Dataset Initial Pre-Processing

This notebook focusses on identifying the techniques that may enhance the images in the dataset

Author: Alexander Goudemond, Student Number: 219030365

Imports

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

import cv2
from PIL.Image import fromarray

from os import mkdir, getcwd, walk
from os.path import join

from IPython.display import Video
```

Summary of Previous Work

Several valuable pieces of information have emerged as a result of the 2nd (002) and 3rd (003) notebooks

From 002 and 003, we can reflect upon the following:

- There are 20 folders, making up the datasets. Together, these images take up ~7.5 GB (Mostly TIFFs)
- There are 10 different kinds of dataset sources, which are listed below:

BF-C2DL-HSC

BF-C2DL-MuSC

DIC-C2DH-HeLa

Fluo-C2DL-Huh7

Fluo-C2DL-MSC

Fluo-N2DH-GOWT1

Fluo-N2DH-SIM+

Fluo-N2DL-HeLa

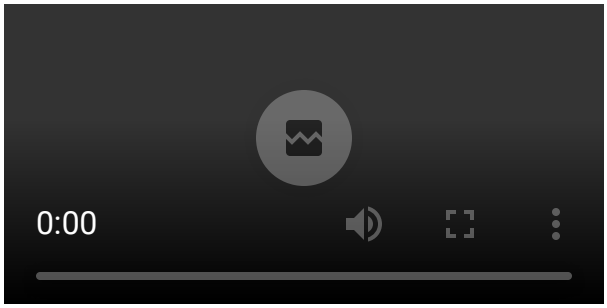
PhC-C2DH-U373

PhC-C2DL-PSC

- Each of those folders have a training set and a challenge set. The challenge set is denoted by a (1). The main distinction between the folders is that the training sets have manually segmented and manually tracked images for some of the images present. The challenge datasets ONLY have the images
- Windows Photo Viewer and OpenCV are unable to view the features contained in these manually segmented and tracked images. They also cannot show the Simulated datasets (Denoted by a plus symbol above)
- Matplotlib.pyplot is able to view all the information in the datasets, and we can use the built in colourmaps to convert the file into a format that Windows Photo Viewer and OpenCV can then use
- Several colourmaps were considered, and cross examined across the 10 datasets. The 3 recurring colourmaps that may be useful are: 'gray', 'Greys' and 'seismic'. The difference between 'gray' and 'Greys' appears to be the variation of whitelight, as 'Greys' appears whiter
- The author generated 4 videos to encapsulate the movement of the cells over time, which includes Colour, Grayscale, Greys and Seismic
- In 003, the author stitched a sample of the 4 videos together. 2 examples are shown below:

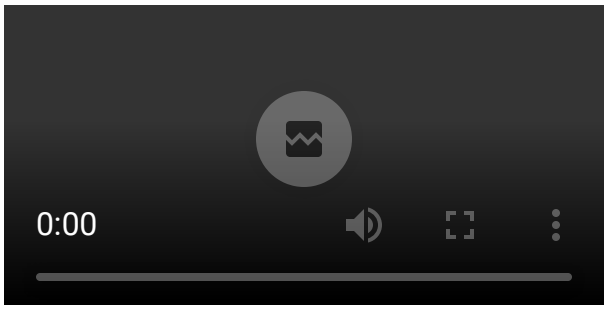
In [2]:
```
Video("003_ColourExploration\\Fluo-N2DH-SIM+_01.mp4")
```

Out[2]:



In [3]:
```
Video("003_ColourExploration\\Fluo-N2DL-HeLa_01.mp4")
```

Out[3]:



- The stitched videos are generated using moviepy, and have 10 frames a second. Thus, some videos stop moving after 5 seconds, whereas other have 10 seconds to display. The reason some videos are shorter is because the original video may be rather short, due to the variation in quantity of images
- Fluo-N2DL-HeLa_01.mp4 appears to be white for Colour and Grayscale, but this is probably due to using OpenCV to generate the initial videos. Future processing steps of each image will first save the images as 'gray' via Matplotlib.pyplot, then be read in via OpenCV

The video comparison reveals valuable information:

- Colour and Grayscale videos are virtually identical

- "Greys" brightens the images, which may saturate cells with low intensity values
- "seismic" creates a kind of heat map, which clearly shows feature movement over time (nucleus, organelles, etc.)
- All 4 videos demonstrate a pulsating light effect, which is most noticeable in the 'seismic' dataset. But, all display it upon close inspection
- The seismic video is fantastic for identifying cells 'hiding' in the background, with low intensity values

All 10 datasets need to be processed to ensure:

- Hidden data is present
- Noise is absent
- Cell Boundaries are clearly recognized

From the synopsis above, we can use this notebook to focus on techniques to process the images!

Test Images generation

Here, we will generate the locations for 10 test images, which will be used to identify valuable processing steps

In [4]:
```
# create directory for work we create
def tryMakeDirectory(current_directory, destination_directory):
    try:
        # join comes from os.path
        mkdir( join(current_directory, destination_directory) )
    except FileExistsError:
        print("Folder already exists!")
        pass
    except:
        print("Unknown Error Encountered...")
```

In [5]:
```
'''
We only need to show every _OTHER_ folder, as each dataset has a
 training and challenge set. So out of 20 files, we need to show 10

First things first, let us create an array of the directory locations
'''

data_sets = "..\\..\\Comp700_DataSets"
current_directory = getcwd()

path = walk(current_directory + "\\" + data_sets)

directory_array = [] # contains the main folders

i = 1
for root, dirs, files in path:
    if (i == 2):
        directory_array = dirs
        break

    i += 1
```

```
print("Directory Array")
print(directory_array)
```

```
Directory Array
['BF-C2DL-HSC', 'BF-C2DL-HSC (1)', 'BF-C2DL-MuSC', 'BF-C2DL-MuSC (1)', 'DIC-C2DH-HeLa',
'DIC-C2DH-HeLa (1)', 'Fluo-C2DL-Huh7', 'Fluo-C2DL-Huh7 (1)', 'Fluo-C2DL-MSC', 'Fluo-C2DL
-MSC (1)', 'Fluo-N2DH-GOWT1', 'Fluo-N2DH-GOWT1 (1)', 'Fluo-N2DH-SIM+', 'Fluo-N2DH-SIM+
(1)', 'Fluo-N2DL-HeLa', 'Fluo-N2DL-HeLa (1)', 'PhC-C2DH-U373', 'PhC-C2DH-U373 (1)', 'PhC
-C2DL-PSC', 'PhC-C2DL-PSC (1)']
```

In [6]:
```
# Now, generate the array of images
test_images = []

i = -1
temp = -1
for root, dirs, files in path:
    # print(dirs)
    for item in files:
        # only execute for first picture in directory
        if ("t0000.tif" == item) or ("t000.tif" == item):
            i += 1

            # skips folder "02" in datasets
            if (i % 2 == 1):
                break

            # print(i)
            temp = i // 2

            # skip Challenge datasets
            if ("(1)" in directory_array[temp]):
                break

            location = ( current_directory + "\\" + data_sets + "\\Extracted\\" + direct
                        "\\" + directory_array[temp] + "\\01\\" + item)
            # print(location)

            img = plt.imread(location)

            test_images.append(img) # place into array

            break

        else:
            break
```

In [7]:
```
# generate labels for test_images
label_array = []

for i in range(20):
    if (i % 2 == 0):
        label_array.append(directory_array[i])

print("\nLabel Array")
print(label_array)
```

```
Label Array
['BF-C2DL-HSC', 'BF-C2DL-MuSC', 'DIC-C2DH-HeLa', 'Fluo-C2DL-Huh7', 'Fluo-C2DL-MSC', 'Flu
o-N2DH-GOWT1', 'Fluo-N2DH-SIM+', 'Fluo-N2DL-HeLa', 'PhC-C2DH-U373', 'PhC-C2DL-PSC']
```

Now, save the images to InitialPreProcessing. Thankfully, Matplotlib.pylot can save them as Tiffs, so our concern in the first notebook is addressed!

In [8]:
```
# save the 10 images to our desired folder
desired_directory = "004_InitialPreProcessing"
```

```
tryMakeDirectory(current_directory, desired_directory)

i = 0
for pic in test_images:
    location = current_directory + "\\" + desired_directory + "\\" + label_array[i] + ".
    plt.imsave(location, pic, cmap="gray")
    i += 1
```

```
Folder already exists!
```

We can read these images in through OpenCV, for our processing needs.

Numpy - Read in Images

Here, we read in the images in our local directory and store them in test_images

Let us try stitch the 10 images together, to show all 10 videos to the user

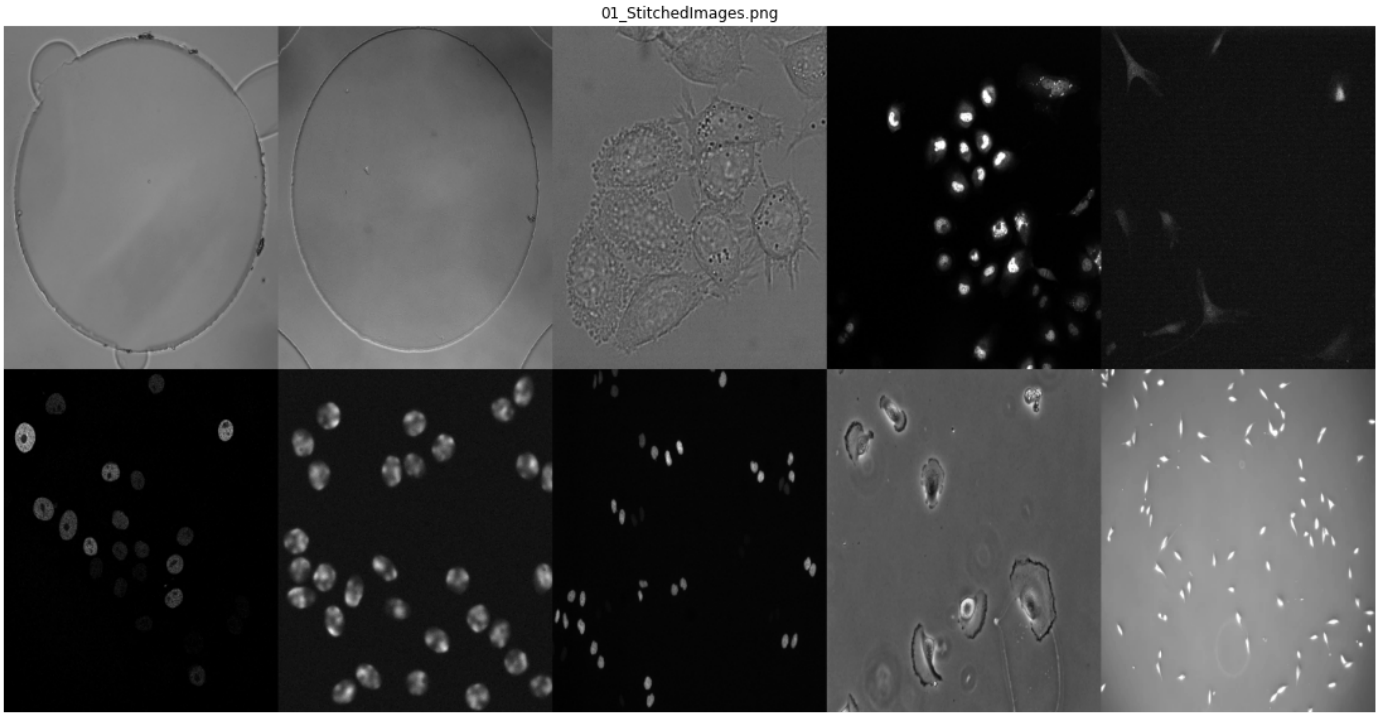
In [9]:
```
path = walk(desired_directory)

test_images = [] # reset variable
test_images_names = []

for root, dirs, files in path:
    for pic_name in files:
        if (".tiff" in pic_name):
            # print(pic_name)
            test_images_names.append(pic_name)
# print(test_images_names)

# read in images in 1 go
for i in range(len(test_images_names)):
    if (i == 0):
        (x, y) = img.shape

    img = cv2.imread(desired_directory + "\\" + test_images_names[i], cv2.IMREAD_GRAYSCA
    img_reshaped = cv2.resize(img, (x // 2, y // 2))
    test_images.append(img_reshaped)
```

Now, let us stitch them together!

In [10]:
```
# from PIL.Image import fromarray

def stitchTogetherPics(array_of_images):
    # top level
    myList = (array_of_images[0], array_of_images[1], array_of_images[2], array_of_image
    numpy_horizontal_top = np.hstack(myList)

    # bottom level
    myList = (array_of_images[5], array_of_images[6], array_of_images[7], array_of_image
    numpy_horizontal_bottom = np.hstack(myList)

    # stick 2 ontop of one another
    myList = (numpy_horizontal_top, numpy_horizontal_bottom)
    numpy_final_pic_concat = np.concatenate(myList, axis=0)

    return numpy_final_pic_concat

# save and show in cell
def saveAndShow(desired_directory, image_array, picName):
    fileName = desired_directory + "\\" + picName
```

```
    # Save pic to file, using Pillow!
    new_img = fromarray(stitchTogetherPics(image_array))
    new_img.save(fileName)  # save using Pillow

    width = 30
    height = 10
    fig = plt.figure()
    fig.set_figwidth(width); fig.set_figheight(height)

    new_img = plt.imread(fileName)
    plt.title(picName)
    plt.axis('off')
    plt.imshow(new_img, cmap='gray')
```

In [11]:
```
saveAndShow(desired_directory, test_images, picName="01_StitchedImages.png")
```



Process the images

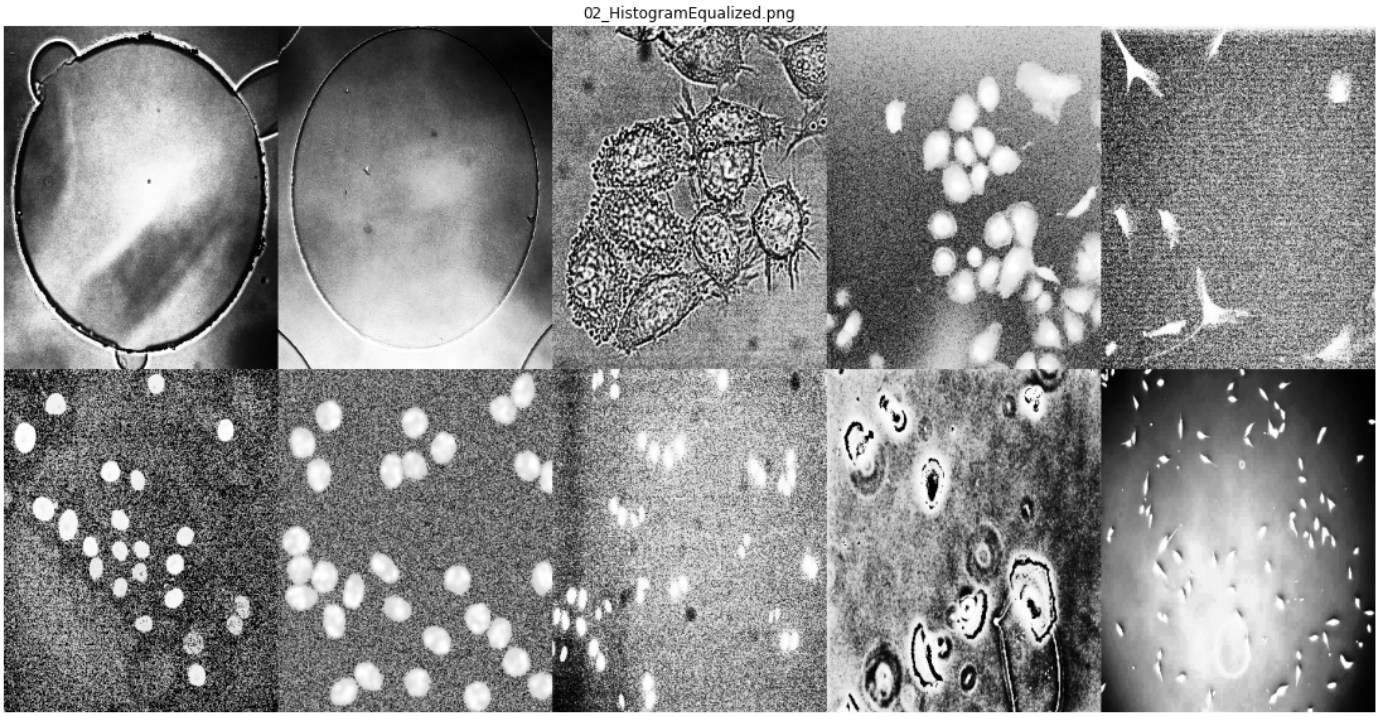
Here, we explore the techniques we can use to process the images. At each junction, JPG versions of the images will be saved to file for reference

Histogram Equalization

In [12]:
```
# Useful functions

def histEqualization(img):
    return cv2.equalizeHist(img)

def gaussianSmooth(img, arraySize):
    return cv2.GaussianBlur(img, (arraySize,arraySize), 0)

def medianSmooth(img, arraySize):
    return cv2.medianBlur(img,arraySize)
```

In [13]:
```
# let us loop through and generate an array of histogram corrected images
```

```
hist_eq_images = []

for i in range(len(test_images)):
    hist_eq_images.append( histEqualization( test_images[i] ) )

picName = "02_HistogramEqualized.png"
saveAndShow(desired_directory, hist_eq_images, picName)
```

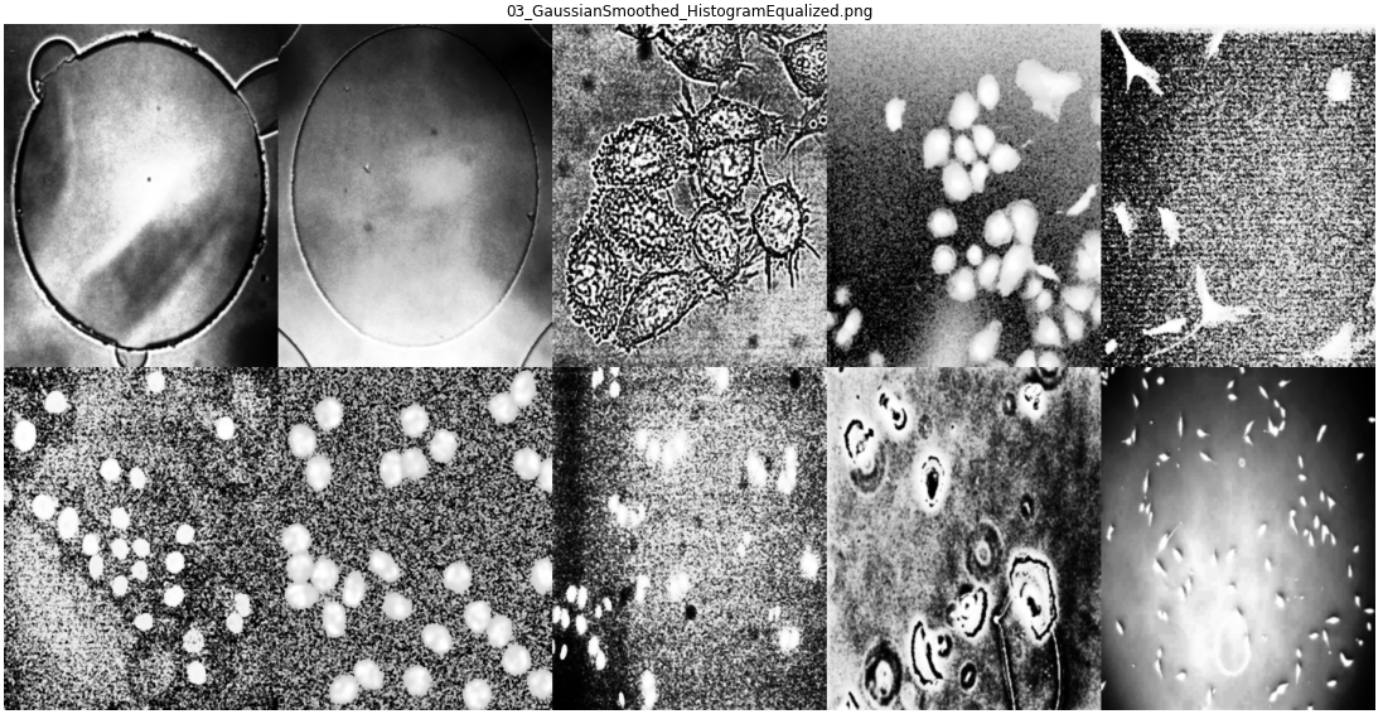


Wow! That is terrible! There is a lot of noise present and a lot of the cell features are lost

Does smoothing improve this?

In [14]:
```
gauss_smoothed_hist_eq_images = []

for i in range(len(hist_eq_images)):
    smoothed_img = gaussianSmooth( hist_eq_images[i], arraySize=3 )
    gauss_smoothed_hist_eq_images.append( histEqualization( smoothed_img ) )

picName = "03_GaussianSmoothed_HistogramEqualized.png"
saveAndShow(desired_directory, gauss_smoothed_hist_eq_images, picName)
```

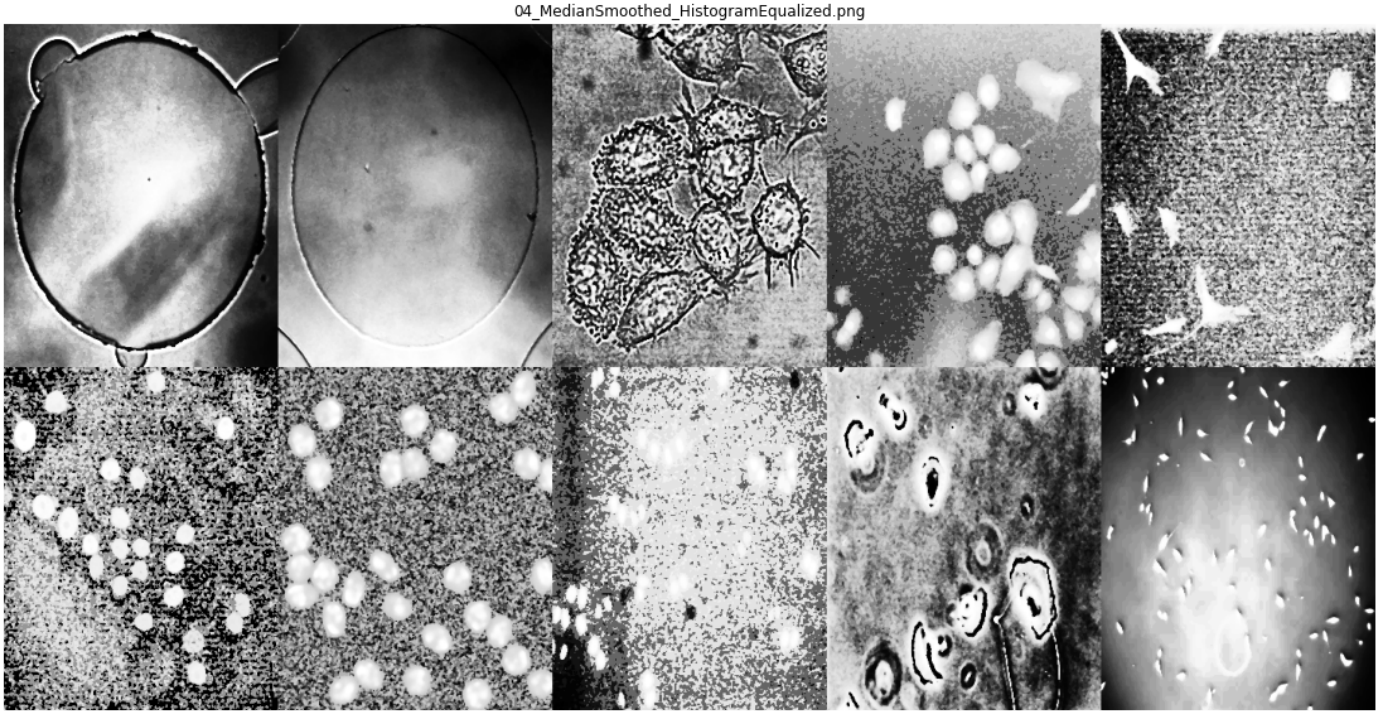
03_GaussianSmoothed_HistogramEqualized.png



That didn't seem to help... what about median smoothing?

In [15]:
```
median_smoothed_hist_eq_images = []

for i in range(len(hist_eq_images)):
    smoothed_img = medianSmooth( hist_eq_images[i], arraySize=3 )
    median_smoothed_hist_eq_images.append( histEqualization( smoothed_img ) )

picName = "04_MedianSmoothed_HistogramEqualized.png"
saveAndShow(desired_directory, median_smoothed_hist_eq_images, picName)
```

04_MedianSmoothed_HistogramEqualized.png



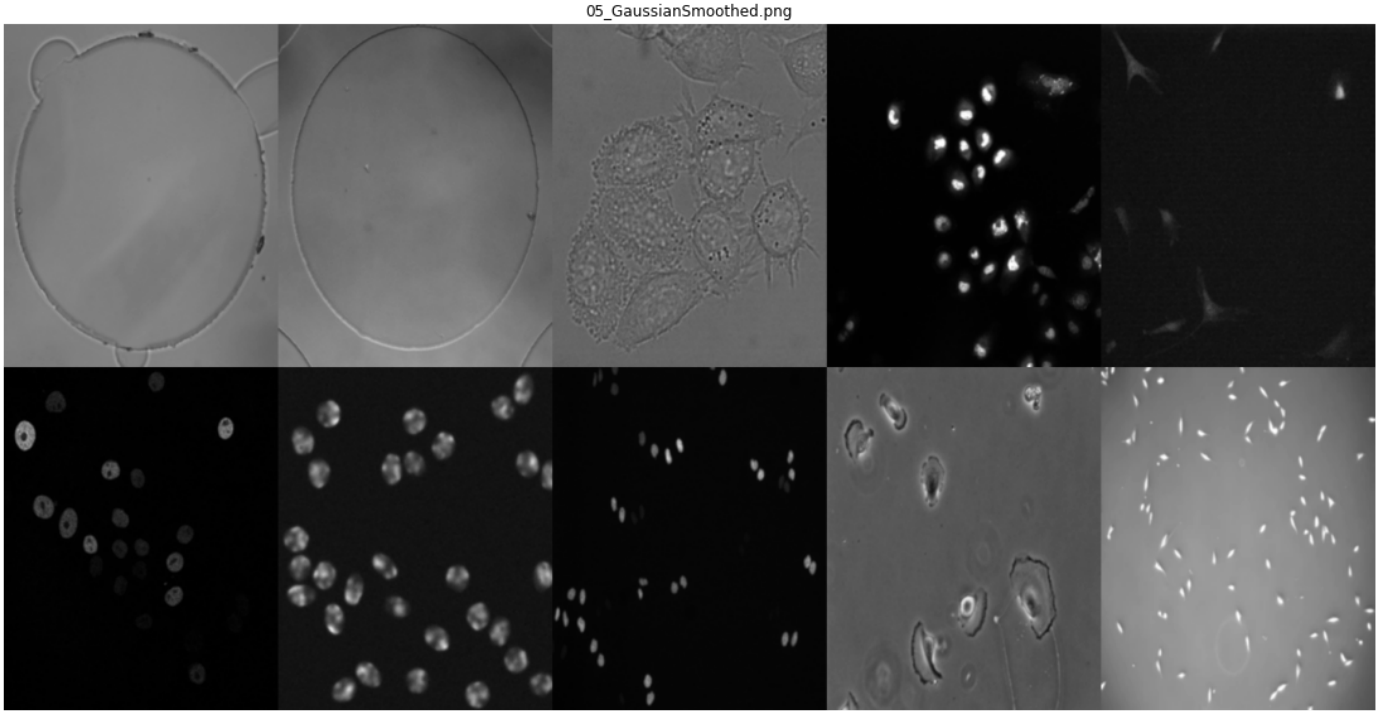
Okay, the smoothing before histogram equalization is not working... Histogram Equalization is clearly a bad idea.

What about just smoothing the images?

Smoothing

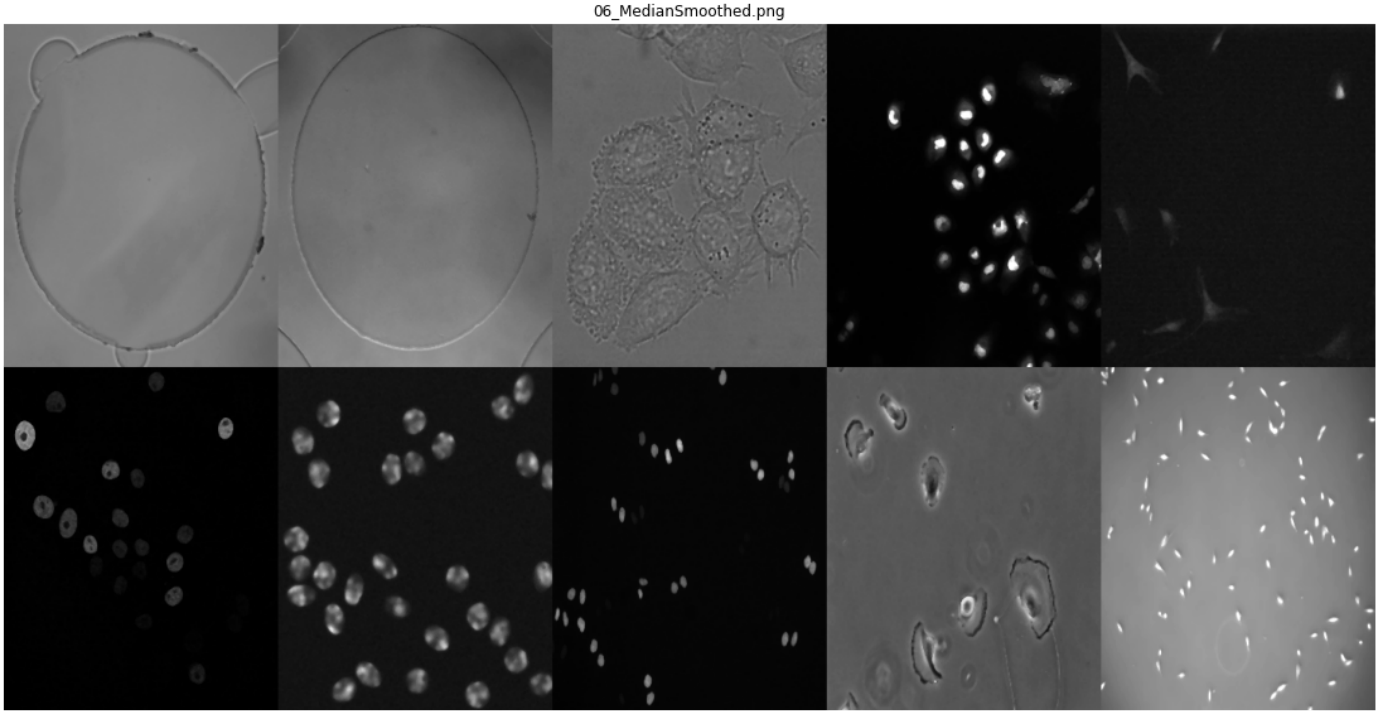
In [16]:
```
gaussian_smoothed_images = []

for i in range(len(test_images)):
    smoothed_img = gaussianSmooth( test_images[i], arraySize=3 )
    gaussian_smoothed_images.append(smoothed_img)

picName = "05_GaussianSmoothed.png"
saveAndShow(desired_directory, gaussian_smoothed_images, picName)
```

05_GaussianSmoothed.png



In [17]:
```
median_smoothed_images = []

for i in range(len(test_images)):
    smoothed_img = medianSmooth( test_images[i], arraySize=3 )
    median_smoothed_images.append(smoothed_img)

picName = "06_MedianSmoothed.png"
saveAndShow(desired_directory, median_smoothed_images, picName)
```

06_MedianSmoothed.png



From the above, The standard Smoothing appears to help with some subtle noise, but arraysize can't exceed 3... It also blurs the images a bit, so applying a filter may not be necessary... The images may not need to have the filter applied.

Let us consider Thresholding next

Thresholding

We need to consider iterative thresholding, as we seek to bring out the underlaying cells (in poorly lit areas)

Let us start with simple thresholding and then iterate:

First, we need to identify the minimum and maximum values of the images

In [18]:
```
for i in range(len(test_images)):
    print("Image", i+1, end=":\t")
    print("Maximum Value:", np.amax(test_images[i]), end=",\t")
    print("Minimum Value:", np.amin(test_images[i]))
```

```
Image 1:        Maximum Value: 219,     Minimum Value: 9
Image 2:        Maximum Value: 206,     Minimum Value: 18
Image 3:        Maximum Value: 240,     Minimum Value: 7
Image 4:        Maximum Value: 255,     Minimum Value: 0
Image 5:        Maximum Value: 211,     Minimum Value: 3
Image 6:        Maximum Value: 230,     Minimum Value: 0
Image 7:        Maximum Value: 248,     Minimum Value: 5
Image 8:        Maximum Value: 253,     Minimum Value: 0
Image 9:        Maximum Value: 243,     Minimum Value: 4
Image 10:       Maximum Value: 255,     Minimum Value: 68
```

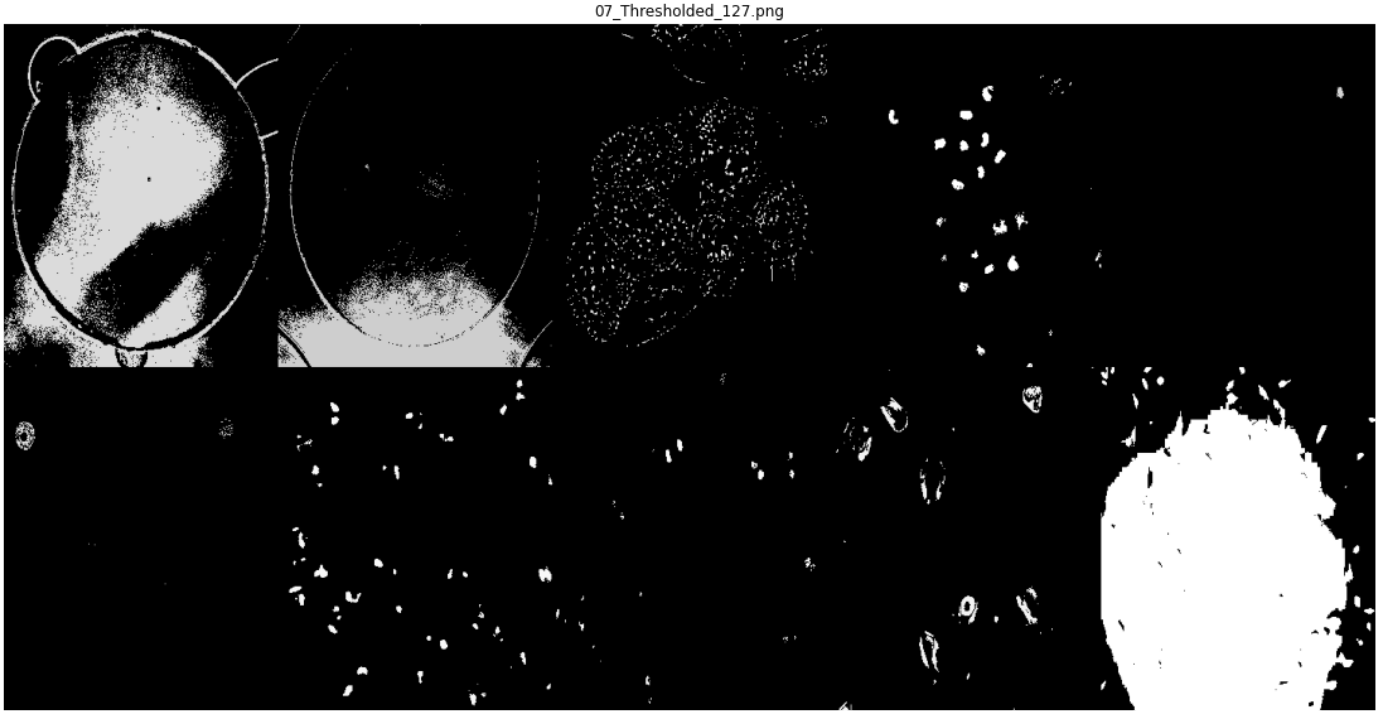
We must use this knowledge to generate a function that considers the threshold of the relevant image:

In [19]:
```
def thresholdTheImage(img, threshold_value):
    max_value = np.amax(img)
    return cv2.threshold(img, threshold_value, max_value, cv2.THRESH_BINARY)
```

Let us try the function with a threshold of 255 // 2 == 127

In [20]:
```
thresholded_images = []

for i in range(len(test_images)):
    _, thresh_img = thresholdTheImage(test_images[i], 127)
    thresholded_images.append(thresh_img)

picName = "07_Thresholded_127.png"
saveAndShow(desired_directory, thresholded_images, picName)
```

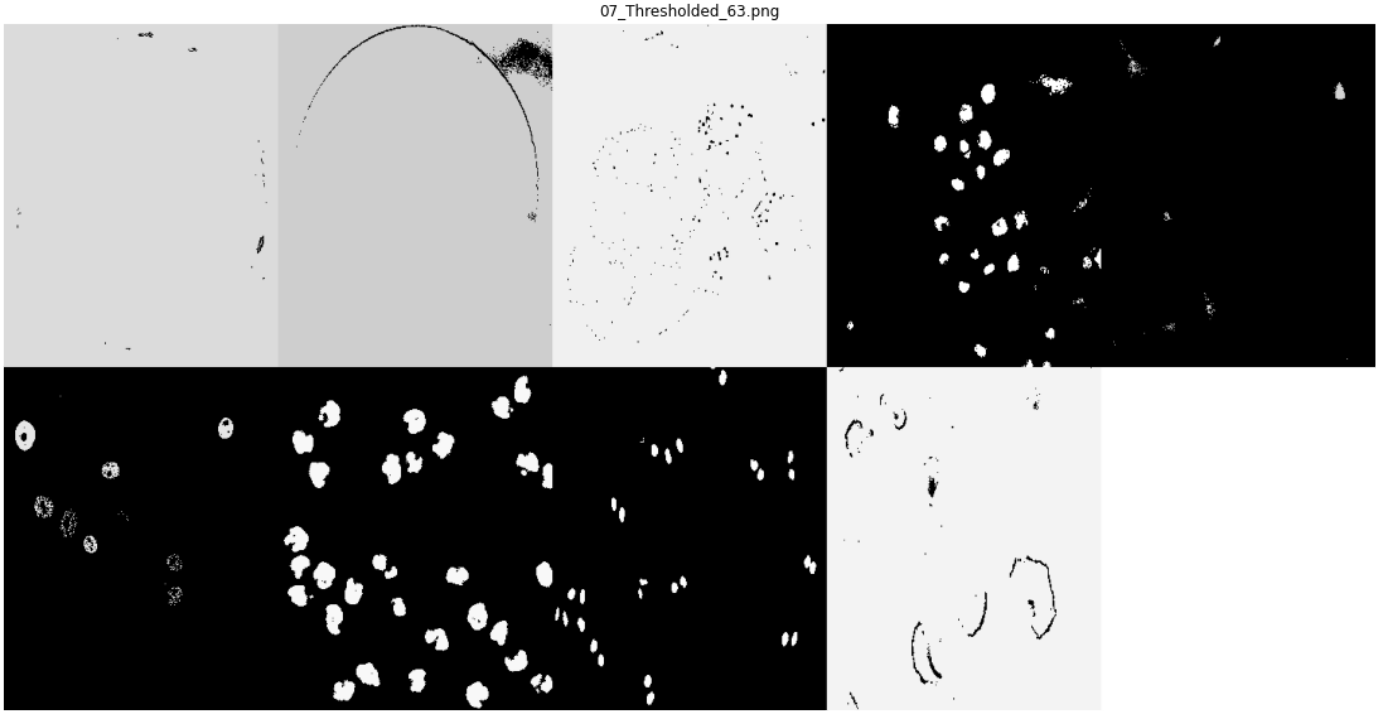
07_Thresholded_127.png



The results are rather poor... Let us try thresholding at a lower level, and a higher level:

In [21]:
```
thresholded_images = []

for i in range(len(test_images)):
    _, thresh_img = thresholdTheImage(test_images[i], 63)
    thresholded_images.append(thresh_img)

picName = "07_Thresholded_63.png"
saveAndShow(desired_directory, thresholded_images, picName)
```

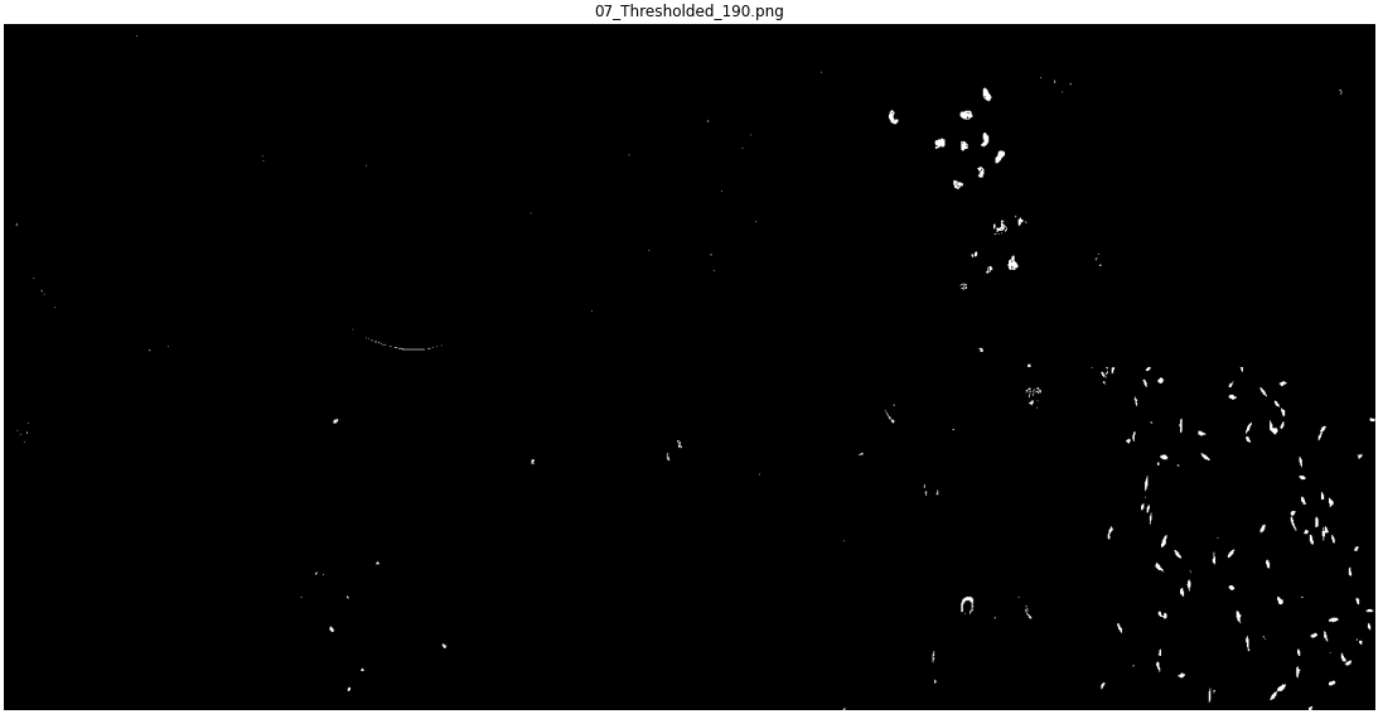
07_Thresholded_63.png



This image appears rather white for many of the images, understandibly

In [22]:
```
thresholded_images = []

for i in range(len(test_images)):
    _, thresh_img = thresholdTheImage(test_images[i], 190)
    thresholded_images.append(thresh_img)

picName = "07_Thresholded_190.png"
saveAndShow(desired_directory, thresholded_images, picName)
```

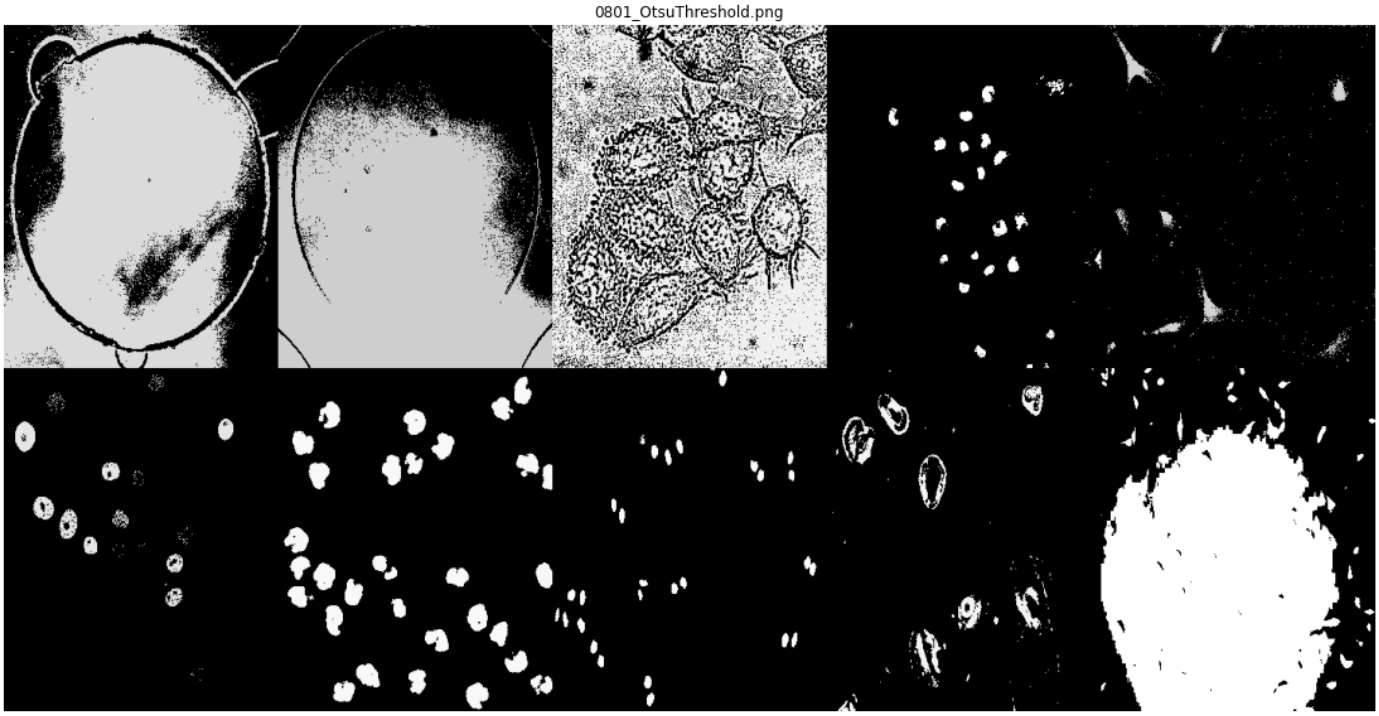
07_Thresholded_190.png



What we should try next is Otsu's Method for Thresholding, as well as Adaptive Thresholding:

In [23]:
```
def otsuThreshold(img):
    max_value = np.amax(img)
    return cv2.threshold(img, 0, max_value, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

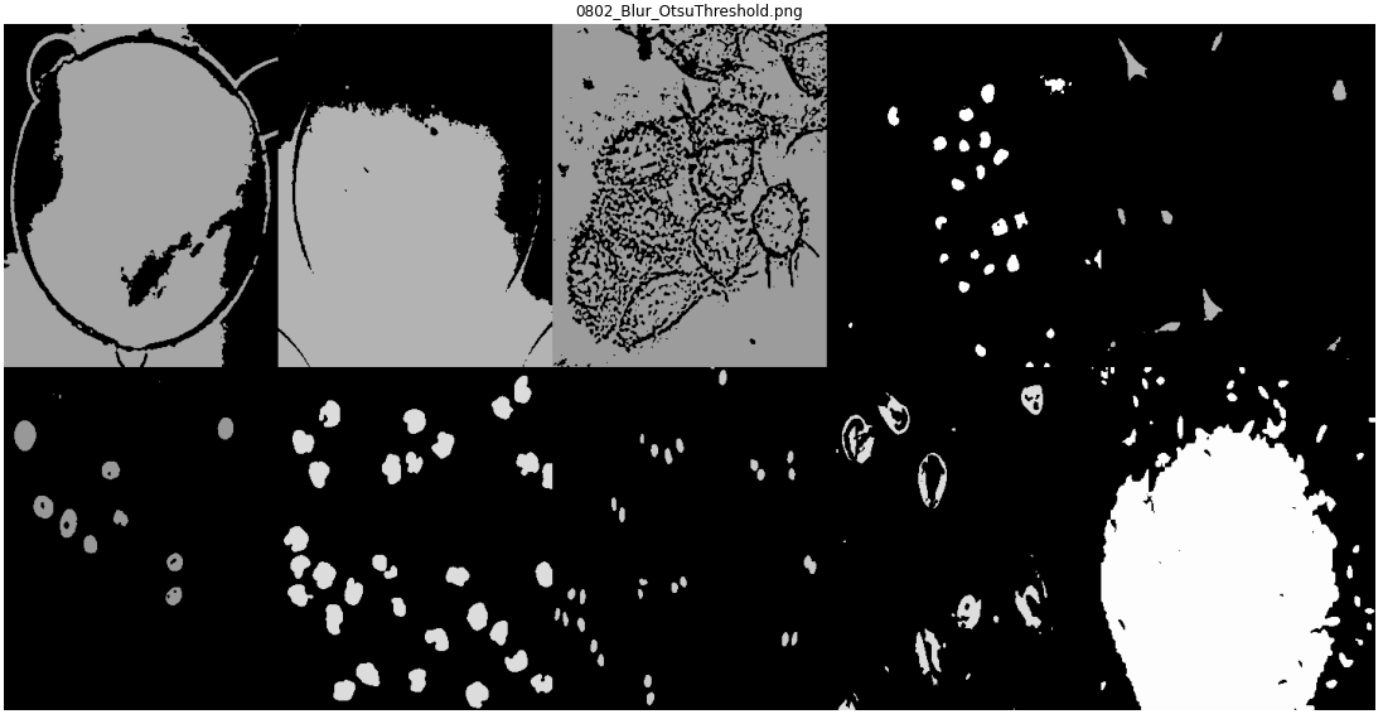
In [24]:
```
otsuThresholded_images = []

for i in range(len(test_images)):
    _, otsuThresh_img = otsuThreshold(test_images[i])
    otsuThresholded_images.append(otsuThresh_img)

picName = "0801_OtsuThreshold.png"
saveAndShow(desired_directory, otsuThresholded_images, picName)
```



This is okay, though some images have undesirable features. Does a Gaussian Blur help?

In [25]:
```
blur_otsuThresholded_images = []

for i in range(len(test_images)):
    blur = cv2.GaussianBlur(test_images[i], (5, 5), 0)
    _, otsuThresh_img = otsuThreshold(blur)
    blur_otsuThresholded_images.append(otsuThresh_img)

picName = "0802_Blur_OtsuThreshold.png"
saveAndShow(desired_directory, blur_otsuThresholded_images, picName)
```

0802_Blur_OtsuThreshold.png

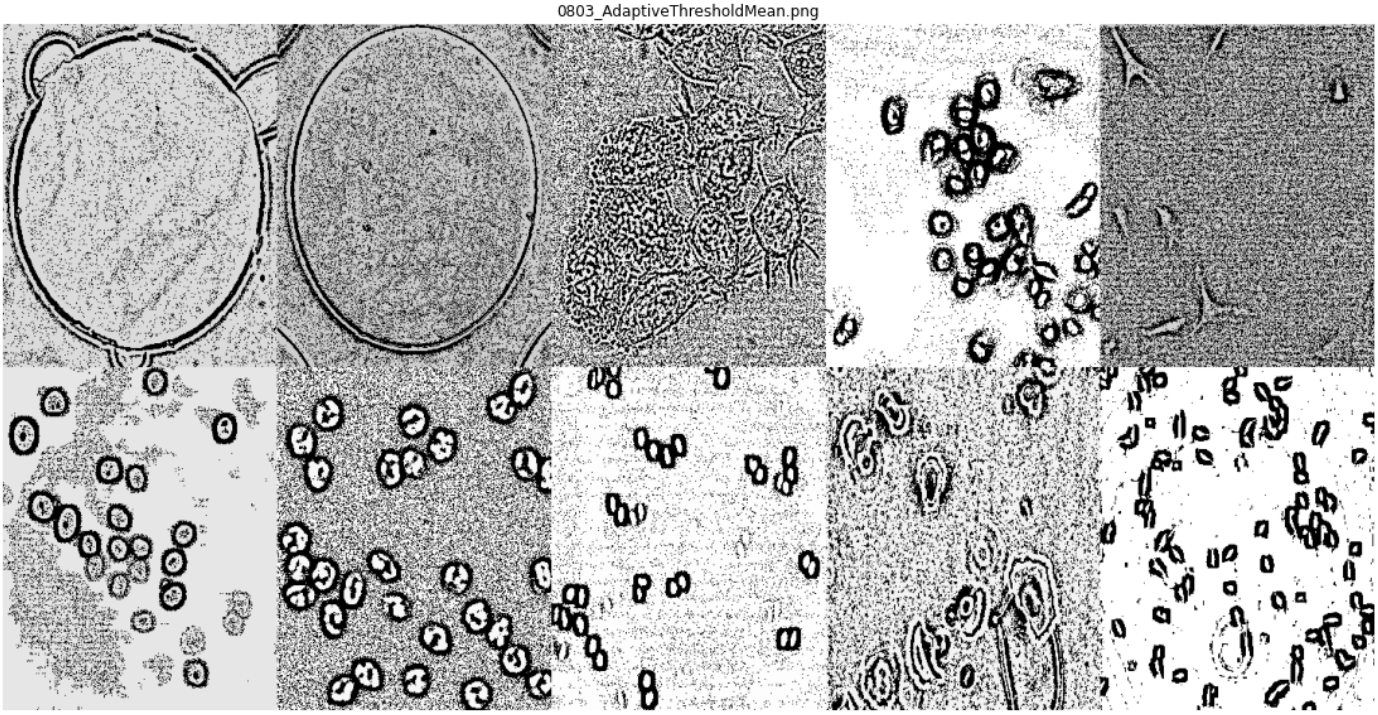


That is the most promising feature yet! Though some pictures have problems... The first image has the cell absorbed by black light, and the last image is saturated in white light... Does adaptive thresholding improve things?

In [26]:
```
def adaptiveThresholdingMean(img):
    max_value = np.amax(img)
    return cv2.adaptiveThreshold(img, max_value, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_

def adaptiveThresholdingGaussian(img):
    max_value = np.amax(img)
    return cv2.adaptiveThreshold(img, max_value, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THR
```

In [27]:
```
adaptiveThresholdedMean_images = []

for i in range(len(test_images)):
    thresh_img = adaptiveThresholdingMean(test_images[i])
    adaptiveThresholdedMean_images.append(thresh_img)

picName = "0803_AdaptiveThresholdMean.png"
saveAndShow(desired_directory, adaptiveThresholdedMean_images, picName)
```

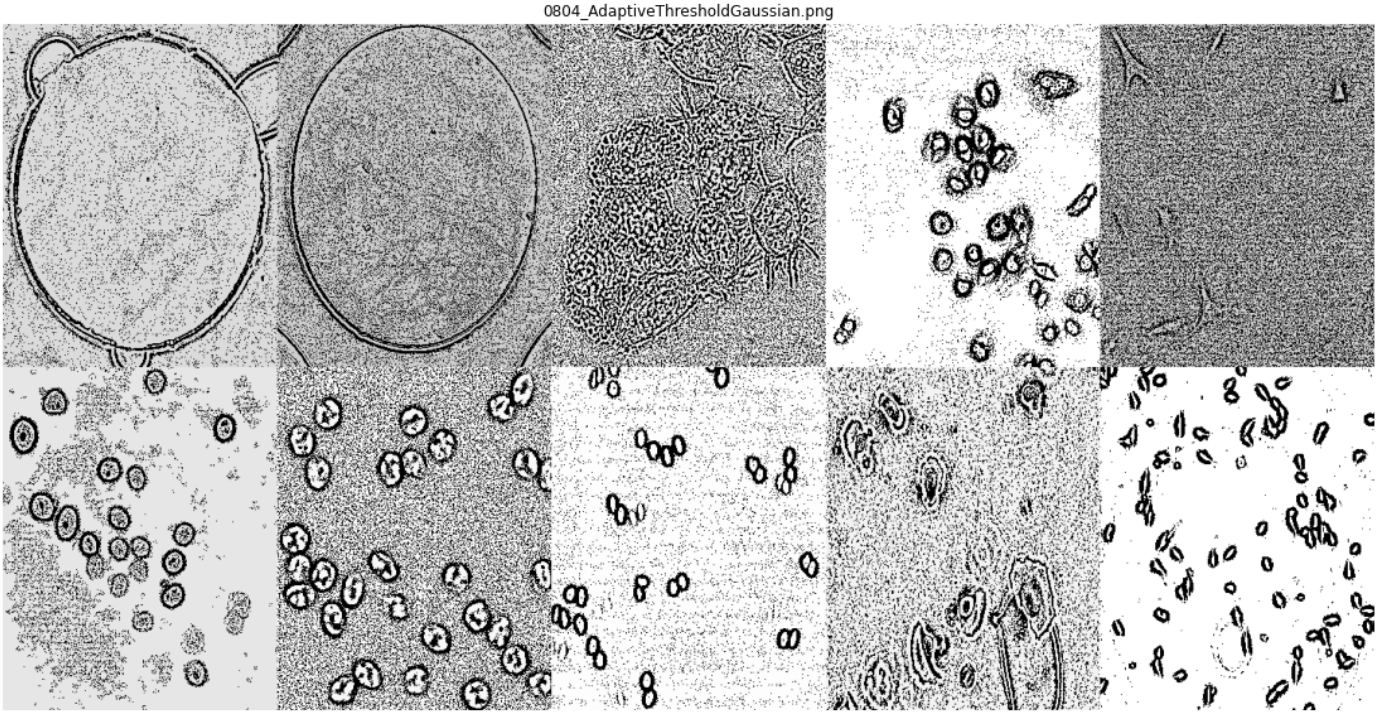
0803_AdaptiveThresholdMean.png



That is quite nice, though a lot of noise is present

In [28]:
```
adaptiveThresholdedGaussian_images = []

for i in range(len(test_images)):
    thresh_img = adaptiveThresholdingGaussian(test_images[i])
    adaptiveThresholdedGaussian_images.append(thresh_img)

picName = "0804_AdaptiveThresholdGaussian.png"
saveAndShow(desired_directory, adaptiveThresholdedGaussian_images, picName)
```

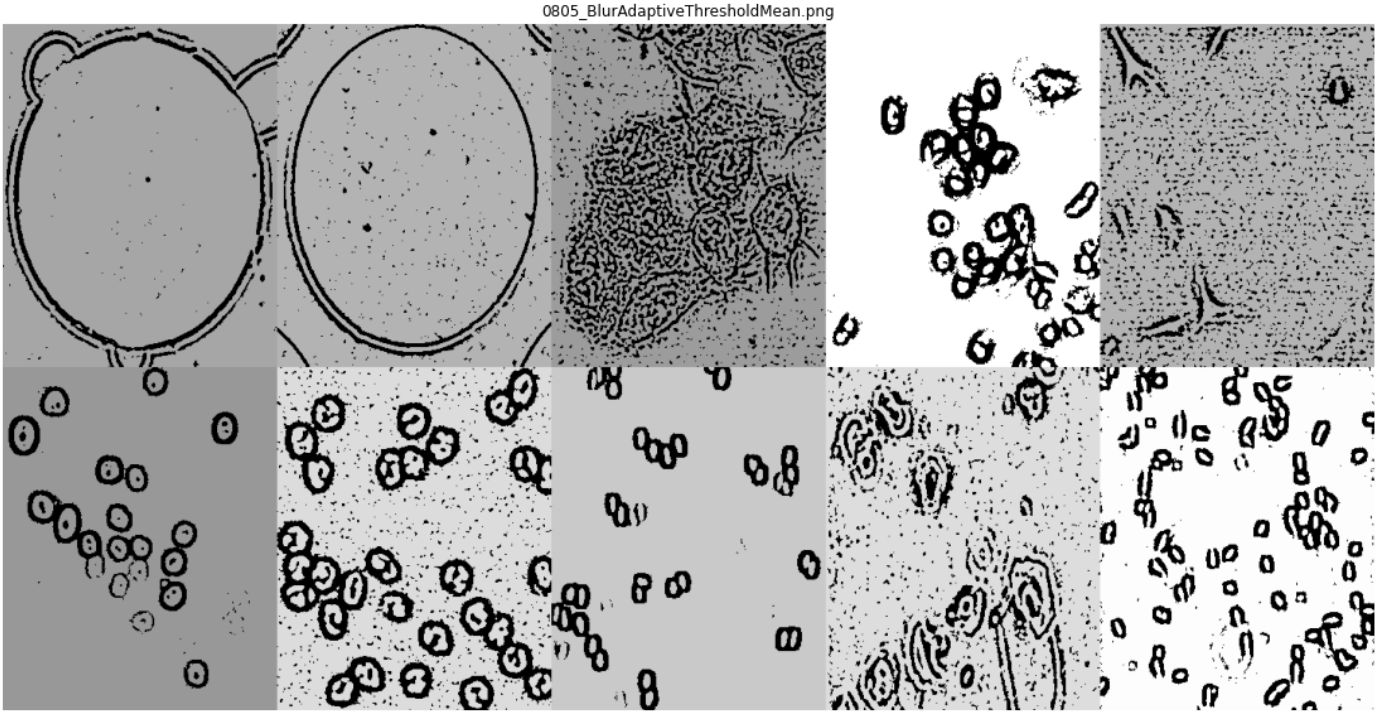
0804_AdaptiveThresholdGaussian.png



Although that is nice, the Adaptive Thresholding Mean images are slightly clearer. Cell shape is preserved, even in the highly lit final image! Though, the presence of noise makes it difficult to find the cells in the first 2 images... What about if we blurred the Adaptive Thresholding Mean images?

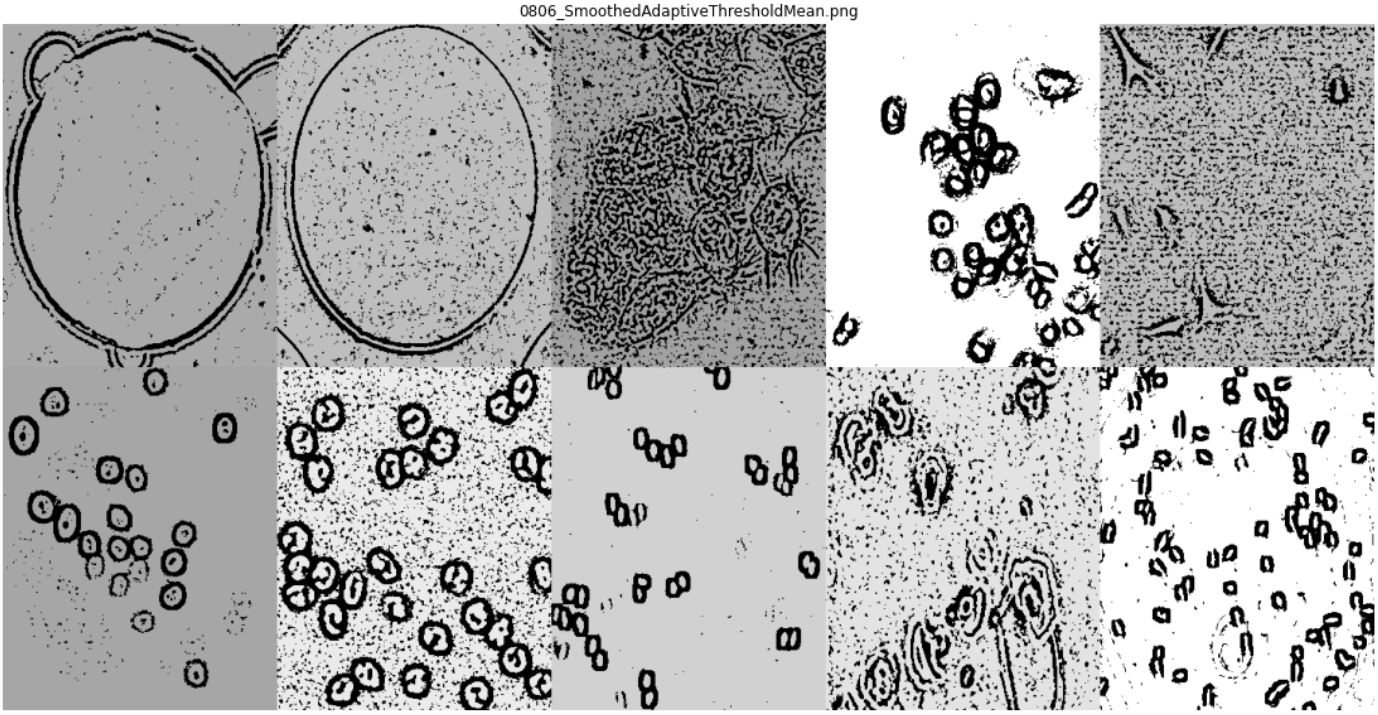
In [29]:
```
blurAdaptiveThresholdedMean_images = []
```

```
for i in range(len(test_images)):
    blur = cv2.GaussianBlur(test_images[i], (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)
    blurAdaptiveThresholdedMean_images.append(thresh_img)

picName = "0805_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, blurAdaptiveThresholdedMean_images, picName)
```



That is the best result so far! Let us recreate it with median smoothing, to compare:

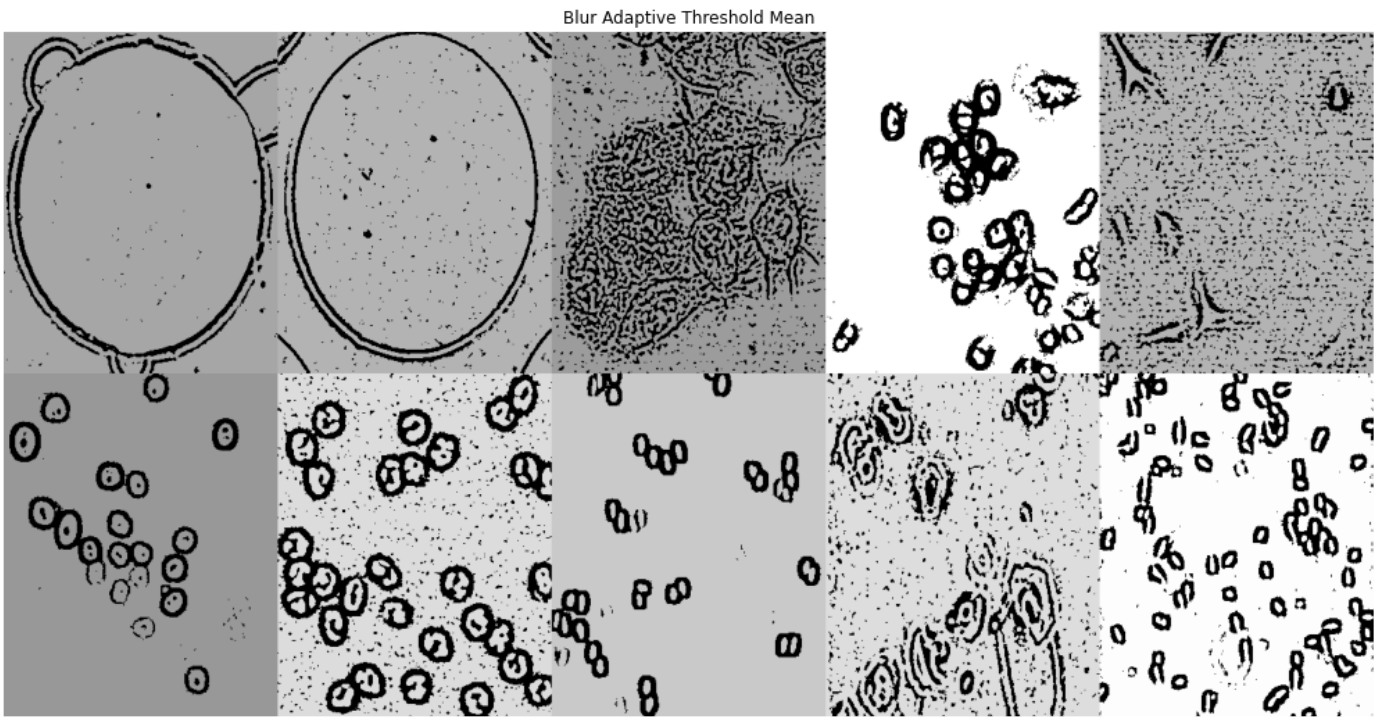
In [30]:
```
smoothedAdaptiveThresholdedMean_images = []

for i in range(len(test_images)):
    smoothed_img = medianSmooth( test_images[i], arraySize=3 )
    thresh_img = adaptiveThresholdingMean(smoothed_img)
    smoothedAdaptiveThresholdedMean_images.append(thresh_img)

picName = "0806_SmoothedAdaptiveThresholdMean.png"
saveAndShow(desired_directory, smoothedAdaptiveThresholdedMean_images, picName)
```



Hands down, the blurred images contain less noise! I think we have a fantastic processing tool for segmentation!

Let us bring that into view with Matplotlib and focus closely on it:

In [31]:
```
fig = plt.figure(figsize=(14, 10))

fig.add_subplot(1, 1, 1)
plt.title("Blur Adaptive Threshold Mean")

img = plt.imread( join(desired_directory, "0805_BlurAdaptiveThresholdMean.png") )
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.tight_layout()

plt.show()
```



Now! if we following the naming convention:

A B C D E F

G H I J K L

Then, we can see that the cells in blocks D - J and L are easy to identify. K may pose a challenge, but the hidden information is coming through. A and B though, the cell is not clearly being shown. Because of the clustering in C, the cells also appear to be speckled, when in reality, they need to de distinct.

An investigation of Morphological Operations on this kind of Thresholding may be ideal.

Morphological Operations on Thresholded Image

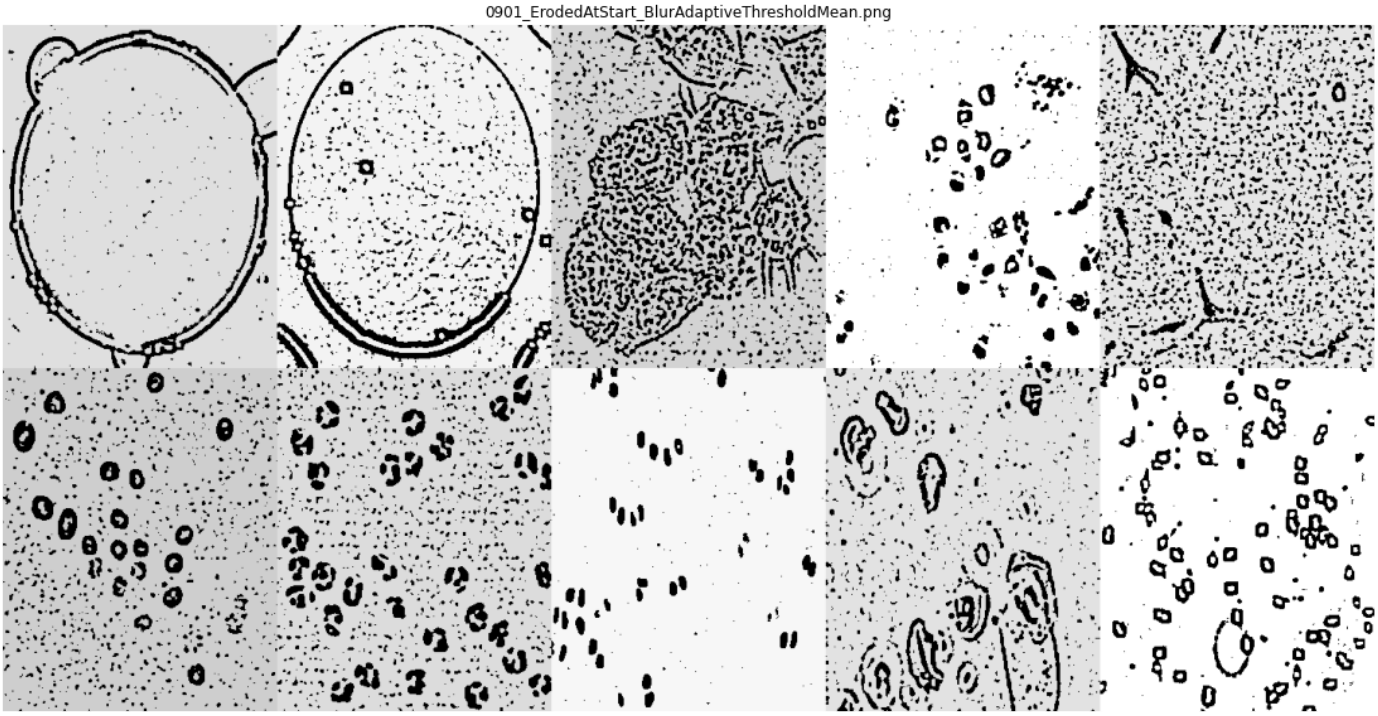
We are just going to focus on applying morphological operations on our best thresholded image, as we are trying to improve on that image

Let us try eroding at the front, as well as the back

In [32]:
```
kernel = np.ones((3,3), np.uint8)
```

In [33]:
```
image_collection = []

for i in range(len(test_images)):
    max_val = np.amax(test_images[i])
    # get negative
    negative = max_val - test_images[i]
    img_erosion = cv2.erode(negative, kernel, iterations=1)
    # get positive
    positive = max_val + img_erosion

    blur = cv2.GaussianBlur(positive, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "0901_ErodedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

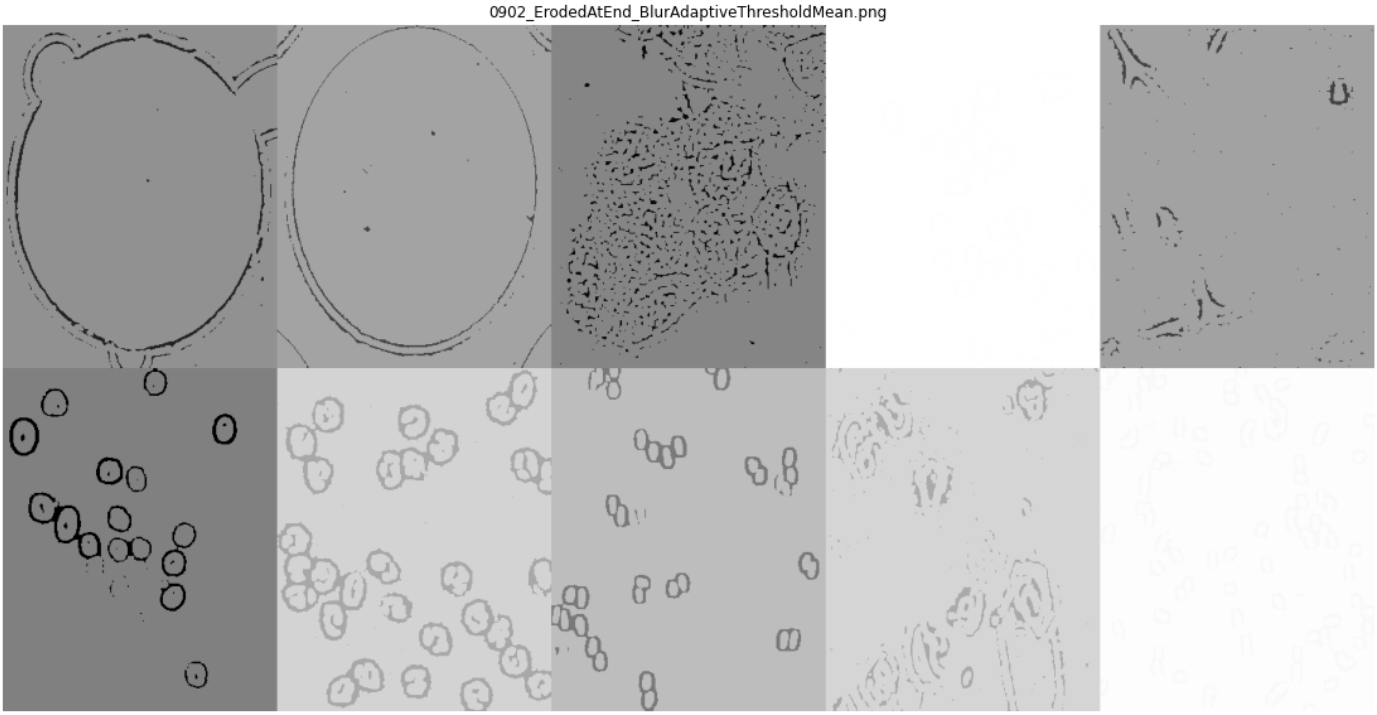
0901_ErodedAtStart_BlurAdaptiveThresholdMean.png



In [34]:

```
image_collection = []

for i in range(len(test_images)):
    blur = cv2.GaussianBlur(test_images[i], (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    max_val = np.amax(thresh_img)
    # get negative
    negative = max_val - thresh_img
    img_erosion = cv2.erode(negative, kernel, iterations=1)
    # get positive
    positive = max_val + img_erosion

    image_collection.append(positive)

picName = "0902_ErodedAtEnd_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

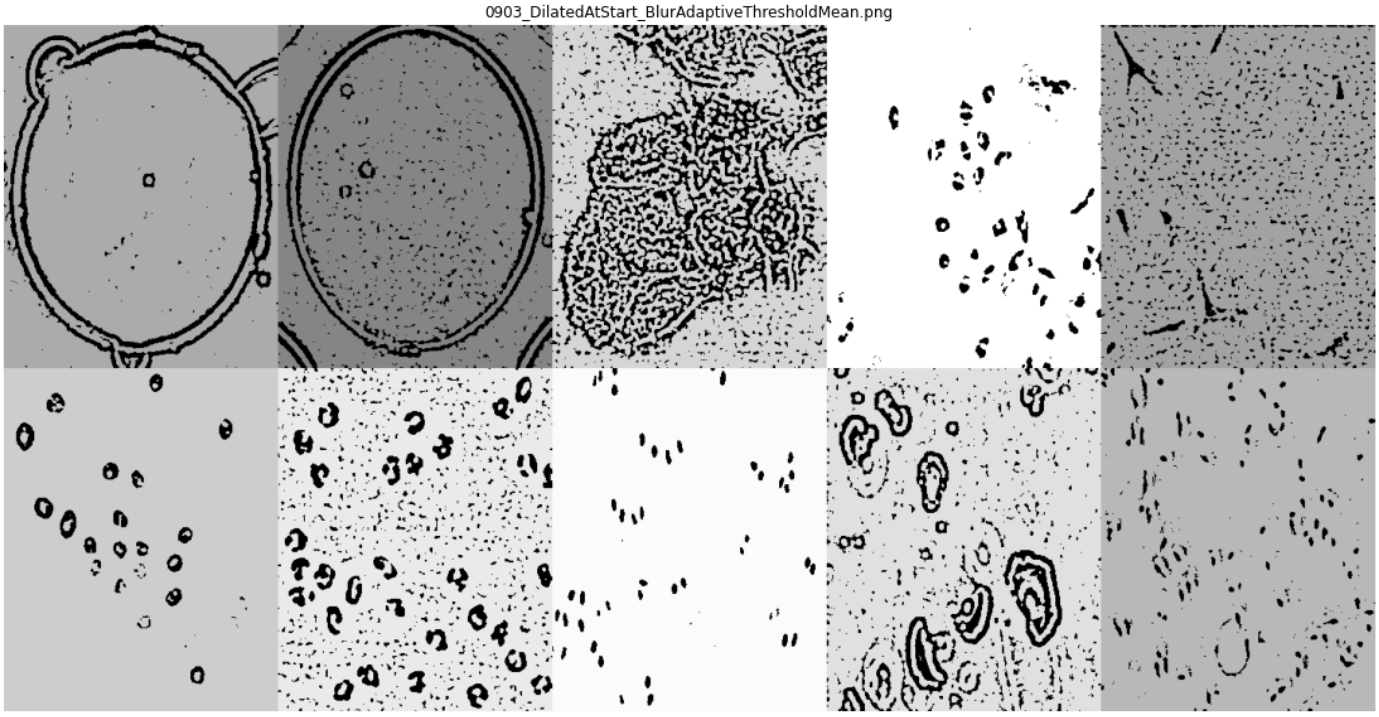
0902_ErodedAtEnd_BlurAdaptiveThresholdMean.png



Now, let us try dilating at front and back

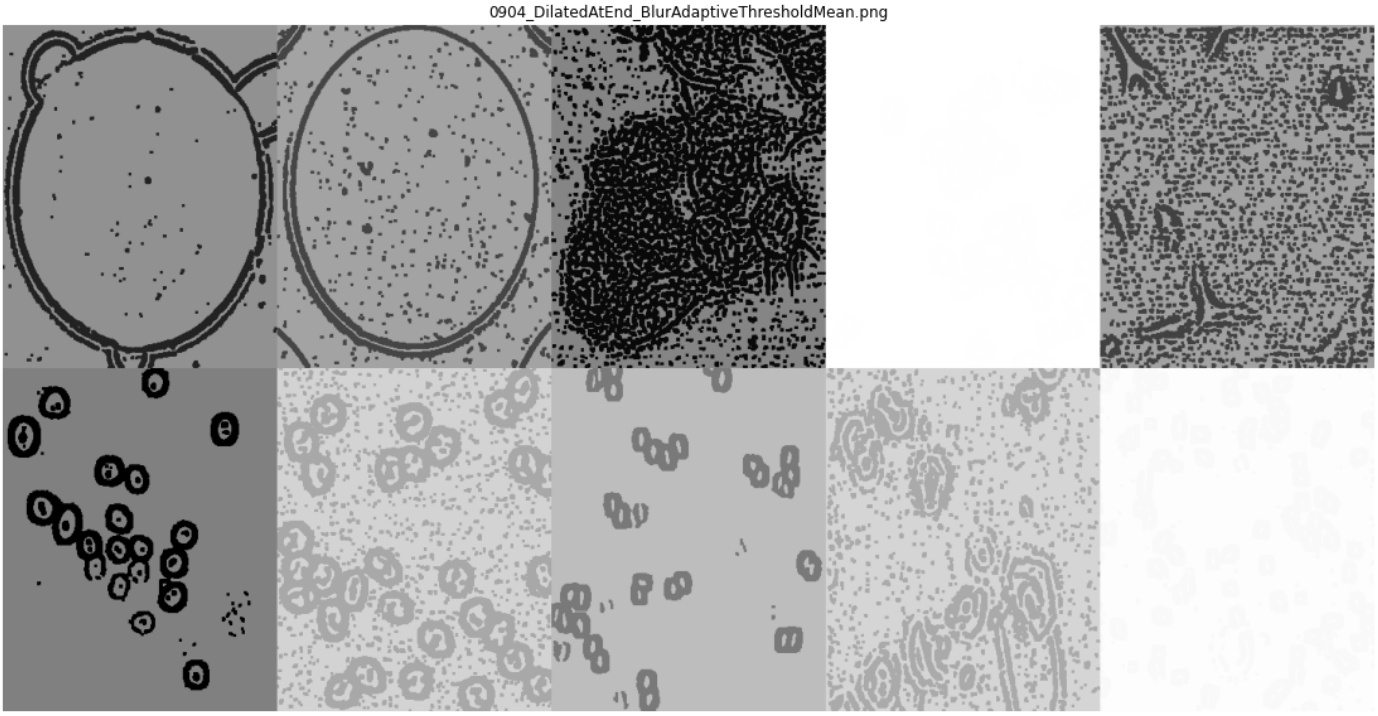
In [35]:
```
image_collection = []

for i in range(len(test_images)):
    max_val = np.amax(test_images[i])
    # get negative
    negative = max_val - test_images[i]
    img_erosion = cv2.dilate(negative, kernel, iterations=1)
    # get positive
    positive = max_val + img_erosion

    blur = cv2.GaussianBlur(positive, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "0903_DilatedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

0903_DilatedAtStart_BlurAdaptiveThresholdMean.png



In [36]:
```
image_collection = []

for i in range(len(test_images)):
    blur = cv2.GaussianBlur(test_images[i], (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    max_val = np.amax(thresh_img)
    # get negative
    negative = max_val - thresh_img
    img_erosion = cv2.dilate(negative, kernel, iterations=1)
    # get positive
    positive = max_val + img_erosion

    image_collection.append(positive)

picName = "0904_DilatedAtEnd_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

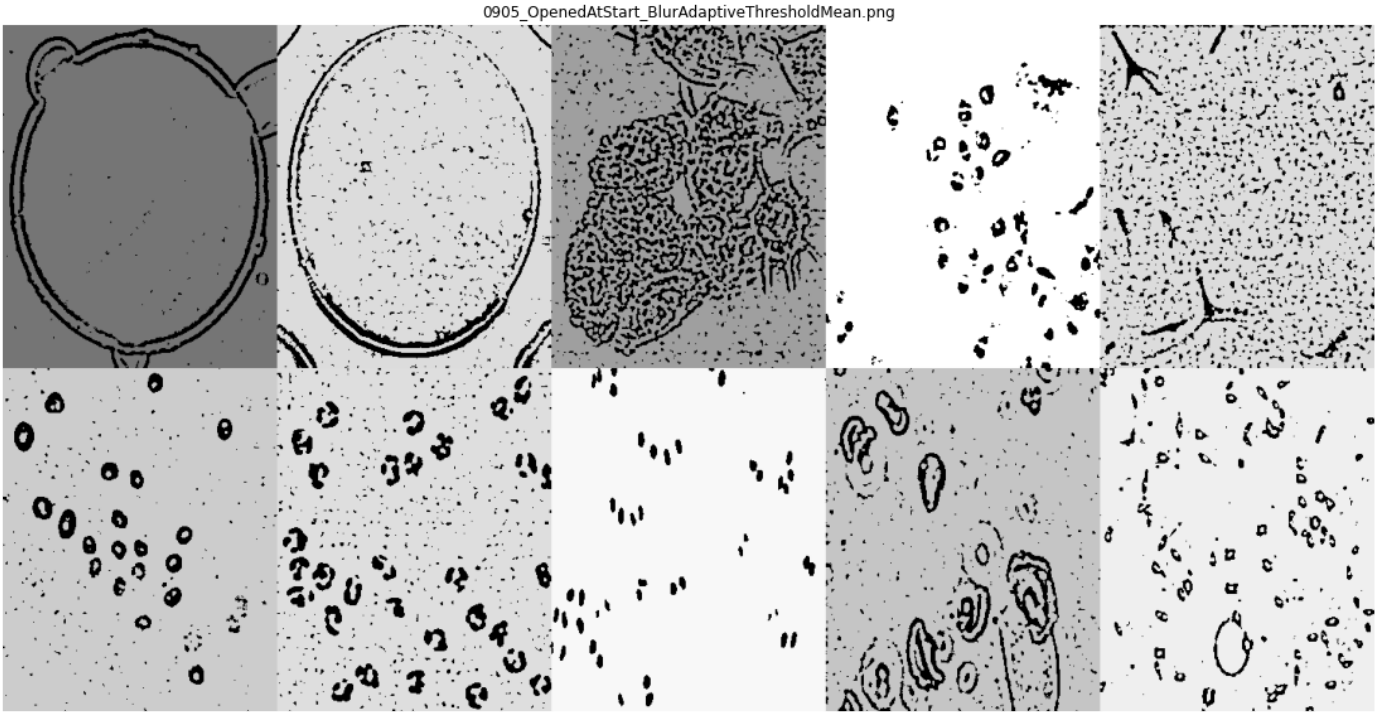
0904_DilatedAtEnd_BlurAdaptiveThresholdMean.png



In both cases, the morphological operation at the end of the code results in a large pixel effect... The dilation and erosion seems to alter the features though. Let us look at opening and closing next:

In [37]:
```
# Opening at front

image_collection = []

for i in range(len(test_images)):
    max_val = np.amax(test_images[i])
    # get negative
    negative = max_val - test_images[i]
    img_erosion = cv2.erode(negative, kernel, iterations=1)
    img_opened = cv2.dilate(img_erosion, kernel, iterations=1)
    # get positive
    positive = max_val + img_opened

    blur = cv2.GaussianBlur(positive, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "0905_OpenedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

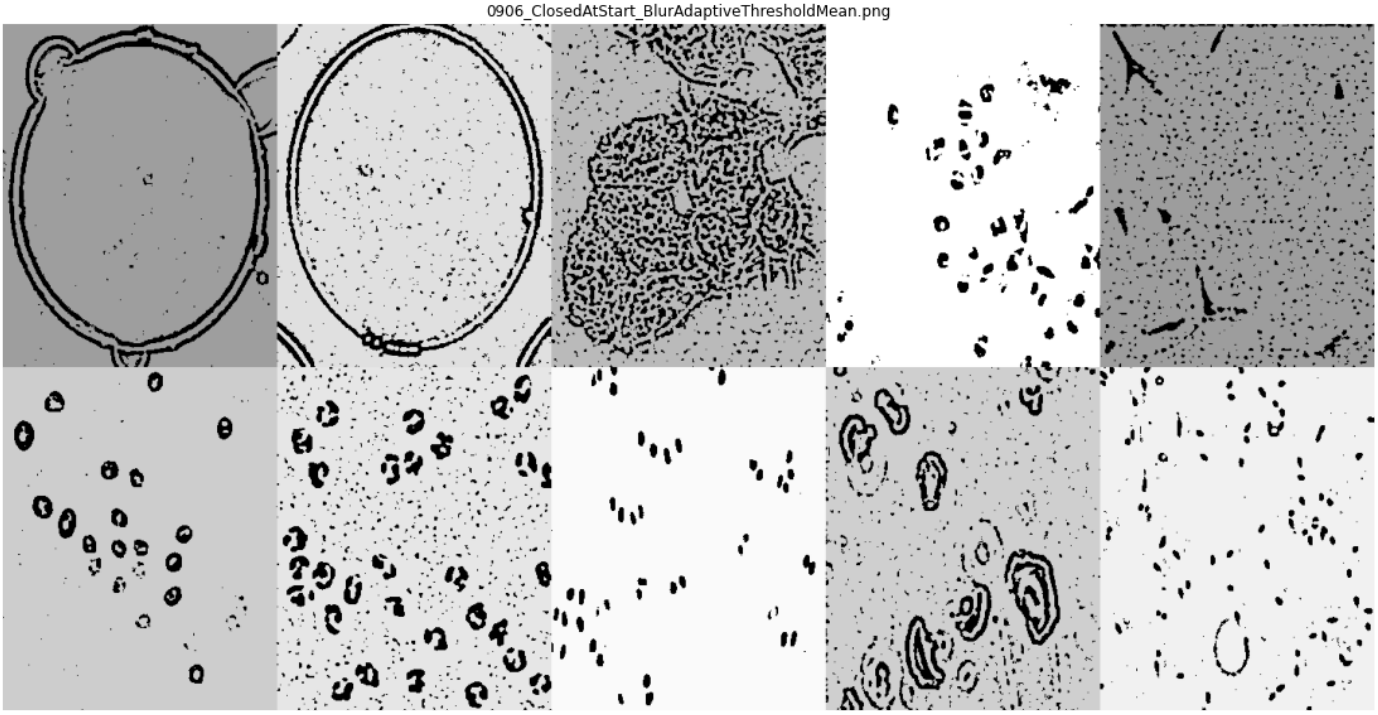
0905_OpenedAtStart_BlurAdaptiveThresholdMean.png



In [38]:

```
# Closing at front

image_collection = []

for i in range(len(test_images)):
    max_val = np.amax(test_images[i])
    # get negative
    negative = max_val - test_images[i]
    img_erosion = cv2.dilate(negative, kernel, iterations=1)
    img_closed = cv2.erode(img_erosion, kernel, iterations=1)
    # get positive
    positive = max_val + img_closed

    blur = cv2.GaussianBlur(positive, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "0906_ClosedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

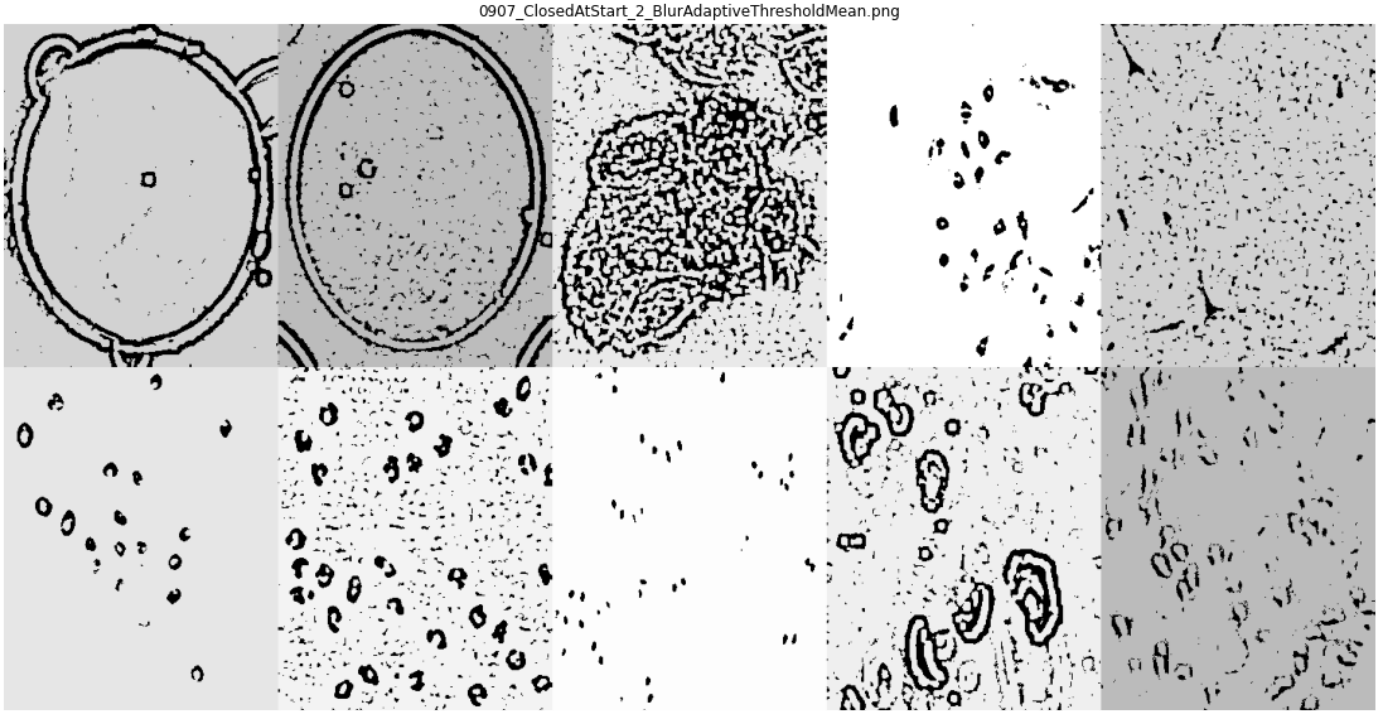
0906_ClosedAtStart_BlurAdaptiveThresholdMean.png



The closing at the front of the algorithm appears to be helping, does 2 iterations improve this?

In [39]:
```
# Closing at front

image_collection = []

for i in range(len(test_images)):
    max_val = np.amax(test_images[i])
    # get negative
    negative = max_val - test_images[i]
    img_erosion = cv2.dilate(negative, kernel, iterations=2)
    img_closed = cv2.erode(img_erosion, kernel, iterations=2)
    # get positive
    positive = max_val + img_closed

    blur = cv2.GaussianBlur(img_erosion, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "0907_ClosedAtStart_2_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

0907_ClosedAtStart_2_BlurAdaptiveThresholdMean.png

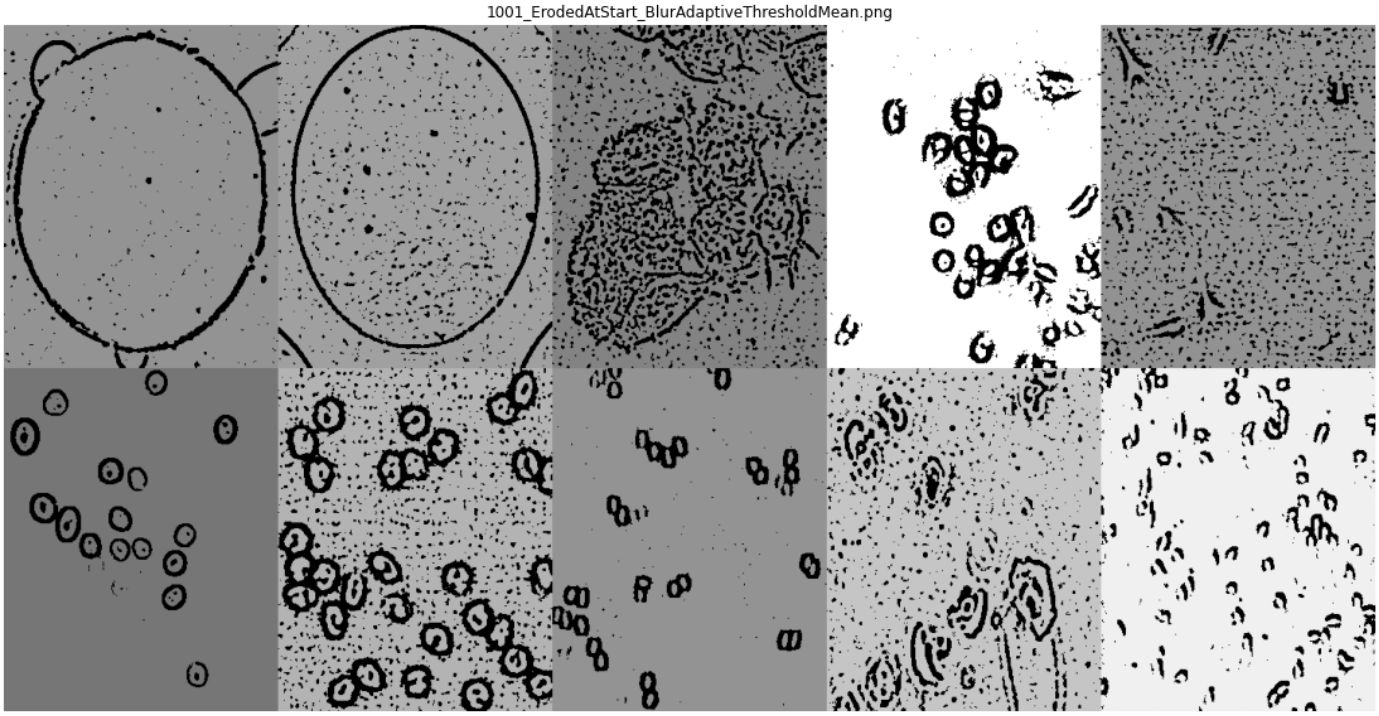


NOPE! Let us stick to 1 iteration!

All of the above was applying a morphological operation on the negative image, then restoring it to normal. What if we do not invert the image?

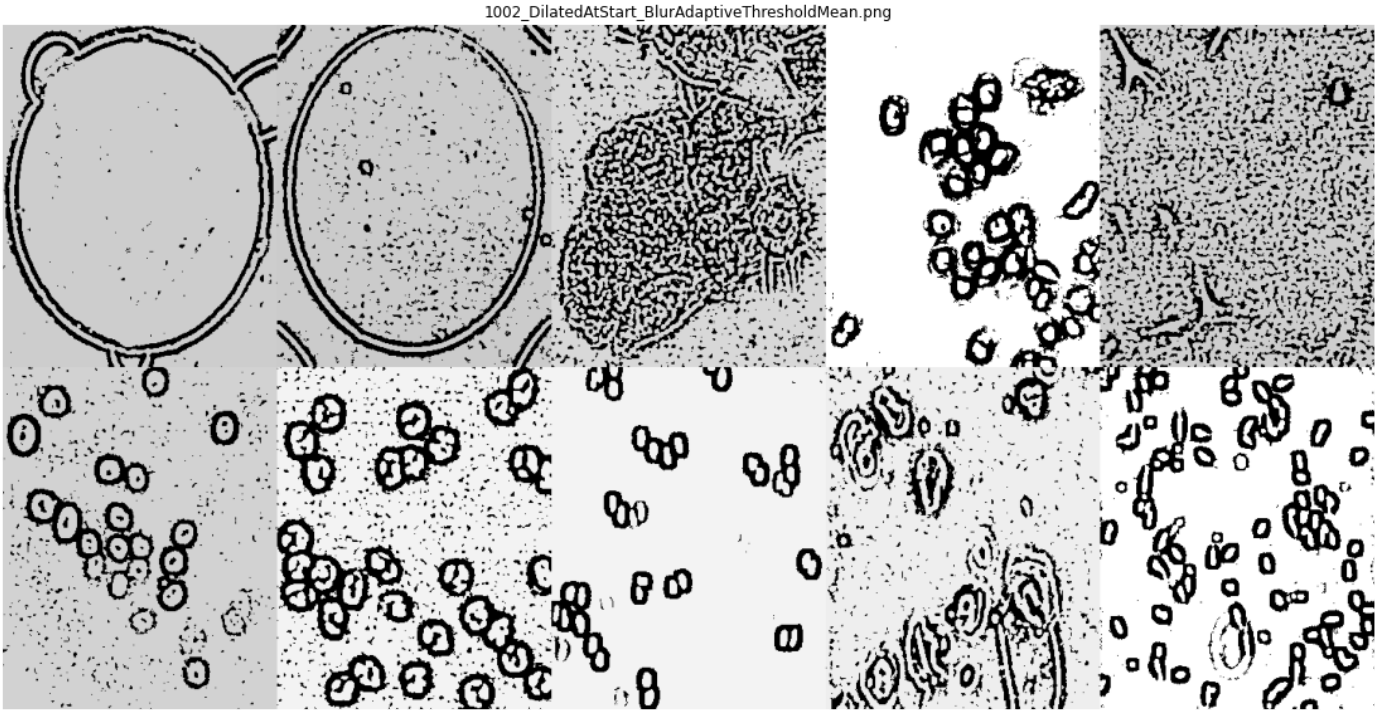
In [40]:
```
image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.erode(test_images[i], kernel, iterations=1)

    blur = cv2.GaussianBlur(img_erosion, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "1001_ErodedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

1001_ErodedAtStart_BlurAdaptiveThresholdMean.png



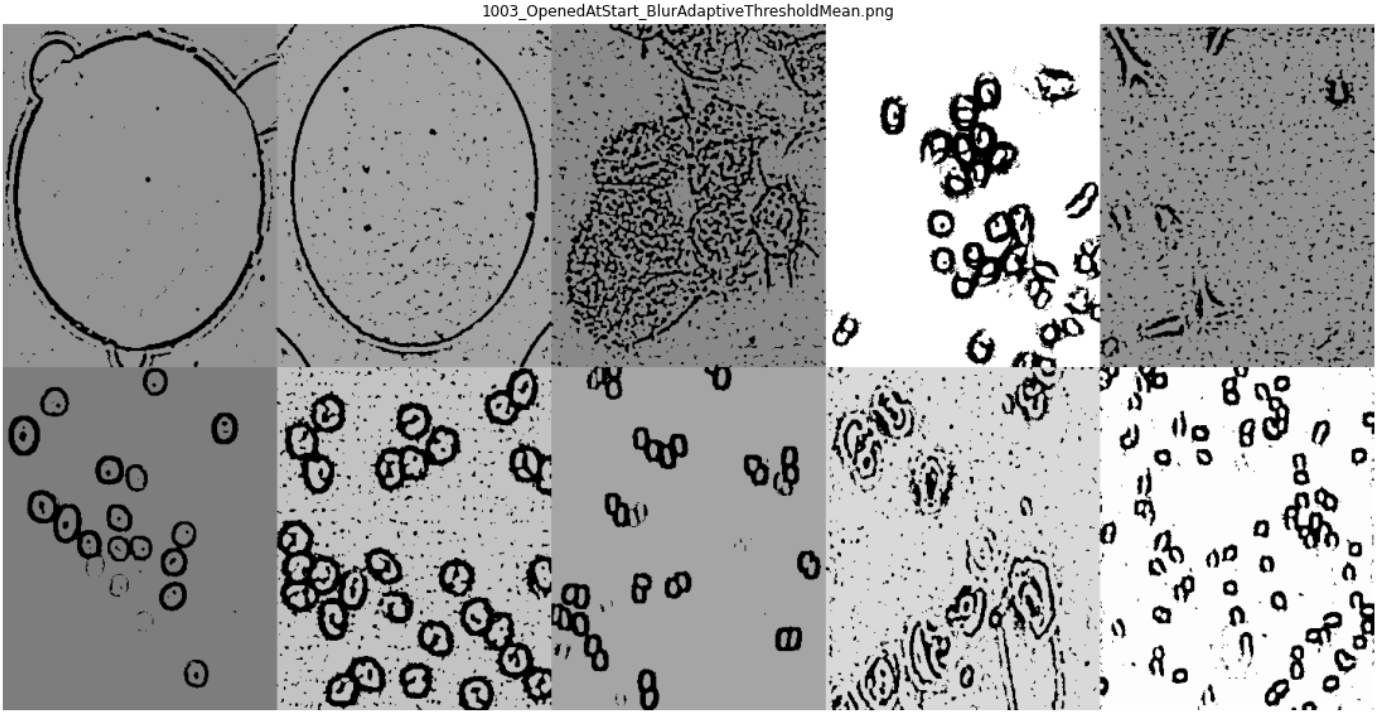
In [44]:
```
image_collection = []

for i in range(len(test_images)):
    img_dilate = cv2.dilate(test_images[i], kernel, iterations=1)

    blur = cv2.GaussianBlur(img_dilate, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "1002_DilatedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

1002_DilatedAtStart_BlurAdaptiveThresholdMean.png



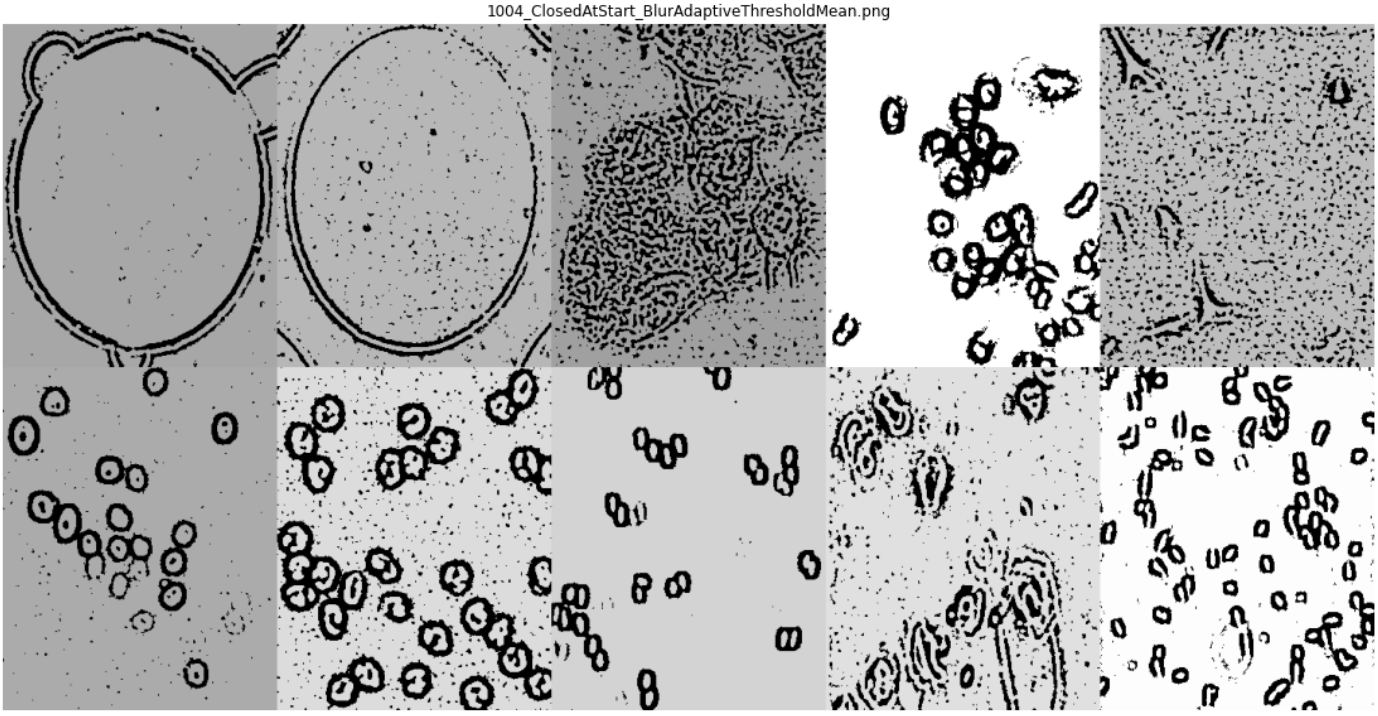
In [42]:
```
# Opening at front

image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.erode(test_images[i], kernel, iterations=1)
    img_opened = cv2.dilate(img_erosion, kernel, iterations=1)

    blur = cv2.GaussianBlur(img_opened, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "1003_OpenedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

1003_OpenedAtStart_BlurAdaptiveThresholdMean.png



In [43]:
```
# Closing at front

image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.dilate(test_images[i], kernel, iterations=1)
    img_closed = cv2.erode(img_erosion, kernel, iterations=1)

    blur = cv2.GaussianBlur(img_closed, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    image_collection.append(thresh_img)

picName = "1004_ClosedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

1004_ClosedAtStart_BlurAdaptiveThresholdMean.png



Discussion of current results

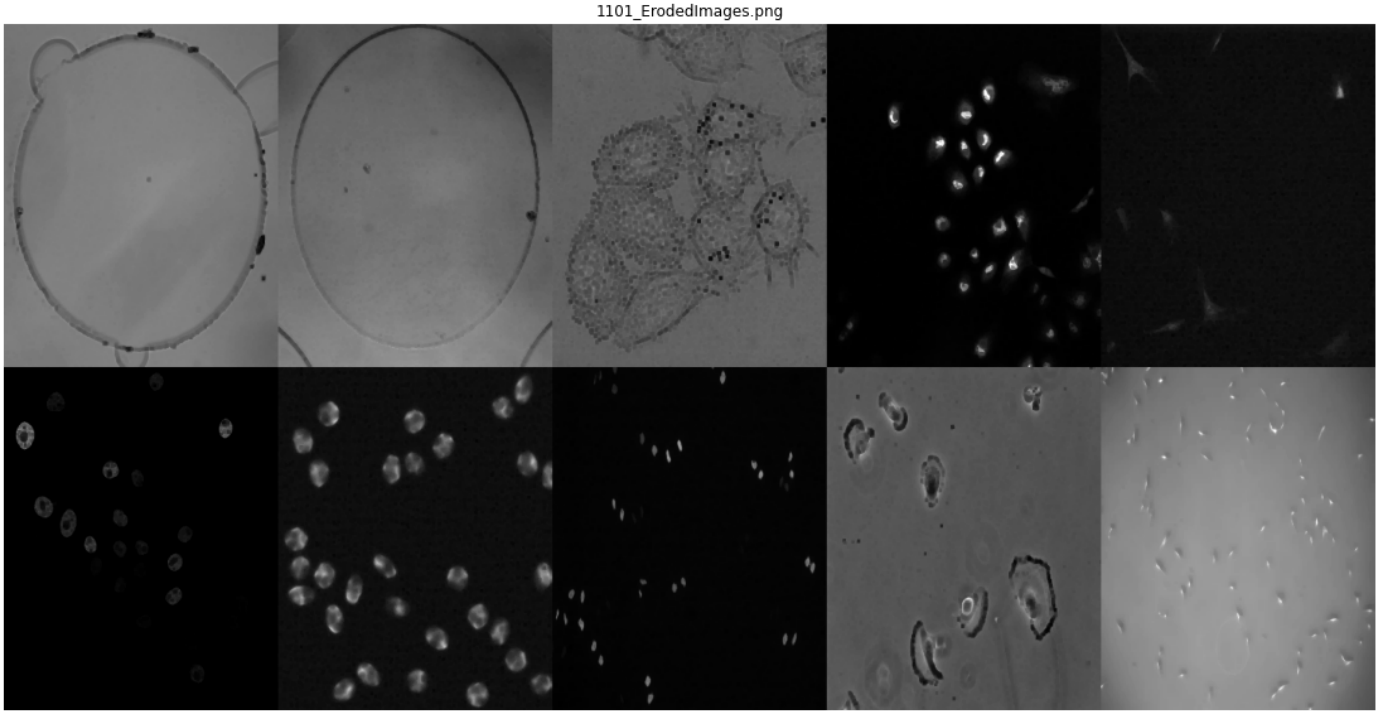
At this point, we have generated many images and tried a few processing options. Let us summarise and then consider next steps:

- Histogram Equalization is not helpful
- Smoothing an image slightly blurs the images, it barely removes any noise so this may not be necessary
- Thresholding is a promising option, particularly a Blurred Adaptive Thresholding Mean process
- Morphological Processing on the identified Blurred Adaptive Thresholding Mean images may improve results, but a global approach is challenging
- Morphological Processing at the beginning of the process preserves faint cell features, but it also preserves noise
- Morphological Processing at the end of the process is often worse, though 0902 was pleasant...
- 0902 and 1003 show great promise, so we should try further improve them
- Effort should also be made to just morphologically change the original images

Morphological Operations on Test Images

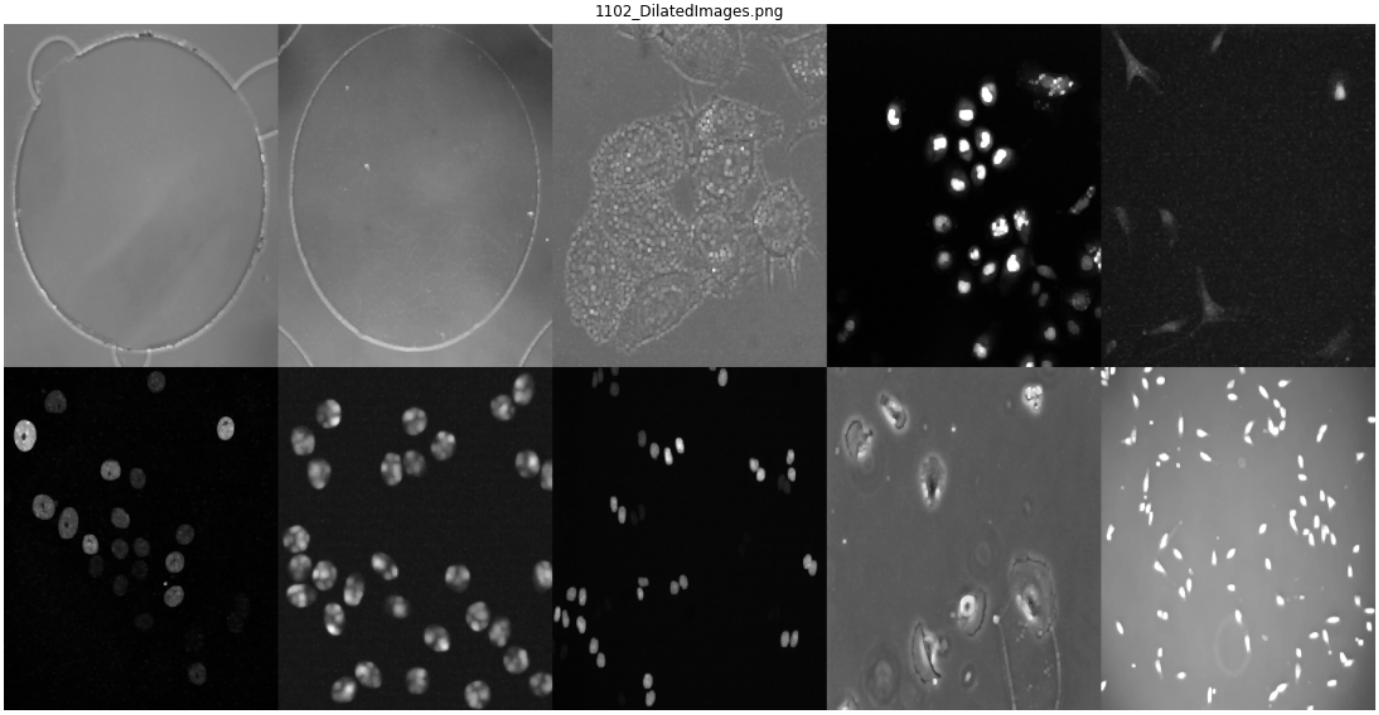
In [51]:
```
# Erosion

image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.erode(test_images[i], kernel, iterations=1)

    image_collection.append(img_erosion)

picName = "1101_ErodedImages.png"
saveAndShow(desired_directory, image_collection, picName)
```

1101_ErodedImages.png



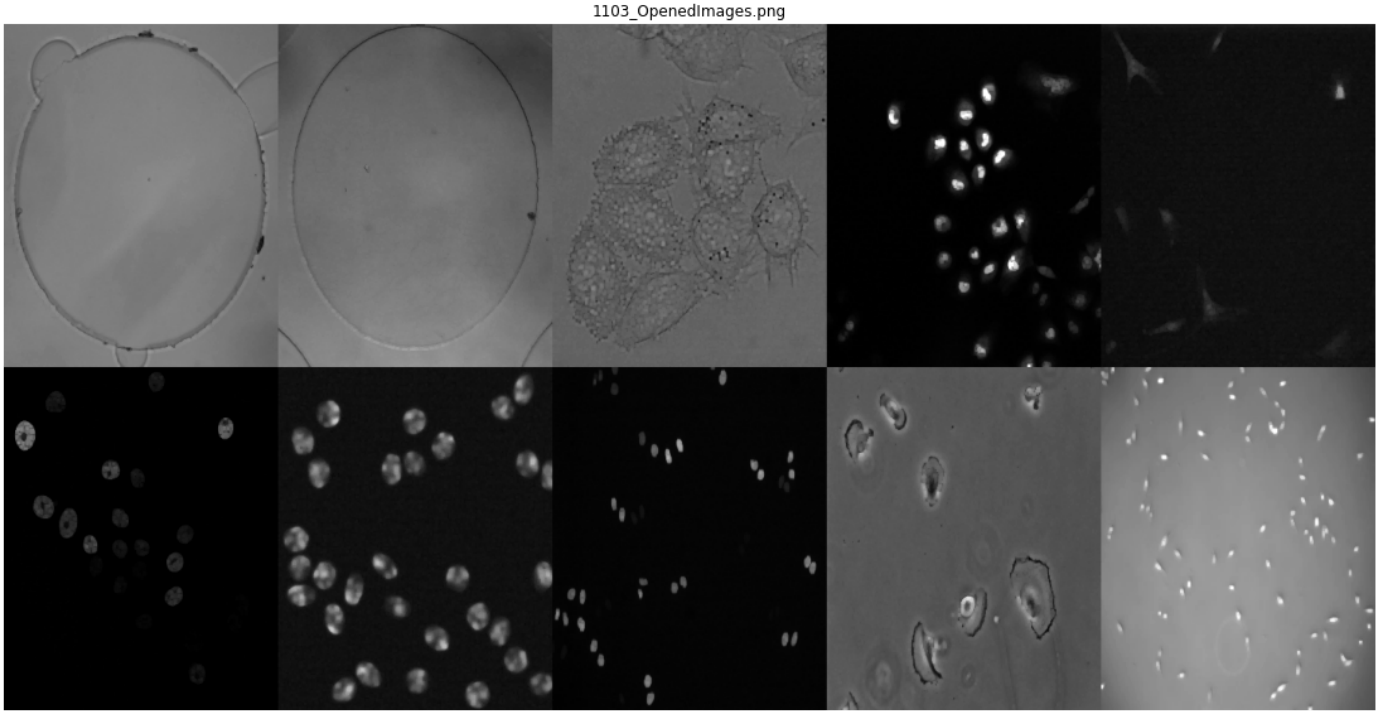
In [47]:
```
# Dilation

image_collection = []

for i in range(len(test_images)):
    img_dilate = cv2.dilate(test_images[i], kernel, iterations=1)

    image_collection.append(img_dilate)

picName = "1102_DilatedImages.png"
saveAndShow(desired_directory, image_collection, picName)
```

1102_DilatedImages.png



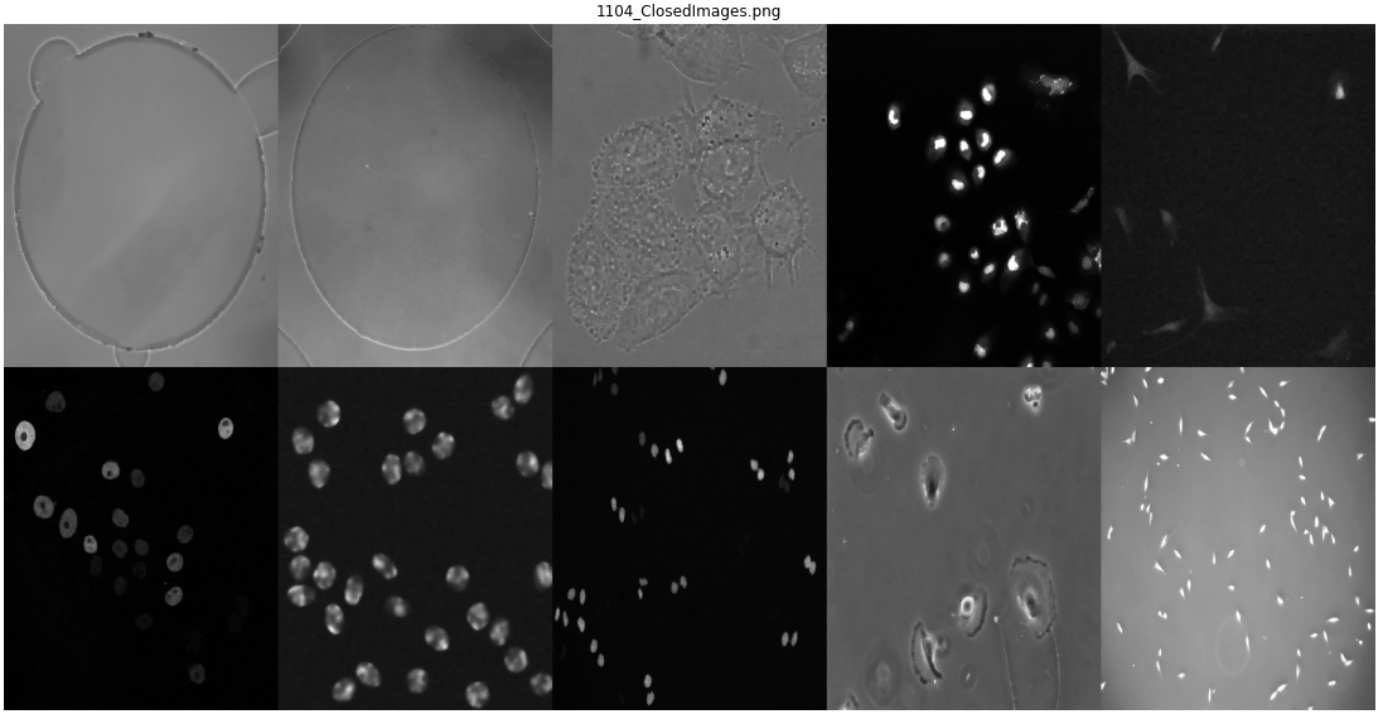
In [48]:
```
# Opening

image_collection = []

for i in range(len(test_images)):
    img_erode = cv2.erode(test_images[i], kernel, iterations=1)
    img_opened = cv2.dilate(img_erode, kernel, iterations=1)

    image_collection.append(img_opened)

picName = "1103_OpenedImages.png"
saveAndShow(desired_directory, image_collection, picName)
```

1103_OpenedImages.png



In [49]:
```
# Closing

image_collection = []

for i in range(len(test_images)):
    img_dilate = cv2.dilate(test_images[i], kernel, iterations=1)
    img_closed = cv2.erode(img_dilate, kernel, iterations=1)

    image_collection.append(img_closed)

picName = "1104_ClosedImages.png"
saveAndShow(desired_directory, image_collection, picName)
```

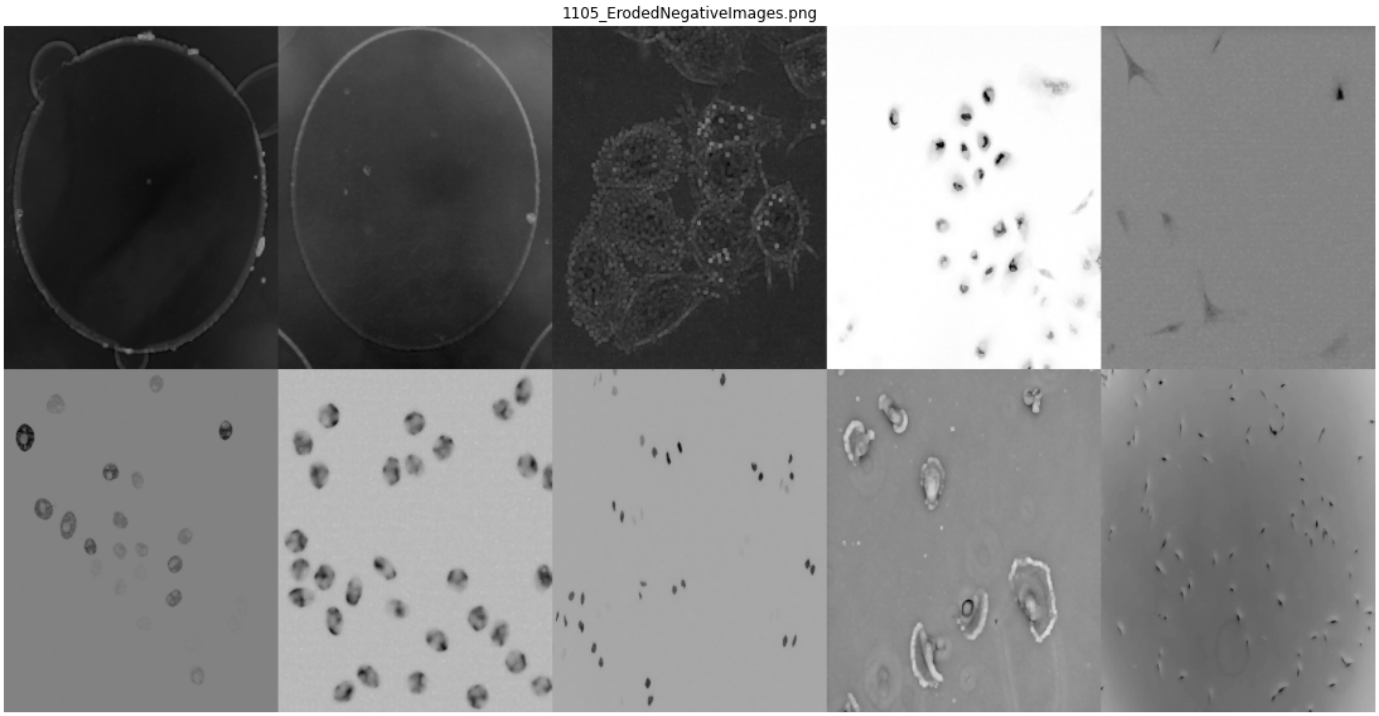
1104_ClosedImages.png



1101 and 1103 show promise, which may correspond to why 1001 is pleasant. What does the negative of each produce?

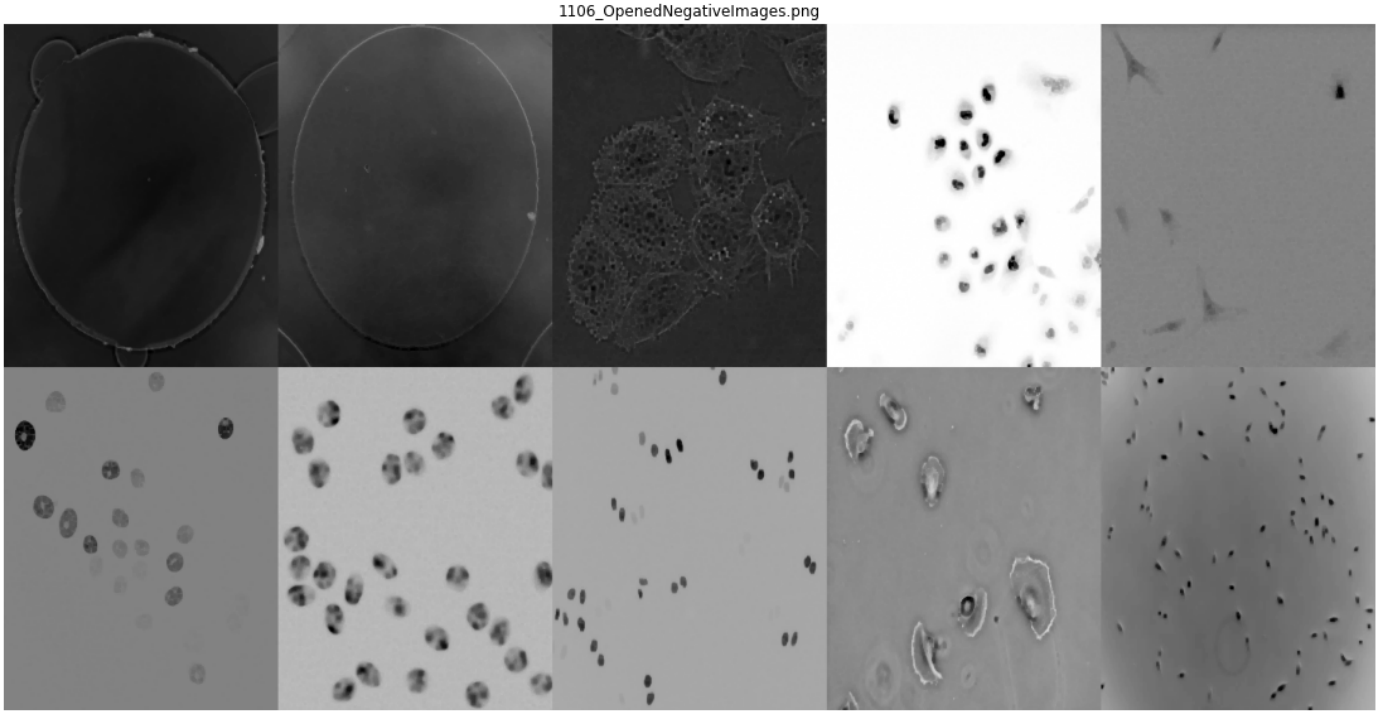
In [52]:
```
# Erosion Negative
```

```
image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.erode(test_images[i], kernel, iterations=1)

    max_val = np.amax(img_erosion)
    neg_img = max_val - img_erosion

    image_collection.append(neg_img)

picName = "1105_ErodedNegativeImages.png"
saveAndShow(desired_directory, image_collection, picName)
```



In [53]:
```
# Opening Negative

image_collection = []

for i in range(len(test_images)):
    img_erode = cv2.erode(test_images[i], kernel, iterations=1)
    img_opened = cv2.dilate(img_erode, kernel, iterations=1)

    max_val = np.amax(img_opened)
    neg_img = max_val - img_opened

    image_collection.append(neg_img)

picName = "1106_OpenedNegativeImages.png"
saveAndShow(desired_directory, image_collection, picName)
```

1106_OpenedNegativeImages.png

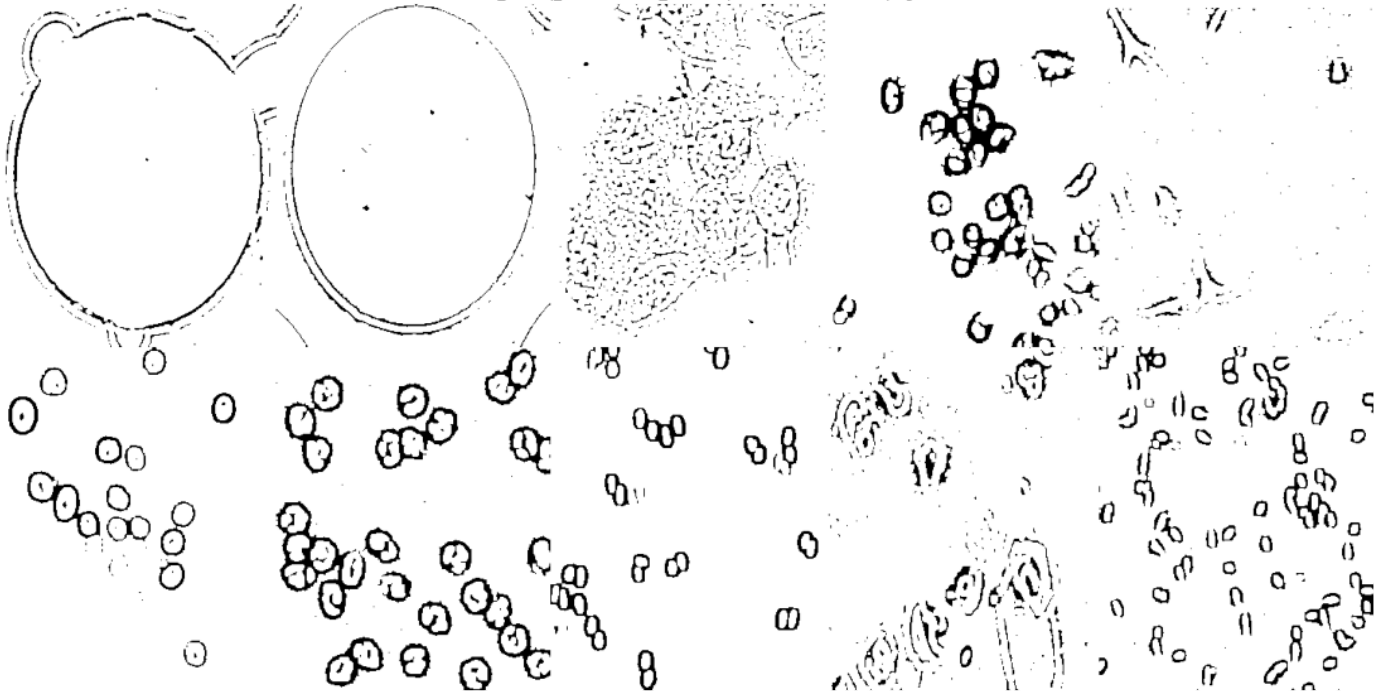


1105 and 1106 look super promising too! Though we may use them for identifying hidden cells to the user, the watershed algorithm may need to non-negative images to perform well....

Histogram Equalization on subset

In [69]:
```
# 0902 + Histogram Equalization

image_collection = []

for i in range(len(test_images)):
    blur = cv2.GaussianBlur(test_images[i], (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    max_val = np.amax(thresh_img)
    # get negative
    negative = max_val - thresh_img
    img_erosion = cv2.erode(negative, kernel, iterations=1)
    # get positive
    positive = max_val + img_erosion

    hist_eq_img = histEqualization(positive)

    image_collection.append(hist_eq_img)

picName = "1201_HistEq_ErodedAtEnd_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

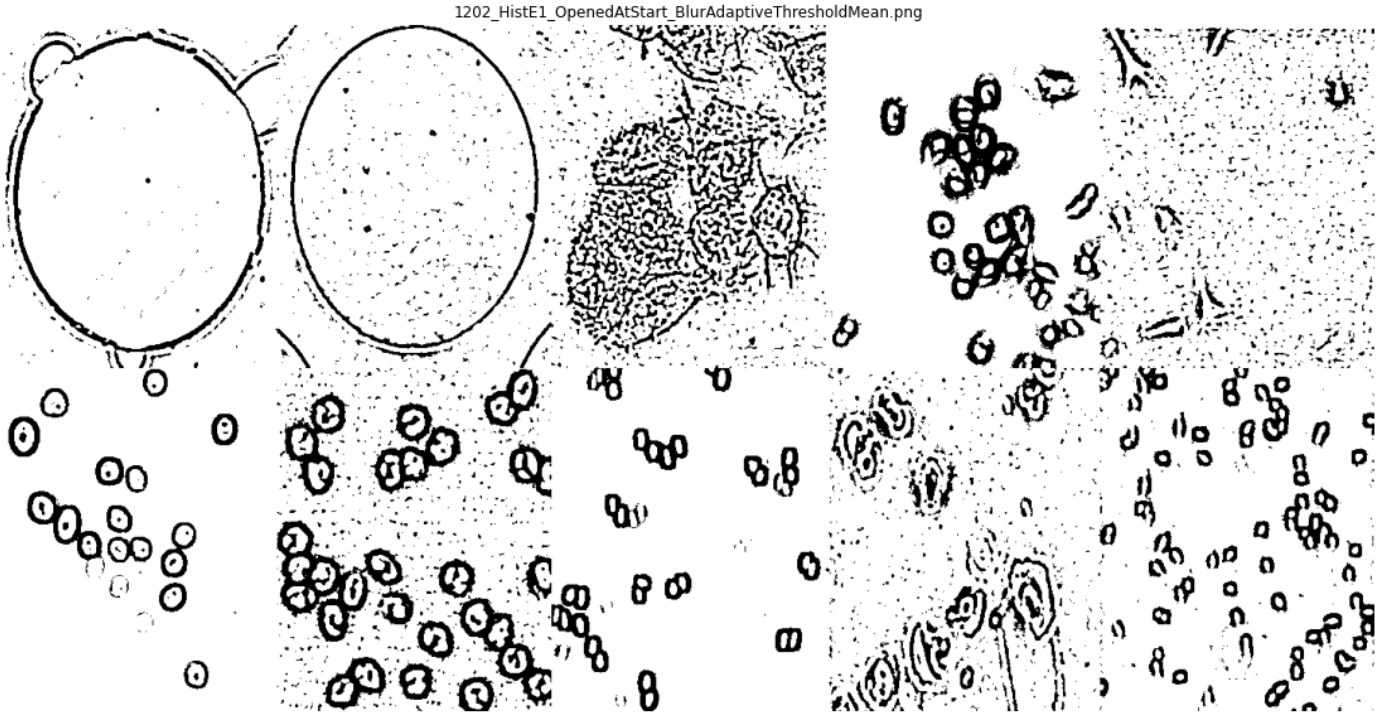
1201_HistEq_ErodedAtEnd_BlurAdaptiveThresholdMean.png



In [70]:

```
# 1003 + Histogram Equalization

image_collection = []

for i in range(len(test_images)):
    img_erosion = cv2.erode(test_images[i], kernel, iterations=1)
    img_opened = cv2.dilate(img_erosion, kernel, iterations=1)

    blur = cv2.GaussianBlur(img_opened, (5, 5), 0)
    thresh_img = adaptiveThresholdingMean(blur)

    hist_eq_img = histEqualization(thresh_img)

    image_collection.append(hist_eq_img)

picName = "1202_HistE1_OpenedAtStart_BlurAdaptiveThresholdMean.png"
saveAndShow(desired_directory, image_collection, picName)
```

1202_HistE1_OpenedAtStart_BlurAdaptiveThresholdMean.png



Initial Conclusion

This part of the notebook summarises the results

1105, 1106, 1201 and 1202 are useful to show the user the hidden information in the images.

0801 and 0802 are super close at being useful, however, they posses undesirable features

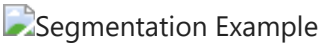
0805, 1101 and 1103 may be useful for segmentation

It may be worth attempting to try introduce an AND gate between 2 images, in order to extract only the common information. Though this may preserve noise...

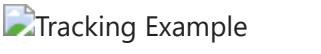
This notebook attempted to process the images, using several techniques from Image Processing. Unfortunately, we did appear to make a lot of headway (though we did exhaust some options). To get an idea of what we are hoping to achieve, let us open 2 images from one of our reference materials: Magnussen (PhD Thesis)

The images will be available in the COMP700 Proposal, if not provided by the author

Segmentation Example



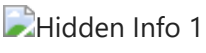
Tracking Example Below:

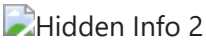


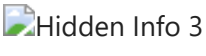
As we can see above, in the segmentation example from Magnussen, (b) is a desirable Processing step before segmentation

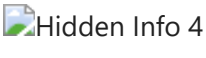
At the moment, we have the following desirable affects:

Hidden Info present (1105, 1106, 1201, 1202):

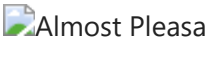


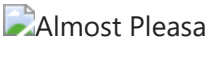




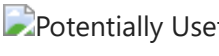


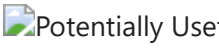
Almost pleasant processed Images (0801, 0802):





Potential Useful Images (0805, 1101, 1103):

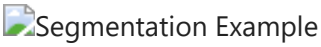




As we can see from the 3 pictures above, The processing we are attempting to produce is not of the same quality of Magnussen YET

One of the challenges we are facing is to find a processing step that assists all 10 datasets, because if we only focussed on 1 dataset we would be able to pick a specific processing option...

For ease of reference, bottom left of the images above corresponds to Magnussen:



Conclusion

This section of the notebook concludes the thoughts so far

This notebook is becoming rather long, and we have generated ~45 pictures already

Let us create a second Notebook to continue the Pre-Processing steps, and try recreate the results from Magnussen there