class `droplet_image(image_sequence)`

A class that collects methods for analyzing droplet and double emulsion images (PIV, droplet tracking, etc.).

```
class droplet_image:
    """Container of functions related to confocal droplet images"""
    def __init__(self, image_sequence):
        """image_sequence: dataframe of image dir info, readdata return value"""
    def process_first_image(self, mask):
        """Compare the provided mask (template) with the first image.
        If the maximum correlation is at the center of the image, the input mask should be corr
        This function also records the measured center coordinates (xc, yc), for computing the
    def droplet_traj(self, mask, xy0):
        """Analyze droplet trajectory"""
    def get_cropped_image(self, index, traj, mask_shape):
        """Retrieve cropped image by index"""
    def get_image(self, index):
        """Retrieve image by index"""
    def check_traj(self, traj, mask_shape, n=10):
        """Check the droplet finding accuracy."""
    def get_image_name(self, index):
        """Retrieve image name."""
    def fixed_mask_piv(self, save_folder, winsize, overlap, dt, mask_dir):
        """Perform fixed mask PIV and save PIV results and parameters in save_folder"""
    def piv_overlay_fixed(self, piv_folder, out_folder, sparcity):
        """Draw PIV overlay for fixed mask PIV data (unify old code in class)"""
    def moving_mask_piv(self, save_folder, winsize, overlap, dt, mask_dir, xy0, mask_shape):
        """Perform moving mask PIV and save PIV results and parameters in save_folder"""
    def piv_overlay_moving(self, piv_folder, out_folder, traj, piv_params, sparcity=1):
        """Draw PIV overlay for moving mask piv data (only on cropped images)"""
```

This class puts together the PIV analysis functions, so that the PIV related scripts can be significantly simplified. Each script only needs to read the parameters from the command line, construct an object and call a method.

Example of script simplification

```
# old PIV overlay script
def determine_arrow_scale(u, v, sparcity):
    row, col = u.shape
    return max(np.nanmax(u), np.nanmax(v)) * col / sparcity / 1.5

if __name__=="__main__": # whether the following script will be executed when run this code
    pivDataFolder = sys.argv[1]
    imgFolder = sys.argv[2]
    output_folder = sys.argv[3]
    if len(sys.argv) == 5:
        sparcity = int(sys.argv[4])
    else:
        sparcity = 1
    if os.path.exists(output_folder) == False:
        os.makedirs(output_folder)
    with open(os.path.join(output_folder, 'log.txt'), 'w') as f:
        f.write('piv_folder: ' + pivDataFolder + '\n')
        f.write('img_folder: ' + imgFolder + '\n')
        f.write('output_folder: ' + output_folder + '\n')
        f.write('sparcity: ' + str(sparcity) + '\n')

    # pivDataDir = dirrec(pivDataFolder, '*.csv')

    l = readdata(pivDataFolder, "csv")

    # compute scale factor of quiver
    x, y, u, v = read_piv(l.Dir[0])
    row, col = x.shape
    scale = determine_arrow_scale(u, v, sparcity)

    for num, i in l.iterrows():
        # PIV data
        folder, pivname = os.path.split(i.Dir)
        x, y, u, v = read_piv(i.Dir)
        row, col = x.shape
        xs = x[0:row:sparcity, 0:col:sparcity]
        ys = y[0:row:sparcity, 0:col:sparcity]
        us = u[0:row:sparcity, 0:col:sparcity]
        vs = v[0:row:sparcity, 0:col:sparcity]
        # overlay image
        imgname = i.Name.split("-")[0]
        imgDir = os.path.join(imgFolder, imgname + '.tif')
        img = to8bit(io.imread(imgDir))
        # bp = bpass(img, 2, 100)
        # fig = plt.figure(figsize=(3, 3*row/col))
        dpi = 300
        figscale = 1
        w, h = img.shape[1] / dpi, img.shape[0] / dpi
        fig = Figure(figsize=(w*figscale, h*figscale)) # on some server `plt` is not supported
        canvas = FigureCanvas(fig) # necessary?
```

```
        ax = fig.add_axes([0, 0, 1, 1])
        ax.imshow(img, cmap='gray')
        ax.quiver(xs, ys, us, vs, color='yellow', width=0.003, \
                    scale=scale, scale_units='width') # it's better to set a fixed scale, see *
        ax.axis('off')
        # outfolder = folder.replace(pivDataFolder, output_folder) #
        # if os.path.exists(outfolder) == False:
        #     os.makedirs(outfolder)
        fig.savefig(os.path.join(output_folder, imgname + '.jpg'), dpi=dpi)
        with open(os.path.join(output_folder, 'log.txt'), 'a') as f:
            f.write(time.asctime() + ' // ' + imgname + ' calculated\n')
# 62 lines
```

```
# New PIV overlay script
if __name__=="__main__": # whether the following script will be executed when run this code
    piv_folder = sys.argv[1]
    image_folder = sys.argv[2]
    out_folder = sys.argv[3]
    sparcity = 1
    if len(sys.argv) > 4:
        sparcity = int(sys.argv[4])

    if os.path.exists(out_folder) == False:
        os.makedirs(out_folder)

    seq = readdata(image_folder, "tif")
    DI = droplet_image(seq)

    DI.piv_overlay_fixed(piv_folder, out_folder, sparcity=sparcity)
# 14 lines
```

The new code can still take different argument and plot PIV overlay for different data. It's just the details are all hidden in the class.

Modify code in the same place

With the new structure, the script files will barely need modification. If we want different behavior, for example, thicker arrows in the PIV overlay images, we can modify the `droplet_image` class. This is another benefit of this code structure: all the functions with related purposes are placed in the same place, so we don't have to jump from file to file when modifying a function.