SWTLoc

Release 2.1.0

Achintya Gupta

Mar 09, 2022

INTRODUCTION

CHAPTER

ONE

SWTLOC : STROKE WIDTH TRANSFORM TEXT LOCALIZER

1.1 Description

This repo contains a python implementation structured as a python package pertaining to the text localization method as in a natural image as outlayed in the Research Paper :-

Detecting Text in Natural Scenes with Stroke Width Transform. Boris Epshtein, Eyal Ofek & Yonatan Wexler (June, 2010)

This library extends the transformation stage of the image for textual content by giving the ability to :

- Localize `Letter`'s : through `SWTImage.localizeLetters`
- Localize `Words`'s, via fusing individual `Letter`'s : through `SWTImage.localizeWords`

The process flow of is depicted in the image below :

1.2 Installation

To install `swtloc`, simply :

```
pip install swtloc
```

1.3 Speed Benchmarking

Below is the speed comparison between different versions of `SWTLoc` and their various engines. The time measured for each test image was calculated based on 10 iterations of 10 runs each. Test Images can be found in `examples/images/` folder in this repository, and the code for generating the below table can be found in - Improvements-in-v2.0.0.ipynb notebook in `examples/` folder.

Test Image	SWT v1.1.1 (Python)	SWT v1.1.1 (Python) [x]	SWT v2.0.0 (Python)	SWT v2.0.0 (Python) [x]	SWT v2.0.0 (numba)	SWT v2.0.0 (numba) [x]
test_img1.jpg	16.929 seconds	1.0x	8.145 seconds	2.078x	0.33 seconds	51.315x
test_img2.jpg	10.107 seconds	1.0x	4.205 seconds	2.404x	0.678 seconds	50.904x
test_img3.jpg	4.545 seconds	1.0x	2.701 seconds	1.683x	0.082 seconds	55.625x
test_img4.jpeg	7.626 seconds	1.0x	3.992 seconds	1.91x	0.142 seconds	53.859x
test_img5.jpg	17.071 seconds	1.0x	7.554 seconds	2.26x	0.302 seconds	56.62x
test_img6.jpg	4.973 seconds	1.0x	3.104 seconds	1.602x	0.094 seconds	53.076x

CHAPTER

TWO

FREQUENTLY USED CODE SNIPPETS

2.1 Performing Stroke Width Transformation

```
# Installation
# !pip install swtloc

# Imports
import swtloc as swt
# Image Path
imgpath = 'examples/images/test_image_5/test_img5.jpg'
# Result Path
respath = 'examples/images/test_image_5/usage_results/'
# Initializing the SWTLocalizer class with the image path
swtl = swt.SWTLocalizer(image_paths=imgpath)
# Accessing the SWTImage Object which is housing this image
swtImgObj = swtl.swtimages[0]
# Performing Stroke Width Transformation
swt_mat = swtImgObj.transformImage(text_mode='db_lf')
```

2.2 Localizing & Annotating Letters and Generating Crops of Letters

```
# Installation
# !pip install swtloc

# Imports
import swtloc as swt
from cv2 import cv2
import numpy as np
# Image Path
imgpath = 'examples/images/test_image_1/test_img1.jpg'
# Read the image
img = cv2.imread(imgpath)
# Result Path
respath = 'examples/images/test_image_1/usage_results/'
# Initializing the SWTLocalizer class with a pre loaded image
swtl = swt.SWTLocalizer(images=img)
swtImgObj = swtl.swtimages[0]
# Perform Stroke Width Transformation
```

(continues on next page)

(continued from previous page)

```
swt_mat = swtImgObj.transformImage(text_mode='db_lf',
                                   maximum_angle_deviation=np.pi/2,
                                   gaussian_blurr_kernel=(11, 11),
                                   minimum_stroke_width=5,
                                   maximum_stroke_width=50,
                                   display=False)  # NOTE: Set display=True
# Localizing Letters
localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
                                              maximum_pixels_per_cc=5200)
letter_labels = [int(k) for k in list(localized_letters.keys())]
```

```
# Some Other Helpful Letter related functions
# # Query a single letter
from swtloc.configs import (IMAGE_ORIGINAL,
                            IMAGE_SWT_TRANSFORMED)
loc_letter, swt_loc, orig_loc = swtImgObj.getLetter(key=letter_labels[5])

# # Iterating over all the letters
# # Specifically useful for jupyter notebooks - Iterate over all
# # the letters, at the same time visualizing the localizations
letter_gen = swtImgObj.letterIterator()
loc_letter, swt_loc, orig_loc = next(letter_gen)

# # Generating a crop of a single letter on any of the available
# # image codes.
# # Crop on SWT Image
swtImgObj.saveCrop(save_path=respath,crop_of='letters',crop_key=6, crop_on=IMAGE_SWT_
↪TRANSFORMED, crop_type='min_bbox')
# # Crop on Original Image
swtImgObj.saveCrop(save_path=respath,crop_of='letters',crop_key=6, crop_on=IMAGE_
↪ORIGINAL, crop_type='min_bbox')
```

2.3 Localizing & Annotating Words and Generating Crops of Words

```
# Installation
# !pip install swtloc
# Imports
import swtloc as swt
# Image Path
imgpath = 'images/test_img2/test_img2.jpg'
# Result Path
respath = 'images/test_img2/usage_results/'
# Initializing the SWTLocalizer class with the image path
swtl = swt.SWTLocalizer(image_paths=imgpath)
swtImgObj = swtl.swtimages[0]
# Perform Stroke Width Transformation
swt_mat = swtImgObj.transformImage(maximum_angle_deviation=np.pi/2,
                                   gaussian_blurr_kernel=(9, 9),
                                   minimum_stroke_width=3,
```

(continues on next page)

(continued from previous page)

```
                                    maximum_stroke_width=50,
                                    include_edges_in_swt=False,
                                    display=False)  # NOTE: Set display=True

# Localizing Letters
localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=400,
                                              maximum_pixels_per_cc=6000,
                                              display=False)  # NOTE: Set display=True

# Calculate and Draw Words Annotations
localized_words = swtImgObj.localizeWords(display=True)  # NOTE: Set display=True
word_labels = [int(k) for k in list(localized_words.keys())]
```

```
# Some Other Helpful Words related functions
# # Query a single word
from swtloc.configs import (IMAGE_ORIGINAL,
                            IMAGE_SWT_TRANSFORMED)
loc_word, swt_loc, orig_loc = swtImgObj.getWord(key=word_labels[8])

# # Iterating over all the words
# # Specifically useful for jupyter notebooks - Iterate over all
# # the words, at the same time visualizing the localizations
word_gen = swtImgObj.wordIterator()
loc_word, swt_loc, orig_loc = next(word_gen)

# # Generating a crop of a single word on any of the available
# # image codes
# # Crop on SWT Image
swtImgObj.saveCrop(save_path=respath, crop_of='words', crop_key=9, crop_on=IMAGE_SWT_
↪TRANSFORMED, crop_type='bubble')
# # Crop on Original Image
swtImgObj.saveCrop(save_path=respath, crop_of='words', crop_key=9, crop_on=IMAGE_
↪ORIGINAL, crop_type='bubble')
```

CHAPTER

THREE

USAGE

These code blocks can be found in SWTloc-Usage-[v2.0.0-onwards].ipynb notebook in `examples/`.

3.1 Initialisation of `SWTLocalizer`

- Initialising the - This is the entry point, which can accept either single image path (str)/ multiple image paths (List[str])/ single image (np.ndarray)/ multiple images (List[np.ndarray]).

```
from swtloc import SWTLocalizer
imgpath = 'images/test_img4/test_img4.jpeg'
respath = 'images/test_img4/usage_results/'
swtl = SWTLocalizer(image_paths=imgpath)
swtImgObj = swtl.swtimages[0]
print(swtImgObj, type(swtImgObj))
swtImgObj.showImage()
```

SWTImage-test_img4 <class 'swtloc.abstractions.SWTImage'>

3.2 Stroke Width Transformation using `SWTImage.transformImage`

```
swt_mat = swtImgObj.transformImage(text_mode='lb_df',
                                   auto_canny_sigma=1.0,
                                   maximum_stroke_width=20)
```

3.3 Localizing Letters using `SWTImage.localizeLetters`

```
localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=100,
                                              maximum_pixels_per_cc=10_000,
                                              acceptable_aspect_ratio=0.2)
letter_labels = list([int(k) for k in localized_letters.keys()])
```

3.4 Query a Letter using SWTImage.getLetter

```
letter_label = letter_labels[3]
locletter = swtImgObj.getLetter(key=letter_label)
```

3.5 Localizing Words using SWTImage.localizeWords

```
localized_words = swtImgObj.localizeWords()
word_labels = list([int(k) for k in localized_words.keys()])
```

3.6 Query a Word using SWTImage.getWord

```
word_label = word_labels[12]
locword = swtImgObj.getWord(key=word_label)
```

3.7 Accessing intermediary stage images using SWTImage.showImage and saving them

```
from swtloc.configs import (IMAGE_ORIGINAL,
                            IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS)
swtImgObj.showImage(image_codes=[IMAGE_ORIGINAL,
                                 IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS],
                    plot_title='Original & Bubble Mask')
```

3.8 Saving Crops of the localized letters and words

```
from swtloc.configs import (IMAGE_ORIGINAL,
                            IMAGE_SWT_TRANSFORMED)
# Letter Crops
swtImgObj.saveCrop(save_path=respath, crop_of='letters', crop_key=4, crop_type='min_bbox
↪', crop_on=IMAGE_ORIGINAL)
swtImgObj.saveCrop(save_path=respath, crop_of='letters', crop_key=4, crop_type='min_bbox
↪', crop_on=IMAGE_SWT_TRANSFORMED)

# Word Crops
swtImgObj.saveCrop(save_path=respath, crop_of='words', crop_key=13, crop_type='bubble',␣
↪crop_on=IMAGE_ORIGINAL)
swtImgObj.saveCrop(save_path=respath, crop_of='words', crop_key=13, crop_type='bubble',␣
↪crop_on=IMAGE_SWT_TRANSFORMED)
```

Letter Crops

Word Crops

```
import os
import numpy as np
import numba as nb
import pandas as pd
from cv2 import cv2
import swtloc as swt
from platform import python_version
```

```
from swtloc import SWTLocalizer
from swtloc.configs import (IMAGE_ORIGINAL,
                            IMAGE_GRAYSCALE,
                            IMAGE_EDGED,
                            IMAGE_SWT_TRANSFORMED,
                            IMAGE_CONNECTED_COMPONENTS_1C,
                            IMAGE_CONNECTED_COMPONENTS_3C,
                            IMAGE_CONNECTED_COMPONENTS_3C_WITH_PRUNED_ELEMENTS,
                            IMAGE_CONNECTED_COMPONENTS_PRUNED_1C,
                            IMAGE_CONNECTED_COMPONENTS_PRUNED_3C,
                            IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS,
                            IMAGE_ORIGINAL_LETTER_LOCALIZATIONS,
                            IMAGE_ORIGINAL_MASKED_LETTER_LOCALIZATIONS,
                            IMAGE_PRUNED_3C_WORD_LOCALIZATIONS,
                            IMAGE_ORIGINAL_WORD_LOCALIZATIONS,
                            IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS,
                            IMAGE_INDIVIDUAL_LETTER_LOCALIZATION,
                            IMAGE_ORIGINAL_INDIVIDUAL_LETTER_LOCALIZATION,
                            IMAGE_INDIVIDUAL_WORD_LOCALIZATION,
                            IMAGE_ORIGINAL_INDIVIDUAL_WORD_LOCALIZATION)
from swtloc.utils import image_1C_to_3C
from swtloc.utils import show_N_images
from swtloc.configs import get_code_descriptions
from swtloc.configs import CODE_NAME_VAR_MAPPINGS
```

```
rawimage_path = 'images/'
```

```
img_paths = []
res_path = []
img_names = []
img_text_modes = ['db_lf', 'lb_df', 'db_lf', 'lb_df', 'db_lf', 'db_lf']

for each_img in [k for k in os.listdir(rawimage_path) if 'test' in k]:
    _ifolder_path = rawimage_path+each_img
    _iname = [k for k in os.listdir(_ifolder_path) if '.' in k][0]
    _img_path = _ifolder_path+'/'+_iname
    img_paths.append(_img_path)
    img_names.append(_iname)
    res_path.append(_ifolder_path+'/usage_results/')
```

CHAPTER

FOUR

SWTLOCALIZER CLASS

`SWTLocalizer` class acts as an entry point for performing transformations to the images. It can accept

- Single Image Path
- Multiple Image Paths
- Single Pre-Loaded Image
- Multiple Pre-Loaded Images

Once the `SWTLocalizer` object is instantiated, the attribute `swtimages` will be populated with all the `SWTImage` objects corresponding to each input image.

Instantiating `SWTLocalizer` with Mixed Input will raise an error

```
swtl = SWTLocalizer(images=[img_paths[0]]+[cv2.imread(img_paths[1])])
print(swtl.swtimages)
```

```
---------------------------------------------------------------------------

SWTLocalizerValueError                    Traceback (most recent call last)

Input In [5], in <module>
----> 1 swtl = SWTLocalizer(images=[img_paths[0]]+[cv2.imread(img_paths[1])])
      2 print(swtl.swtimages)


File D:\Personal Stuff\swtloc-project\venvs\py39VenvDev\lib\site-packages\swtloc\
↪swtlocalizer.py:173, in SWTLocalizer.__init__(self, multiprocessing, images, image_
↪paths)
    170 self.swtimages: List[SWTImage] = []
    172 # Sanity Checks
--> 173 res_pack = self._sanityChecks(images=images, image_paths=image_paths)
    174 transform_inputs, transform_input_flags, transform_input_image_names = res_pack
    176 # Instantiate each transform_input as SWTImage


File D:\Personal Stuff\swtloc-project\venvs\py39VenvDev\lib\site-packages\swtloc\
↪swtlocalizer.py:254, in SWTLocalizer._sanityChecks(images, image_paths)
    251 for each_image in images:
    252     # Check if all the elements in the list are images
    253     if not isinstance(each_image, np.ndarray):
--> 254         raise SWTLocalizerValueError(
```

(continues on next page)

(continued from previous page)

```
    255             "If a list is provided to `images`, each element should be an np.
↪ndarray")
    256     # Check if its whether 3d or 1d image
    257     if not (len(each_image.shape) in [3, 2]):


SWTLocalizerValueError: If a list is provided to `images`, each element should be an np.
↪ndarray
```

Instantiating `SWTLocalizer` with Single Pre-Loaded Image

```
single_image = cv2.imread(img_paths[0])
swtl = SWTLocalizer(images=single_image)
print(swtl.swtimages)
```

```
[SWTImage-SWTImage_982112]
```

Instantiating `SWTLocalizer` with Multiple Pre-Loaded Images

```
multiple_images = [cv2.imread(each_path) for each_path in img_paths]
swtl = SWTLocalizer(images=multiple_images)
print(swtl.swtimages)
```

```
[SWTImage-SWTImage_982112, SWTImage-SWTImage_571388, SWTImage-SWTImage_866821, SWTImage-
↪SWTImage_182401, SWTImage-SWTImage_241787, SWTImage-SWTImage_631871]
```

Instantiating `SWTLocalizer` with Single Image Path

```
swtl = SWTLocalizer(image_paths=img_paths[0])
print(swtl.swtimages)
```

```
[SWTImage-test_img1]
```

Instantiating `SWTLocalizer` with Multiple Image Paths

```
swtl = SWTLocalizer(image_paths=img_paths)
print(swtl.swtimages)
```

```
[SWTImage-test_img1, SWTImage-test_img2, SWTImage-test_img3, SWTImage-test_img4,␣
↪SWTImage-test_img5, SWTImage-test_img6]
```

Once the `swtimages` attribute of the `SWTLocalizer` class has been populated with `SWTImage` object, to perform operations like *transforming*, *pruning*, *localization* etc, grab any of the prepared `SWTImage` object and access its functions for performing the said operations

```
swtImgObj = swtl.swtimages[0]
swtImgObj
```

```
SWTImage-test_img1
```

CHAPTER

FIVE

SWT TRANSFORMATION

This section is devoted to explaining the parameters which are available in the `SWTImage.transformImage` function

- text_mode
- engine
- gaussian_blurr
- gaussian_blurr_kernel
- edge_function
- auto_canny_sigma
- minimum_stroke_width
- maximum_stroke_width
- check_angle_deviation
- maximum_angle_deviation
- include_edges_in_swt
- display

Image Codes which become available after running of `SWTImage.transfromImage` function

Image-Code	Description
IMAGE_ORIGINAL	Original Image
IMAGE_GRAYSCALE	Gray-Scaled Image
IMAGE_EDGED	Edge Image
IMAGE_SWT_TRANSFORMED	SWT Transformed Image

Our Muse () for this Section

```
swtImgObj0 = swtl.swtimages[0]
swtImgObj0.showImage()
```

5.1 SWTImage.transform.text_mode

Text Mode Parameter [default = 'lb_df']

This is image specific, but an extremely important parameter that takes one of the two value :-

- db_lf :> **D**ark **B**ackground **L**ight **F**oreground i.e Light color text on Dark color background
- lb_df :> **L**ight **B**ackground **D**ark **F**oreground i.e Dark color text on Light color background

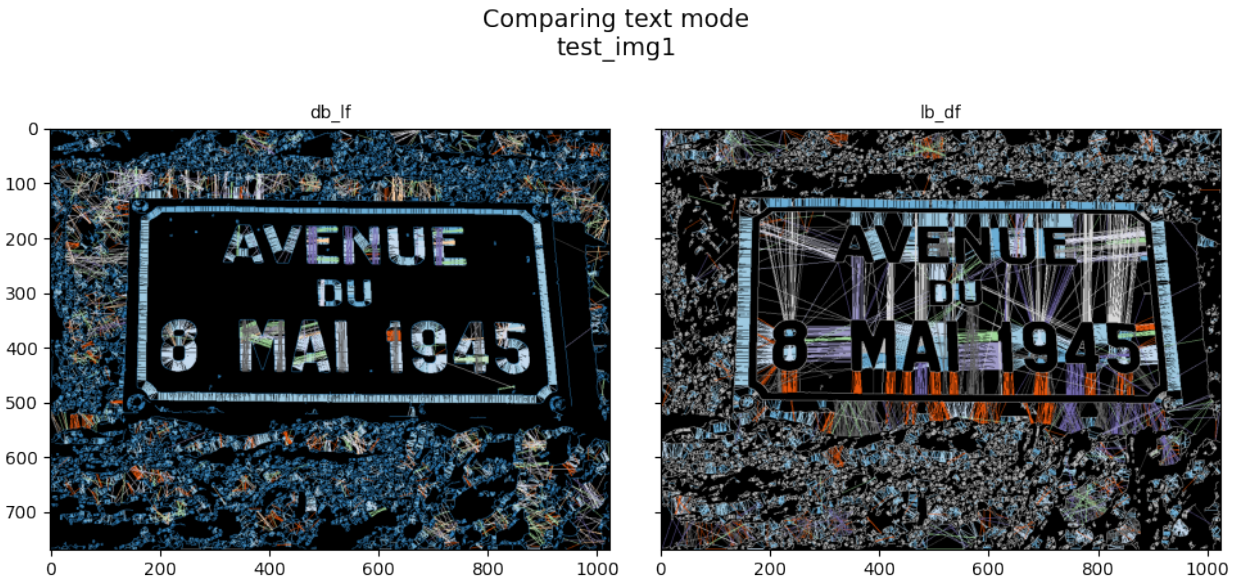
This parameters affect how the gradient vectors (the direction) are calulated, since gradient vectors of **db_lf** are in $-ve$ direction to that of **lb_df** gradient vectors

For the image below, the `text_mode` parameter must be "db_lf"

```
swt_mat = swtImgObj0.transformImage(text_mode='db_lf')
```

If we were to use text_mode = 'lb_df' (*default value*) in the above image, the strokes would have been made in the direction opposite to what we would want :

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', display=False)
swt_mat2 = swtImgObj0.transformImage(text_mode='lb_df', display=False)

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=['db_lf', 'lb_df'],
              plot_title='Comparing text mode\n',
              sup_title=swtImgObj0.image_name)
```



For the image below, the `text_mode` parameter must be “lb_df”

```
swtImgObj1 = swtl.swtimages[1]
swt_mat = swtImgObj1.transformImage(text_mode='lb_df')
swtImgObj1._resetSWTTransformParams()
```

5.2 SWTImage.transform.engine

Engine Parameter [default='numba']

This parameter was added from v2.0.0 onwards, there are two available engines for stroke width transformation

- `engine="numba"` : Numba No-Python, jit compilation to compine the `findStrokes` function (in core.py) to machine code, hence enhancing the speed of transformation
- `engine="python"` : Vanilla Python

```
%%timeit -n 10 -r 10
# Speed Benchmarking using numba engine
_=swtImgObj0.transformImage(text_mode='db_lf', display=False)
```

```
232 ms ± 11.6 ms per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

```
%%timeit -n 10 -r 10
# Speed Benchmarking using python engine
_=swtImgObj0.transformImage(text_mode='db_lf', engine='python', display=False)
```

```
5.33 s ± 159 ms per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

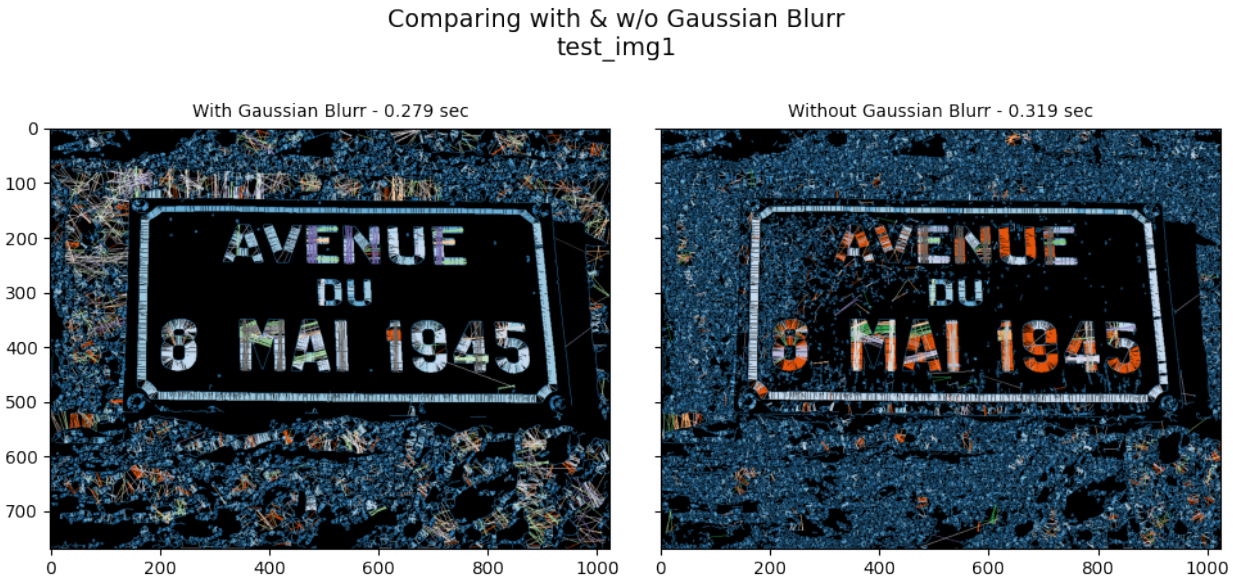
Clearly, there is nearly 50x speed enhancement while using `engine=`"numba", its also the default parameter. Also, while testing the same with other test images, similar speed improvements were found. You can find the Speed Benchmarking for other test images in `README.md`

5.3 SWTImage.transform.gaussian_blurr

Gaussian Blurr Parameter [default = True]

Whether to apply gaussian blurr or not.

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', gaussian_blurr=True,␣
↪display=False)
gs_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', gaussian_blurr=False,␣
↪display=False)
nogs_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'With Gaussian Blurr - {gs_time}', f'Without Gaussian␣
↪Blurr - {nogs_time}'],
              plot_title='Comparing with & w/o Gaussian Blurr\n',
              sup_title=swtImgObj0.image_name)
```

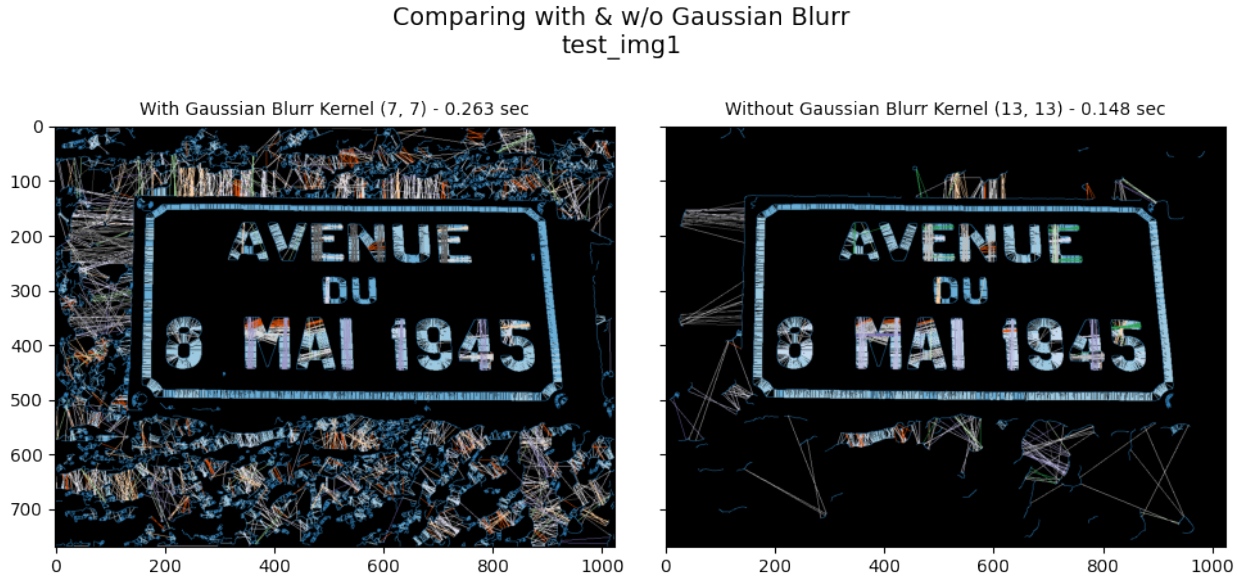


5.4 SWTImage.transform.gaussian_blurr_kernel

Gaussian Blurr Kernel Parameter [default = (5, 5)]

Kernel to use for gaussian blurring when `gaussian_blurr` parameter is set to True

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', gaussian_blurr=True,
                                     gaussian_blurr_kernel=(7, 7), display=False)
k1_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', gaussian_blurr=True,
                                     gaussian_blurr_kernel=(13, 13), display=False)
k2_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'With Gaussian Blurr Kernel (7, 7) - {k1_time}',
                                 f'Without Gaussian Blurr Kernel (13, 13) - {k2_time}'],
              plot_title='Comparing with & w/o Gaussian Blurr\n',
              sup_title=swtImgObj0.image_name)
```



5.5 SWTImage.transform.edge_function

Image Edging [default = 'ac']

Finding the Edge of the image is a tricky part, this is pertaining to the fact that in most of the cases the images we deal with are of not that standard that applying just a opencv Canny operator would result in the desired Edge Image.

Sometimes (In most cases) there is some custom processing required before edging, for that reason alone this function accepts one of the following two arguments :-

- 'ac' :> Auto-Canny function, an in-built function which will generate the Canny Image from the original image, internally calculating the threshold parameters, although, to tune it even further 'ac_sigma' parameter is provided which can take any value between 0.0 <–> 1.0 .

- *A custom function* : This function should have its signature as mentioned below :

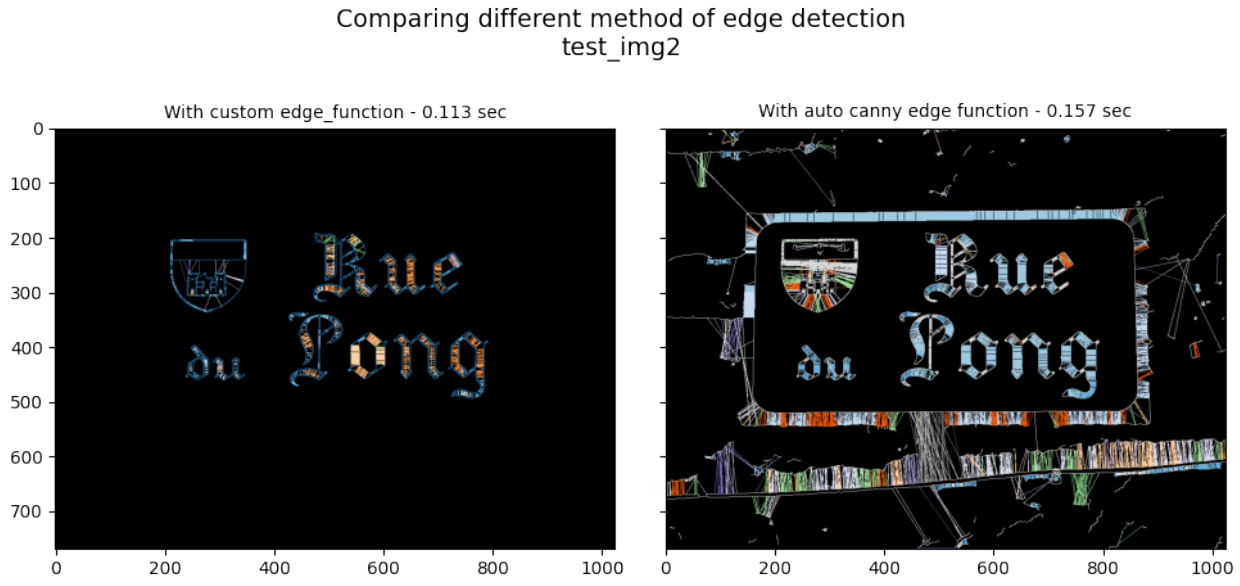
  def custom_edge_func(gauss_image):

```
Your Function Logic...

edge_image = ...

return edge_image
```

Comparing Auto Canny Sigma Values

```
def custom_edge_func(gray_image):
    gauss_image = cv2.GaussianBlur(gray_image, (5,5), 1)
    laplacian_conv = cv2.Laplacian(gauss_image, -1, (5,5))
    canny_edge = cv2.Canny(laplacian_conv, 20, 140)
    return canny_edge
```

```
swt_mat1 = swtImgObj1.transformImage(edge_function=custom_edge_func, display=False)
cef_time = swtImgObj1.transform_time
swt_mat2 = swtImgObj1.transformImage(edge_function='ac', display=False)
ac_time = swtImgObj1.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'With custom edge_function - {cef_time}',
                                 f'With auto canny edge function - {ac_time}'],
              plot_title='Comparing different method of edge detection\n',
              sup_title=swtImgObj1.image_name)
swtImgObj1._resetSWTTransformParams()
```

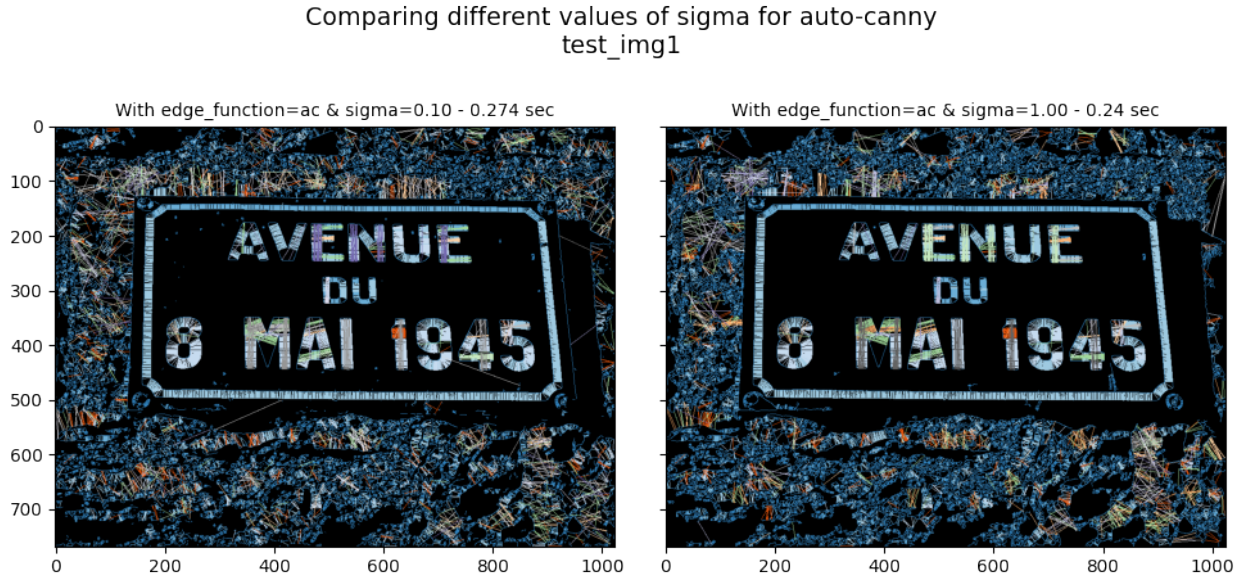


5.6 SWTImage.transform.auto_canny_sigma

Sigma Values of Auto Canny Edge Function [default = 0.33]

Sigma value of the Auto Canny Edge function, only applicable when edge_function = "ac"

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', auto_canny_sigma=0.1,␣
↪display=False)
acd_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', auto_canny_sigma=1.0,␣
↪display=False)
ac1_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'With edge_function=ac & sigma=0.10 - {acd_time}',
                                 f'With edge_function=ac & sigma=1.00 - {ac1_time}'],
              plot_title='Comparing different values of sigma for auto-canny\n',
              sup_title=swtImgObj.image_name)
```



5.7 SWTImage.transform.include_edges_in_swt

Parameter to include those edges from which no stroke was finalised [default = True]

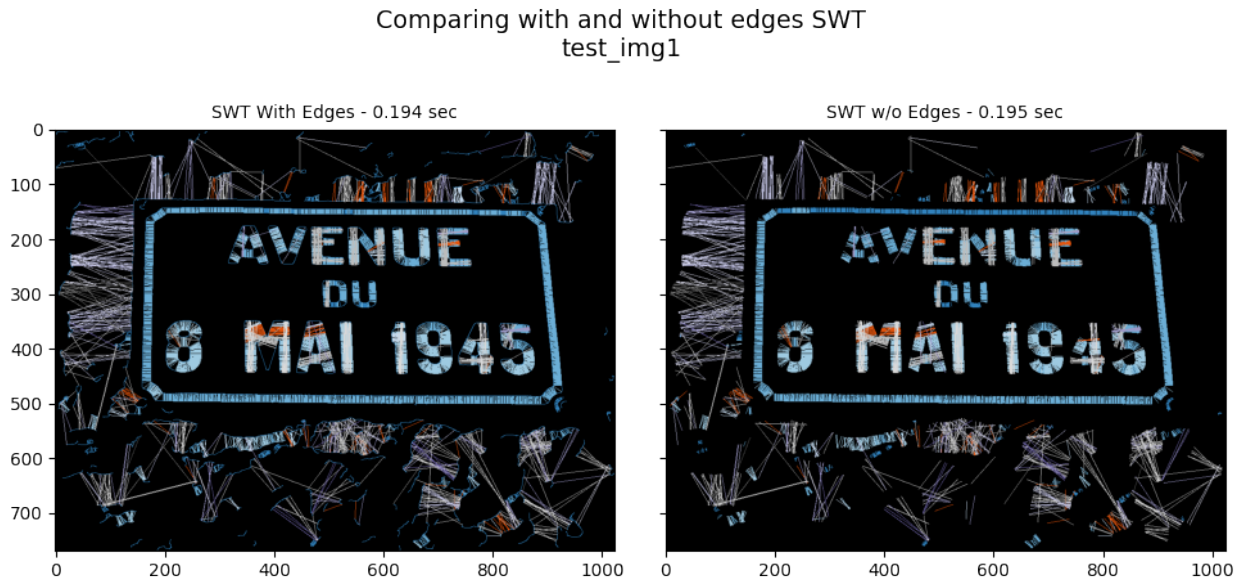
Not all edges end up generating an eligible strokes from them, this parameter gives an option whether to include those edges in or not. Edges hold the value=1 in the swt image

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', include_edges_in_swt=True,
                                     gaussian_blurr_kernel = (11, 11), display=False)
we_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', include_edges_in_swt=False,
                                     gaussian_blurr_kernel = (11, 11), display=False)
```

(continues on next page)

(continued from previous page)

```
woe_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'SWT With Edges - {we_time}',
                                 f'SWT w/o Edges - {woe_time}'],
              plot_title='Comparing with and without edges SWT\n',
              sup_title=swtImgObj0.image_name)
```



5.8 SWTImage.transform.minimum_stroke_width

Minimum Permissible Stroke Length Parameter [default = 3]

While looking for a eligibile stroke emanating from any edge co-ordinate in the image, this parameter governs the minimum stroke length below which the stroke will be disqualified.

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', minimum_stroke_width=10,
                                     include_edges_in_swt=False, display=False)
acd_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', minimum_stroke_width=20,
                                     include_edges_in_swt=False, display=False)
ac1_time = swtImgObj0.transform_time
```

```
np.unique(swt_mat1)
```

```
array([  0,  10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  20,  21,
        22,  23,  24,  25,  26,  27,  28,  29,  30,  31,  32,  33,  34,
        35,  36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,
        48,  49,  50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,
        61,  62,  63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,
        74,  75,  76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,
```

(continues on next page)

(continued from previous page)

```
        89,  91,  92,  93,  94,  95,  98, 100, 101, 102, 104, 106, 107,
       109, 110, 112, 113, 114, 115, 116, 118, 153, 175, 185])
```

```
np.unique(swt_mat2)
```

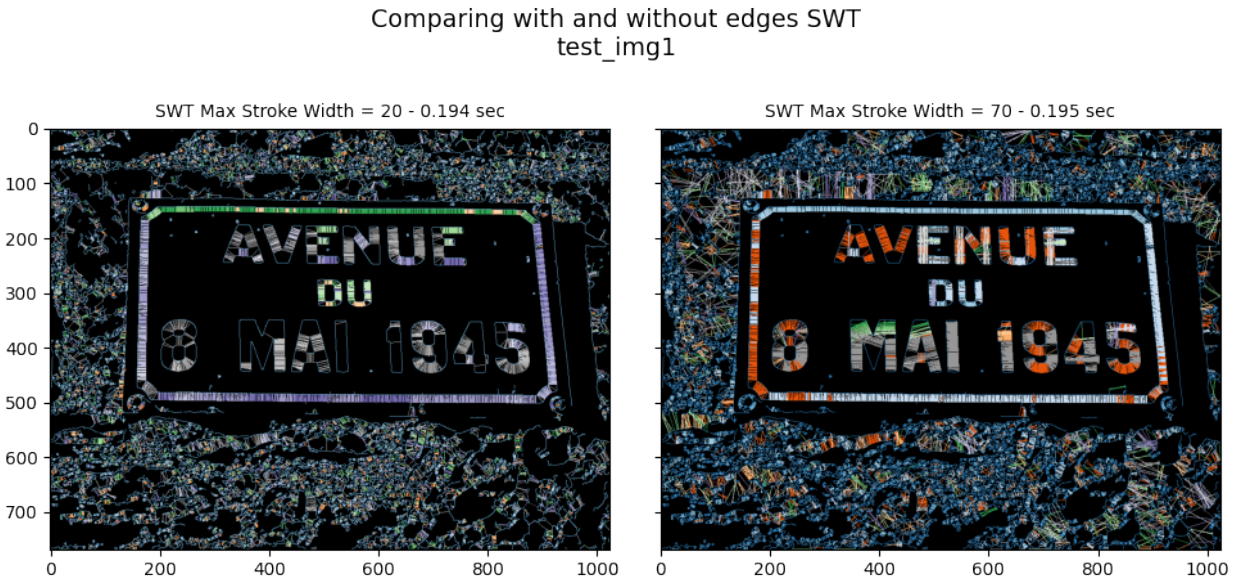
```
array([  0,  20,  21,  22,  23,  24,  25,  26,  27,  28,  29,  30,  31,
        32,  33,  34,  35,  36,  37,  38,  39,  40,  41,  42,  43,  44,
        45,  46,  47,  48,  49,  50,  51,  52,  53,  54,  55,  56,  57,
        58,  59,  60,  61,  62,  63,  64,  65,  66,  67,  68,  69,  70,
        71,  72,  73,  74,  75,  76,  77,  78,  79,  80,  81,  82,  83,
        84,  85,  86,  89,  91,  92,  93,  94,  95,  98, 100, 101, 102,
       104, 106, 107, 109, 110, 112, 113, 114, 115, 116, 118, 153, 175,
       185])
```

5.9 SWTImage.transform.maximum_stroke_width

Maximum Permissible Stroke Length Parameter [default = 200]

While looking for a eligibile stroke emanating from any edge co-ordinate in the image, this parameter governs the maximum stroke length beyond which the stroke will be disqualified.

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', maximum_stroke_width=20,␣
↪display=False)
acd_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', maximum_stroke_width=70,␣
↪display=False)
ac1_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'SWT Max Stroke Width = 20 - {we_time}',
                                 f'SWT Max Stroke Width = 70 - {woe_time}'],
              plot_title='Comparing with and without edges SWT\n',
              sup_title=swtImgObj0.image_name)
```



5.10 SWTImage.transform.check_angle_deviation

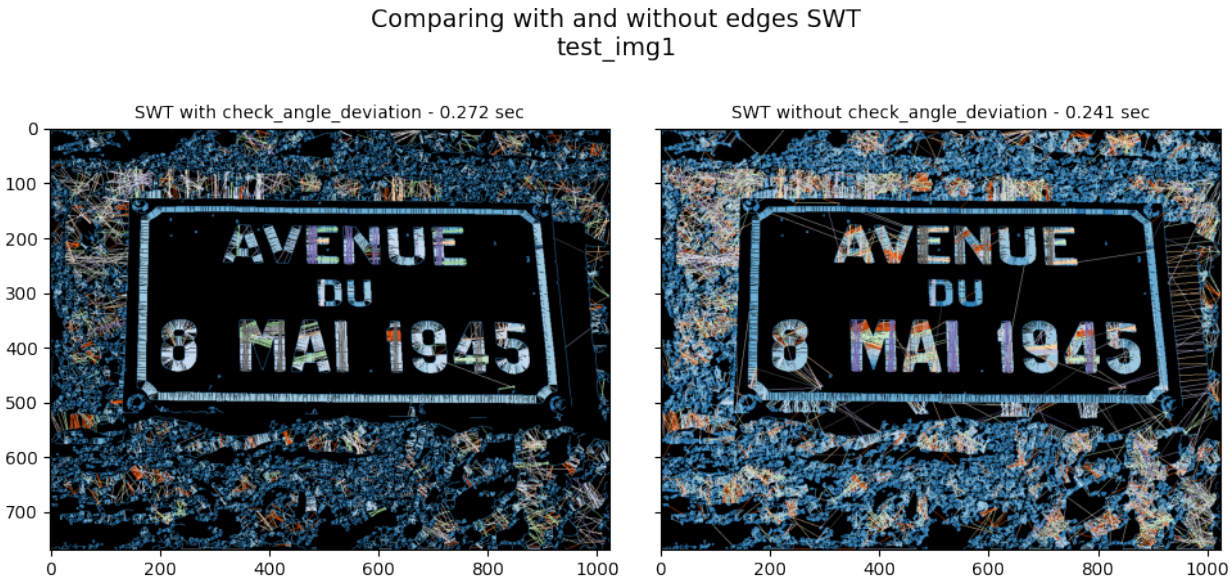
Whether to check for angle deviation [default = True]

Let's take an edge point **E1**, from where we start to form a ray in the direction of the gradient vector, and after few steps that ray lands on another edge point **E2**. Now that we have found an edge point **E2** after having started from E1, how can we be sure that this edge point **E2** is of the same letter i.e how can we be sure that the ray is an eligible stroke?

For this, the deviation between the gradient vectors of edge point **E1** and **E2** are measured, this parameter `check_angle_deviation` governs whether to check that deviation. If set to `False`, the first edge point that ray ends up meeting will stop the ray generation process and that ray will become an eligible stroke.

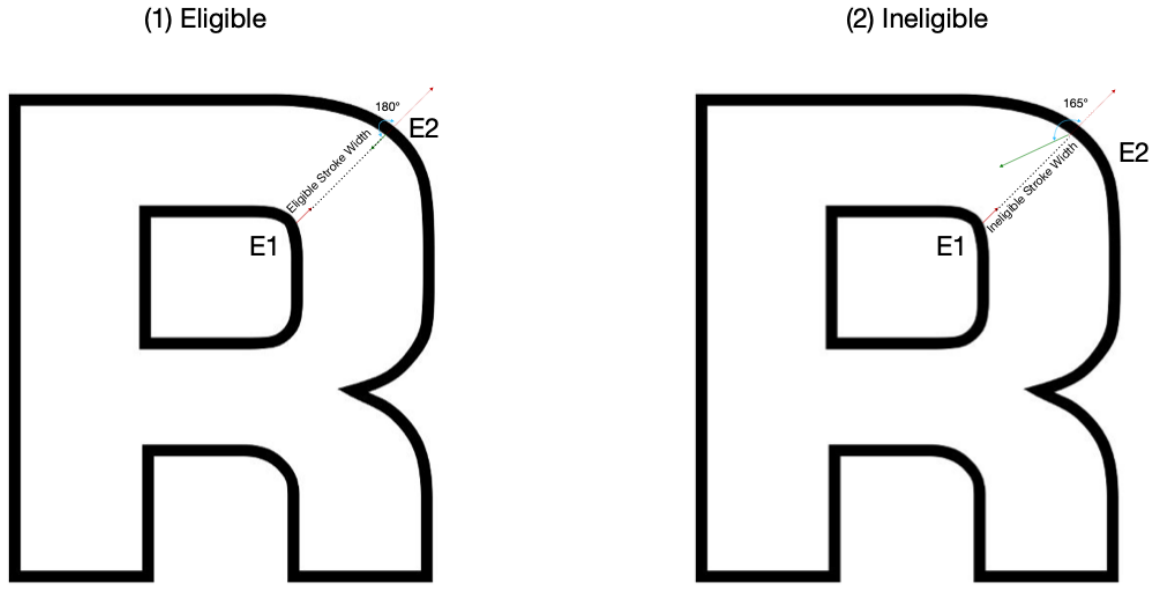
If se to `True` it will be ensured, before locking in the stroke, that the gradient vector of **E2** is nearly 180 degrees. The *nearly* part in last statement is quantified by the parameter maximum_angle_deviation

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', check_angle_deviation=True,␣
↪display=False)
ad_time = swtImgObj0.transform_time
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', check_angle_deviation=False,␣
↪display=False)
wad_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'SWT with check_angle_deviation - {ad_time}',
                                 f'SWT without check_angle_deviation - {wad_time}'],
              plot_title='Comparing with and without edges SWT\n',
              sup_title=swtImgObj0.image_name)
```



5.11 SWTImage.transform.maximum_angle_deviation

Acceptable angle deviation in the incoming edge point's gradient vector[default = np.pi/6]



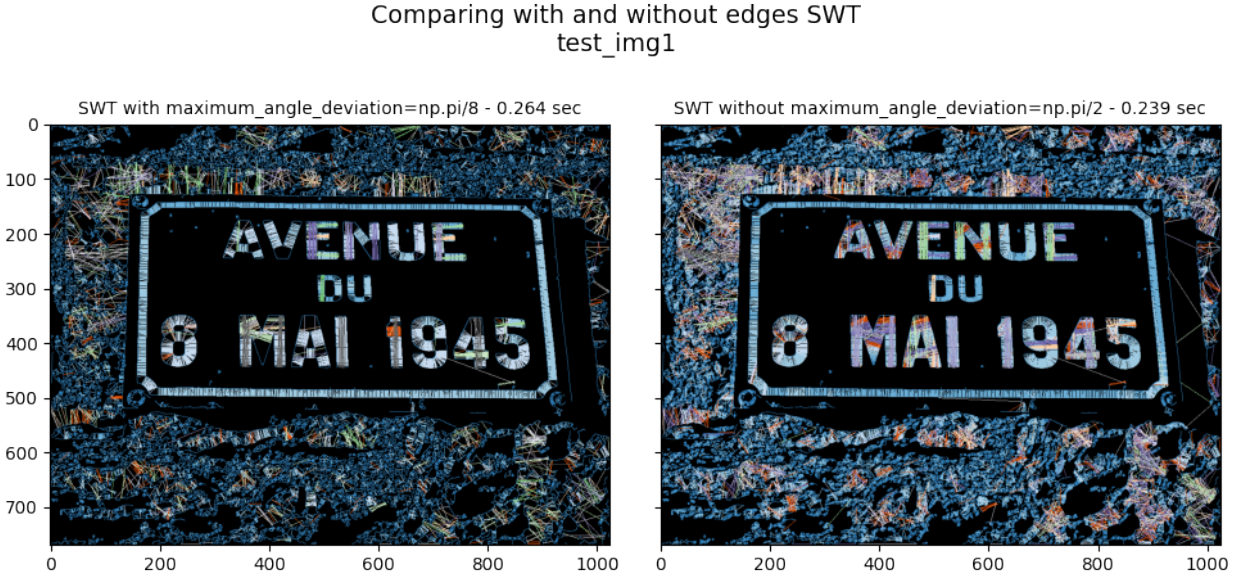
As explained in the check_angle_deviation, this parameter quantifies whats the acceptable deviation between the gradient angles of the **E1** and any other incoming **E2**.

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', maximum_angle_deviation=np.pi/8,␣
↪display=False)
ad8_time = swtImgObj0.transform_time
```

(continues on next page)

(continued from previous page)

```
swt_mat2 = swtImgObj0.transformImage(text_mode='db_lf', maximum_angle_deviation=np.pi/2,␣
↪display=False)
ad2_time = swtImgObj0.transform_time

show_N_images([image_1C_to_3C(swt_mat1),
               image_1C_to_3C(swt_mat2)],
              individual_titles=[f'SWT with maximum_angle_deviation=np.pi/8 - {ad8_time}
↪',
                                 f'SWT without maximum_angle_deviation=np.pi/2 - {ad2_
↪time}'],
              plot_title='Comparing with and without edges SWT\n',
              sup_title=swtImgObj0.image_name)
```



5.12 SWTImage.transform.display

This parameter just governs whether to display that particular stage's process. Its interpretation is same for all the functions in `SWTImage` class.

```
swt_mat1 = swtImgObj0.transformImage(text_mode='db_lf', maximum_angle_deviation=np.pi/8,␣
↪display=True)
```


SWT
Transform Time - 0.282 sec
Original Image
Gray-Scaled Image
Edge Image
SWT Transformed Image
AVENUE
DU
8 MAI 1945
0
100
200
300
400
500
600
700
0
200
400
600
800
1000

CHAPTER

SIX

LETTER LOCALIZATIONS, ANNOTATIONS & QUERY

`SWTImage.localizeLetters` function is responsible for localizing letters based on the following parameters. Apart from localization this function also calculates and generates boundaries across each individual letters.

- minimum_pixels_per_cc
- maximum_pixels_per_cc
- acceptable_aspect_ratio
- localize_by
- padding_pct

Connected Components are found using OpenCV's `connectedComponentsWithStats` function, in which each pixel is grouped to a blob of pixels who share their borders. Once these components have been found, pruning is done based on three parameters as stated above and explained below.

After the pruning it is assumed, whatever components which remain are worthy of being called as `Letter`'s.

These are the Image Codes which become available after running of `SWTImage.localizeLetters` function :

Image-Code	Description
IMAGE_CONNECTED_COMPONENTS_1C	Connected Components Single Channel
IMAGE_CONNECTED_COMPONENTS_3C	Connected Components RGB Channel
IMAGE_CONNECTED_COMPONENTS_3C_WITH_PRUNED_ELEMENTS	Connected Components Regions which were pruned (in red)
IMAGE_CONNECTED_COMPONENTS_PRUNED_1C	Pruned Connected Components Single Channel
IMAGE_CONNECTED_COMPONENTS_PRUNED_3C	Pruned Connected Components RGB Channel
IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS	Pruned RGB Channel SWT Image With Letter Localizations
IMAGE_ORIGINAL_LETTER_LOCALIZATIONS	Original Image With Letter Localizations
IMAGE_ORIGINAL_MASKED_LETTER_LOCALIZATIONS	Original Image With Masked Letter Localizations

Our Muse () for this Section

```
swtImgObj1 = swtl.swtimages[1]
swtImgObj1.showImage()
```

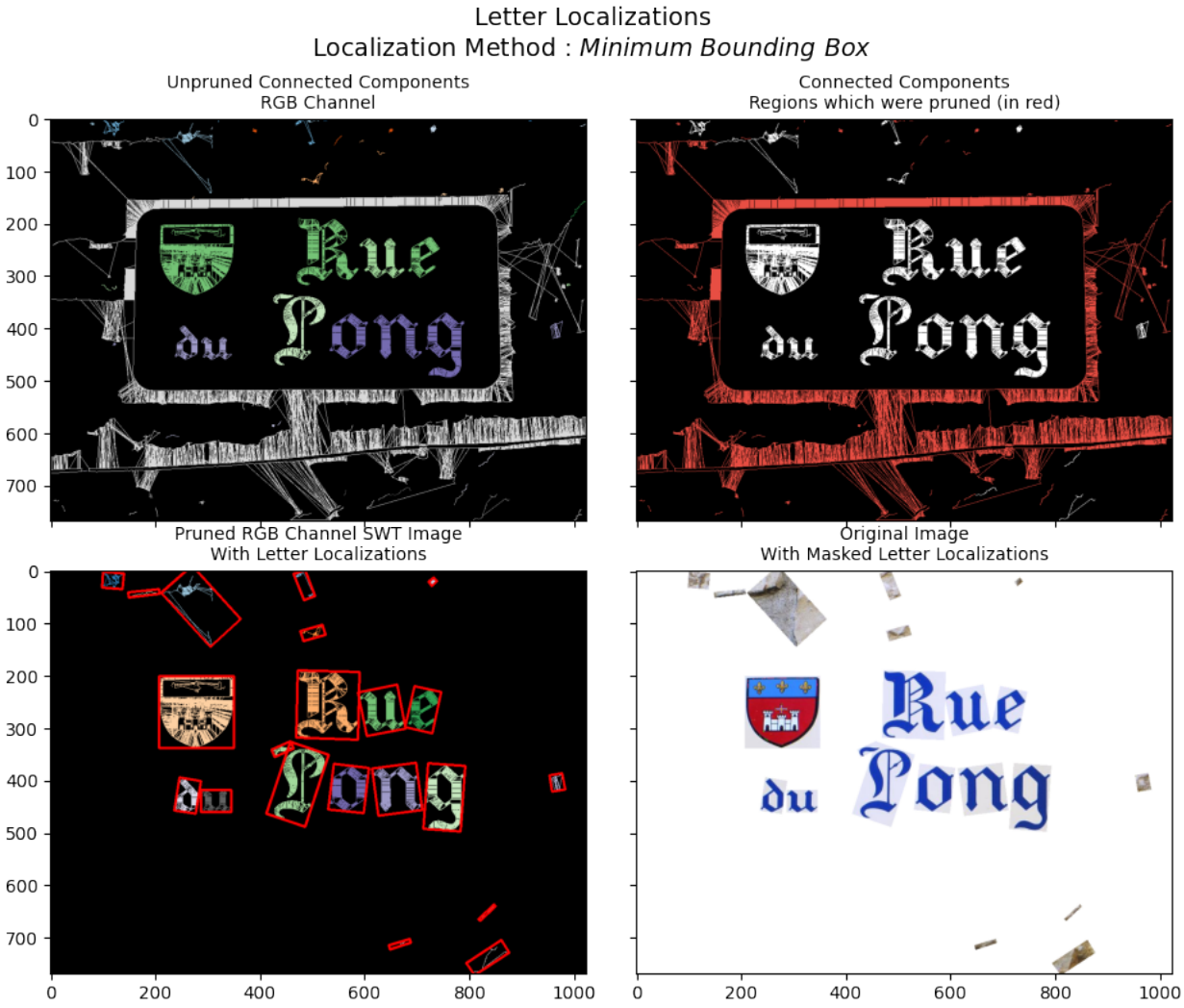
Transformation

```
swt_mat = swtImgObj1.transformImage(text_mode='lb_df',
                                    maximum_angle_deviation=np.pi/4,
                                    include_edges_in_swt=True)
```

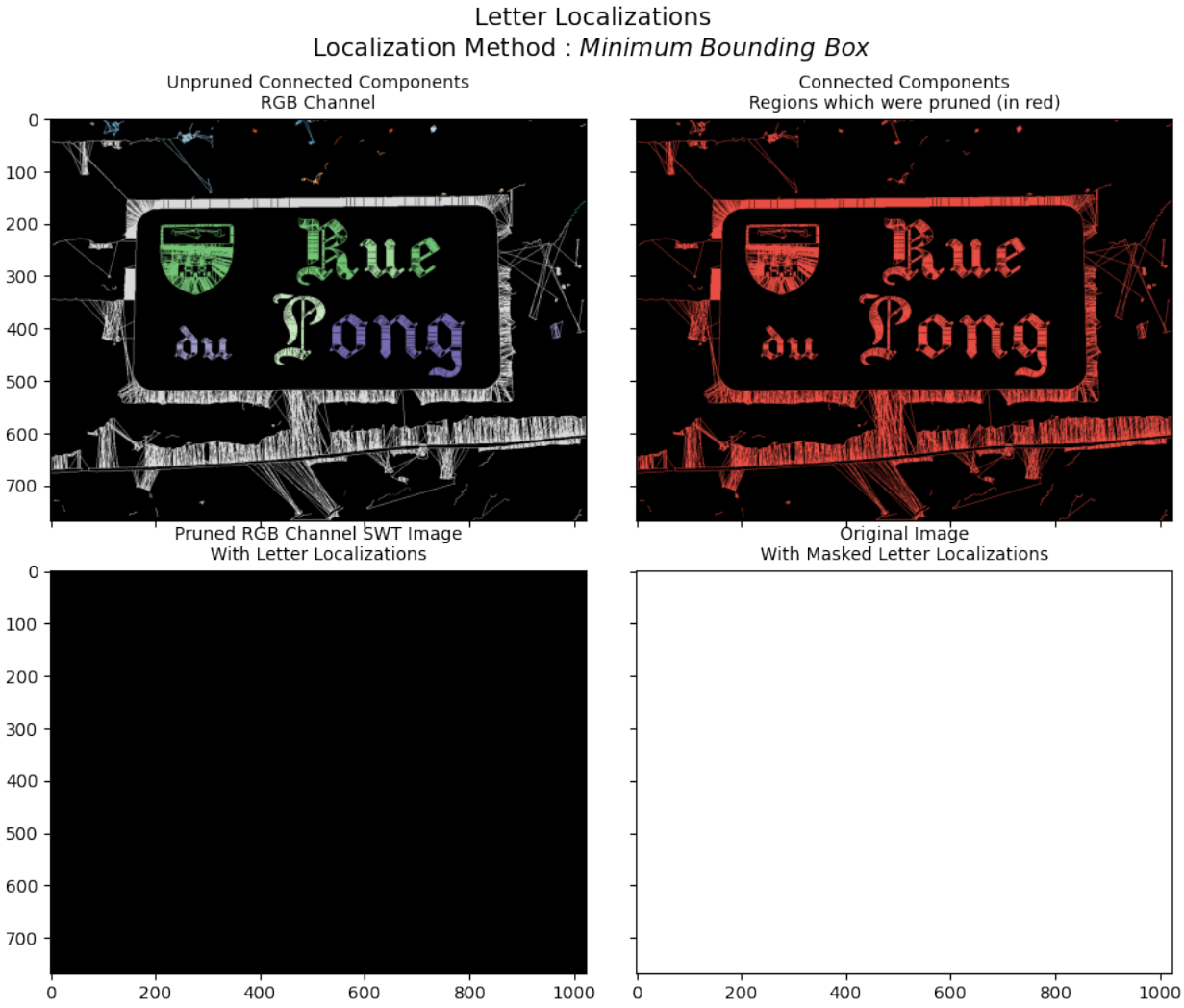
6.1 SWTImage.localizeLetters.minimum_pixels_per_cc

All those connected components whose area(number of pixels in that connected component) is less than `minimum_pixels_per_cc` will be pruned

```
localized_letters = swtImgObj1.localizeLetters()
```



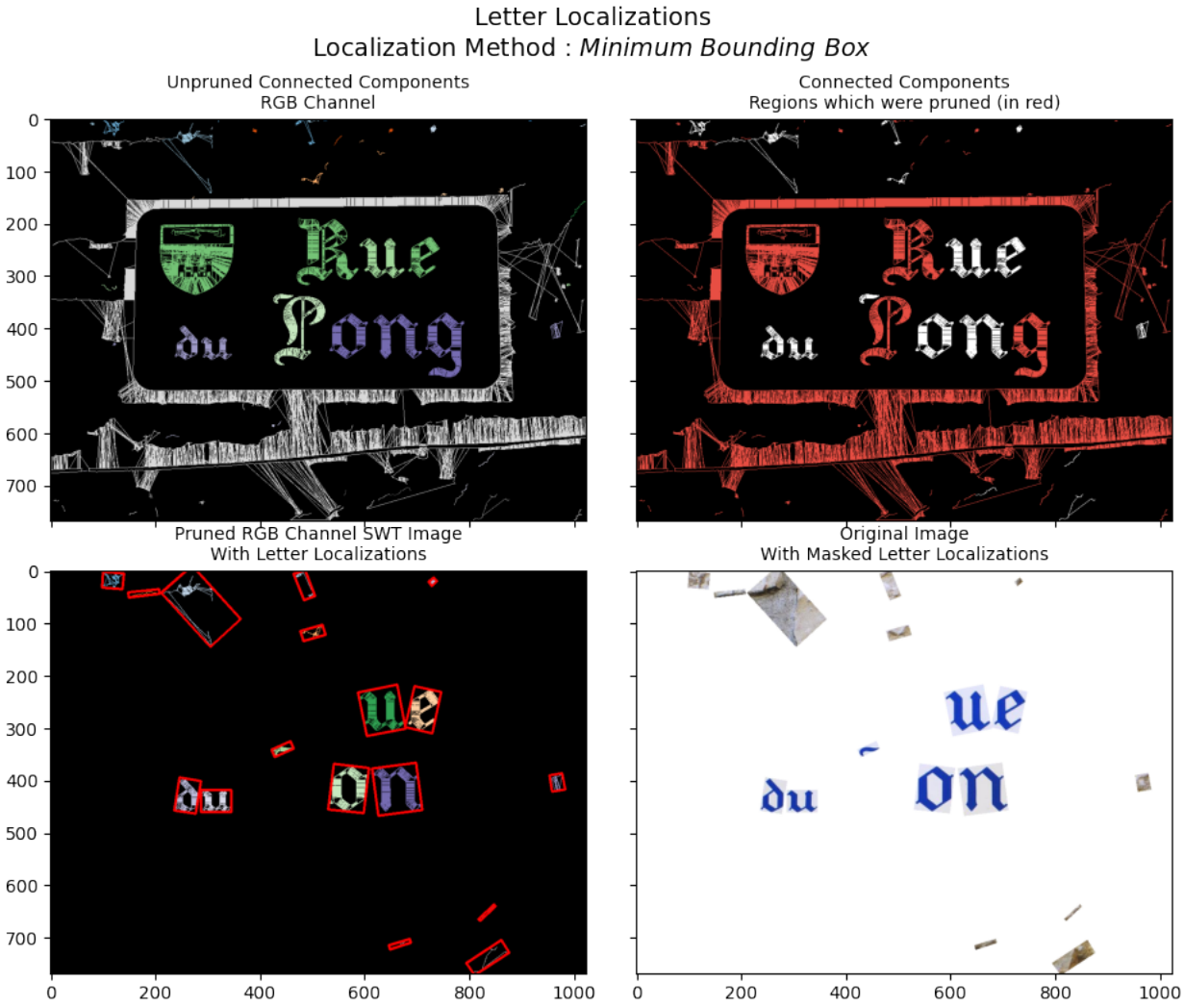

```
localized_letters = swtImgObj1.localizeLetters(minimum_pixels_per_cc=9000)
```



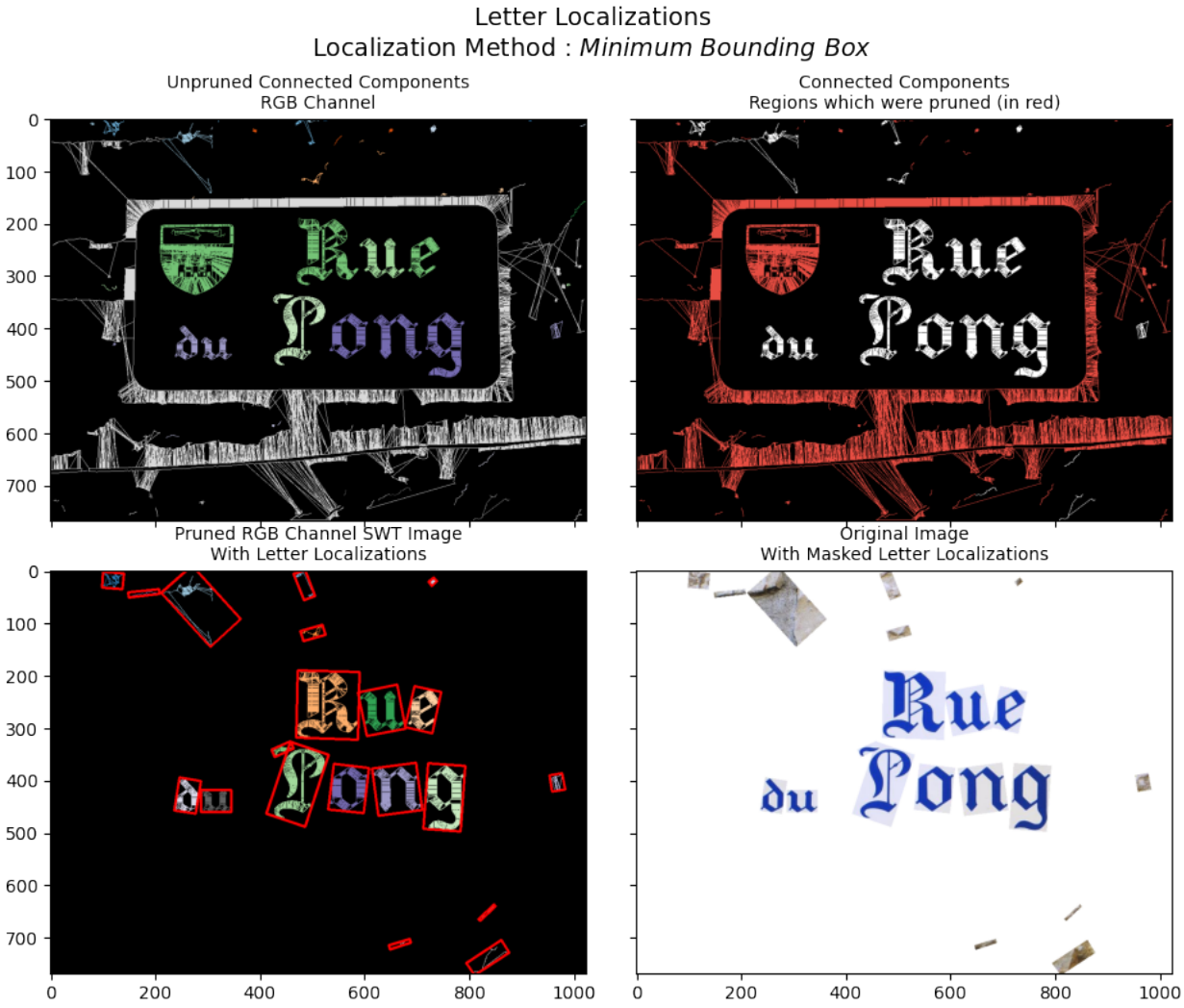
6.2 SWTImage.localizeLetters.maximum_pixels_per_cc

All those connected components whose area(number of pixels in that connected component) is more than `maximum_pixels_per_cc` will be pruned

```
localized_letters = swtImgObj1.localizeLetters(maximum_pixels_per_cc=3_000)
```




```
localized_letters = swtImgObj1.localizeLetters(maximum_pixels_per_cc=5_000)
```

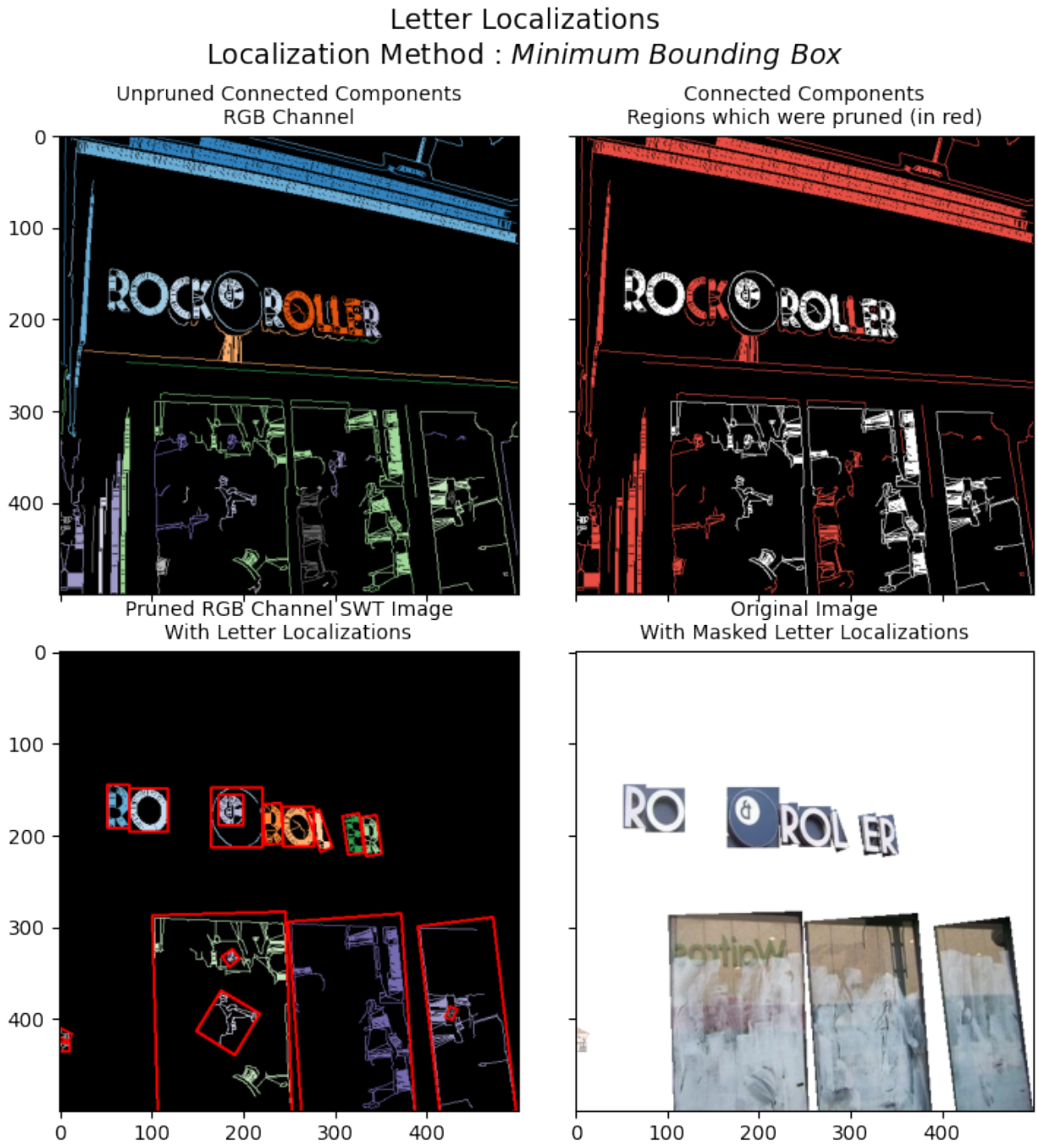


6.3 SWTImage.localizeLetters.acceptable_aspect_ratio

All those connected components whose aspect ratio (width/height) is less than `acceptable_aspect_ratio` will be pruned

```
swtImgObj2 = swtl.swtimages[2]
swt_mat = swtImgObj2.transformImage(text_mode='db_lf',
                                    maximum_stroke_width=30,
                                    maximum_angle_deviation=np.pi/8)
```


```
localized_letters = swtImgObj2.localizeLetters(acceptable_aspect_ratio=0.5)
```




```
localized_letters = swtImgObj2.localizeLetters(acceptable_aspect_ratio=0.005)
```


Letter Localizations
Localization Method : *Minimum Bounding Box*
Unpruned Connected Components
RGB Channel
Connected Components
Regions which were pruned (in red)
Pruned RGB Channel SWT Image
With Letter Localizations
Original Image
With Masked Letter Localizations
0
100
200
300
400

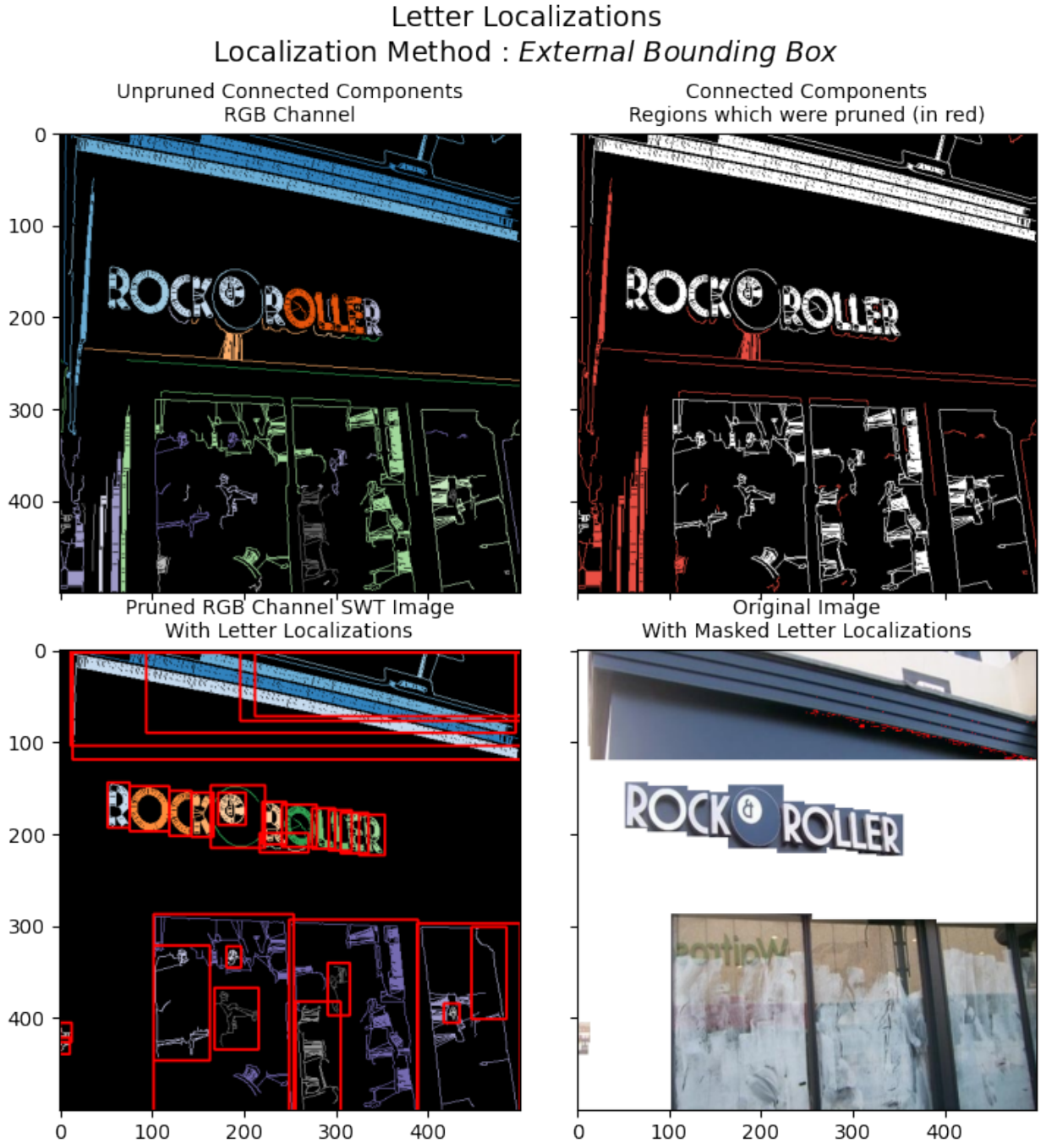
6.4 SWTImage.localizeLetters.localize_by

There are three possible parameters which can be given to this function

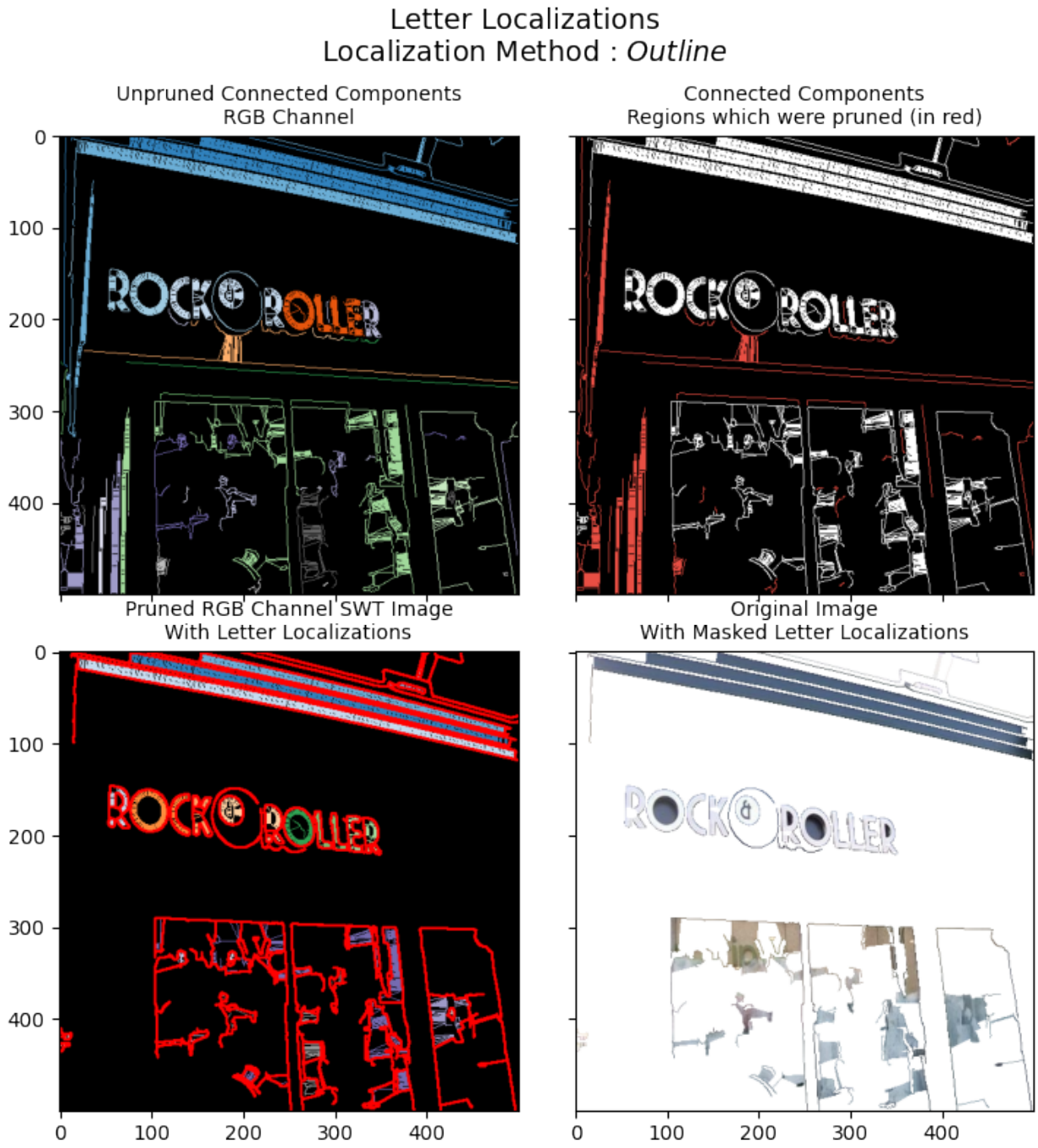
- min_bbox : Minimum Bounding Box Boundary calculation & annotation
- ext_bbox : Extreme Bounding Box Boundary calculation & annotation
- outline : Outline (Contour) Boundary calculation & annotation

```
localized_letters = swtImgObj2.localizeLetters(localize_by='min_bbox')
```

```
localized_letters = swtImgObj2.localizeLetters(localize_by='ext_bbox')
```




```
localized_letters = swtImgObj2.localizeLetters(localize_by='outline')
```


Letter Localizations
Localization Method : Outline
Unpruned Connected Components
RGB Channel
Connected Components
Regions which were pruned (in red)
Pruned RGB Channel SWT Image
With Letter Localizations
Original Image
With Masked Letter Localizations
0
100
200
300
400

6.5 SWTImage.localizeLetters.padding_pct

While annotating, sometimes the annotation are quite tightly packed to the component itself, `padding_pct` will inflate the boundary which are calculated. This parameter is only applicable for `min_bbox` and `ext_bbox` localizations

```
localized_letters = swtImgObj2.localizeLetters()
```


```
localized_letters = swtImgObj2.localizeLetters(padding_pct=0.1)
```


Letter Localizations
Localization Method : Minimum Bounding Box
Unpruned Connected Components
RGB Channel
Connected Components
Regions which were pruned (in red)
Pruned RGB Channel SWT Image
With Letter Localizations
Original Image
With Masked Letter Localizations
0
100
200
300
400

CHAPTER

SEVEN

WORD LOCALIZATION, ANNOTATIONS & QUERY

After having the letters localizer, there exists a core class `Fusion` whose job is to group individual components into `Words`, comparing aspects like :

- Proximity of letters to each other
- Relative minimum bounding box rotation angle from each other
- Deviation in color between from one component to the other
- Ratio of stroke widths from one to the other
- Ratio of minimum bounding box height of one to the other

The above comparisions are taken care of by the following parameters :

- localize_by
- [lookup_radius_multiplier]
- [acceptable_stroke_width_ratio]
- [acceptable_color_deviation]
- [acceptable_height_ratio]
- [acceptable_angle_deviation]
- [polygon_dilate_iterations]
- [polygon_dilate_kernel]

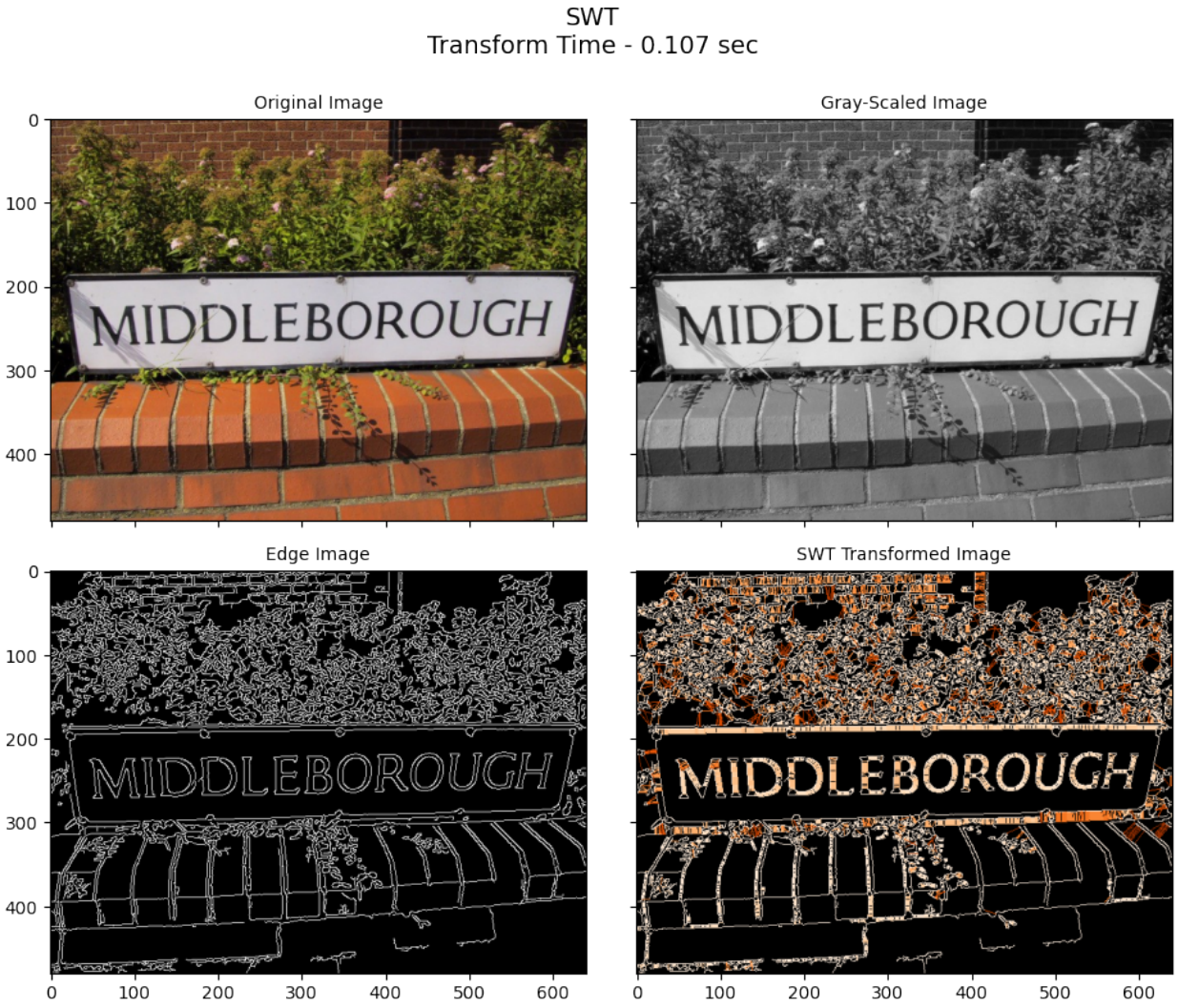
Image-Code	Description
IMAGE_PRUNED_3C_WORD_LOCALIZATIONS	Pruned RGB Channel SWT Image With Words Localizations
IMAGE_ORIGINAL_WORD_LOCALIZATIONS	Original Image With Words Localizations
IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS	Original Image With Masked Words Localizations

Our Muse () for this Section

```
swtImgObj3 = swtl.swtimages[3]
swtImgObj3.showImage()
```

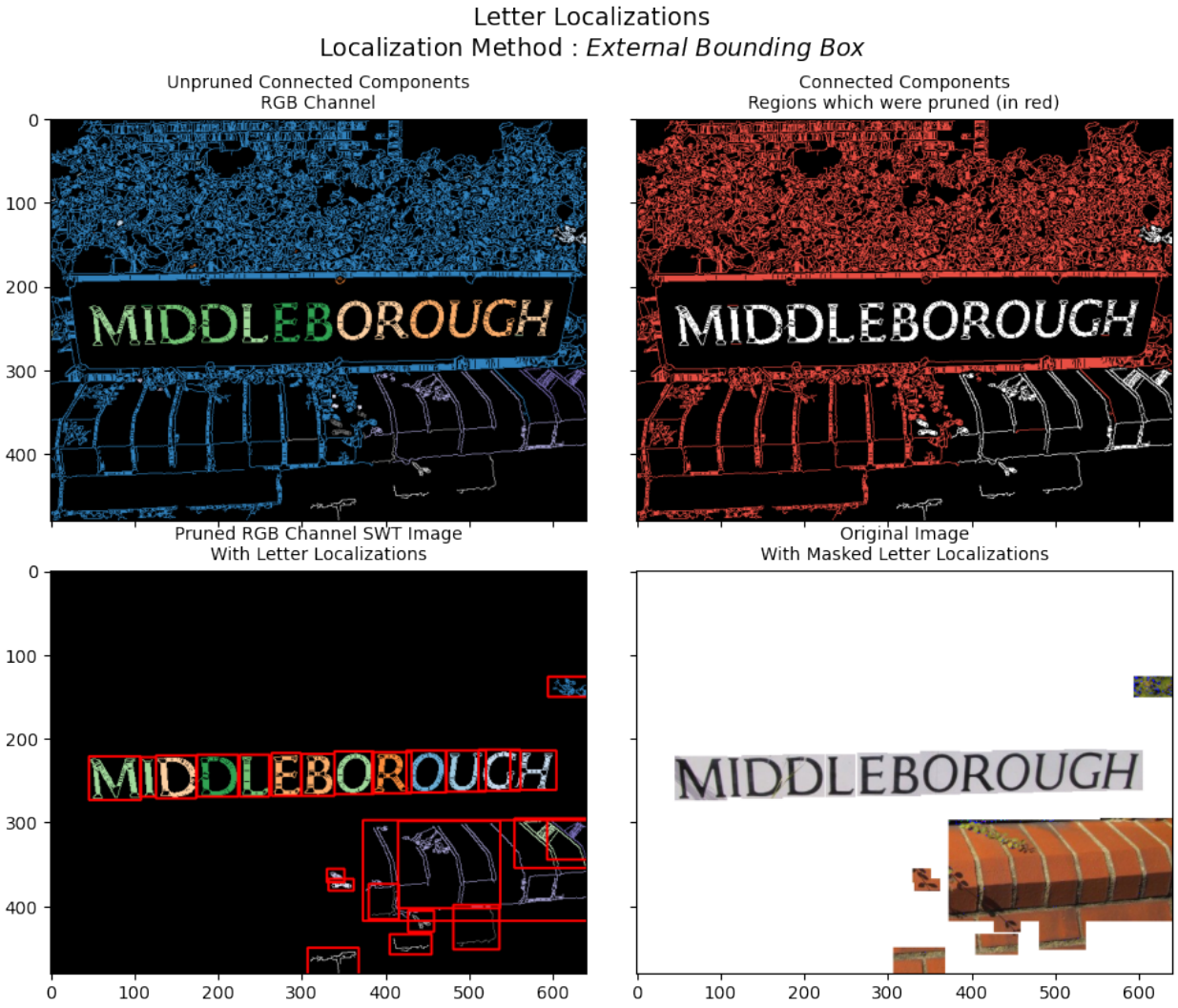
Applying Transformations

```
swt_mat = swtImgObj3.transformImage(auto_canny_sigma=1.0,
                                    minimum_stroke_width=3,
                                    maximum_stroke_width=20,
                                    maximum_angle_deviation=np.pi/6)
```



Localizing Letters

```
# Localising Letters
localized_letters = swtImgObj3.localizeLetters(localize_by='ext_bbox')
```



SWTImageProcess Error will be raised if `.localizeLetters(localize_by='min_bbox')` hasnt been run already, as the properties which are calculated while localzing letters by `min_bbox` such as the min_bbox circum radius are required to fuse letters into words.

```
# Localising Letters
localized_letters = swtImgObj3.localizeWords()
```

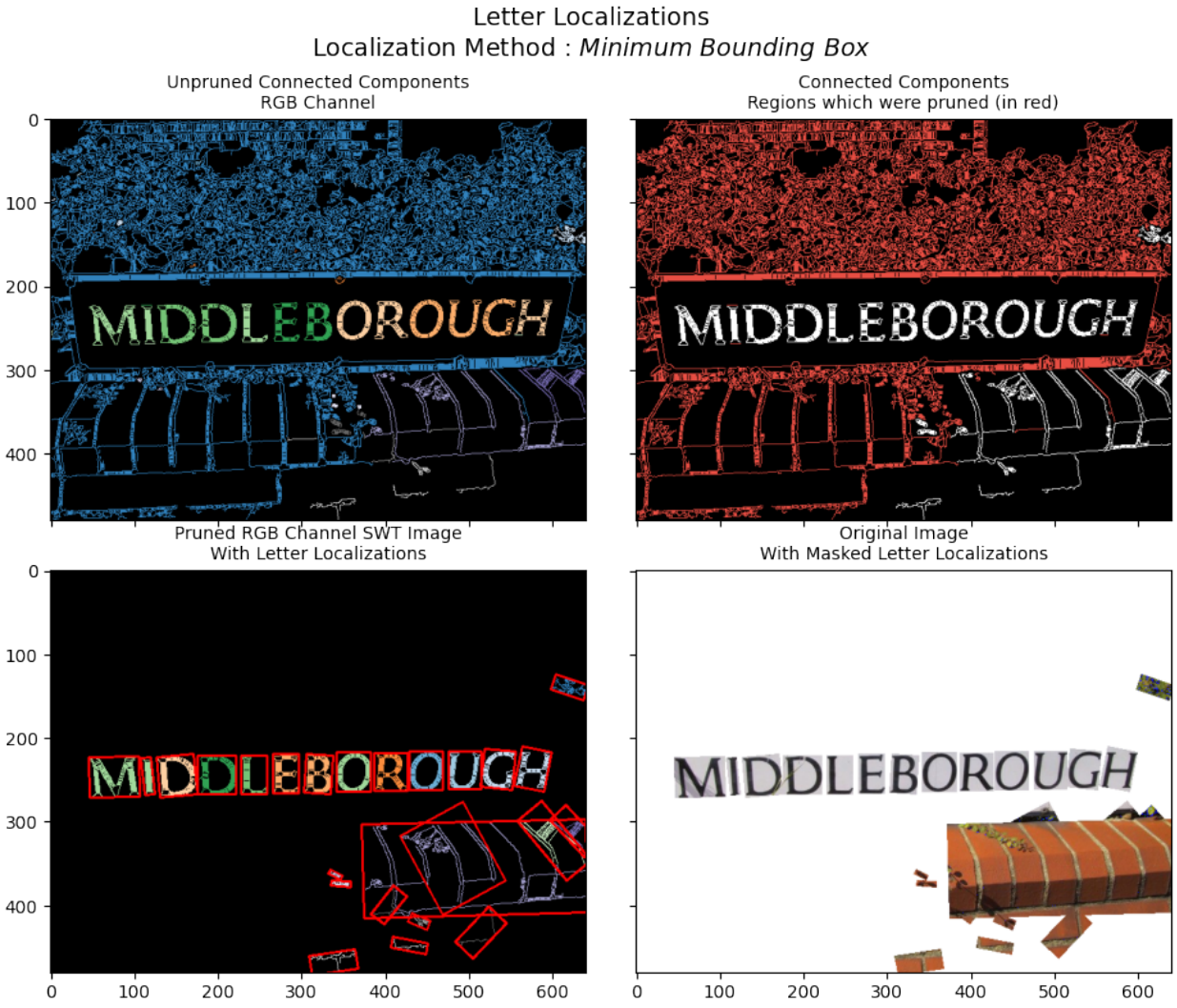
```
---------------------------------------------------------------------------

SWTImageProcessError                      Traceback (most recent call last)

Input In [48], in <module>
      1 # Localising Letters
----> 2 localized_letters = swtImgObj3.localizeWords()


File D:\Personal Stuff\swtloc-project\venvs\py39VenvDev\lib\site-packages\swtloc\
↪abstractions.py:1158, in SWTImage.localizeWords(self, localize_by, lookup_radius_
↪multiplier, acceptable_stroke_width_ratio, acceptable_color_deviation, acceptable_
↪height_ratio, acceptable_angle_deviation, polygon_dilate_iterations, polygon_dilate_
↪kernel, display)
```

(continued on next page)

(continued from previous page)

```
   1154 # TODO : Add the functionality to detect whether the changes were made to the
   1155 # TODO : localizations parameters or just annotation parameter and accordingly␣
↪make the resets.
   1156 # Check if transform stage has been run first or not
   1157 if not self.letter_min_done:
-> 1158     raise SWTImageProcessError(
   1159         "`SWTImage.localizeLetters` with localize_by='min_bbox' must be called␣
↪before this function")
   1160 # Update configs & Initialise
   1161 self.cfg[CONFIG__SWTIMAGE__LOCALIZEWORDS_LOCALIZE_BY] = localize_by


SWTImageProcessError: `SWTImage.localizeLetters` with localize_by='min_bbox' must be␣
↪called before this function
```

```
# Solution : Run .localizeLetters default localize_by = 'min_bbox' (default)
# before running .localizeWords
localized_letters = swtImgObj3.localizeLetters()
```



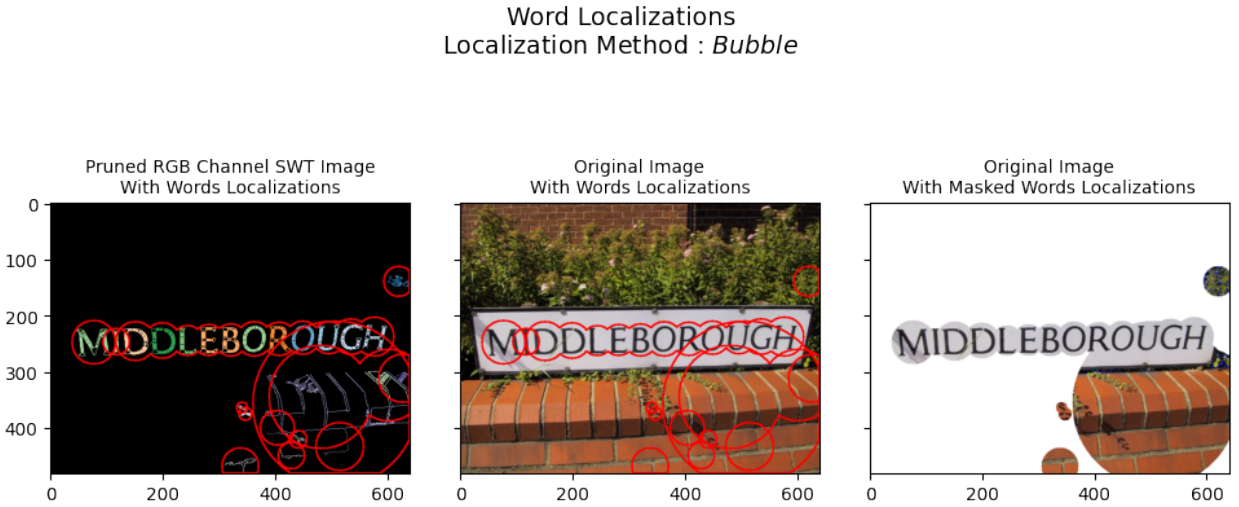
7.1 SWTImage.localizeWords.localize_by

This parameter switches between various annotations which can be added to the words [default = 'bubble']

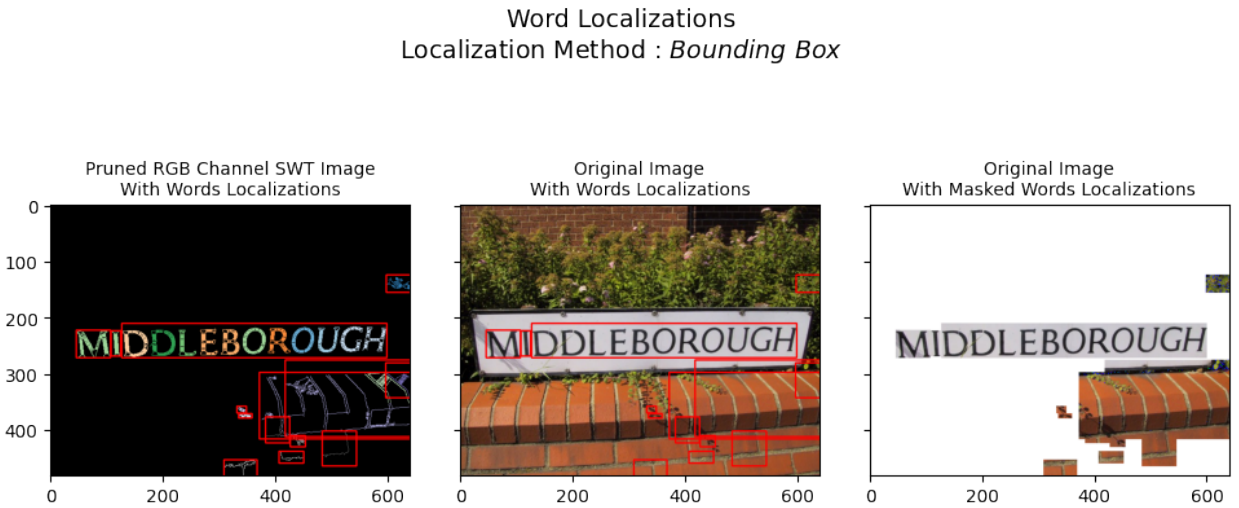
Possible values are :

- 'bubble' : Bubble Boundary
- 'bbox' : Bounding Box
- 'polygon' : Contour Boundary

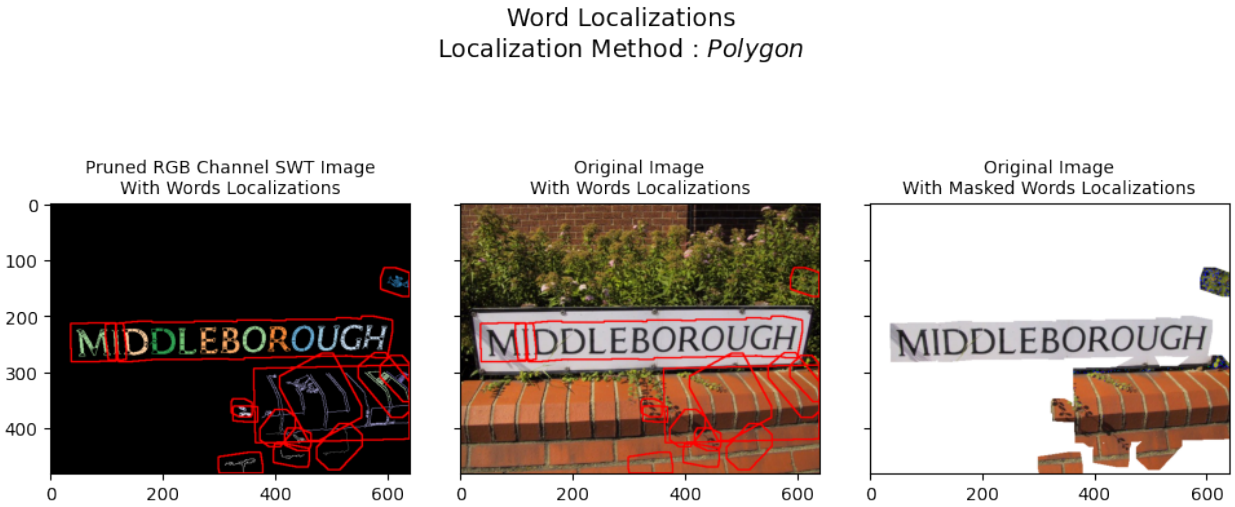
```
localized_words = swtImgObj3.localizeWords(localize_by='bubble')
```




```
localized_words = swtImgObj3.localizeWords(localize_by='bbox')
```




```
localized_words = swtImgObj3.localizeWords(localize_by='polygon')
```



7.2 Display Intermediary Images

From v2.0.0 onwards, a provision was added to save the intermediary stage images by access the `SWTImage.showImage` function.

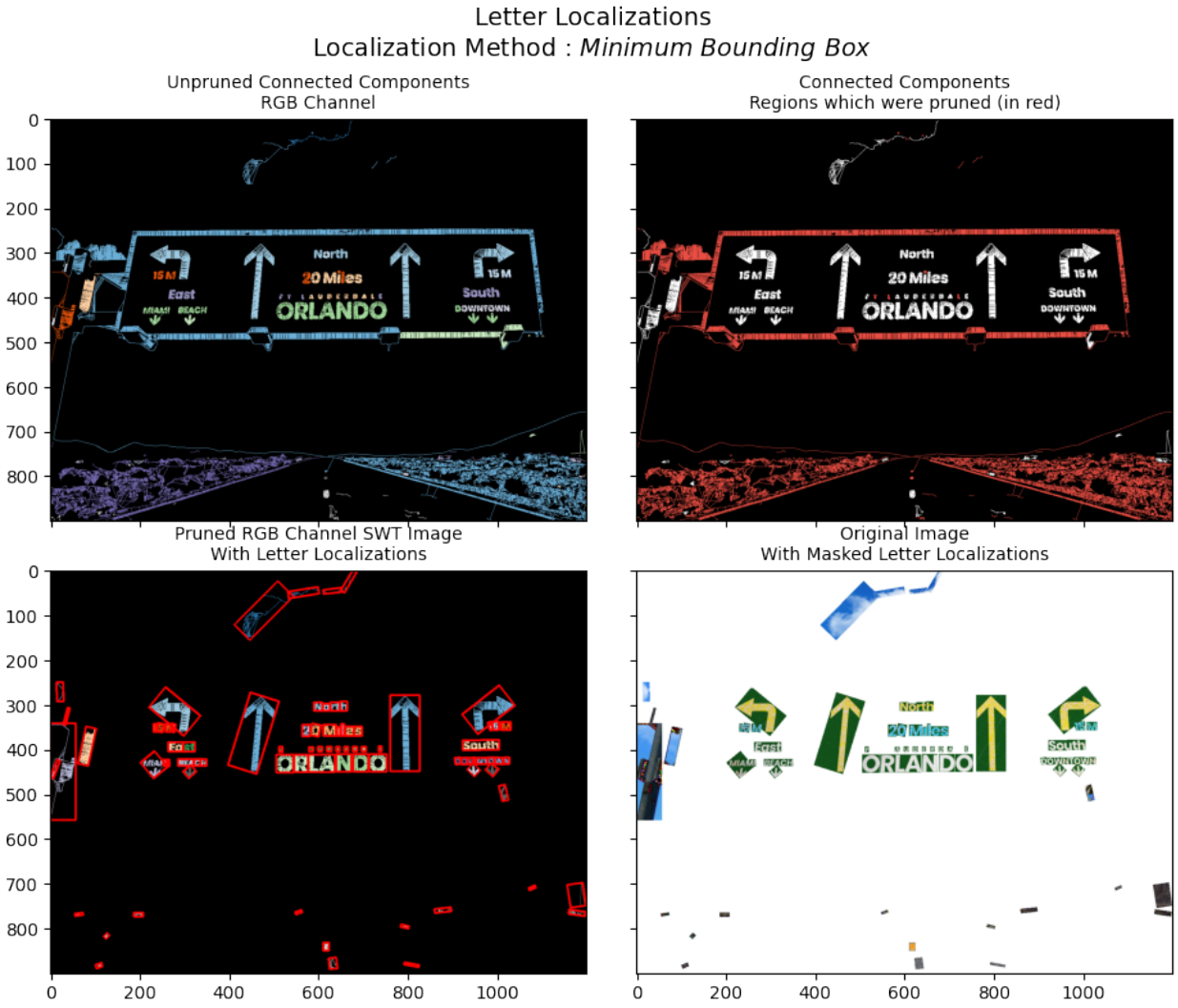
Available Image Codes

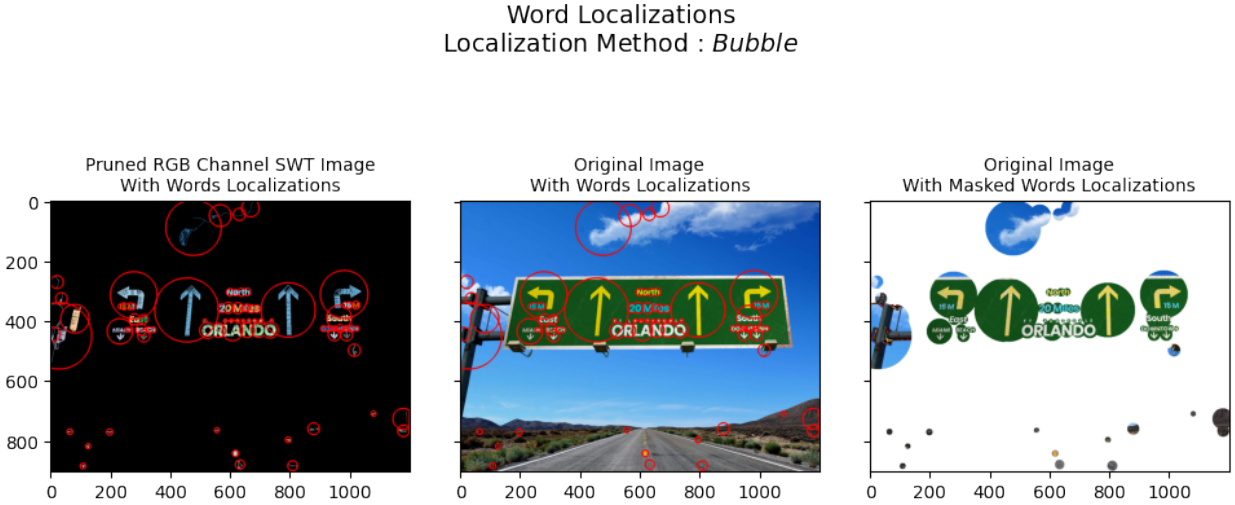
```
image_code_df = pd.DataFrame(columns=['Description'])

for each_code_name, each_code in CODE_NAME_VAR_MAPPINGS.items():
    image_code_df.loc[each_code_name] = get_code_descriptions(each_code).replace('\n', '
↪')
image_code_df.style.set_properties(**{'width': '600px', 'text-align': 'center'})
```

Our Muse () for this Section

```
swtImgObj4 = swtl.swtimages[4]
swtImgObj4.showImage()
```

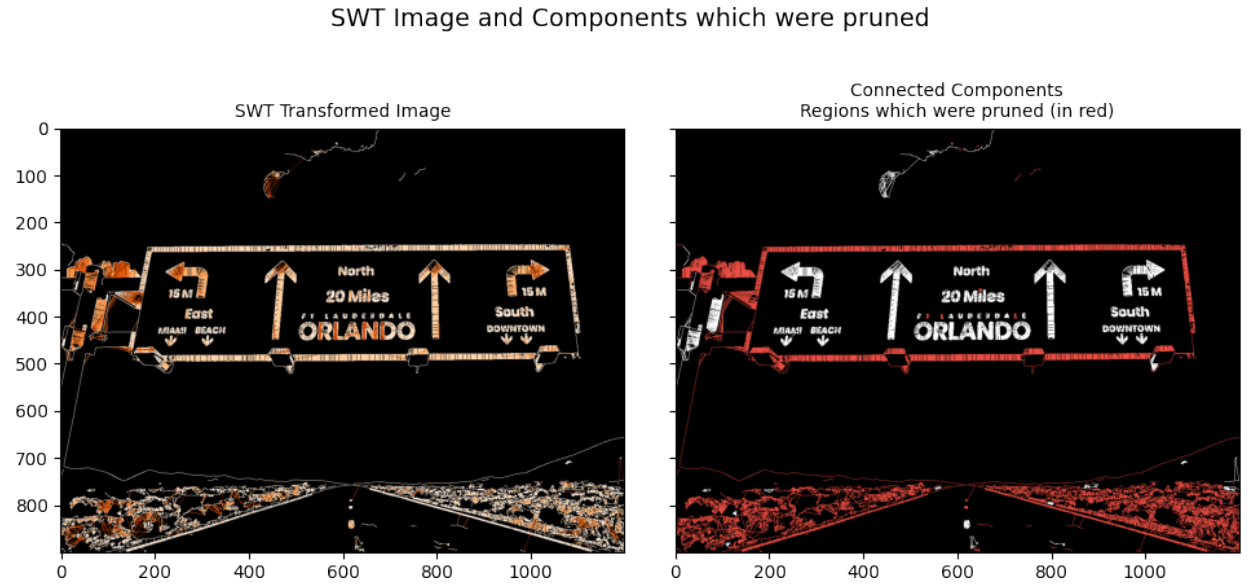

```
swt_mat = swtImgObj4.transformImage(text_mode='db_lf', maximum_angle_deviation=np.pi/2,
                                    edge_function='ac',
                                    minimum_stroke_width=3, maximum_stroke_width=50,␣
↪display=False)
localized_letters = swtImgObj4.localizeLetters()
localized_words = swtImgObj4.localizeWords()
```





Display Multiple Images

```
swtImgObj4.showImage(image_codes=[IMAGE_SWT_TRANSFORMED,
                                  IMAGE_CONNECTED_COMPONENTS_3C_WITH_PRUNED_ELEMENTS],
                     plot_title='SWT Image and Components which were pruned')
```



Save Prepared Images

```
swtImgObj4.showImage(image_codes=[IMAGE_SWT_TRANSFORMED,
                                  IMAGE_CONNECTED_COMPONENTS_3C_WITH_PRUNED_ELEMENTS],
                     plot_title='SWT Image and Components which were pruned',
                     save_fig=True, save_dir=res_path[4])
```

```
'images/test_img5/usage_results/test_img5_04_07.jpg'
```

CHAPTER

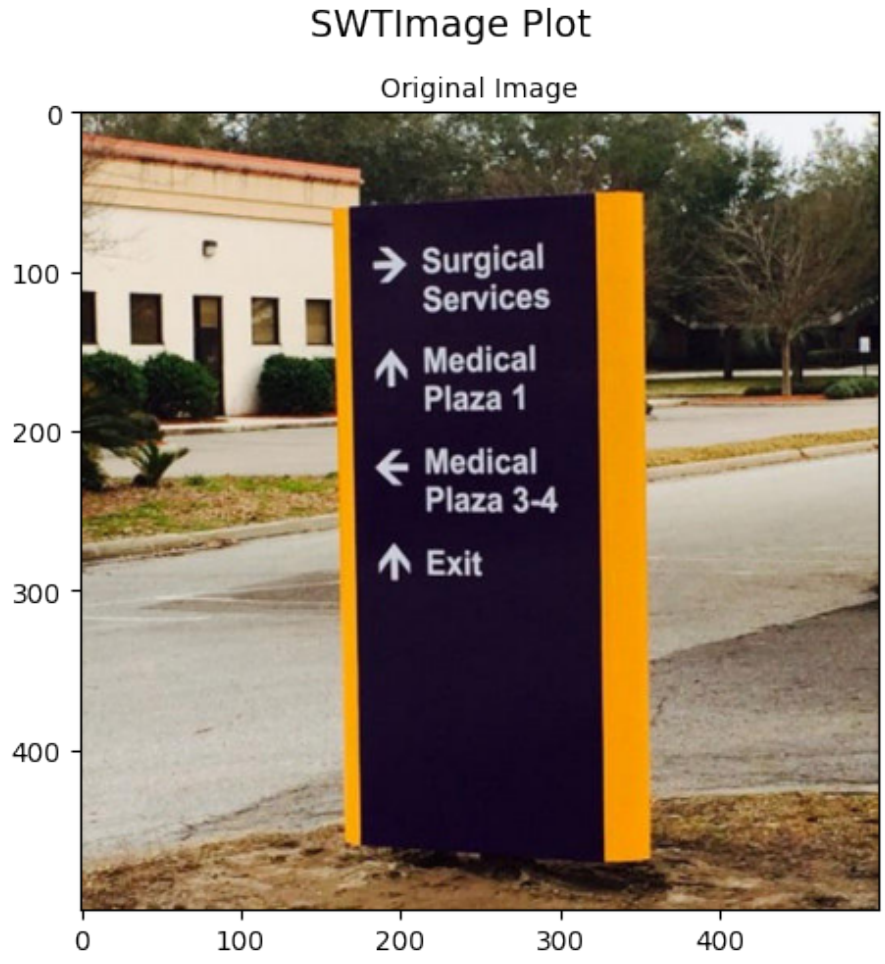
EIGHT

SAVE CROPS OF LETTER'S AND WORD'S

From v2.0.0 onwards, provision to save a crop of the `Letter` or a `Word` has been added via the function `SWTImage.saveCrop`. The crops can be made on any one of the available image codes for a particular `letter` or `word` key

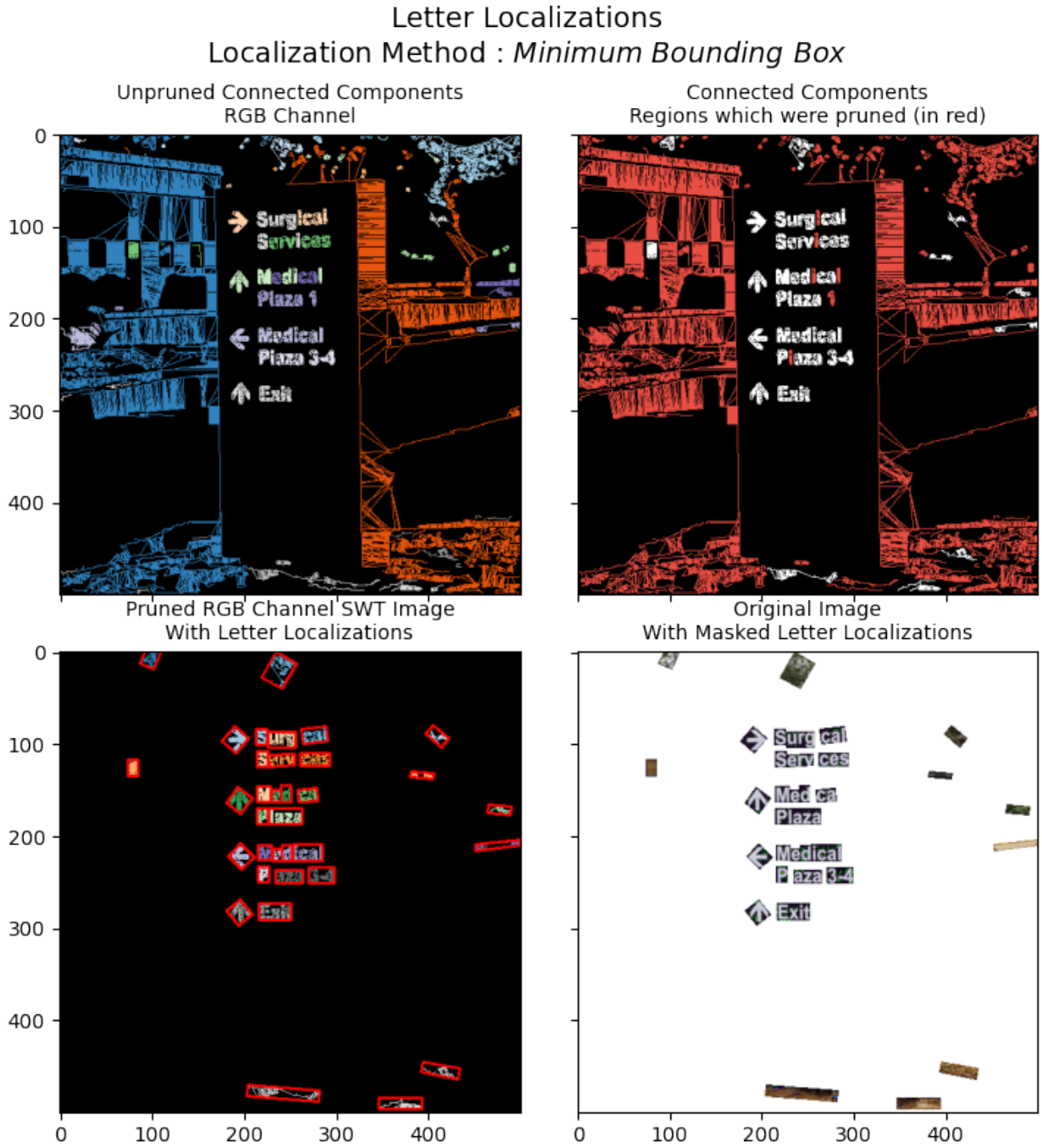
Our Muse () for this Section

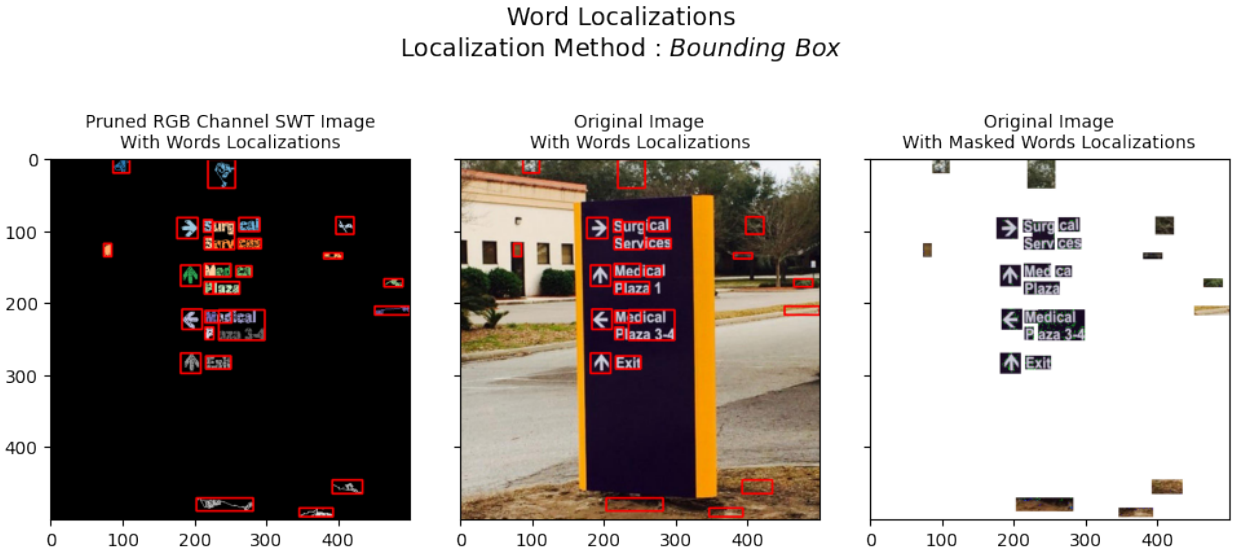
```
swtImgObj5 = swtl.swtimages[5]
swtImgObj5.showImage()
```



Performing Transformations and Localizations

```
swt_mat = swtImgObj5.transformImage(text_mode='db_lf', maximum_angle_deviation=np.pi/2,
                                    edge_function='ac',
                                    minimum_stroke_width=3, maximum_stroke_width=50,␣
↪display=False)
localized_letters = swtImgObj5.localizeLetters(minimum_pixels_per_cc=80,
                                               maximum_pixels_per_cc=400)
localized_words = swtImgObj5.localizeWords(localize_by='bbox')
```

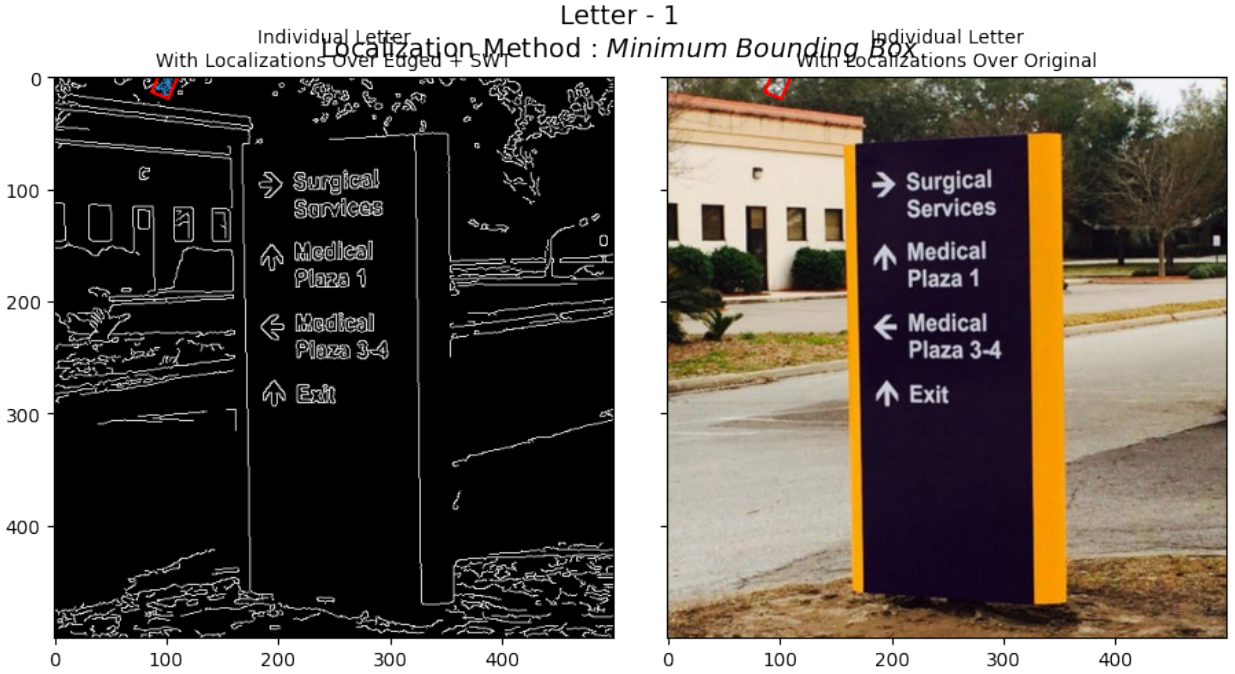




While sifting through the multitudes of letters and words localized, it becomes cumbersome to query a function for every label and then see which letter/word is of interest to use the crop of. For this purpose, two functions `letterIterator` and `wordIterator` are provided in the `SWTImage` object which return a generator for a particular `localize_type`

```
# Letter Iterator
letter_iterator = swtImgObj5.letterIterator()
word_iterator = swtImgObj5.wordIterator(localize_by='bbox')
```

```
letter, orig_letter, orig_masked_letter = next(letter_iterator)
```



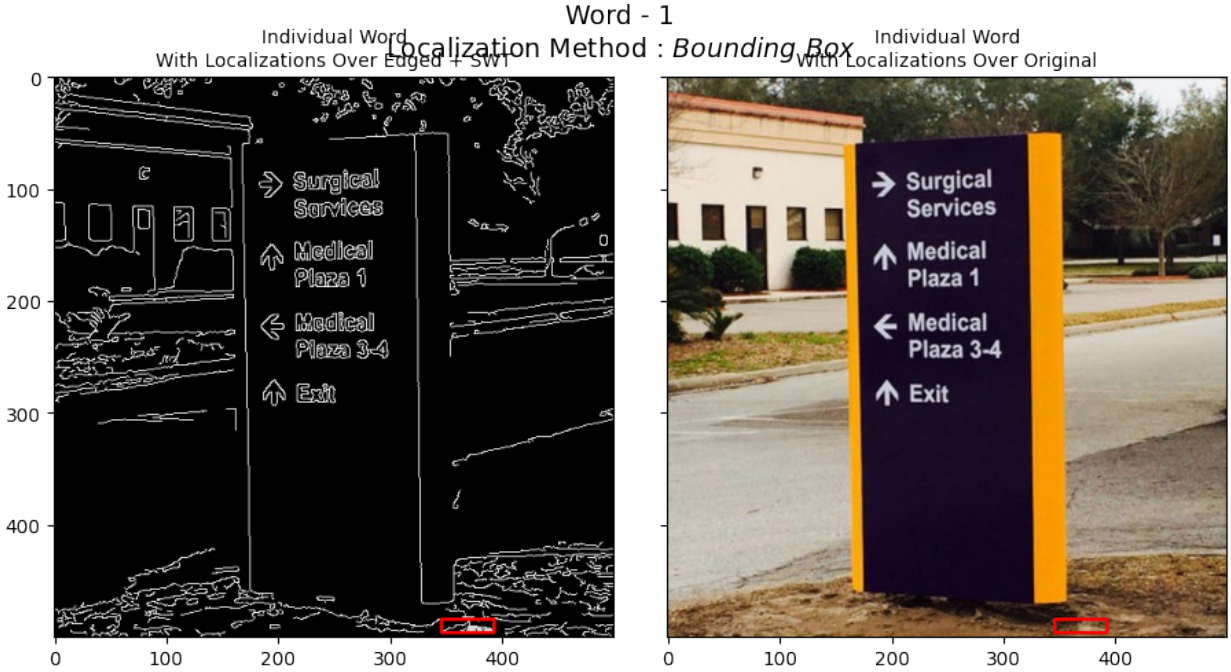

```
swtImgObj5.saveCrop(save_path=res_path[5], crop_of='letters',
```

(continues on next page)

(continued from previous page)

```
                    crop_key=int(letter.label), crop_on=IMAGE_SWT_TRANSFORMED, crop_type=
↪'min_bbox')
swtImgObj5.saveCrop(save_path=res_path[5], crop_of='letters',
                    crop_key=int(letter.label), crop_on=IMAGE_ORIGINAL, crop_type='min_
↪bbox')
```

```
word, orig_word, orig_masked_word = next(word_iterator)
```




```
swtImgObj5.saveCrop(save_path=res_path[5], crop_of='words',
                    crop_key=int(word.label), crop_on=IMAGE_SWT_TRANSFORMED, crop_type=
↪'bbox')
swtImgObj5.saveCrop(save_path=res_path[5], crop_of='words',
                    crop_key=int(word.label), crop_on=IMAGE_ORIGINAL, crop_type='bbox')
```

CHAPTER

NINE

VERSION LOGS

---

v2.1.0 : Minor Release - Refactoring, Add Docs, Add Tests

- ReadTheDocs integration.
- Removal of deprecated codebase.
- Removal of `README_old.md`
- Removal of deprecated files.
- Add `tests/` to house tests.
- Update *setup.py* & *setup_dev.py* files for `packages` parameter.
- Update Release Logs.

---

v2.0.0 : Major Release - Refactoring, New Engines, Abstraction Addition (Py36, Py37, Py38, Py39 & Py310 Compatible)

- Following refactoring Additions/Changes were made:
  - Core algorithms moved to core.py
  - Older files deprecated and names changed, these files would be removed in v2.0.1
    - `bubble_bbox.py` → `_bubble_bbox.py`
    - `swt.py` → `_swt.py`
    - `utils.py` → `_utils.py`
  - Add README.md (v2.0.0 onwards)
  - Add History.md : File to house history logs
  - Add Usage.md : Gives overview of the usage of the package
  - Newer files added:
    - core.py : To house all the core algorithms - `findStrokes`, `Fusion`.
    - abstractions.py : To house all the abstractions - `SWTImage`, `Letter` and `Word`.
- Dependency Changes
  - (+) Numba
  - (-) imutils
- Algorithmic Changes:

- (+/-) Major deprecation in `SWTLocalizer`, almost all codebase moved to other locations. This will be, henceforth, a driver class.
- (+) New abstractions.py file:
  - Addition of class `SWTImage` - An abstraction for an individual images sent in for processing. Has following major functions :
    - transformImage : To transform the original image to its stroke widths (*1)
    - localizeLetters : To localize letters
    - localizeWords : To letters into words
    - getLetter : To retrieve an individual letters
    - letterIterator : Returns a generator over all the letters with visualization capabilities
    - getWord : To retrieve an individual word
    - wordIterator : Returns a generator over all the words with visualization capabilities
    - saveCrop : To crop and save a letter or a word
    - showImage: To display one/multiple images using the Image Codes defined in configs.py, also has the ability to save the prepared image
  - Addition of class `Letter` - Represent and houses properties of possible letters
    - Functionality : Add various localization annotation to input image
  - Addition of class `Words` - Represent and houses properties of possible words
    - Functionality : Add various localization annotation to input image
- (+) New core.py file
  - Addition of `swt_strokes` & `swt_strokes_jitted` function corresponding to the `python` and `numba` engines
  - Addition of `Fusion` & `ProxyLetter` for grouping of letters into probable letters
- (+) New base.py file
  - Addition of `IndividualComponentBase` : A base class to be inherited by `Letter`
  - Addition of `GroupedComponentsBase` : A base class to be inherited by `Word`
  - Addition of `TextTransformBase` : A base class to be inherited by `SWTImage`
- (+) New configs.py file
  - Houses configurations for the Stroke Width Transform
- (+) Add Improvements in v2.0.0.ipynb notebook
- (+) Add README Code Blocks.ipynb notebook
- (+) Add QnA [v2.0.0 onwards].ipynb notebook
- (+) Add SWTloc Usage [v2.0.0 onwards].ipynb notebook

v1.1.1 : Refine Versioning System

- New versioning system defined : x[Major Update].x[Minor Update].x[Fixes]
- Tag 1.1.x Represents all bug fixed versions of 1.1.
- Bug Fixes

v1.0.0.3 : Add Individual Image Processing

- Functionality to transform pre-loaded image
- Minor Bug Fixes
- Add support for Python 3.6
- Reduce Dependency

v1.0.0.2 : Few bug fixes and addendum's

- Add parameter to govern the width of BubbleBBox
- Add Examples - StackOverflow Q/A
- Add image resizing utility function to the utils.py

v1.0.0.1 : Original Package

- Add SWTlocaliser to the package
- Add the logic for Bubble Bounding Boxes
- Add Examples

(+) -> Addition (-) -> Deletion (+/-) -> Modification

CHAPTER

TEN

SWTLOC

API Documentation for SWTLoc

swtloc Stroke Width Transform Localization Library

Modules

10.1 swtloc.swtlocalizer

Description

Classes

SWTLocalizer([multiprocessing, images, ...])	`SWTLocalizer` acts as an entry point for performing Transformations and Localizations.

10.1.1 swtloc.swtlocalizer.SWTLocalizer

class swtloc.swtlocalizer.**SWTLocalizer**(*multiprocessing: Optional[bool] = False*, *images: Optional[Union[numpy.ndarray, List[numpy.ndarray]]] = None*, *image_paths: Optional[Union[str, List[str]]] = None*)

Bases: `object`

`SWTLocalizer` acts as an entry point for performing Transformations and Localizations.

It creates and houses a list of `SWTImage` objects in *swtimages* attribute, after sanity checks have been performed

on the input given.

The inputs can be a path (string) to an image file or an numpy array of the image. Inputs can also be just a single image filepath (string) or a single pre-loaded image (np.ndarray) or it could be a list of either image filepath or list of np.ndarray.

But both the parameters i.e *image_paths* and *images* cannot be provided. Once the inputs provided to the `SWTLocalizer` class, sanity checks are performed on the input, and in case of *images* being provided, random numerical names are assigned to each image(s).

Example:

```
>>> # Import the SWTLocalizer class
>>> from swtloc import SWTLocalizer
>>> from cv2 import cv2
>>>
>>> root_path = 'examples/images/'
>>>
>>> # Single Image Path (NOTE : Use your own image paths)
>>> single_image_path = root_path+'test_image_1/test_img1.jpg'
>>> swtl = SWTLocalizer(image_paths=single_image_path)
>>>
>>> # Multiple Image Paths (NOTE : Use your own image paths)
>>> multiple_image_paths = [root_path+'test_image_2/test_img2.jpg',
>>>                         root_path+'test_image_3/test_img3.jpg',
>>>                         root_path+'test_image_4/test_img4.jpeg' ]
>>> swtl = SWTLocalizer(image_paths=multiple_image_paths)

>>> # Single Pre-Loaded Image - Agnostic to image channels
>>> single_image = cv2.imread(root_path+'test_image_1/test_img1.jpg')
>>> swtl = SWTLocalizer(images=single_image)

>>> # Multiple Pre-Loaded Image
>>> multiple_images = [cv2.imread(each_path) for each_path in [root_path+'test_
↪image_2/test_img2.jpg',
>>>                                                            root_path+'test_
↪image_3/test_img3.jpg',
>>>                                                            root_path+'test_
↪image_4/test_img4.jpeg' ]]
>>> swtl = SWTLocalizer(images=multiple_images)

>>> # Accessing `SWTImage` objects from the `SWTLocalizer`
>>> multiple_images = [cv2.imread(each_path) for each_path in [root_path+'test_
↪image_2/test_img2.jpg',
>>>                                                            root_path+'test_
↪image_3/test_img3.jpg',
>>>                                                            root_path+'test_
↪image_4/test_img4.jpeg' ]]
>>> swtl = SWTLocalizer(images=multiple_images)
>>> print(swtl.swtimages, type(swtl.swtimages[0]))
[Image-SWTImage_982112, Image-SWTImage_571388, Image-SWTImage_866821] <class
↪'swtloc.abstractions.SWTImage'>

>>> # Empty Initialisation -> Raises SWTLocalizerValueError (from v2.1.0)
>>> swtl = SWTLocalizer()
```

(continues on next page)

(continued from previous page)

```
SWTLocalizerValueError: Either `images` or `image_paths` parameters should be␣
↪provided.
>>>
>>>
>>> # Mixed input given  -> Raises SWTLocalizerValueError
>>> mixed_input = [root_path+'test_image_1/test_img1.jpg' , cv2.imread(root_path+
↪'test_image_1/test_img1.jpg')]
>>> swtl = SWTLocalizer(images=mixed_input)
SWTLocalizerValueError: If a list is provided to `images`, each element should be␣
↪an np.ndarray
>>>
>>> # Wrong input type given  -> Raises SWTLocalizerValueError
>>> wrong_input = [True, 1, 'abc', root_path+'test_image_1/test_img1.jpg']
>>> swtl = SWTLocalizer(image_paths=wrong_input)
SWTLocalizerValueError: `image_paths` should be a `list` of `str`
>>>
>>>
>>> # If the file is not present at the location (NOTE : Use your own image paths) -
↪> Raises FileNotFoundError
>>> multiple_image_paths = [root_path+'test_image_2/test_img2.jpg',
>>>                         root_path+'test_image_/image_not_there.jpg',
>>>                         root_path+'test_image_4/test_img4.jpeg' ]
>>> swtl = SWTLocalizer(image_paths=multiple_image_paths)
FileNotFoundError: No image present at ../swtloc/examples/test_images/image_not_
↪there.jpg

>>> # Random Names being assigned to each image when `images` parameter is provided
>>> multiple_images = [cv2.imread(each_path) for each_path in [root_path+'test_
↪image_2/test_img2.jpg',
>>>                                                            root_path+'test_
↪image_3/test_img3.jpg',
>>>                                                            root_path+'test_
↪image_4/test_img4.jpeg' ]]
>>> swtl = SWTLocalizer(images=multiple_images)
>>> print([each_image.image_name for each_image in swtl.swtimages])
['SWTImage_982112', 'SWTImage_571388', 'SWTImage_866821']
```

Methods

`SWTLocalizer.__init__`([multiprocessing, ...])	Create a `SWTLocalizer` object which will house a list of `SWTImage` objects in *swtimage* attribute.

10.2 swtloc.base

Description

Classes

GroupedComponentsBase(label, image_height, ...)	Base class representing the Grouped Components found in an transformed image for example: a Word.
IndividualComponentBase(label, image_height, ...)	Base class representing the Individual Components found in an transformed image for example: a *Letter*.
TextTransformBase(image, image_name, ...)	Base class for various transformation classes.

10.2.1 swtloc.base.GroupedComponentsBase

class swtloc.base.**GroupedComponentsBase**(*label: int*, *image_height: int*, *image_width: int*)

Bases: object

Base class representing the Grouped Components found in an transformed image for example: a Word.

Methods

GroupedComponentsBase.__init__(label, ...)	Create an IndividualComponentBase object which will house the grouped components properties such as : - Various Bounding Shapes which house that particular grouped component entirely :param label: A unique identifier for this Component :type label: int :param image_height: Height of the image in which this component resides :type image_height: int :param image_width: Width of the image in which this component resides :type image_width: int
GroupedComponentsBase.addLocalization(image, ...)	Add a specific *localize_type* of localization to the input *image*.

addLocalization(*image: numpy.ndarray*, *localize_type: str*, *fill: bool*) → numpy.ndarray

Add a specific *localize_type* of localization to the input *image*. *fill* parameter tells whether to fill the component or not.

Parameters

- **image** (*np.ndarray*) – Image on which localization needs to be added
- **localize_type** (*str*) – Type of the localization that will be added. Can be only one of ['bbox', 'bubble', 'polygon']. Where
  - *bbox* : Bounding Box
  - *bubble* : Bubble Boundary
  - *polygon* : Contour Boundary
- **fill** (*bool*) – Whether to fill the added localization or not

Returns (np.ndarray) - annotated image

10.2.2 swtloc.base.IndividualComponentBase

class swtloc.base.**IndividualComponentBase**(*label: int*, *image_height: int*, *image_width: int*)

Bases: object

Base class representing the Individual Components found in an transformed image for example: a *Letter*.

Methods

IndividualComponentBase.__init__(label, ...)	Create an IndividualComponentBase object which will house the components properties such as : - Minimum Bounding Box & its related properties - External Bounding Box & its related properties - Outline (Contour) - Original Image Properties
IndividualComponentBase.addLocalization(...)	Add a specific *localize_type* of localization to the input *image*.

addLocalization(*image: numpy.ndarray*, *localize_type: str*, *fill: bool*, *radius_multiplier: float = 1.0*) → numpy.ndarray

Add a specific *localize_type* of localization to the input *image*. *fill* parameter tells whether to fill the component or not.

Parameters

- **image** (*np.ndarray*) – Image on which localization needs to be added
- **localize_type** (*str*) – Type of the localization that will be added. Can be only one of ['min_bbox', 'ext_bbox', 'outline', 'circular']. Where :
  - *min_bbox* : Minimum Bounding Box
  - *ext_bbox* : External Bounding Box
  - *outline* : Contour
  - *circular* : Circle - With Minimum Bounding Box Centre coordinate and

    radius = Minimum Bounding Box Circum Radius * radius_multiplier
- **fill** (*bool*) – Whether to fill the added localization or not
- **radius_multiplier** (*float*) – Minimum Bounding Box Circum Radius inflation parameter. [default = 1.0].

Returns (np.ndarray) - annotated image

10.2.3 swtloc.base.TextTransformBase

class swtloc.base.**TextTransformBase**(*image: numpy.ndarray*, *image_name: str*, *input_flag: ByteString*, *cfg: Dict*)

Bases: object

Base class for various transformation classes. for example: a SWTImage.

Methods

TextTransformBase.__init__(image, ...)	Create a *TextTransformBase* which will house the properties of the input image, its name, its input type flag, transform configuration and various other parameters corresponding to various stages in the transformation process.

10.3 swtloc.abstractions

Description

Classes

Letter(label, image_height, image_width)	Letter class represents, a letter - an individual component which houses various properties of that individual letter.
SWTImage(image, image_name, input_flag, cfg)	This class houses the procedures for
Word(label, letters, image_height, image_width)	Word class represents, a word - connected component which houses various properties of that individual word.

10.3.1 swtloc.abstractions.Letter

class swtloc.abstractions.**Letter**(*label: int*, *image_height: int*, *image_width: int*)

Bases: *swtloc.base.IndividualComponentBase*

Letter class represents, a letter - an individual component which houses various properties of that individual letter.

Methods

Letter.__init__(label, image_height, image_width)	Create an Letter object which will house the components properties such as : - Minimum Bounding Box & its related properties - External Bounding Box & its related properties - Outline (Contour) - Original Image Properties - Stroke Width Properties :param label: A unique identifier for this Component :type label: int :param image_height: Height of the image in which this component resides :type image_height: int :param image_width: Width of the image in which this component resides :type image_width: int
Letter.addLocalization(image, localize_type, ...)	Add a specific *localize_type* of localization to the input *image*.

addLocalization(*image: numpy.ndarray*, *localize_type: str*, *fill: bool*, *radius_multiplier: float = 1.0*) → numpy.ndarray

Add a specific *localize_type* of localization to the input *image*. *fill* parameter tells whether to fill the

component or not.

Parameters

- **image** (*np.ndarray*) – Image on which localization needs to be added
- **localize_type** (*str*) – Type of the localization that will be added. Can be only one of ['min_bbox', 'ext_bbox', 'outline', 'circular']. Where :
  - *min_bbox* : Minimum Bounding Box
  - *ext_bbox* : External Bounding Box
  - *outline* : Contour
  - *circular* : Circle - With Minimum Bounding Box Centre coordinate and

    radius = Minimum Bounding Box Circum Radius * radius_multiplier
- **fill** (*bool*) – Whether to fill the added localization or not
- **radius_multiplier** (*float*) – Minimum Bounding Box Circum Radius inflation parameter. [default = 1.0].

Returns (np.ndarray) - annotated image

10.3.2 swtloc.abstractions.SWTImage

class swtloc.abstractions.**SWTImage**(*image: numpy.ndarray*, *image_name: str*, *input_flag: ByteString*, *cfg: Dict*)

Bases: *swtloc.base.TextTransformBase*

This class houses the procedures for

- Transforming
- Localizing Letters
- Localizing Words

Objects of this class are made and stored in `SWTLocalizer` class attribute *swtimages*

This class serves as an abstraction to various operations that can be performed via transforming the image through the Stroke Width Transform. This class also includes helper functions to extend the ability to save, show and crop various localizations and intermediary stages as well.

Methods

`SWTImage.__init__`(image, image_name, ...)	Create an `SWTImage`, an abstraction to various procedures to be performed on a ***single*** input image.
SWTImage.getLetter(key[, localize_by, display])	
SWTImage.getWord(key[, localize_by, display])	
SWTImage.letterIterator([localize_by, display])	
SWTImage.localizeLetters([...])	

continues on next page

Table 10 – continued from previous page

SWTImage.localizeWords([localize_by, ...])	
SWTImage.saveCrop(save_path[, crop_of, ...])	
SWTImage.showImage([image_codes, ...])	Function to display a group of ImageCodes (maximum 4), explanation for those codes can be found in the table below : .. csv-table:: :header: Image Code, Explanation.
SWTImage.transformImage([text_mode, engine, ...])	Transform the input image into its Stroke Width Transform.
SWTImage.wordIterator([localize_by, display])	

getLetter(*key: int*, *localize_by: Optional[str] = 'min_bbox'*, *display: Optional[bool] = True*)

Note: This function need to be run only after *localizeLetters* has been run.

Get a particular letter being housed in *letters* attribute

Parameters

- **key** (*int*) – Letter key associated to *letters* attribute
- **localize_by** (*Optional[str]*) – Which localization to apply [default = 'min_bbox'] 1) *min_bbox* - Minimum Bounding Box (Rotating Bounding Box) 2) *ext_bbox* - External Bounding Box 3) *outline* - Contour
- **display** (*Optional[bool]*) – If set to True this will display the following images [default = True] IMAGE_INDIVIDUAL_LETTER_LOCALIZATION = b'17' -> Individual Letter Localized over Pruned RGB Image IMAGE_ORIGINAL_INDIVIDUAL_LETTER_LOCALIZATION = b'18' -> Individual Letter Localized over Original Image

Returns Individual `Letter` which was queried (np.ndarray) : Localization on Edge and SWT Image (np.ndarray) : Localization on Original Image

Return type (*Letter*)

Raises *SWTImageProcessError*, *SWTValueError* –

Example:

```
>>> # Localizing Letters
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixls_per_cc=5200,
```

(continues on next page)

(continued from previous page)

```
>>>                                                 localize_by='min_bbox',␣
↪display=False)

>>> # Access all the letters which have been localized
>>> swtImgObj.letters
{1: Letter-1, 2: Letter-2, 3: Letter-3, 4: Letter-4 ...

>>> # Accessing an individual letter by its key in `swtImgObj.letters`␣
↪dictionary
>>> _letter, _edgeswt_letter, _orig_image_letter = swtImgObj.getLetter(1,␣
↪display=True)

>>> # Accessing `getLetter` for a `localize_by` which hasn't been run already by␣
↪the
>>> # `localizeLetters` function will raise an error -> SWTImageProcessError␣
↪will be raised
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                               maximum_pixels_per_cc=5200,
>>>                                               localize_by='min_bbox',␣
↪display=False)
>>> # Accessing `min_bbox` wont raise any error as that has been run already by␣
↪the localizeLetters function
>>> _letter, _edgeswt_letter, _orig_image_letter = swtImgObj.getLetter(1,␣
↪localize_by='min_bbox', display=True)
>>> # Accessing `ext_bbox` when `ext_bbox` hasn't been run already by the␣
↪localizeLetters function
>>> _letter, _edgeswt_letter, _orig_image_letter = swtImgObj.getLetter(1,␣
↪localize_by='ext_bbox', display=True)
SWTImageProcessError: 'SWTImage.localizeLetters' with localize_by='ext_bbox'␣
↪should be run before this.
>>> # Solution : Run the `localizeLetters` function with `ext_bbox` and then␣
↪access getLetter for `ext_bbox`
>>> localized_letters = swtImgObj.localizeLetters(localize_by='ext_bbox',␣
↪display=False)
>>> _letter, _edgeswt_letter, _orig_image_letter = swtImgObj.getLetter(1,␣
↪localize_by='min_bbox', display=True)
>>> _letter, _edgeswt_letter, _orig_image_letter = swtImgObj.getLetter(1,␣
↪localize_by='ext_bbox', display=True)
```

getWord(*key*, *localize_by: Optional[str] = 'bubble'*, *display: Optional[bool] = True*)

Note: This function can run only after *localizeWords* has been run with parameter *localize_type* parameter.

Get a particular word being housed in *words* attribute

Parameters

- **key** (*int*) – Word key associated to *words* attribute
- **localize_by** (*Optional[str]*) – Which localization to apply 1) *bubble* - Bubble Boundary 2) *bbox* - Bounding Box 3) *polygon* - Contour Boundary
- **display** (*Optional[bool]*) – If set to True, this will show [default = True] IMAGE_INDIVIDUAL_WORD_LOCALIZATION = b'19' -> Individual word localized over Pruned RGB Image IMAGE_ORIGINAL_INDIVIDUAL_WORD_LOCALIZATION = b'20' -> Individual word localized over Original Image

Returns Individual `Word` which was queried (np.ndarray) : Localization on Edge and SWT Image (np.ndarray) : Localization on Original Image

Return type (*Word*)

Raises ***SWTImageProcessError***, ***SWTValueError***, ***SWTTypeError*** –

Example:

```
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              localize_by='min_bbox',␣
↪display=False)
>>> localized_words = swtImgObj.localizeWords(display=False)
>>> # Access all the words which have been localized
>>> swtImgObj.words
{0: Word-0, 1: Word-1, 2: Word-2, 3: Word-3, 4: Word-4, ...
>>> # Accessing an individual word by its key in `swtImgObj.words` dictionary
>>> _word, _edgeswt_word, _orig_image_word = swtImgObj.getWord(1, display=True)

>>> # Accessing `getWord` for a `localize_by` which hasn't been run already by the
>>> # `localizeLetters` function will raise an error -> SWTImageProcessError␣
↪will be raised
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
```

(continues on next page)

(continued from previous page)

```
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              localize_by='min_bbox',␣
↪display=False)
>>> localized_words = swtImgObj.localizeWords(display=False)
>>> # Accessing an individual word by its key in `swtImgObj.words` dictionary
>>> _word, _edgeswt_word, _orig_image_word = swtImgObj.getWord(1, localize_by=
↪'polygon', display=True)
SWTImageProcessError: 'SWTImage.localizeWords' with localize_by='polygon'␣
↪should be run before this.

>>> # Solution:  Run the `localizeWords` function with localize_by=`polygon` and␣
↪then access getWord for `polygon`
>>> localized_words = swtImgObj.localizeWords(localize_by='polygon',␣
↪display=False)
>>> _word, _edgeswt_word, _orig_image_word = swtImgObj.getWord(4, localize_by=
↪'polygon', display=True)
```

letterIterator(*localize_by: Optional[str] = 'min_bbox'*, *display: Optional[bool] = True*)

Note: This function can run only after *localizeLetters* has been for the particular *localize_type*.

Generator to Iterate over all the letters in IPython/Jupyter interactive environment.

Args:

localize_by (Optional[str]) [Which localization to apply [defautl = 'min_bbox']]

1) *min_bbox* - Minimum Bounding Box (Rotating Bounding Box)

2) *ext_bbox* - External Bounding Box

3) *outline* - Contour

display (Optional[bool]) [If set to True this will display the following images [default = True]] IMAGE_INDIVIDUAL_LETTER_LOCALIZATION = b'17' -> Individual Letter Localized over Pruned RGB Image IMAGE_ORIGINAL_INDIVIDUAL_LETTER_LOCALIZATION = b'18' -> Individual Letter Localized over Original Image

Returns Individual `Letter` which was queried (np.ndarray) : Localization on Edge and SWT Image (np.ndarray) : Localization on Original Image

Return type (*Letter*)

Example:

```
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
```

(continues on next page)

(continued from previous page)

```
>>>                                          minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                               maximum_pixels_per_cc=5200,
>>>                                               localize_by='min_bbox',␣
↪display=False)
>>> # (A plot will be displayed as well at every `next` call to this generator␣
↪since display=True)
>>> # Ensure the localize_by parameter has already been run in␣
↪`localizeLetters` function.
>>> localized_letter_generator = swtImgObj.letterIterator(localize_by='min_bbox
↪', display=False)
>>> _letter, _edgeswt_letter, _orig_image_letter = next(localized_letter_
↪generator)
```

localizeLetters(*minimum_pixels_per_cc: Optional[int] = 50*, *maximum_pixels_per_cc: Optional[int] = 10000*, *acceptable_aspect_ratio: Optional[float] = 0.2*, *localize_by: Optional[str] = 'min_bbox'*, *padding_pct: Optional[float] = 0.01*, *display: Optional[bool] = True*) → Dict[int, *swtloc.abstractions.Letter*]

Note: This function need to be run only after *SWTImage.transformImage* has been run.

After having found and pruned the individual connected components, this function add boundaries to the *Letter*'s so found in the *SWTImage.transformImage*.

Parameters

- **minimum_pixels_per_cc** (*Optional[int]*) – Minimum pixels for each components to make it eligible for being a *Letter*. [default = 50]
- **maximum_pixels_per_cc** (*Optional[int]*) – Maximum pixels for each components to make it eligible for being a *Letter*. [default = 10_000]
- **acceptable_aspect_ratio** (*Optional[float]*) – Acceptable Aspect Ratio of each component to make it eligible for being a *Letter*. [default = 0.2]
- **localize_by** (*Optional[str]*) – Which method to localize the letters from : [default = 'min_bbox'] 1) *min_bbox* - Minimum Bounding Box (Rotating Bounding Box) 2) *ext_bbox* - External Bounding Box 3) *outline* - Contour
- **padding_pct** (*Optional[float]*) – How much padding to apply to each localizations [default = 0.01]
- **display** (*Optional[bool]*) – If set to True, this will display the following [default = True] IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS = b'11' -> Localization on Pruned RGB channel image IMAGE_ORIGINAL_LETTER_LOCALIZATIONS = b'12' -> Localization on Original image IMAGE_ORIGINAL_MASKED_LETTER_LOCALIZATIONS = b'13' -> Localization masked on original image

Returns A dictionary with keys as letter labels and values as `Letter` class objects

Return type Dict[int, *Letter*]

Raises ***SWTImageProcessError***, ***SWTValueError***, ***SWTTypeError*** –

Example:

```
>>> # Localizing Letters
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> # (A plot will be displayed as well)
>>>  localized_letters = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                                maximum_pixels_per_cc=5200,
>>>                                                localize_by='min_bbox')

>>> # Running `localizeLetters` before having run `transformImage` -> Raises␣
↪SWTImageProcessError
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> localized_letters = swtImgObj.localizeLetters(localize_by='min_bbox')
SWTImageProcessError: `SWTImage.transformImage` must be called before this␣
↪function
```

localizeWords(*localize_by: Optional[str] = 'bubble'*, *lookup_radius_multiplier: Optional[float] = 1.1*, *acceptable_stroke_width_ratio: Optional[float] = 2.0*, *acceptable_color_deviation: Optional[List] = [13, 13, 13]*, *acceptable_height_ratio: Optional[float] = 1.5*, *acceptable_angle_deviation: Optional[float] = 30.0*, *polygon_dilate_iterations: Optional[int] = 5*, *polygon_dilate_kernel: Optional[int] = (5, 5)*, *display: Optional[bool] = True*) → Dict[int, *swtloc.abstractions.Word*]

Note: This function can run only after *localizeLetters* has been for the particular *localize_type="min_bbox"*.

Once the `letters` attribute has been populated with the pruned connected components, these components can be fused together into `Word`'s. This fusion process is taken care of by the `Fusion` class which groups a `Letter` with another based on comparisons such as :

- Ratio between two individual `Letter`'s
- Ratio between two individual `Letter`'s heights
- Difference between two individual `Letter`'s minimum bounding box rotation angle
- Difference between two individual `Letter`'s color vectors

`Letter`'s which come under consideration of being grouped for a particular `Letter`, will be in the close proximity of the `Letter`, which is gauged by components minimum bouncing box circum circle.

Dilation is performed before finding the localization for a word when *localize_by* parameter is "polygon", so as to merge the nearby bounding box.

Parameters

- **localize_by** (*Optional[str]*) – One of the three localizations can be performed : [default = 'bubble'] - 'bubble' : Bubble Boundary - 'bbox' : Bounding Box - 'polygon' : Contour Boundary

- **lookup_radius_multiplier** (*Optional[float]*) – Circum Radius multiplier, to inflate the lookup range. [default = 1.1]
- **acceptable_stroke_width_ratio** (*Optional[float]*) – Acceptable stroke width ratio between two `Letter`'s to make them eligible to be a part of a word. [default = 2.0]
- **acceptable_color_deviation** (*Optional[List]*) – Acceptable color deviation between two `Letter`'s to make them eligible to be a part of a word.. [default = [13, 13, 13]]
- **acceptable_height_ratio** (*Optional[float]*) – Acceptable height ratio between two `Letter`'s to make them eligible to be a part of a word.. [default = 1.5]
- **acceptable_angle_deviation** (*Optional[float]*) – Acceptable angle deviation between two `Letter`'s to make them eligible to be a part of a word.. [default = 30.0]
- **polygon_dilate_iterations** (*Optional[int]*) – Only required when localize_by = 'polygon'. Number of iterations to be performed before finding contour. [default = 5]
- **polygon_dilate_kernel** (*Optional[int]*) – Only required when localize_by = 'polygon', dilation kernel. [default = (5,5)]
- **display** (*Optional[bool]*) – If set tot True, this function will display . [default = 'bubble'] IMAGE_PRUNED_3C_WORD_LOCALIZATIONS = b'14' -> Pruned RGB Image with Word Localizations IMAGE_ORIGINAL_WORD_LOCALIZATIONS = b'15' -> Original Image with Word Localizations IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS = b'16' -> Original Image mask with Word Localizations

Returns A dictionary with keys as word labels and values as `Word` class objects

Return type Dict[int, *Word*]

Raises ***SWTImageProcessError***, ***SWTValueError***, ***SWTTypeError*** –

Example:

```
>>> # To Localize Words, after having localized Letters
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              localize_by='min_bbox',␣
↪display=False)
>>> # (A plot will be displayed as well)
>>> localized_words = swtImgObj.localizeWords()

>>> # If `localizeWords` is run before having run `localizeLetters`, it will
>>> # raise an error -> SWTImageProcessError will be raised
```

(continues on next page)

(continued from previous page)

```
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_words = swtImgObj.localizeWords()
SWTImageProcessError: `SWTImage.localizeLetters` with localize_by='min_bbox'␣
↪must be called before this function

>>> # Before running `localizeWords` its required that `localizeLetters` has been
>>> # run with localize_by='min_bbox' parameter. Otherwise SWTImageProcessError␣
↪is raised
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              localize_by='ext_bbox',␣
↪display=False)
>>> localized_words = swtImgObj.localizeWords()
SWTImageProcessError: `SWTImage.localizeLetters` with localize_by='min_bbox'␣
↪must be called before this function
```

saveCrop(*save_path: str*, *crop_of: Optional[str] = 'words'*, *crop_key: Optional[int] = 0*, *crop_on: Optional[ByteString] = b'01'*, *crop_type: Optional[str] = 'bubble'*, *padding_pct: Optional[float] = 0.05*)

Note:

- To see the full list of *ImageCodes* (value for *crop_on*) available and their meaning , look at the

showImage function documentation

- For crop_of = 'words', ensure *localizeWords* function has been run prior to this with the same *localize_type* as *crop_type*
- For crop_of = 'letters', ensure *localizeLetters* function has been run prior to this with the same *localize_type* as *crop_type*

Parameters

- **save_path** (*str*) – The directory to save the image at

- **crop_of** (*Optional[str]*) – Generate the crop of 'letters' or 'words'. [default = 'words']
- **crop_key** (*Optional[int]*) – Which key to query from *letters* (if crop_of='letters') or *words* (if crop_of = 'words').[default = 0]
- **crop_on** (*Optional[ByteString]*) – [default = IMAGE_ORIGINAL]
- **crop_type** (*Optional[str]*) – Which localization to crop with. [default = 'bubble'] For crop_of = 'words', available options are :
  - bubble
  - bbox
  - polygon

  For crop_of = 'letters',available options are
  - min_bbox
  - ext_bbox
  - outline
- **padding_pct** (*Optional[float]*) – Padding applied to each localization [default = 0.05]

Raises *SWTValueError*, *SWTImageProcessError*, *SWTTypeError* –

Example:

```
>>> from swtloc import SWTLocalizer
>>> from swtloc.configs import IMAGE_PRUNED_3C_WORD_LOCALIZATIONS
>>> from swtloc.configs import IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              display=False)
>>> localized_words = swtImgObj.localizeWords(display=False)
>>> # To generate and save the crops of `letters`
>>> swtImgObj.saveCrop(save_path='../', crop_of='letters', crop_key=3, crop_
↪on=IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS,
>>>                    crop_type='outline', padding_pct=0.01)
>>> # To generate and save the crops of `words`
>>> swtImgObj.saveCrop(save_path='../', crop_of='words', crop_key=8, crop_
↪on=IMAGE_PRUNED_3C_WORD_LOCALIZATIONS,
>>>                    crop_type='bubble', padding_pct=0.01)
```

(continues on next page)

(continued from previous page)

```
>>> # An error will be raised if `.saveCrops` functions is called for `crop_of=
↪'letters'`
>>> # even before `.localizeLetters` for localize_by = crop_type hasn't been␣
↪called before
>>> # -> SWTImageProcessError will be raised
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> swtImgObj.saveCrop(save_path='../', crop_of='letters', crop_key=3, crop_
↪on=IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS,
>>>                    crop_type='outline', padding_pct=0.01)
Call .localizeLetters method for this Image Code to be populated
SWTImageProcessError: None of the [b'11'] are available!

>>> # An error will be raised if `.saveCrops` functions is called for `crop_of=
↪'words'`
>>> # even before `.localizeWords` for localize_by = crop_type hasn't been␣
↪called before
>>> # -> SWTImageProcessError will be raised
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              display=False)
>>> swtImgObj.saveCrop(save_path='../', crop_of='words', crop_key=8, crop_
↪on=IMAGE_PRUNED_3C_WORD_LOCALIZATIONS,
>>>                    crop_type='bubble', padding_pct=0.01)
Call .localizeWords method for this Image Code to be populated
SWTImageProcessError: None of the [b'14'] are available!
```

showImage(*image_codes: Optional[List[ByteString]] = None*, *plot_title: Optional[str] = 'SWTImage Plot'*, *plot_sup_title: Optional[str] = ''*, *save_dir: Optional[str] = ''*, *save_fig: Optional[bool] = False*, *dpi: Optional[int] = 200*)

Function to display a group of ImageCodes (maximum 4), explanation for those codes can be found in the table below : .. csv-table:

```
:header: Image Code, Explanation

IMAGE_ORIGINAL,  "Original Image"
IMAGE_GRAYSCALE, "Gray-Scaled Image"
IMAGE_EDGED, "Edge Image"
```

(continues on next page)

(continued from previous page)

```
IMAGE_SWT_TRANSFORMED, "SWT Transformed Image"
IMAGE_CONNECTED_COMPONENTS_1C, "Connected Components Single Channel"
IMAGE_CONNECTED_COMPONENTS_3C, "Connected Components RGB Channel"
IMAGE_CONNECTED_COMPONENTS_3C_WITH_PRUNED_ELEMENTS, "Connected Components␣
↪Regions which were pruned (in red)"
IMAGE_CONNECTED_COMPONENTS_PRUNED_1C, "Pruned Connected Components Single␣
↪Channel"
IMAGE_CONNECTED_COMPONENTS_PRUNED_3C, "Pruned Connected Components RGB Channel"
IMAGE_CONNECTED_COMPONENTS_OUTLINE, "Connected Components Outline"
IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS, "Pruned RGB Channel SWT Image With␣
↪Letter Localizations"
IMAGE_ORIGINAL_LETTER_LOCALIZATIONS, "Original Image With Letter Localizations"
IMAGE_ORIGINAL_MASKED_LETTER_LOCALIZATIONS, "Original Image With Masked Letter␣
↪Localizations"
IMAGE_PRUNED_3C_WORD_LOCALIZATIONS, "Pruned RGB Channel SWT Image With Words␣
↪Localizations"
IMAGE_ORIGINAL_WORD_LOCALIZATIONS, "Original Image With Words Localizations"
IMAGE_ORIGINAL_MASKED_WORD_LOCALIZATIONS, "Original Image With Masked Words␣
↪Localizations"
IMAGE_INDIVIDUAL_LETTER_LOCALIZATION, "Individual Letter With Localizations␣
↪Over Edged + SWT"
IMAGE_ORIGINAL_INDIVIDUAL_LETTER_LOCALIZATION, "Individual Letter With␣
↪Localizations Over Original"
IMAGE_INDIVIDUAL_WORD_LOCALIZATION, "Individual Word With Localizations Over␣
↪Edged + SWT"
IMAGE_ORIGINAL_INDIVIDUAL_WORD_LOCALIZATION, "Individual Word With␣
↪Localizations Over Original"
```

Parameters

- **image_codes** (*Optional[List[ByteString]]*) – List of image codes to display. [default = IMAGE_ORIGINAL]
- **plot_title** (*Optional[str]*) – Title of the plot
- **plot_sup_title** (*Optional[str]*) – Sub title of the plot
- **save_dir** (*Optional[str]*) – Directory in which to save the prepared plot
- **save_fig** (*Optional[bool]*) – Whether to save the prepared plot or not
- **dpi** (*Optional[int]*) – DPI of the figure to be saved

Raises *SWTValueError*, *SWTTypeError* –

Returns Returns the location where the image was saved if save_dir=True and save_path is given.

Return type (str)

Example:

```
>>> from swtloc import SWTLocalizer
>>> from swtloc.configs import IMAGE_ORIGINAL
>>> from swtloc.configs import IMAGE_SWT_TRANSFORMED
>>> from swtloc.configs import IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS
```

(continues on next page)

(continued from previous page)

```
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              display=False)
>>> swtImgObj.showImage(image_codes=[IMAGE_ORIGINAL,
>>>                                  IMAGE_SWT_TRANSFORMED,
>>>                                  IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS],
>>>                     plot_title="Process Flow",
>>>                     plot_sup_title="Original -> SWT -> Pruned Letters")

>>> # (A plot will be displayed as well) + Save the prepared plot
>>> localized_letter = swtImgObj.localizeLetters(display=False)
>>> swtImgObj.showImage(image_codes=[IMAGE_ORIGINAL,
>>>                                  IMAGE_SWT_TRANSFORMED,
>>>                                  IMAGE_PRUNED_3C_LETTER_LOCALIZATIONS],
>>>                     plot_title="Process Flow",
>>>                     plot_sup_title="Original -> SWT -> Pruned Letters",
>>>                     save_dir='../', save_fig=True, dpi=130)
```

transformImage(*text_mode: Optional[str] = 'lb_df'*, *engine: Optional[str] = 'numba'*, *gaussian_blurr: Optional[bool] = True*, *gaussian_blurr_kernel: Optional[Tuple] = (5, 5)*, *edge_function: Optional[Union[str, Callable]] = 'ac'*, *auto_canny_sigma: Optional[float] = 0.33*, *minimum_stroke_width: Optional[int] = 3*, *maximum_stroke_width: Optional[int] = 200*, *check_angle_deviation: Optional[bool] = True*, *maximum_angle_deviation: Optional[float] = 0.5235987755982988*, *include_edges_in_swt: Optional[bool] = True*, *display: Optional[bool] = True*) → numpy.ndarray

Transform the input image into its Stroke Width Transform. The entire transformation follows the following flow

- Step-1 : Convert To Gray-Scale
- Step-2 : Apply Gaussian Blurr
- Step-3 : Find Edge of the Image
- Step-4 : Calculate the Image Gradient Theta Angle
- Step-5 : Calculate the Step Matrices
- Step-6 : Apply Stroke Width Transformation

This function also stores the time taken to complete all the above mentioned stages in the class attribute *transform_time*

Parameters

- **text_mode** (*Optional[str]*) – Contrast of the text present in the image, which needs to be transformed. Two possible values :

1) ”db_lf” :> Dark Background Light Foreground i.e Light color text on Dark color background

2) ”lb_df” :> Light Background Dark Foreground i.e Dark color text on Light color background

This parameters affect how the gradient vectors (the direction) are calculated, since gradient vectors of db_lf are in direction to that of lb_df gradient vectors. [default = ‘lb_df’]

- **engine** (*Optional[str]*) – Which engine to use for applying the Stroke Width Transform. [default = ‘numba’] 1) “python” : Use *Python* for running the *findStrokes* function 2) “numba” : Use *numba* for running the *findStrokes* function
- **gaussian_blurr** (*Optional[bool]*) – Whether to apply gaussian blurr or not. [default = True]
- **gaussian_blurr_kernel** (*Optional[Tuple]*) – Kernel to use for gaussian blurr. [default = (5, 5)]
- **edge_function** (*Optional[str, Callable]*) – Finding the Edge of the image is a tricky part, this is pertaining to the fact that in most of the cases the images we deal with are not of standard that applying just a opencv Canny operator would result in the desired Edge Image. Sometimes (In most cases) there is some custom processing required before edging, for that reason alone this parameter accepts one of the following two values :-

  1.) ‘ac’ :> Auto-Canny function, an in-built function which will generate the Canny Image from the original image, internally calculating the threshold parameters, although, to tune it even further ‘ac_sigma : float, default(0.33)’ parameter is provided which can take any value between 0.0 <–> 1.0.

  2.) A custom function : This function should have its signature as mentioned below :

```
>>> def custom_edge_func(gray_image):
>>>     # Your Function Logic...
>>>     edge_image =
>>>     return edge_image
```

- **auto_canny_sigma** – (Optional[float]) : Value of the sigma to be used in the edging function, if *edge_function* parameter is given the value “ac”. [default = 0.33]
- **minimum_stroke_width** (*Optional[int]*) – Maximum permissible stroke width. [default = 0.33]
- **maximum_stroke_width** (*Optional[int]*) – Minimum permissible stroke width. [default = 0.33]
- **check_angle_deviation** (*Optional[bool]*) – Whether to check the angle deviation to terminate the ray. [default = 0.33]
- **maximum_angle_deviation** (*Optional[float]*) – Maximum Angle Deviation which would be permissible. [default = 0.33]
- **include_edges_in_swt** (*Optional[bool]*) – Whether to include edges (those edges, from which no stroke was able to be determined) in the final swt transform
- **display** (*Optional[bool]*) – If set to True, the images corresponding to following image codes will be displayed . [default = True]

IMAGE_ORIGINAL = b'01' -> Original Image IMAGE_GRAYSCALE = b'02' -> Gray Sclaed Image IMAGE_EDGED = b'03' -> Edged Image IMAGE_SWT_TRANSFORMED = b'04' -> SWT Transformed image converted to three channels .. note :

IMAGE_SWT_TRANSFORMED is not the same as the image array returned from this function.

Returns Stroke Width Transform of the image.

Return type (np.ndarray)

Raises *SWTValueError*, *SWTTypeError* –

Example:

```
>>> # Transform the image using the default engine [default : engine='numba']
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=True)
>>> # (A plot will be displayed as well)
>>> print('Time Taken', swtImgObj.transform_time)
Time Taken 0.193 sec

>>> # Python engine been used for `transformImage` for engine = 'python'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, engine='python')
>>> # (A plot will be displayed as well)
>>> print('Time Taken', swtImgObj.transform_time)
Time Taken 3.822 sec

>>> # Wrong Input given -> SWTValueError/SWTTypeError will be raised
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='asc', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5.1, maximum_
↪stroke_width=50)
SWTTypeError: `minimum_stroke_width` value should be one of these types : [
↪<class 'int'>]. Not mixed either.

>>># Custom edge function been given to the `transformImage`
>>> def custom_edge_func(gray_image):
```

(continues on next page)

(continued from previous page)

```
>>>     gauss_image = cv2.GaussianBlur(gray_image, (5,5), 1)
>>>     laplacian_conv = cv2.Laplacian(gauss_image, -1, (5,5))
>>>     canny_edge = cv2.Canny(laplacian_conv, 20, 90)
>>>     return canny_edge
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function=custom_edge_func,␣
↪gaussian_blurr_kernel=(3,3),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50)
```

wordIterator(*localize_by: Optional[str] = 'bubble'*, *display: Optional[bool] = True*)

Note: This function can run only after *localizeWords* has been run with parameter *localize_type* parameter.

Get a particular word being housed in *words* attribute

Parameters

- **localize_by** (*Optional[str]*) – Which localization to apply - *bubble* - Bubble Boundary - *bbox* - Bounding Box - *polygon* - Contour Boundary
- **display** (*Optional[bool]*) – If set to True, this will show [default = True] IMAGE_INDIVIDUAL_WORD_LOCALIZATION = b'19' -> Individual word localized over Pruned RGB Image IMAGE_ORIGINAL_INDIVIDUAL_WORD_LOCALIZATION = b'20' -> Individual word localized over Original Image

Returns Individual `Word` which was queried (np.ndarray) : Localization on Edge and SWT Image (np.ndarray) : Localization on Original Image

Return type (*Word*)

Raises ***SWTImageProcessError***, ***SWTValueError***, ***SWTTypeError*** –

Example:

```
>>> from swtloc import SWTLocalizer
>>> root_path = 'examples/images/'
>>> swtl = SWTLocalizer(image_paths=root_path+'test_image_1/test_img1.jpg')
>>> swtImgObj = swtl.swtimages[0]
>>> swt_image = swtImgObj.transformImage(text_mode='db_lf', maximum_angle_
↪deviation=np.pi/2,
>>>                                      edge_function='ac', gaussian_blurr_
↪kernel=(11, 11),
>>>                                      minimum_stroke_width=5, maximum_
↪stroke_width=50, display=False)
>>> localized_letter = swtImgObj.localizeLetters(minimum_pixels_per_cc=950,
>>>                                              maximum_pixels_per_cc=5200,
>>>                                              localize_by='min_bbox',␣
↪display=False)
>>> localized_words = swtImgObj.localizeWords(localize_by='polygon',␣
↪display=False, polygon_dilate_iterations=3)
```

(continues on next page)

(continued from previous page)

```
>>> # Creating a generator for a specific localize_by
>>> word_iterator = swtImgObj.wordIterator(localize_by='polygon', display=True)

>>> _word, _edgeswt_word, _orig_image_word = next(word_iterator)
```

10.3.3 swtloc.abstractions.Word

class swtloc.abstractions.**Word**(*label: int*, *letters: List[*swtloc.abstractions.Letter*]*, *image_height: int*, *image_width: int*)

Bases: *swtloc.base.GroupedComponentsBase*

Word class represents, a word - connected component which houses various properties of that individual word.

Methods

Word.__init__(label, letters, image_height, ...)	Create an Word object which will house the grouped components properties such as : - Various Bounding Shapes which house that particular grouped component entirely :param letters: Letters which can be grouped into this word. :type letters: List[Letter] :param label: A unique identifier for this Component :type label: int :param image_height: Image height :type image_height: int :param image_width: Image Width :type image_width: int.
Word.addLocalization(image, localize_type, fill)	Add a specific *localize_type* of localization to the input *image*.

addLocalization(*image: numpy.ndarray*, *localize_type: str*, *fill: bool*) → numpy.ndarray

Add a specific *localize_type* of localization to the input *image*. *fill* parameter tells whether to fill the component or not.

Parameters

- **image** (*np.ndarray*) – Image on which localization needs to be added
- **localize_type** (*str*) – Type of the localization that will be added. Can be only one of ['bbox', 'bubble', 'polygon']. Where
  - *bbox* : Bounding Box
  - *bubble* : Bubble Boundary
  - *polygon* : Contour Boundary
- **fill** (*bool*) – Whether to fill the added localization or not

Returns (np.ndarray) - annotated image

10.4 swtloc.core

Description

Functions

swt_strokes(edged_image, hstep_mat, ...)	Core Logic for Stroke Width Transform.
swt_strokes_jitted(edged_image, hstep_mat, ...)	Core Logic for Stroke Width Transform.

10.4.1 swtloc.core.swt_strokes

swtloc.core.**swt_strokes**(*edged_image*, *hstep_mat*, *vstep_mat*, *dstep_mat*, *max_stroke_width*, *min_stroke_width*, *image_height*, *image_width*, *check_angle_deviation*, *image_gradient_theta*, *max_angle_deviation*, *include_edges_in_swt*)

Core Logic for Stroke Width Transform. Implementing the work of [Boris Epshtein, Eyal Ofek & Yonatan Wexler](https://www.microsoft.com/en-us/research/publication/detecting-text-in-natural-scenes-with-stroke-width-transform/)

Objective of this function is to, given an edged input image, find the stroke widths conforming to the following rules :

- Each Stroke Width has be in the range of : min_stroke_width<= stroke_widths<=max_stroke_width
- A ray emanating from each edge point, traveling in its gradients direction, when met with another

edge point will terminate its journey only when the difference between their gradient directional angles is np.pi - max_angle_deviation <= theta_diff <= np.pi + max_angle_deviation

Parameters

- **edged_image** (*np.ndarray*) – Edges of the Original Input Image. Same size as the original image
- **hstep_mat** (*np.ndarray*) – For each pixel, cos(gradient_theta), where gradient_theta is the gradient angle for that pixel, representing length of horizontal movement for every unit movement in gradients direction. Same size as the original image
- **vstep_mat** (*np.ndarray*) – For each pixel, sin(gradient_theta), where gradient_theta is the gradient angle for that pixel, representing length of vertical movement for every unit movement in gradients direction. Same size as the original image
- **dstep_mat** (*np.ndarray*) – np.sqrt(hstep_mat**2+vstep_mat**2)
- **max_stroke_width** (*int*) – Maximum Stroke Width which would be permissible
- **min_stroke_width** (*int*) – Minimum Stroke Width which would be required
- **image_height** (*int*) – Height of the image
- **image_width** (*int*) – Width of the image
- **check_angle_deviation** (*bool*) – Whether to check the angle deviation to terminate the ray
- **image_gradient_theta** (*np.ndarray*) – Gradient array of the input image

- **max_angle_deviation** (*float*) – Maximum Angle Deviation which would be permissible
- **include_edges_in_swt** (*bool*) – Whether to include edges in the final SWT result

Returns Stroke Width Transformed Image, each stroke filled with stroke length.

Return type (np.ndarray)

10.4.2 swtloc.core.swt_strokes_jitted

swtloc.core.**swt_strokes_jitted**(*edged_image*, *hstep_mat*, *vstep_mat*, *dstep_mat*, *max_stroke_width*, *min_stroke_width*, *image_height*, *image_width*, *check_angle_deviation*, *image_gradient_theta*, *max_angle_deviation*, *include_edges_in_swt*)

Core Logic for Stroke Width Transform. Implementing the work of [Boris Epshtein, Eyal Ofek & Yonatan Wexler](https://www.microsoft.com/en-us/research/publication/detecting-text-in-natural-scenes-with-stroke-width-transform/)

Objective of this function is to, given an edged input image, find the stroke widths conforming to the following rules :

- Each Stroke Width has be in the range of : min_stroke_width<= stroke_widths<=max_stroke_width
- A ray emanating from each edge point, traveling in its gradients direction, when met with another

edge point will terminate its journey only when the difference between their gradient directional angles is np.pi - max_angle_deviation <= theta_diff <= np.pi + max_angle_deviation

Parameters

- **edged_image** (*np.ndarray*) – Edges of the Original Input Image. Same size as the original image
- **hstep_mat** (*np.ndarray*) – For each pixel, cos(gradient_theta), where gradient_theta is the gradient angle for that pixel, representing length of horizontal movement for every unit movement in gradients direction. Same size as the original image
- **vstep_mat** (*np.ndarray*) – For each pixel, sin(gradient_theta), where gradient_theta is the gradient angle for that pixel, representing length of vertical movement for every unit movement in gradients direction. Same size as the original image
- **dstep_mat** (*np.ndarray*) – np.sqrt(hstep_mat**2+vstep_mat**2)
- **max_stroke_width** (*int*) – Maximum Stroke Width which would be permissible
- **min_stroke_width** (*int*) – Minimum Stroke Width which would be required
- **image_height** (*int*) – Height of the image
- **image_width** (*int*) – Width of the image
- **check_angle_deviation** (*bool*) – Whether to check the angle deviation to terminate the ray
- **image_gradient_theta** (*np.ndarray*) – Gradient array of the input image
- **max_angle_deviation** (*float*) – Maximum Angle Deviation which would be permissible
- **include_edges_in_swt** (*bool*) – Whether to include edges in the final SWT result

Returns Stroke Width Transformed Image, each stroke filled with stroke length.

Return type (np.ndarray)

Classes

Fusion(letters, ...)	Class for fusing Individual Components (Letters) into Grouped Components Words, comparing aspects like : - Proximity of letters to each other - Relative minimum bounding box rotation angle from each other - Deviation in color between from one component to the other - Ratio of stroke widths from one to the other - Ratio of minimum bounding box height of one to the other
ProxyLetter(label, sw_median, color_median, ...)	A proxy class for the `Letters` object, housing only those properties which would be required by the Fusion Class.

10.4.3 swtloc.core.Fusion

class swtloc.core.**Fusion**(*letters: dict*, *acceptable_stroke_width_ratio: float*, *acceptable_color_deviation: List[int]*, *acceptable_height_ratio: float*, *acceptable_angle_deviation: float*)

Bases: `object`

Class for fusing Individual Components (Letters) into Grouped Components Words, comparing aspects like :

- Proximity of letters to each other
- Relative minimum bounding box rotation angle from each other
- Deviation in color between from one component to the other
- Ratio of stroke widths from one to the other
- Ratio of minimum bounding box height of one to the other

Methods

`Fusion.__init__`(letters, ...)	Create `Fusion` object
Fusion.getProximityLetters(anchor_letter, ...)	Finds all the labels which are in proximity of anchor_letter amongst the remaining_letters
Fusion.groupEligibility(curr_letter, ...)	Check whether two ProxyLetters are eligible to be grouped with one another.
Fusion.groupLetters(curr_letter, ...)	Groups curr_letter with its proximity labels which are eligible to be grouped to it.
Fusion.runGrouping()	Fuses eligible individual components (letters) together which can be eligible to form a *word* out of them.

getProximityLetters(*anchor_letter:* swtloc.core.ProxyLetter, *remaining_letters: dict*) → List[int]

Finds all the labels which are in proximity of anchor_letter amongst the remaining_letters

Parameters

- **anchor_letter** (`ProxyLetter`) – Letter with respect to which proximity labels are

to be searched.

- **remaining_letters** (*dict*) – A dictionary, with labels as keys, mapped to their corresponding ProxyLetter object.

Returns List of all the labels which are in the proximity of anchor_letter

Return type (List[int])

groupEligibility(*curr_letter*, *proximity_letter*) → bool

Check whether two ProxyLetters are eligible to be grouped with one another.

Parameters

- **curr_letter** (ProxyLetter) – Current Letter
- **proximity_letter** (ProxyLetter) – A letter in proximity of Current Letter

Returns Whether curr_letter and proximity_letter are eligible to be grouped with each other

Return type (bool)

groupLetters(*curr_letter*, *remaining_letters*, *grouping*) → List[*swtloc.core.ProxyLetter*]

Groups curr_letter with its proximity labels which are eligible to be grouped to it. [Recursive Function]

Parameters

- **curr_letter** (ProxyLetter) – ProxyLetter whose grouping needs to be mapped.
- **remaining_letters** (*dict*) – Dictionary with keys as the ProxyLetter label and the corresponding values as the ProxyLetter themselves.
- **grouping** (*list*) – A list of lists containing ProxyLetters which can be assumed to be *words*.

Returns

A list containing ProxyLetters which can be assumed to be a *word*.

Return type (List[*ProxyLetter*])

runGrouping() → List[List[*swtloc.core.ProxyLetter*]]

Fuses eligible individual components (letters) together which can be eligible to form a *word* out of them.

Returns

A list of lists containing ProxyLetter which can be assumed to be *words* amongst the pool of individual components provided to the Fusion class.

Return type (List[List[*ProxyLetter*]])

10.4.4 swtloc.core.ProxyLetter

class swtloc.core.**ProxyLetter**(*label*, *sw_median*, *color_median*, *min_height*, *min_angle*, *inflated_radius*, *circular_mask*, *min_label_mask*)

Bases: object

A proxy class for the Letters object, housing only those properties which would be required by the Fusion Class. This is to support application of *numba* onto the Fusion Class as the Letter class object wont be acceptable by Fusion class were it to be run on nopython-jit mode

Methods

ProxyLetter.__init__(label, sw_median, ...)	Create a ProxyLetter object :param label: Letter identifier :type label: int :param sw_median: Median stroke width of this letter :type sw_median: float :param color_median: Median Color of this letter :type color_median: float :param min_height: Minimum Bounding Box height of this letter :type min_height: int :param min_angle: Rotation angle of the Minimum Bounding Box of this letter :type min_angle: float :param inflated_radius: Inflated Circum-Radius of the Minimum Bounding Box of this letter :type inflated_radius: int :param circular_mask: Circular filled mask of this letter of radius=inflated_radius and :type circular_mask: np.ndarray :param centre=Centre Co-Ordinates of the Minimum Bounding Box.: :param min_label_mask: Filled Minimum Bounding Box of the letter :type min_label_mask: np.ndarray

10.5 swtloc.utils

Description

Functions

auto_canny(img[, sigma])	Autocanny Function.
deprecated_wrapper(reason, in_favour_of, ...)	A decorator to mark the deprecated functions and give the reason of deprecation. Alongside the reason, also inform in which version will the function be removed. If the function is being relocated then also inform in favour of which function will this function will be relocated :param reason: Reason for relocation/deprecation :type reason: str :param in_favour_of: If relocated, in favour of which function is this function being relocated :type in_favour_of: str :param removed_in: In which version will the function be removed :type removed_in: str :param relocated: Is the deprecation for being relocated :type relocated: Optional[bool].
generate_random_swtimage_names(n)	Generates random image names made of random numbers :param n: Number of names to generate :type n: int
get_connected_components_with_stats(img)	Function to find the connected components alongside their stats using oepncv's connectedComponentWithStats function for any given image.
image_1C_to_3C(image_1C[, all_colors, ...])	Prepare the RGB channel image from a single channel image (gray-scale).
perform_type_sanity_checks(cfg, cfg_of)	Perform type sanity checks for the values provided in *cfg* for a particular *cfg_of* configuration type.
print_in_red(text)	Print a red color text.

continues on next page

Table 16 – continued from previous page

show_N_images(images, individual_titles[, ...])	Display n (<=4) images in a grid.
unique_value_counts(image[, remove_0])	Calculate unique integers in an image and their counts

10.5.1 swtloc.utils.auto_canny

swtloc.utils.**auto_canny**(*img: numpy.ndarray*, *sigma: Optional[float] = 0.33*) → numpy.ndarray

Autocanny Function. Taken from : https://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/ Function to find Edge image from a grayscale image based on the thresholding parameter sigma.

Parameters

- **img** (*np.ndarray*) – Input gray-scale image.
- **sigma** (*Optional[float]*) – Sigma Value,default : 0.33

Example:

```
>>> # Generating Canny Edge Image
>>> root_path = '../swtloc/examples/test_images/'
>>> single_image_path = root_path+'test_img1.jpg'
>>> original_image = cv2.imread(single_image_path)
>>> edge_image = auto_canny(img=original_image, sigma=0.2)
>>> print(original_image.shape, edge_image.shape)
(768, 1024, 3) (768, 1024)
```

10.5.2 swtloc.utils.deprecated_wrapper

swtloc.utils.**deprecated_wrapper**(*reason: str*, *in_favour_of: str*, *removed_in: str*, *relocated: Optional[bool] = False*)

A decorator to mark the deprecated functions and give the reason of deprecation. Alongside the reason, also inform in which version will the function be removed. If the function is being relocated then also inform in favour of which function will this function will be relocated :param reason: Reason for relocation/deprecation :type reason: str :param in_favour_of: If relocated, in favour of which function is this

function being relocated

Parameters

- **removed_in** (*str*) – In which version will the function be removed
- **relocated** (*Optional[bool]*) – Is the deprecation for being relocated

10.5.3 swtloc.utils.generate_random_swtimage_names

swtloc.utils.**generate_random_swtimage_names**(*n: int*) → List[str]

Generates random image names made of random numbers :param n: Number of names to generate :type n: int

Returns List of string names to generate.

Return type (List[str])

Example:

```
>>> # Generating `n` random integer string (names)
>>> generate_random_swtimage_names(3)
['SWTImage_982112', 'SWTImage_571388', 'SWTImage_866821']
```

10.5.4 swtloc.utils.get_connected_components_with_stats

swtloc.utils.**get_connected_components_with_stats**(*img: numpy.ndarray*)

Function to find the connected components alongside their stats using oepncv's connectedComponentWithStats function for any given image.

Parameters **img** (*np.ndarray*) – Input Image

Returns Results of opencv connectedComponentsWithStats

Return type (Tuple[int, np.ndarray, np.ndarray, np.ndarray])

10.5.5 swtloc.utils.image_1C_to_3C

swtloc.utils.**image_1C_to_3C**(*image_1C: numpy.ndarray*, *all_colors: Optional[List[Tuple[int]]] = None*, *scale_with_values: Optional[bool] = False*) → numpy.ndarray

Prepare the RGB channel image from a single channel image (gray-scale). Each unique integer in *image_1C* will be given a unique color, unless *all_colors* parameter is provided. *scale_with_values* parameter ensures color so generated (if *all_colors* parameter not given) will be generated using the *sequential* matplotlib color scheme.

Parameters

- **image_1C** (*np.ndarray*) – Input single channel image, which needs to be transformed
- **all_colors** (*Optional[List[Tuple[int]]]*) – Colors corresponding to each unique integer in the image
- **scale_with_values** (*Optional[bool]*) – Whether to use matplotlib *sequential* color map or not.

Returns Three channel image, after the conversions of the single channel image

Return type (np.ndarray)

10.5.6 swtloc.utils.perform_type_sanity_checks

swtloc.utils.**perform_type_sanity_checks**(*cfg: Dict*, *cfg_of: str*) → None
Perform type sanity checks for the values provided in *cfg* for a particular *cfg_of* configuration type. :param cfg: Configuration dictionary :type cfg: dict :param cfg_of: Configuration of which Type Sanity Checks need to be performed :type cfg_of: str

Raises *SWTTypeError*, *SWTValueError* –

10.5.7 swtloc.utils.print_in_red

swtloc.utils.**print_in_red**(*text: str*) → None
Print a red color text. :param text: Text to print :type text: str

Example:

```
>>> # Printing a red text
>>> print_in_red('This is a red text')
This is a red text
```

10.5.8 swtloc.utils.show_N_images

swtloc.utils.**show_N_images**(*images: List[numpy.ndarray]*, *individual_titles: List[str]*, *plot_title: Optional[str] = 'SWTLoc Plot'*, *sup_title: Optional[str] = ''*, *return_img: Optional[bool] = False*) → Optional[numpy.ndarray]
Display n (<=4) images in a grid.

Parameters

- **images** (*List[np.ndarray]*) – List of images to display
- **individual_titles** (*List[str]*) – Title for each image to be displayed
- **plot_title** (*Optional[str]*) – Plot Title. [default = 'SWTLoc Plot']
- **sup_title** (*Optional[str]*) – Plot sub title. [default = '']
- **return_img** (*Optional[bool]*) – Whether to return the plotted figure or not. [default = False]

Raises *SWTValueError* –

Returns Plotted figure if the *return_fig* parameter was given as *True*

Return type (matplotlib.figure.Figure)

10.5.9 swtloc.utils.unique_value_counts

swtloc.utils.**unique_value_counts**(*image: numpy.ndarray*, *remove_0: Optional[bool] = True*) → Dict
Calculate unique integers in an image and their counts

Parameters

- **image** (*np.ndarray*) – Image of which the unique values need to be calculated
- **remove_0** (*Optional[bool]*) – Whether to remove integer 0 from the calculated dictionary or not.

Returns

Dictionary containing key as unique integer and value as counts of that integer in the image

Return type (dict)

Exceptions

SWTImageProcessError	Raised when functions don't follow a proper flow for SWTImage
SWTLocExceptions	Base class for SWTLoc Exceptions
SWTLocalizerValueError	Raised when wrong input is provided to the *SWTLocalizer* class
SWTTypeError	Raised when non-acceptable type is received in the parameters
SWTValueError	Raised when non-acceptable value is received in the parameters

10.5.10 swtloc.utils.SWTImageProcessError

class swtloc.utils.**SWTImageProcessError**

Bases: *swtloc.utils.SWTLocExceptions*

Raised when functions don't follow a proper flow for SWTImage

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

10.5.11 swtloc.utils.SWTLocExceptions

class swtloc.utils.**SWTLocExceptions**

Bases: Exception

Base class for SWTLoc Exceptions

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

10.5.12 swtloc.utils.SWTLocalizerValueError

class swtloc.utils.**SWTLocalizerValueError**

Bases: *swtloc.utils.SWTLocExceptions*

Raised when wrong input is provided to the *SWTLocalizer* class

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

10.5.13 swtloc.utils.SWTTypeError

class swtloc.utils.**SWTTypeError**
Bases: *swtloc.utils.SWTLocExceptions*

Raised when non-acceptable type is received in the parameters

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

10.5.14 swtloc.utils.SWTValueError

class swtloc.utils.**SWTValueError**
Bases: *swtloc.utils.SWTLocExceptions*

Raised when non-acceptable value is received in the parameters

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

T

U

W