 = np.zeros([l1+l2-1])
    temp2 = np.zeros([l1+l2-1])
```

```
    temp1[:l1] = vec1
    temp2[:l2] = vec2

    nxcval = np.real(fftshift(ifft(fft(temp1)*np.conj(fft(temp2)))))

    peakVal = np.max(nxcval)
    peakIdx = np.where(nxcval==peakVal)[0][0]

    if len(nxcval)%2:
        shift = (len(nxcval)-1)/2.0 - peakIdx
    else:
        shift = len(nxcval)/2.0 - peakIdx



    if doPlots:
        plt.figure()
        plt.subplot(3,1,1)
        plt.plot(vec1)
        plt.subplot(3,1,2)
        plt.plot(vec2)
        plt.subplot(3,1,3)
        plt.plot(nxcval)
        plt.show()

    return shift,peakVal/len(vec1),nxcval


def RPS(im,ds=1.0):
    sy,sx = im.shape
    edge = max(sy,sx)
    fim = np.fft.fft2(im,s=(edge,edge))
    fim = np.fft.fftshift(fim)
    ps = np.abs(fim)

    fedge = float(edge)
    if edge%2:
        dcCoordinate = (fedge-1)/2
    else:
        dcCoordinate = fedge/2

    vec = np.arange(fedge)-dcCoordinate
    xx,yy = np.meshgrid(vec,vec)
    d = np.sqrt(xx**2.0+yy**2.0)

    mask = np.zeros(d.shape)
    mask[np.where(d<=np.floor(fedge/2.0))] = 1

    ps = ps * mask
```

```
    freq = np.fft.fftshift(np.fft.fftfreq(int(fedge),ds))


    freq = freq[dcCoordinate:]
    x = np.arange(dcCoordinate,fedge) - dcCoordinate

    xp = d.ravel()
    yp = ps.ravel()

    valid = np.where(xp<=len(freq))[0]
    xp = xp[valid]
    yp = yp[valid]

    y = np.interp(x,xp,yp)
    return y,freq



def crossCorrelate(im1,im2,axis=-1,normalize=False):
    try:
        sy1 = im1.shape[0]
    except:
        sy1 = 1
    try:
        sx1 = im1.shape[1]
    except:
        sx1 = 1
    try:
        sy2 = im2.shape[0]
    except:
        sy2 = 1
    try:
        sx2 = im2.shape[1]
    except:
        sx2 = 1


    im1 = (im1 - np.mean(im1))/np.std(im1)
    im2 = (im2 - np.mean(im2))/np.std(im2)

    if axis==-1:
        temp1 = np.zeros([sy1+sy2-1,sx1+sx2-1])
        temp2 = np.zeros([sy1+sy2-1,sx1+sx2-1])
        temp1[0:sy1,0:sx1] = im1
        temp2[0:sy2,0:sx2] = im2

        nxcval = np.real(fftshift(ifft2(fft2(temp1)*np.conj(fft2(temp2)))))
        if normalize:
            nxcval = nxcval/(sx1*sy1)
    elif axis==0:
```

```
        if not sx1==sx2:
            sys.exit('For axis 0 nxc, im1.shape[0] must equal im2.shape[0]')
        else:
            temp1 = np.zeros([sy1+sy2-1,sx1])
            temp2 = np.zeros([sy1+sy2-1,sx1])
            temp1[0:sy1,0:sx1] = im1
            temp2[0:sy2,0:sx2] = im2
            nxcval = np.real(fftshift(ifft(fft(temp1,axis=0)*np.conj(fft(temp2,axis=0)),axis=0),axes=(0,)))
            if normalize:
                nxcval = nxcval/np.max(nxcval)
    elif axis==1:
        if not sy1==sy2:
            sys.exit('For axis 1 nxc, im1.shape[1] must equal im2.shape[1]')
        else:
            temp1 = np.zeros([sy1,sx1+sx2-1])
            temp2 = np.zeros([sy1,sx1+sx2-1])
            temp1[0:sy1,0:sx1] = im1
            temp2[0:sy2,0:sx2] = im2
            nxcval = np.real(fftshift(ifft(fft(temp1,axis=1)*np.conj(fft(temp2,axis=1)),axis=1),axes=(1,)))
            if normalize:
                nxcval = nxcval/np.max(nxcval)

    return nxcval


def vectorCrossCorrelate(vec1,vec2,axis=-1,normalize=False):
    try:
        sy1 = vec1.shape[0]
    except:
        sy1 = 1
    try:
        sx1 = vec1.shape[1]
    except:
        sx1 = 1
    try:
        sy2 = vec2.shape[0]
    except:
        sy2 = 1
    try:
        sx2 = vec2.shape[1]
    except:
        sx2 = 1


    vec1 = (vec1 - np.mean(vec1))/np.std(vec1)
    vec2 = (vec2 - np.mean(vec2))/np.std(vec2)

    if axis==-1:
        temp1 = np.zeros([sy1+sy2-1,sx1+sx2-1])
        temp2 = np.zeros([sy1+sy2-1,sx1+sx2-1])
```

```
        temp1[0:sy1,0:sx1] = vec1
        temp2[0:sy2,0:sx2] = vec2

        nxcval = np.real(fftshift(ifft2(fft2(temp1)*np.conj(fft2(temp2)))))
        if normalize:
            nxcval = nxcval/np.max(nxcval)
    elif axis==0:
        if not sx1==sx2:
            sys.exit('For axis 0 nxc, vec1.shape[0] must equal vec2.shape[0]')
        else:
            temp1 = np.zeros([sy1+sy2-1,sx1])
            temp2 = np.zeros([sy1+sy2-1,sx1])
            temp1[0:sy1,0:sx1] = vec1
            temp2[0:sy2,0:sx2] = vec2
            nxcval = np.real(fftshift(ifft(fft(temp1,axis=0)*np.conj(fft(temp2,axis=0)),axis=0),axes=(0,)))
            if normalize:
                nxcval = nxcval/np.max(nxcval)
    elif axis==1:
        if not sy1==sy2:
            sys.exit('For axis 1 nxc, vec1.shape[1] must equal vec2.shape[1]')
        else:
            temp1 = np.zeros([sy1,sx1+sx2-1])
            temp2 = np.zeros([sy1,sx1+sx2-1])
            temp1[0:sy1,0:sx1] = vec1
            temp2[0:sy2,0:sx2] = vec2
            nxcval = np.real(fftshift(ifft(fft(temp1,axis=1)*np.conj(fft(temp2,axis=1)),axis=1),axes=(1,)))
            if normalize:
                nxcval = nxcval/np.max(nxcval)

    return nxcval


def chooseFile(path,filt='*.*',selection=None):
    import glob
    flist = glob.glob(path+filt)
    print flist
    err = True
    while err:
        idx = 0
        for f in flist:
            print '[%d]: %s'%(idx,f)
            idx = idx + 1
        print '[%s]: quit'%'q'
        if selection==None:
            choice = raw_input('Please choose one of the files above: ')
            if choice.lower()=='q':
                sys.exit('User quit')
            else:
                try:
                    out = flist[int(choice)]
```

```
                return out
            except IndexError:
                print 'Choice out of range.'
            except ValueError:
                print 'Invalid choice.'

        else:
            print 'File %d selected in function call'%selection
            try:
                out = flist[selection]
                return out
            except IndexError:
                print 'Selection out of range.'
                sys.exit()
            except ValueError:
                print 'Invalid selection.'
                sys.exit()

def sgsmooth(y, window_size, order, deriv=0, rate=1):
    r"""Smooth (and optionally differentiate) data with a Savitzky-Golay filter.
    The Savitzky-Golay filter removes high frequency noise from data.
    It has the advantage of preserving the original shape and
    features of the signal better than other types of filtering
    approaches, such as moving averages techniques.
    Parameters
    ----------
    y : array_like, shape (N,)
        the values of the time history of the signal.
    window_size : int
        the length of the window. Must be an odd integer number.
    order : int
        the order of the polynomial used in the filtering.
        Must be less then 'window_size' - 1.
    deriv: int
        the order of the derivative to compute (default = 0 means only smoothing)
    Returns
    -------
    ys : ndarray, shape (N)
        the smoothed signal (or it's n-th derivative).
    Notes
    -----
    The Savitzky-Golay is a type of low-pass filter, particularly
    suited for smoothing noisy data. The main idea behind this
    approach is to make for each point a least-square fit with a
    polynomial of high order over a odd-sized window centered at
    the point.
    Examples
    --------
    t = np.linspace(-4, 4, 500)
    y = np.exp( -t**2 ) + np.random.normal(0, 0.05, t.shape)
```

```
    ysg = savitzky_golay(y, window_size=31, order=4)
    import matplotlib.pyplot as plt
    plt.plot(t, y, label='Noisy signal')
    plt.plot(t, np.exp(-t**2), 'k', lw=1.5, label='Original signal')
    plt.plot(t, ysg, 'r', label='Filtered signal')
    plt.legend()
    plt.show()
    References
    ----------
    .. [1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of
       Data by Simplified Least Squares Procedures. Analytical
       Chemistry, 1964, 36 (8), pp 1627-1639.
    .. [2] Numerical Recipes 3rd Edition: The Art of Scientific Computing
       W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
       Cambridge University Press ISBN-13: 9780521880688
    """
    import numpy as np
    from math import factorial

    try:
        window_size = np.abs(np.int(window_size))
        order = np.abs(np.int(order))
    except ValueError, msg:
        raise ValueError("window_size and order have to be of type int")
    if window_size % 2 != 1 or window_size < 1:
        raise TypeError("window_size size must be a positive odd number")
    if window_size < order + 2:
        raise TypeError("window_size is too small for the polynomials order")
    order_range = range(order+1)
    half_window = (window_size -1) // 2
    # precompute coefficients
    b = np.mat([[k**i for i in order_range] for k in range(-half_window, half_window+1)])
    m = np.linalg.pinv(b).A[deriv] * rate**deriv * factorial(deriv)
    # pad the signal at the extremes with
    # values taken from the signal itself
    firstvals = y[0] - np.abs( y[1:half_window+1][::-1] - y[0] )
    lastvals = y[-1] + np.abs(y[-half_window-1:-1][::-1] - y[-1])
    y = np.concatenate((firstvals, y, lastvals))
    return np.convolve( m[::-1], y, mode='valid')


class LRPModel:

    def __init__(self,lrp,microns_per_pixel=1.85):
        self.lrp = lrp
        self.labels = {}
        self.labels_allowed = ['ILM','NFL','IPL','INL','OPL','ONL','ELM','CISCOS','COST','RISROS','ROST','RPE','BM','C
H']
        self.microns_per_pixel = microns_per_pixel
        self.z_um = np.arange(len(lrp))*self.microns_per_pixel
```

```
    def write_to_h5(self,h5):
        try:
            del h5['/lrp_model']
        except Exception as e:
            print e
        h5.create_dataset('/lrp_model/lrp',data=self.lrp)
        h5.create_dataset('/lrp_model/z_um',data=self.z_um)
        h5.create_dataset('/lrp_model/microns_per_pixel',data=self.microns_per_pixel)
        for k in self.labels.keys():
            label = self.labels[k]
            h5.create_dataset('/lrp_model/%s'%label,data=k)


    def auto_label(self):
        peaks = find_peaks(self.lrp)
        ordered_layers = ['ILM','IPL','OPL','ELM','ISOS','COST','RPE']
        normalized_brightness = {'ISOS':1.0,'COST':0.6,'RPE':0.8,'OPL':0.1,'ILM':0.1,'IPL':0.9}
        print peaks


    def add_labels(self,auto_peak=True):
        peaks = find_peaks(self.lrp)
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(self.lrp)

        def onclick(event):
            print 'button=%d, x=%d, y=%d, xdata=%f, ydata=%f'%(event.button, event.x, event.y, event.xdata, event.ydat
a)
            x = int(round(event.xdata))
            if auto_peak:
                x = peaks[np.argmin(np.abs(x-peaks))]
            y = self.lrp[x]
            self.plot(ax)
            my_label = raw_input('Label for point at %d,%d? (Return to skip.) '%(x,y))
            if len(my_label):
                if my_label in self.labels_allowed:
                    self.labels[x] = my_label
                    ax.cla()
                    self.plot(ax)
                    plt.draw()
            else:
                plt.close()

        cid = fig.canvas.mpl_connect('button_press_event', onclick)
        plt.show()
```

```
    def plot(self,ax):
        ax.plot(self.lrp)
        for key in self.labels.keys():
            x,y = key,self.lrp[key]
            ax.plot(x,y,'gs')
            ax.text(x,y,self.labels[key],ha='center',va='bottom')


class Centroider:

    def __init__(self,sy,sx):
        self.xx,self.yy = np.meshgrid(np.arange(sx),np.arange(sy))

    def get(self,im):
        denom = np.sum(im)
        return np.sum(im*self.xx)/denom,np.sum(im*self.yy)/denom


def strel(kind='disk',diameter=15):
    if kind=='disk':
        xx,yy = np.meshgrid(np.arange(diameter),np.arange(diameter))
        xx = xx - float(diameter-1)/2.0
        yy = yy - float(diameter-1)/2.0
        d = np.sqrt(xx**2+yy**2)
        out = np.zeros(xx.shape)
        out[np.where(d<=diameter/2.0)] = 1.0
        return out

def background_subtract(im,strel=strel()):
    if len(im.shape)==2:
        bg = morphology.grey_opening(im,structure=strel)
    elif len(im.shape)==3:
        bg = np.zeros(im.shape)
        for k in range(3):
            bg[:,:,k] = morphology.grey_opening(im[:,:,k],structure=strel)
    return im-bg

def gaussian_blur(im,sigma=1.0,kernel_size=11):
    xx,yy = np.meshgrid(np.arange(kernel_size),np.arange(kernel_size))
    xx = xx.astype(np.float) - (kernel_size-1)/2
    yy = yy.astype(np.float) - (kernel_size-1)/2
    rad = xx**2+yy**2
    g = np.exp(-rad/(2*sigma**2))
    gsum = np.sum(g)
    if len(im.shape)==2:
        data = signal.convolve2d(im,g,'same')/gsum
    elif len(im.shape)==3:
        data = np.zeros(im.shape)
        for idx in range(im.shape[2]):
            data[:,:,idx] = signal.convolve2d(im[:,:,idx],g,'same')/gsum
```

```
    return data

def bmpify(im):
    return np.clip(np.round(im),0,255).astype(np.uint8)

def cstretch(im):
    return (im - np.min(im))/(np.max(im)-np.min(im))

def high_contrast(r,g):
    out = np.zeros((r.shape[0],r.shape[1],3)).astype(np.uint8)
    out[:,:,0] = bmpify(cstretch(r)*255)
    out[:,:,1] = bmpify(cstretch(g)*255)
    return out

def centroid_objects(im,mask):
    cyvec,cxvec = [],[]
    sy,sx = im.shape
    c = Centroider(sy,sx)
    labeled,n_objects = ndimage.label(mask)
    for k in range(1,n_objects):
        tempmask = np.zeros(mask.shape)
        tempmask[np.where(labeled==k)] = 1.0
        tempim = im*tempmask
        cx,cy = c.get(tempim)
        cyvec.append(cy)
        cxvec.append(cx)
    return cxvec,cyvec

def poisson(k, lamb):
    return (lamb**k/factorial(k)) * np.exp(-lamb)

def get_poisson_threshold(g,frac=0.01,nbins=50,p0=4.0):
    normed_counts, bin_edges = np.histogram(g.ravel(),bins=nbins,range=[g.min()-.5,g.max()+.5],normed=True)
    bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])

    # poisson function, parameter lamb is the fit parameter
    parameters, cov_matrix = curve_fit(poisson, bin_centers, normed_counts,p0=[p0])
    x_plot = np.linspace(np.min(bin_centers), np.max(bin_centers), 1000)
    y_plot = poisson(x_plot,*parameters)
    threshold = round(x_plot[np.where(y_plot>frac)[0][-1]])
    return threshold

def threshold(g,sigma,frac,nbins,erosion_diameter):
    g = background_subtract(g,strel())
    g = gaussian_blur(g,sigma)
    g = np.round(g)
    gthreshold = get_poisson_threshold(g,frac=frac,nbins=nbins)
    gt = g.copy()
    gt[np.where(g<gthreshold)] = 0.0
    gto = morphology.grey_erosion(gt,footprint=strel(diameter=erosion_diameter))
```

```
    return gto

def nxcorr(vec1,vec2,doPlots=False):
    '''Given two vectors TARGET and REFERENCE, nxcorr(TARGET,REFERENCE)
    will return a pair (tuple) of values, (SHIFT, CORR). CORR is a quantity
    corresponding to the Pearson correlation of the two vectors, accounting
    for a time delay between the two. Put slightly differently, CORR is the
    Pearson correlation of the best alignment of the two vectors.
    SHIFT gives the number of pixels of delay between the two, such that
    shifting TARGET by SHIFT pixels (rightward for positive, leftward for
    negative) will produce the optimal alignment of the vectors.'''

    l1 = len(vec1)
    l2 = len(vec2)

    vec1 = (vec1 - np.mean(vec1))/np.std(vec1)
    vec2 = (vec2 - np.mean(vec2))/np.std(vec2)

    temp1 = np.zeros([l1+l2-1])
    temp2 = np.zeros([l1+l2-1])

    temp1[:l1] = vec1
    temp2[:l2] = vec2

    nxcval = np.real(fftshift(ifft(fft(temp1)*np.conj(fft(temp2)))))

    peakVal = np.max(nxcval)
    peakIdx = np.where(nxcval==peakVal)[0][0]


    if False:
        if l1%2!=l2%2:
            shift = np.fix(peakIdx-len(nxcval)/2.0)
        else:
            shift = np.fix(peakIdx-len(nxcval)/2.0) + 1

    if len(nxcval)%2:
        shift = (len(nxcval)-1)/2.0 - peakIdx
    else:
        shift = len(nxcval)/2.0 - peakIdx



    if doPlots:
        plt.figure()
        plt.subplot(3,1,1)
        plt.plot(vec1)
        plt.subplot(3,1,2)
        plt.plot(vec2)
        plt.subplot(3,1,3)
```

```
        plt.plot(nxcval)
        plt.show()
        sys.exit()

    return shift,peakVal/len(vec1)

def findShift(vec1,vec2):
    shift,peakVal = nxcorr(vec1,vec2)
    return shift

def shear(im,order,oversample=1.0,sameshape=False,y1=None,y2=None):
    sy,sx = im.shape
    sy0 = sy
    newsy = int(np.round(float(sy)*float(oversample)))

    if y1 is None:
        cropY1 = 0
    else:
        cropY1 = y1

    if y2 is None:
        cropY2 = sy
    else:
        cropY2 = y2

    imOriginal = np.zeros(im.shape)
    imOriginal[:] = im[:]

    im = im[cropY1:cropY2,:]

    fim = fft(im,axis=0)
    fim = fftshift(fim,axes=0)
    im = np.abs(ifft(fim,axis=0,n=newsy))

    windowSize = 5

    refIdx = 0
    tarIdx = refIdx

    rx1 = refIdx
    rx2 = rx1 + windowSize

    tx1 = rx1
    tx2 = rx2

    xVector = []
    yVector = []

    ref = im[:,rx1:rx2]
    ref = np.mean(ref,axis=1)
```

```
    while tx2<sx:
        tar = im[:,tx1:tx2]
        tar = np.mean(tar,axis=1)
        shift = -findShift(ref,tar)
        xVector.append(tx1-rx1)
        yVector.append(shift)
        tx1 = tx1 + 1
        tx2 = tx2 + 1


    p = np.polyfit(xVector,yVector,order)
    newY = np.round(np.polyval(p,range(sx)))
    newY = newY - np.min(newY)


    newim = np.zeros([newsy+np.max(newY),sx])
    for ix in range(sx):
        newim[newY[ix]:newY[ix]+newsy,ix] = im[:,ix]

    newSum = np.sum(newim)

    osy,osx = newim.shape

    outsy = int(float(osy)/float(oversample))

    if oversample!=1.0:
        newim = imresize(newim,(outsy,sx),interp='bicubic')*oversample

    newim = newim/np.sum(newim)*newSum
    resampledSum = np.sum(newim)


    if sameshape:
        dy = newim.shape[0] - sy
        newim = newim[dy/2:dy/2+sy,:]

    return newim
```