py-lmd

Release 1.0.1

Georg Wallmann, Sophia Mädler and Niklas Schmacke

Feb 18, 2022

CONTENTS:

Table of Contents

Welcome to the show

CHAPTER

ONE

QUICK START

1.1 Installation from Github

To install the py-lmd library clone the Github repository and use pip to install the library in your current environment. It is recommended to use the library with a Conda environment. Please make sure that the package is installed editable like described. Otherwise static glyph files might not be available.

```
git clone https://github.com/HornungLab/py-lmd
pip install -e .
```

Once installed the modules can be loaded as following:

```
from lmd.lib import Collection, Shape
```

1.2 Generating Shapes

As first example we will create a cutting data for a rectangle and visualize it. First we need to think of a calibration points for our coordinate system. The calibration points are specified as Numpy array and should have the shape *(3, 2)*. When calibrating the file on the Leica LMD after loading the file, the order of selecting the calibration points is the same as the order of points set here. If no other orientation transform has been specified, the calibration points are defined in the *(x, y)* coordinate system. More on the usage of different coordinate systems can be found under XXX.

```
import numpy as np
from lmd.lib import Collection, Shape

calibration = np.array([[0, 0],
                        [0, 100],
                        [50, 50]])
```

With these calibration coordinates we can create our `Collection` object. The `Collection` is the base object for creating cutting data and holds a list of `Shape` objects. Furthermore, it allows to read and write the Leica LMD XML format and handles the coordinate system.

```
my_first_collection = Collection(calibration_points = calibration)
```

We can then create our first rectangle by using the `Shape` object and passing it to our collection by using the `add_shape` method. For creating the Shape object we need to pass the vertices as Numpy array. List of vertices should always be closed with the last vertex equaling the first one.

```
rectangle_coordinates = np.array([[10,10],
                                  [40,10],
                                  [40,40],
```

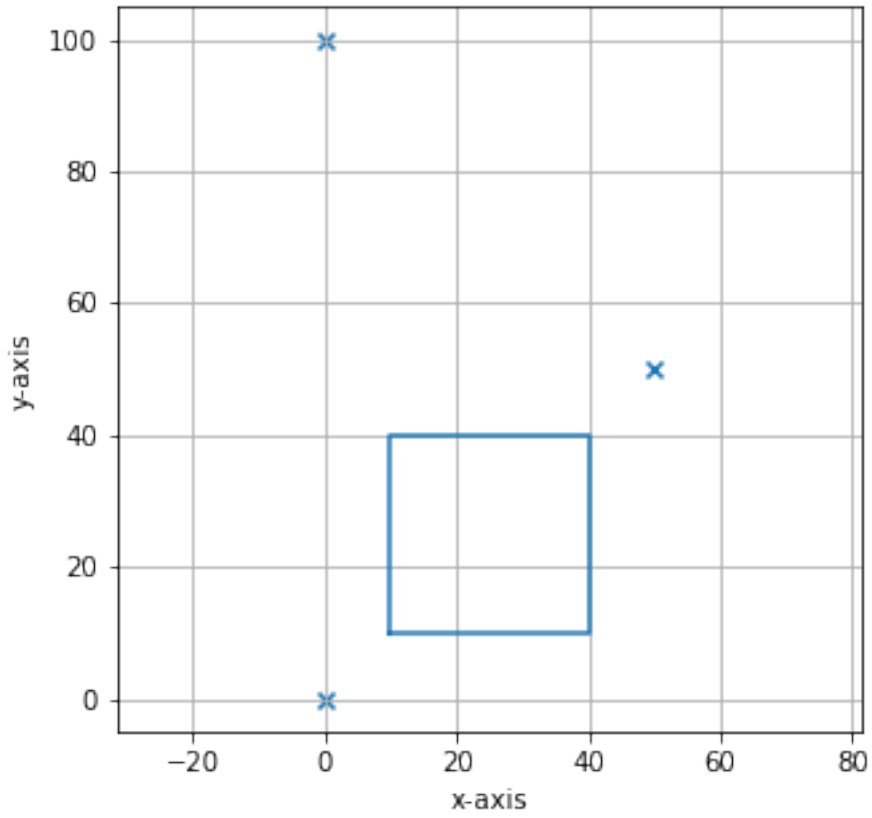
(continues on next page)

(continued from previous page)

```
                                    [10,40],
                                    [10,10]])
rectangle = Shape(rectangle_coordinates)
my_first_collection.add_shape(rectangle)
```

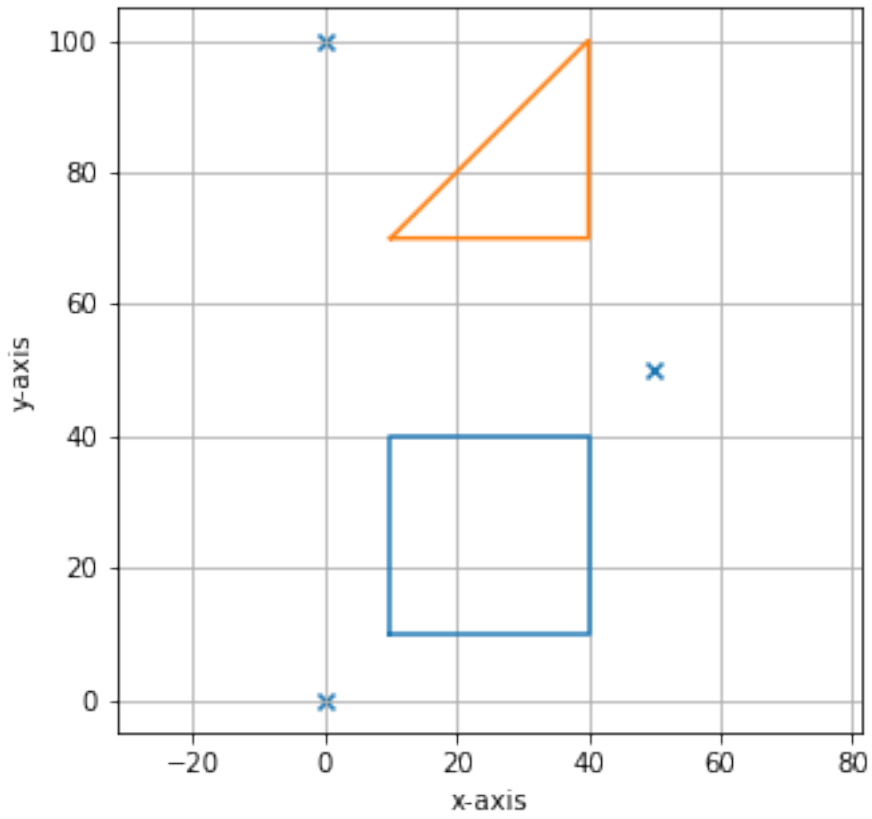
We can visualize our collection with the `plot` method. Using the *calibration* = *True* parameter will include the calibration coordinates in the plot.

```
my_first_collection.plot(calibration = True)
```



We can generate a second shape in form of a triangle. This time we will be using the `new_shape` method of the collection object.

```
triangle_coordinates = np.array([[10,70], [40,70], [40,100], [10,70]])
my_first_collection.new_shape(triangle_coordinates)

my_first_collection.plot(calibration = True)
```



We can then export and save our collection of shapes into xml cutting data.

```
my_first_collection.save("first_collection.xml")
```

```
<?xml version='1.0' encoding='UTF-8'?>
<ImageData>
  <GlobalCoordinates>1</GlobalCoordinates>
  <X_CalibrationPoint_1>0</X_CalibrationPoint_1>
  <Y_CalibrationPoint_1>0</Y_CalibrationPoint_1>
  <X_CalibrationPoint_2>0</X_CalibrationPoint_2>
  <Y_CalibrationPoint_2>10000</Y_CalibrationPoint_2>
  <X_CalibrationPoint_3>5000</X_CalibrationPoint_3>
  <Y_CalibrationPoint_3>5000</Y_CalibrationPoint_3>
  <ShapeCount>2</ShapeCount>
  <Shape_1>
    <PointCount>5</PointCount>
    <X_1>1000</X_1>
    <Y_1>1000</Y_1>
    <X_2>4000</X_2>
    <Y_2>1000</Y_2>
    <X_3>4000</X_3>
    <Y_3>4000</Y_3>
    <X_4>1000</X_4>
    <Y_4>4000</Y_4>
    <X_5>1000</X_5>
    <Y_5>1000</Y_5>
```

(continues on next page)

(continued from previous page)

```
  </Shape_1>
  <Shape_2>
    <PointCount>4</PointCount>
    <X_1>1000</X_1>
    <Y_1>7000</Y_1>
    <X_2>4000</X_2>
    <Y_2>7000</Y_2>
    <X_3>4000</X_3>
    <Y_3>10000</Y_3>
    <X_4>1000</X_4>
    <Y_4>7000</Y_4>
  </Shape_2>
</ImageData>
```

Looking at the generated xml output we can see the calibration points and different shapes. Furthermore, we see that the coordinate system has been scaled by a linear scaling factor. As all points are defined as integers scaling by a linear factor allows to use decimal numbers as coordinates.

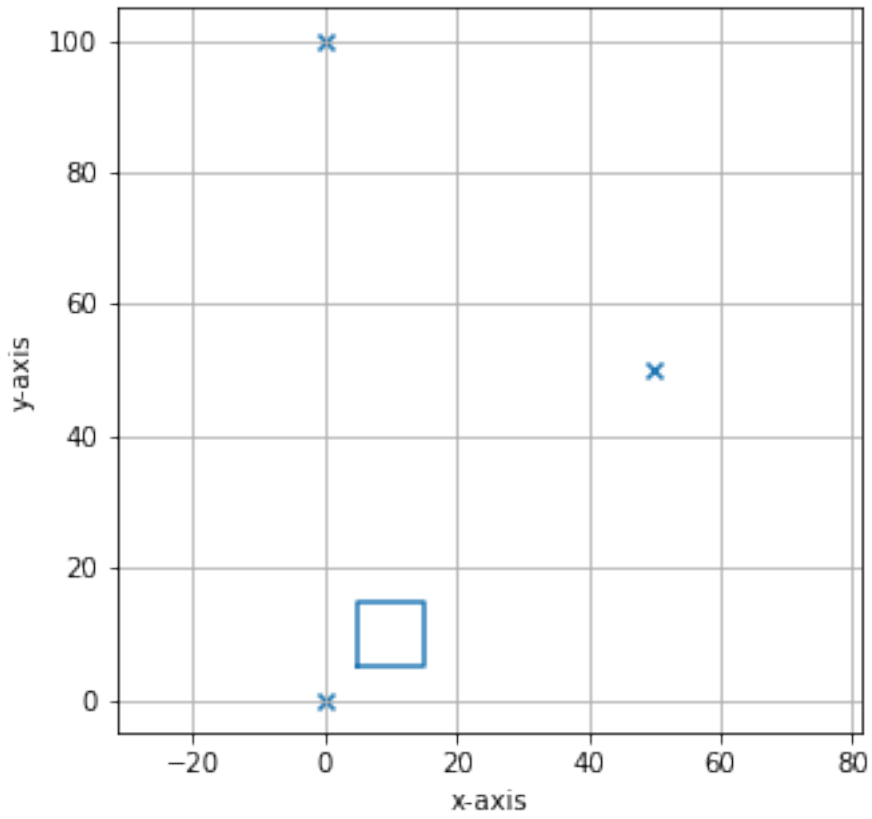
1.3 Using the py-lmd tools

A lot uf usefull functionality is included in the tools module of the py-lmd package. We will first use the rectangle functionality to create rectangle shapes fast.

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [50, 50]])
my_first_collection = Collection(calibration_points = calibration)
```

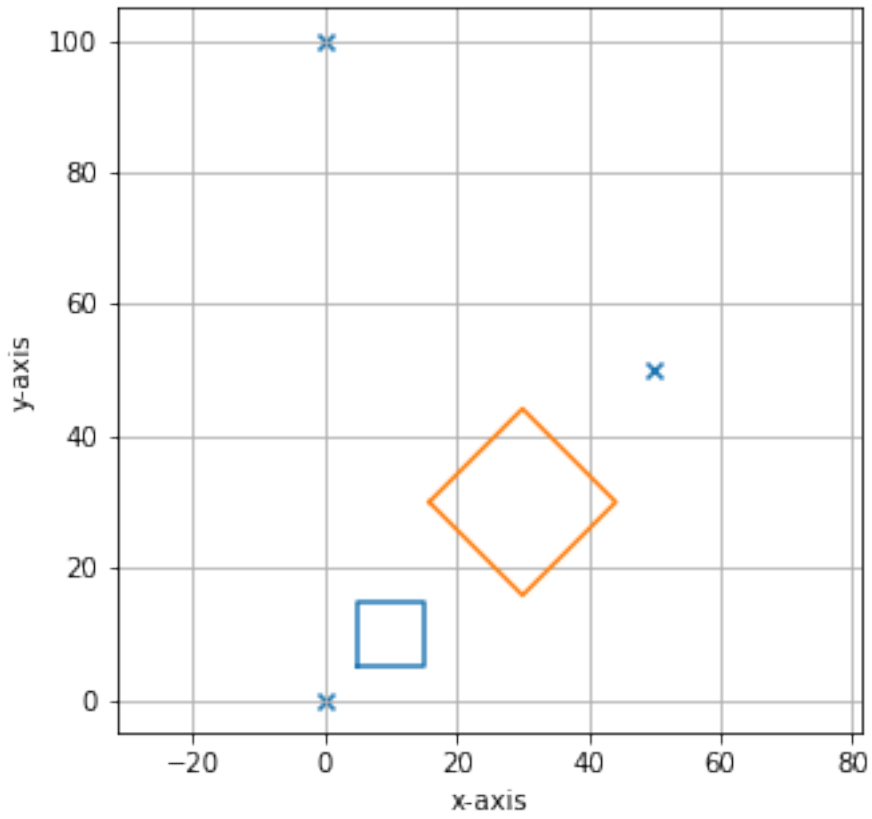
After initiating the coordinate system we can use the `rectangle()` helper function to create a `Shape` object with a rectangle with specified size and position.

```
my_square = tools.rectangle(10, 10, offset=(10,10))
my_first_collection.add_shape(my_square)
my_first_collection.plot(calibration = True)
```



We can further specify an angle of rotation.

```
my_square = tools.rectangle(20, 20, offset=(30,30), rotation = np.pi/4)
my_first_collection.add_shape(my_square)
my_first_collection.plot(calibration = True)
```

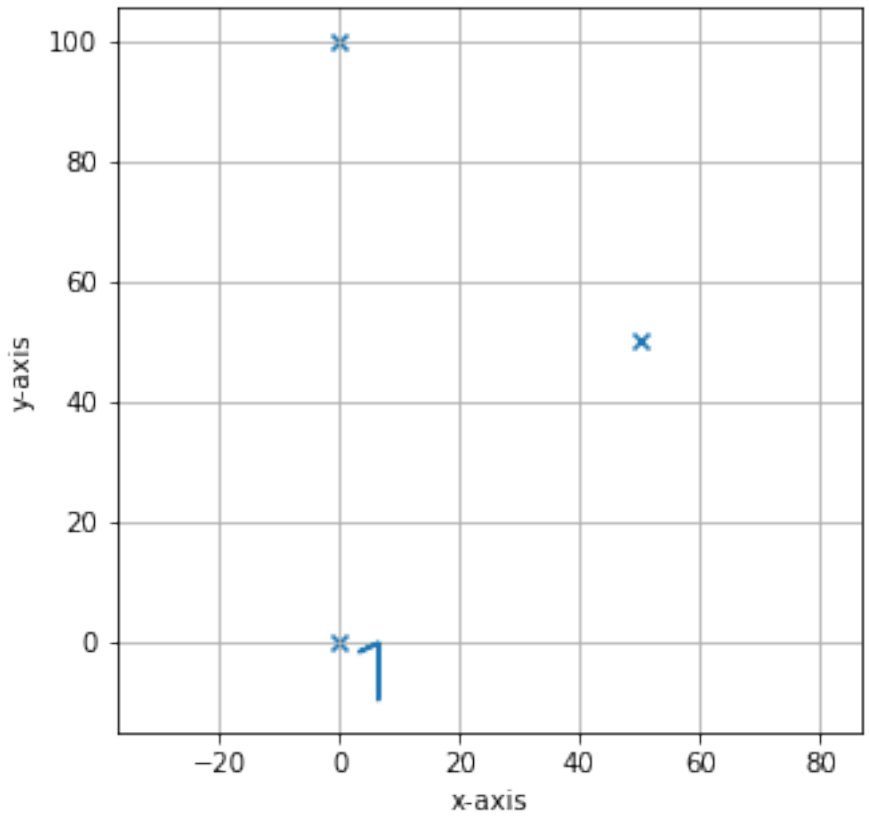


1.4 Numbers and Letters

The py-lmd tools offer a limited support for numbers and some capital letters. The following glyphs are available: *ABCDEFGHI0123456789-_*. They were included in the package as they allow for the development of more consistent calibration and sample indexing.In screens with multiple slides, samples can be unambiguously identified from imaged data.

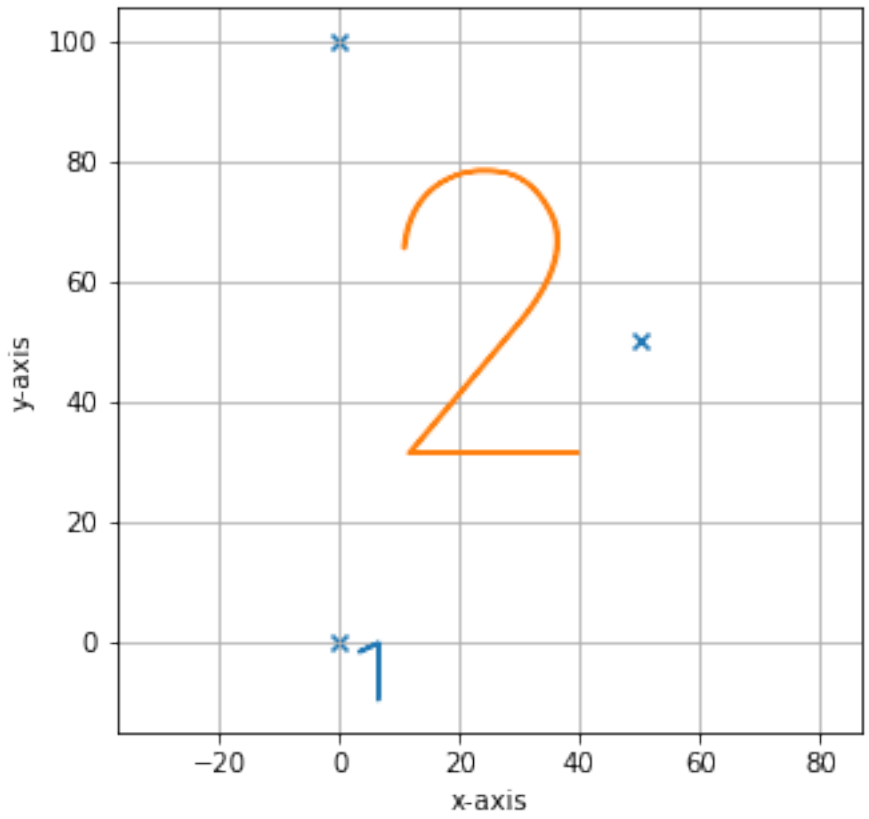
We will first use `glyphs()` to load single glyphs. The glyphs are included in the py-lmd package as SVG files and are loaded by the `svg_to_lmd()` into an uncalibrated `Collection`. This uncalibrated collection is returned and can be joined with a calibrated collection with the `join()` function.

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [50, 50]])
my_first_collection = Collection(calibration_points = calibration)

digit_1 = tools.glyph(1)
my_first_collection.join(digit_1)
my_first_collection.plot(calibration = True)
```



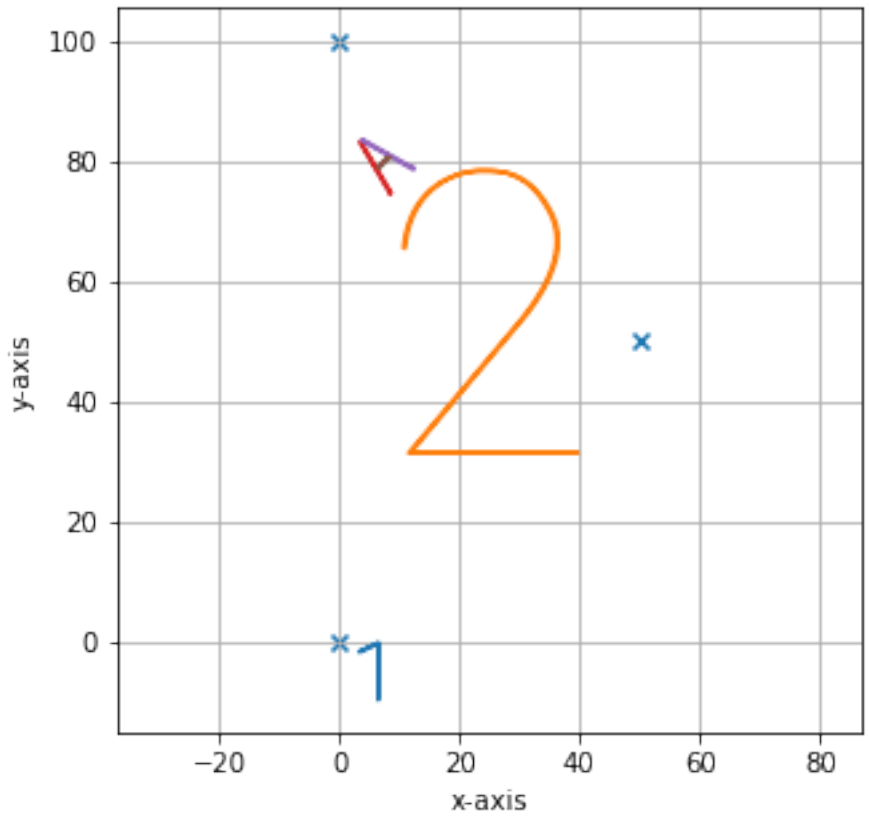
By default glyphs and text have a height of ten units and are located by the top left corner. We can use the *offset* and *multiplier* parameters to change the size and position.

```
digit_2 = tools.glyph(2, offset = (0,80), multiplier = 5)
my_first_collection.join(digit_2)
my_first_collection.plot(calibration = True)
```



Like with the previous rectangle example we can also use the *rotation* parameter to set a clockwise rotation.

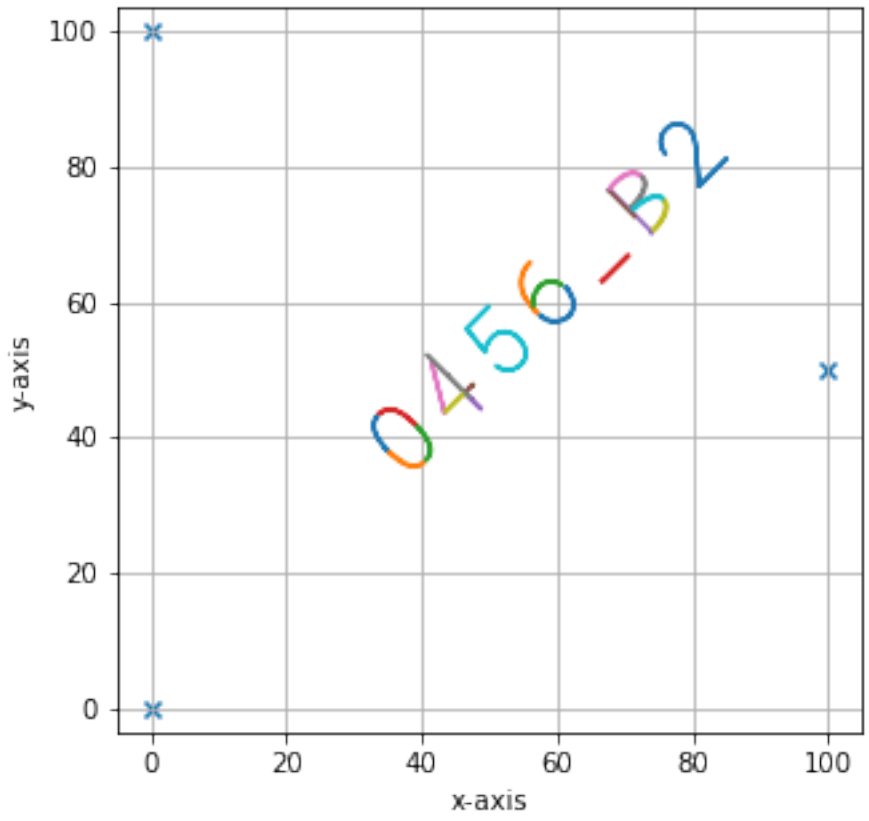
```
glyph_A = tools.glyph('A', offset=(0,80), rotation =-np.pi/4)
my_first_collection.join(glyph_A)
my_first_collection.plot(calibration = True)
```



1.5 Text

Next to individual glyphs the `text()` method can be used to write text with specified position, size and rotation.

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [100, 50]])
my_first_collection = Collection(calibration_points = calibration)

identifier_1 = tools.text('0456_B2', offset=np.array([30, 40]), rotation = -np.pi/4)
my_first_collection.join(identifier_1)
my_first_collection.plot(calibration = True)
```

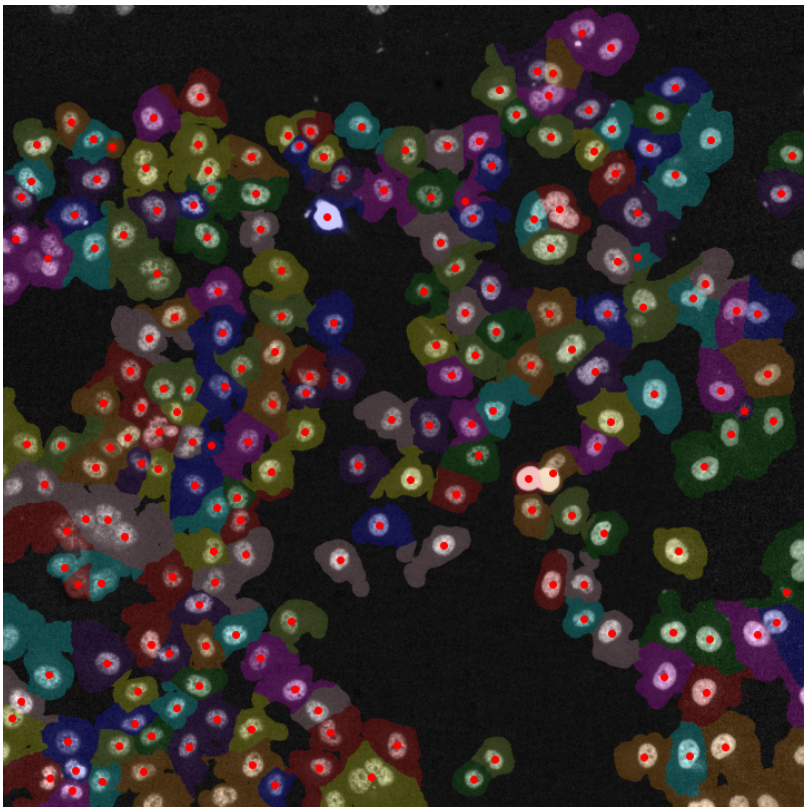

0456-B2
y-axis
x-axis
0
20
40
60
80
100

CHAPTER

TWO

USING SEGMENTED IMAGES

2.1 Background



Although the py-lmd package is meant to serve as framework for creating your own workflows, generating cutting data based on segmentations is the central application of this package. When biological images are segmented, every pixel receives a class or label. Labels can be used to identify single cells and distinguish them from the background or can categorize cells or areas based on phenotypes, functions or location.

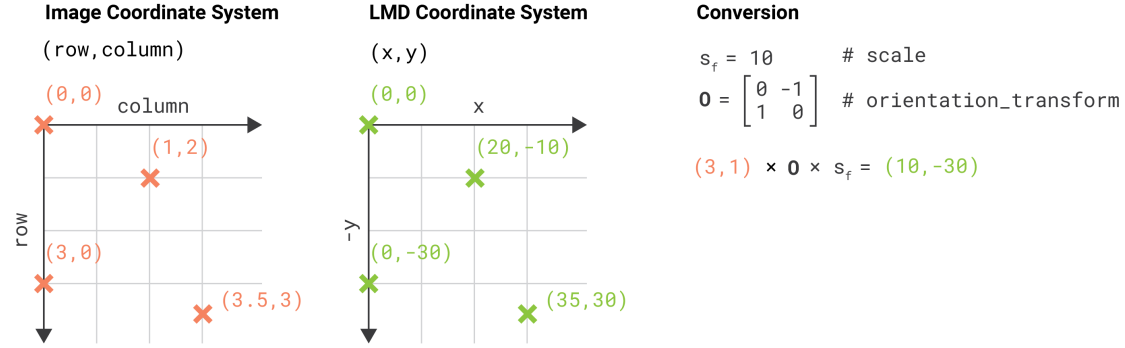
In the following example we will assume that a segmentation was performed to assign labels to individual cells and distinguish their cytosol from the background. The procedures are though applicable to all types of labels.

As this process is so central to the usage of the Leica LMD, the `SegmentationLoader` can be used to create cutting data based on segmentation data. The workflow was specifically designed to work with whole slide images, as large as the LMD membrane slides, and large numbers of single cells. Therefore, different processing steps are included which optimize single cell shapes and decrease overall cutting time.

2.2 Different Coordinate Systems

Using images to generate cutting data with the py-lmd package makes it necessary to transform the image coordinate system to the Leica LMD coordinate system. Although this functionality is part of the package, it is important to highlight the differences in the coordinate systems and to keep in mind what coordinate system is used when calibration points are determined from image data.

The coordinates for the Leica LMD are defined as *(x, y)* coordinates with the x-axis extending to the right and the y-axis extending to the top. All cutting data should exist in this coordinate system and should be calibrated accordingly. When cutting data is generated based on whole slide images we have to keep in mind that images are often indexed differently. Images in Fiji or Numpy are indexed as *(row, column)* with the rows extending downwards and the columns extending to the right. If we want to identify positions in image data - like calibration crosses or single cells - we have to translate their position in the *(row, column)* format to the *(x, y)* format.



The py-lmd library has been designed in a way which allows to transform the coordinate system prior to saving. Therefore one can specify all coordinates in the image coordinate system and rely on the library to handle the transformation. In this case the *orientation_transform* attribute needs to be set when the Collection is created.

```
calibration = np.array([[10, 10], [1000, 10], [500, 500]])

collection = Collection(calibration_points = calibration)
collection.orientation_transform = np.array([[0, -1], [1, 0]])
```

In this case all coordinates for calibration points and shapes can be set in form of *(row, column)* coordinates. The orientation transform is only applied when the Collection is saved or, if desired, when the Collection is plotted.

2.3 Getting started with the SegmentationLoader

Before we can start with the `SegmentationLoader`, we have to load our image which contains the labels for our cells. The segmentation loader expects a numpy array of integers where the background is assigned to label 0. There are no further restrictions to the shape or number of labels other than being continuos. All pixel with a certain label need to be in contact with each other. Functional labels which assign cells based on the cell type must be converted.

As example we will use the cytosol segmentation seen above which can be found under *notebooks/Image_Segmentation/segmentation_cytosol.tiff*. First, we will load this image and convert it to a numpy array.

```
import numpy as np
from PIL import Image

```

(continues on next page)

(continued from previous page)

```
im = Image.open('segmentation_cytosol.tiff')
segmentation = np.array(im).astype(np.uint32)
```

Based on this segmentation we have to select group of cells. These groups can be assigned to separate wells and intersecting shapes and cutting paths will be optimized separately for every group. In our case, all cells will be selected and assigned to the same well A1.

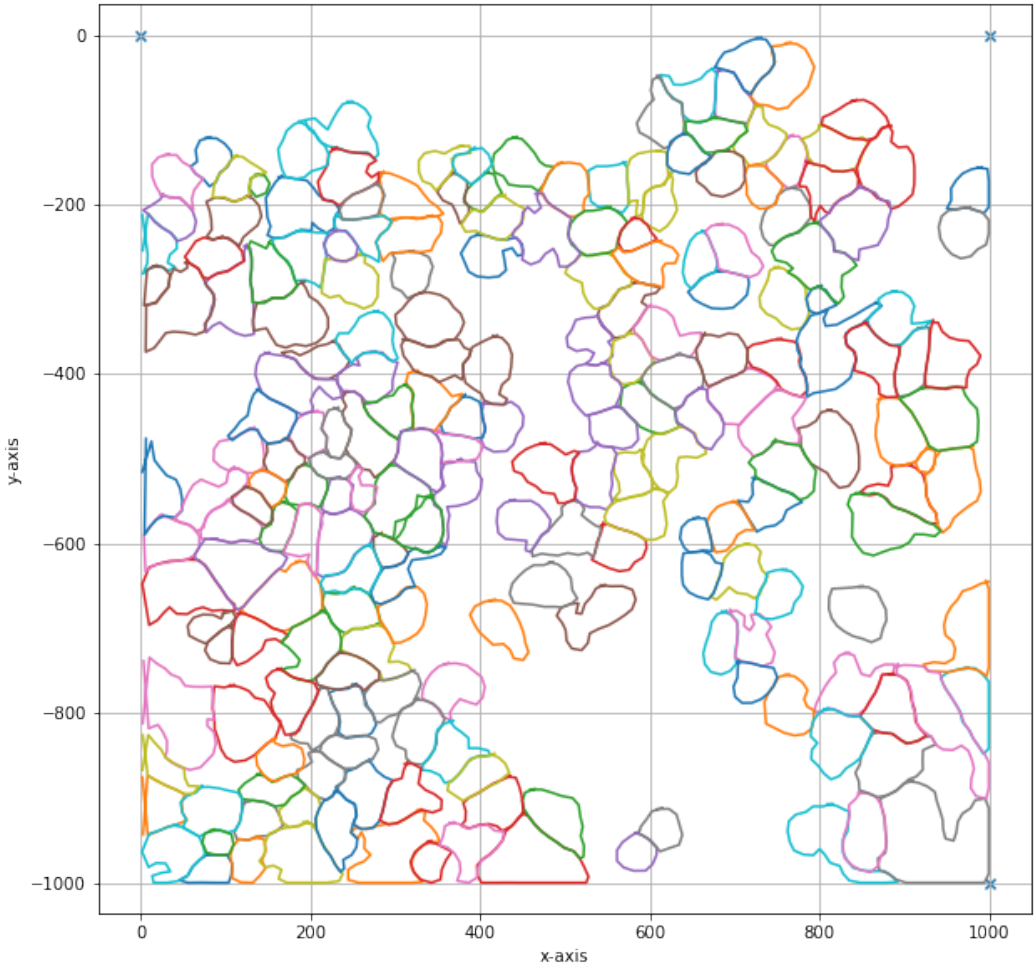
```
all_classes = np.unique(segmentation)
cell_sets = [{"classes": all_classes, "well": "A1"}]
```

Next we need to specify the calibration points which were identified in the image and the coordinate transform which should be applied. By default, the `SegmentationLoader` will read all coordinates as *(row, column)* based on the top left origin. Therefore, the calibration points should be specified in the same way.

```
calibration_points = np.array([[0,0],[0,1000],[1000,1000]])

loader_config = {
    'orientation_transform': np.array([[0, -1],[1, 0]])
}
```

We can now create an instance of the `SegmentationLoader` and generate the cutting data.

```
from lmd.lib import SegmentationLoader
sl = SegmentationLoader(config = loader_config)
shape_collection = sl(segmentation,
                      cell_sets,
                      calibration_points)

shape_collection.plot(fig_size = (10, 10))
```


0
−200
−400
−600
−800
−1000
y-axis
0
200
400
600
800
1000
x-axis

2.4 Overview of Configuration

Table 1: Overview of Configuration Parameters.

Parameter	Default Value	Description
`shape_dilation`	`0`	dilation of the cutting mask in pixel before intersecting shapes in a selection group are merged
`shape_erosion`	`0`	erosion of the cutting mask in pixel before intersecting shapes in a selection group are merged
`binary_smoothing`	`3`	Cutting masks are transformed by binary dilation and erosion
`convolution_smoothing`	`15`	number of datapoints which are averaged for smoothing. The resolution of datapoints is twice as high as the resolution of pixels.
`poly_compression_factor`	`10`	fold reduction of datapoints for compression
`path_optimization`	`"hilbert"`	Optimization of the cutting path inbetween shapes. Optimized paths improve the cutting time and the microscopes focus. valid options are `["none", "hilbert", "greedy"]`
`hilbert_p`	`7`	Paramter required for hilbert curve based path optimization. Defines the order of the hilbert curve used, which needs to be tuned with the total cutting area.
`greedy_k`	`20`	Parameter required for greedy path optimization. Instead of a global distance matrix, the k nearest neighbours are approximated. The optimization problem is then greedily solved for the known set of nearest neighbours until the first set of neighbours is exhausted.Established edges are then removed and the nearest neighbour approximation is recursivly repeated.
`distance_heuristic`	`300`	Overlapping shapes are merged based on a nearest neighbour heuristic. All selected shapes closer than distance_heuristic pixel are checked for overlap.

To be extended…

Here's a grid table followed by a simple table:

Order of processing:

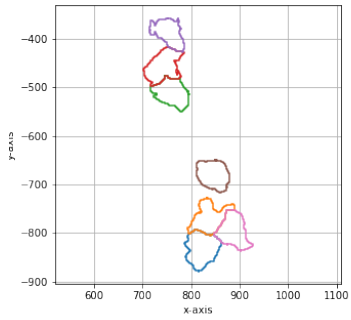
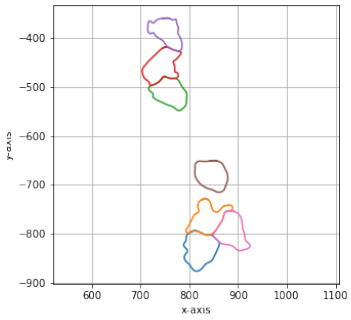
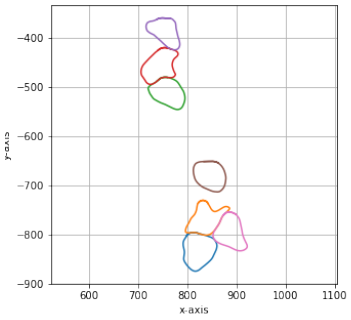
shape_dilation shape_erosion join_intersecting

binary_smoothing binary_fill_holes convolution_smoothing poly_compression

path optimization

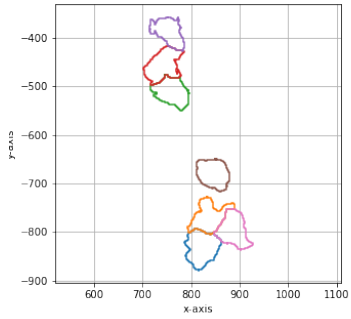
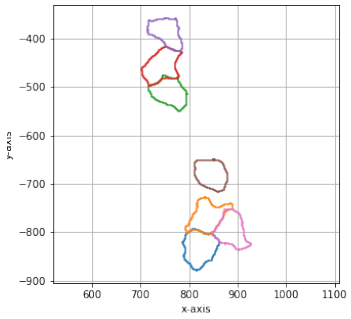
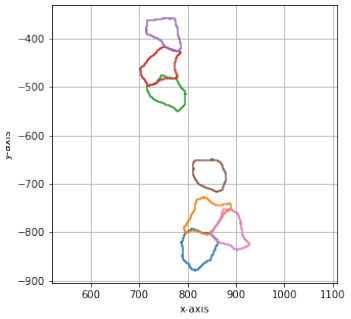
convolution_smoothing

Smooth the polygon by applying a circular, linear convolution of given size. The default convolution kernel with `n` elements is `[1/n, 1/n, ... , 1/n]`. By default a value of 15 is used. Values below 3 are not recommended. In contrast to `binary_smoothing`, `convolution_smoothing` does not increase the convex hull of the shape. When there are many deep recessions in the shape `convolution_smoothing` might not be able to smooth these out and `binary_smoothing` should be used. `convolution_smoothing` does not change the number of vertices in the polygon of a shape. Please see `poly_compression_factor`.

`convolution_smoothing: 1`	`convolution_smoothing: 40`	`convolution_smoothing: 80`
		

binary_smoothing

Smooth the polygon by applying a circular, linear convolution of given size. The default convolution kernel with `n` elements is `[1/n, 1/n, ... , 1/n]`. In contrast to `binary_smoothing`, `convolution_smoothing` does not increase the convex hull of the shape. When there are many deep recessions in the shape `convolution_smoothing` might not be able to smooth these out and `binary_smoothing` should be used. `convolution_smoothing` does not change the number of vertices in the polygon of a shape. Please see `poly_compression_factor`.

`convolution_smoothing: 1`	`convolution_smoothing: 40`	`convolution_smoothing: 80`
		

CHAPTER

THREE

MODULES

3.1 lmd.lib

3.1.1 Collection

class lmd.lib.**Collection**(*calibration_points: Optional[numpy.ndarray] = None*)

Class which is used for creating shape collections for the Leica LMD6 & 7. Contains a coordinate system defined by calibration points and a collection of various shapes.

Parameters **calibration_points** – Calibration coordinates in the form of $(3, 2)$.

shapes

Contains all shapes which are part of the collection.

Type List[Shape]

calibration_points

Calibration coordinates in the form of $(3, 2)$.

Type Optional[np.ndarray]

orientation_transform

defines transformations performed on the provided coordinate system prior to export as XML. This orientation_transform is always applied to shapes when there is no individual orienation_transform provided.

Type np.ndarray

add_shape(*shape: lmd.lib.Shape*)

Add a new shape to the collection.

Parameters **shape** – Shape which should be added.

join(*collection: lmd.lib.Collection*)

Join the collection with the shapes of a different collection. The calibration markers of the current collection are kept. Please keep in mind that coordinate systems and calibration points must be compatible for correct joining of collections.

Parameters **collection** – Collection which should be joined with the current collection object.

Returns returns self

load(*file_location: str*)

Can be used to load a shape file from XML. Both, XMLs generated with py-lmd and the Leica software can be used. :param file_location: File path pointing to the XML file.

new_shape(*points: numpy.ndarray*, *well: Optional[str] = None*, *name: Optional[str] = None*)

Directly create a new Shape in the current collection.

Parameters

- **points** – Array or list of lists in the shape of *(N,2)*. Contains the points of the polygon forming a shape.
- **well** – Well in which to sort the shape after cutting. For example A1, A2 or B3.
- **name** – Name of the shape.

plot(*calibration: bool = True*, *mode: str = 'line'*, *fig_size: tuple = (5, 5)*, *apply_orientation_transform: bool = True*, *apply_scale: bool = False*, *save_name: Optional[str] = None*, ***kwargs*)
This function can be used to plot all shapes of the corresponding shape collection.

Parameters

- **calibration** – Controls wether the calibration points should be plotted as crosshairs. Deactivating the crosshairs will result in the size of the canvas adapting to the shapes. Can be especially usefull for small shapes or debugging.
- **fig_size** – Defaults to $(10, 10)$ Controls the size of the matplotlib figure. See matplotlib documentation for more information.
- **apply_orientation_transform** – Define wether the orientation transform should be applied before plotting.
- **(Optional[str]** (*save_name*) – None): Specify a filename for saving the generated figure. By default *None* is provided which will not save a figure.
- **default** – None): Specify a filename for saving the generated figure. By default *None* is provided which will not save a figure.

save(*file_location: str*, *encoding: str = 'utf-8'*)
Can be used to save the shape collection as XML file.

file_location: File path pointing to the XML file.

stats()
Print statistics about the Collection in the form of:

```
===== Collection Stats =====
Number of shapes: 208
Number of vertices: 126,812
============================
Mean vertices: 609.67
Min vertices: 220.00
5% percentile vertices: 380.20
Median vertices: 594.00
95% percentile vertices: 893.20
Max vertices: 1,300.00
```

svg_to_lmd(*file_location*, *offset=[0, 0]*, *divisor=3*, *multiplier=60*, *rotation_matrix=numpy.eye*, *orientation_transform=None*)
Can be used to save the shape collection as XML file.

Parameters

- **file_location** – File path pointing to the SVG file.
- **orientation_transform** – Will superseed the global transform of the Collection.
- **rotation_matrix** –

3.1.2 Shape

class lmd.lib.**Shape**(*points: numpy.ndarray*, *well: Optional[str] = None*, *name: Optional[str] = None*, *orientation_transform=None*)

Class for creating a single shape object.

from_xml(*root*)

Load a shape from an XML shape node. Used internally for reading LMD generated XML files.

Parameters **root** – XML input node.

to_xml(*id: int*, *orientation_transform: numpy.ndarray*, *scale: int*)

Generate XML shape node needed internally for export.

Parameters

- **id** – Sequential identifier of the shape as used in the LMD XML format.
- **orientation_transform** (*np.array*) – Pass orientation_transform which is used if no local orientation transform is set.
- **scale** (*int*) – Scalling factor used to enable higher decimal precision.

Note: If the Shape has a custom orientation_transform defined, the custom orientation_transform is applied at this point. If not, the oritenation_transform passed by the parent Collection is used. This highlights an important difference between the Shape and Collection class. The Collection will always has an orientation transform defined and will use *np.eye(2)* by default. The Shape object can have a orientation_transform but can also be set to *None* to use the Collection value.

3.1.3 SegmentationLoader

class lmd.lib.**SegmentationLoader**(*config={}*, *verbose=False*)

Select single cells from a segmentation and generate cutting data

Parameters

- **config** (*dict*) – Dict containing configuration parameters. See Note for further explanation.
- **cell_sets** (*list(dict)*) – List of dictionaries containing the sets of cells which should be sorted into a single well.
- **calibration_marker** (*np.array*) – Array of size '(3,2)' containing the calibration marker coordinates in the '(row, column)' format.

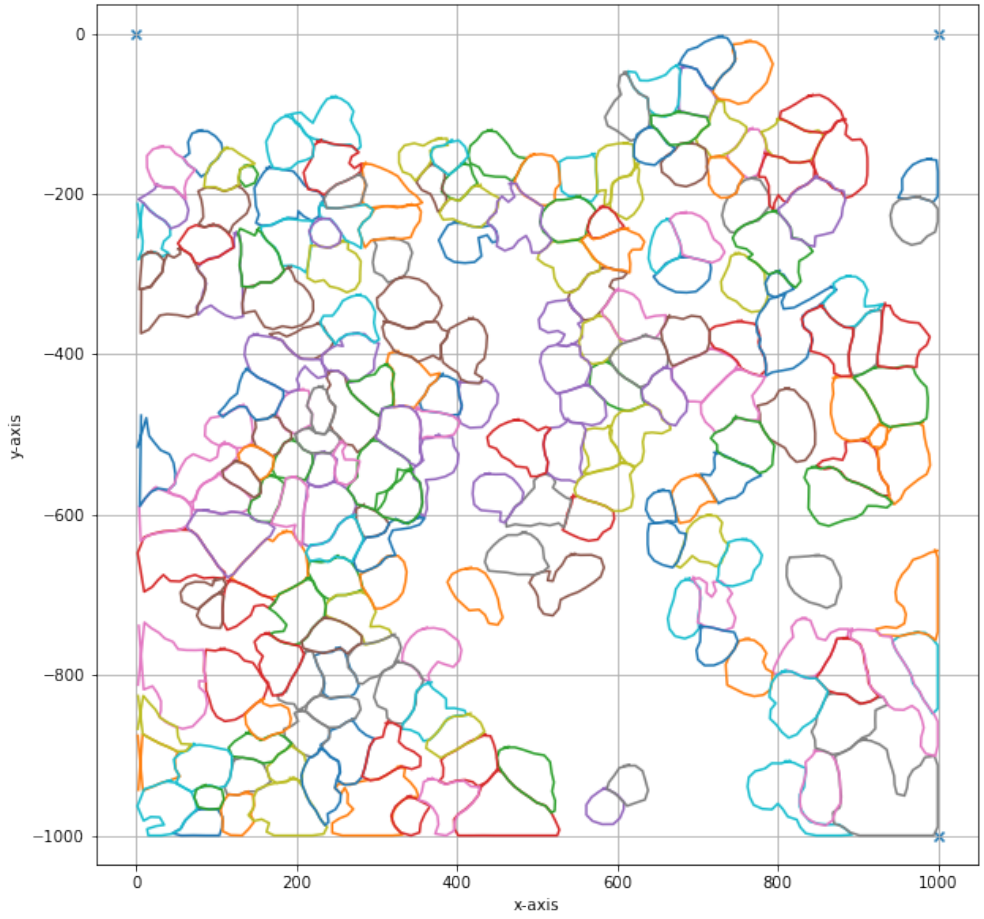
Example

```
import numpy as np
from PIL import Image
from lmd.lib import SegmentationLoader

im = Image.open('segmentation_cytosol.tiff')
segmentation = np.array(im).astype(np.uint32)

all_classes = np.unique(segmentation)

cell_sets = [{"classes": all_classes, "well": "A1"}]

calibration_points = np.array([[0,0],[0,1000],[1000,1000]])
```

(continues on next page)

(continued from previous page)

```
loader_config = {
    'orientation_transform': np.array([[0, -1],[1, 0]])
}

sl = SegmentationLoader(config = loader_config)
shape_collection = sl(segmentation,
                      cell_sets,
                      calibration_points)

shape_collection.plot(fig_size = (10, 10))
```



Note: Basic explanation of the parameters in the config dict:

```
# dilation of the cutting mask in pixel before intersecting shapes in a␣
↪selection group are merged
shape_dilation: 0

# erosion of the cutting mask in pixel before intersecting shapes in a selection␣
↪group are merged
```

(continues on next page)

(continued from previous page)

```
shape_erosion: 0

# Cutting masks are transformed by binary dilation and erosion
binary_smoothing: 3

# number of datapoints which are averaged for smoothing
# the resoltion of datapoints is twice as high as the resolution of pixel
convolution_smoothing: 15

# fold reduction of datapoints for compression
poly_compression_factor: 30

# Optimization of the cutting path inbetween shapes
# optimized paths improve the cutting time and the microscopes focus
# valid options are ["none", "hilbert", "greedy"]
path_optimization: "hilbert"

# Paramter required for hilbert curve based path optimization.
# Defines the order of the hilbert curve used, which needs to be tuned with the␣
↪total cutting area.
# For areas of 1 x 1 mm we recommend at least p = 4,  for whole slides we␣
↪recommend p = 7.
hilbert_p: 7

# Parameter required for greedy path optimization.
# Instead of a global distance matrix, the k nearest neighbours are approximated.
# The optimization problem is then greedily solved for the known set of nearest␣
↪neighbours until the first set of neighbours is exhausted.
# Established edges are then removed and the nearest neighbour approximation is␣
↪recursivly repeated.
greedy_k: 20

# Overlapping shapes are merged based on a nearest neighbour heuristic.
# All selected shapes closer than distance_heuristic pixel are checked for␣
↪overlap.
distance_heuristic: 300
```

---

check_cell_set_sanity(*cell_set*)

Check if cell_set dictionary contains the right keys

load_classes(*cell_set*)

Identify cell class definition and load classes

Identify if cell classes are provided as list of integers or as path pointing to a csv file. Depending on the type of the cell set, the classes are loaded and returned for selection.

3.2 lmd.tools

lmd.tools.**ellipse**(*major_axis*, *minor_axis*, *offset=(0, 0)*, *rotation=0*, *polygon_resolution=1*)
Get a lmd.lib.Shape for ellipse of choosen dimensions.

Parameters

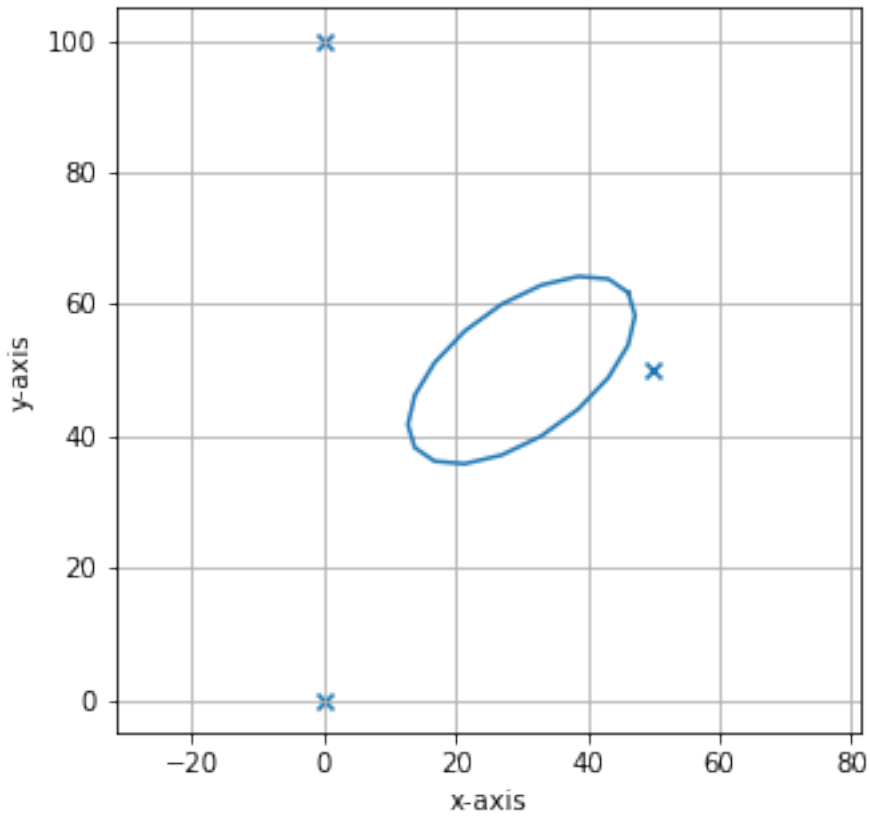
- **major_axis** (*float*) – Major axis of the ellipse. The major axis is defined from the center to the perimeter and therefore half the diameter. The major axis is placed along the x-axis before rotation.
- **minor_axis** (*float*) – Minor axis of the ellipse. The minor axis is defined from the center to the perimeter and therefore half the diameter. The minor axis is placed along the y-axis before rotation.
- **offset** (*np.ndarray*) – Location of the ellipse based on the center given in the form of *(x, y)*. Default value: np.array((0, 0))
- **rotation** (*float*) – Clockwise rotation in radian.
- **polygon_resolution** (*float*) – The polygon resolution defines how far the vertices should be spaced on average. A polygon_resolution of 10 will place a vertex on average every ten units.

Returns Shape which contains the ellipse.

Return type lmd.lib.Shape

Example

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [50, 50]])
my_first_collection = Collection(calibration_points = calibration)

my_ellipse = tools.ellipse(20, 10, offset = (30, 50), polygon_resolution = 5,␣
↪rotation = 1.8*np.pi)
my_first_collection.add_shape(my_ellipse)
my_first_collection.plot(calibration = True)
```



lmd.tools.**get_rotation_matrix**(*angle: float*)
Returns a rotation matrix for clockwise rotation.

Parameters **angle** (*float*) – Rotation in radian.

Returns Matrix in the shape of (2, 2).

Return type np.ndarray

lmd.tools.**glyph**(*glyph*, *offset=numpy.array*, *rotation=0*, *divisor=10*, *multiplier=1*, ***kwargs*)
Get an uncalibrated lmd.lib.Collection for a glyph of interest.

Parameters

- **glyph** (*str*) – Single glyph as string.
- **divisor** (*int*) – Parameter which determines the resolution when creating a polygon from a SVG. A larger divisor will lead to fewer datapoints for the glyph. Default value: 10
- **offset** (*np.ndarray*) – Location of the glyph based on the top left corner. Default value: np.array((0, 0))
- **multiplier** (*float*) – Scaling parameter for defining the size of the glyph. The default height of a glyph is 10 units. Default value: 1

Returns Uncalibrated Collection which contains the Shapes for the glyph.

Return type lmd.lib.Collection

lmd.tools.**glyph_path**(*glyph*)
Returns the path for a glyph of interest. Raises a NotImplementedError if an unknown glyph is requested.

Parameters **glyph** (*str*) – Single glyph as string.

Returns Path for the glyph.

Return type str

lmd.tools.**makeCross**(*center*, *arms*, *width*, *dist*)
Generate lmd.lib.Shapes to represent a crosshair and add them to an exisiting lmd.lib.Collection.

Parameters

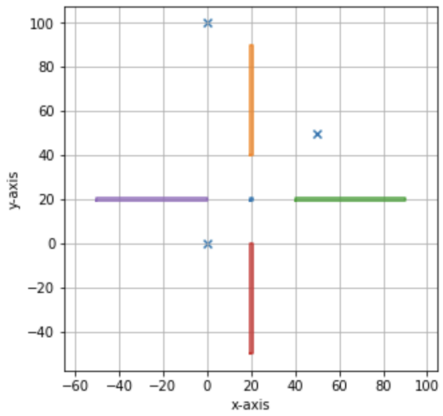
- **center** (*numpy.array*) – center of the new crosshair
- **arms** (*numpy.array*) – length of the individual arms [top, right, bottom, left]
- **width** (*float*) – width of each individual element of the crosshair
- **dist** (*float*) – distance between the center of the cross hair and each arm

Returns Uncalibrated Collection which contains the Shapes for the calibration cross.

Return type lmd.lib.Collection

Example

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [50, 50]])
my_first_collection = Collection(calibration_points = calibration)

cross_1 = tools.makeCross([20, 20], [50,50,50,50], 1, 10)
my_first_collection.join(cross_1)
my_first_collection.plot(calibration = True)
```



lmd.tools.**rectangle**(*width*, *height*, *offset=(0, 0)*, *rotation=0*, *rotation_offset=(0, 0)*)
Get a lmd.lib.Shape for rectangle of choosen dimensions.

Parameters

- **width** (*float*) – Width of the rectangle.
- **offset** (*np.ndarray*) – Location of the rectangle based on the center. Default value: np.array((0, 0))
- **rotation** (*float*) – Rotation in radian.
- **rotation_offset** (*np.ndarray*) – Location of the center of rotation relative to the center of the rectangle. Default value: np.array((0, 0))

Returns Shape which contains the rectangle.

Return type lmd.lib.Shape

Example:

lmd.tools.**text**(*text*, *offset=numpy.array*, *divisor=1*, *multiplier=1*, *rotation=0*, ***kwargs*)
Get an uncalibrated lmd.lib.Collection for a text.

Parameters

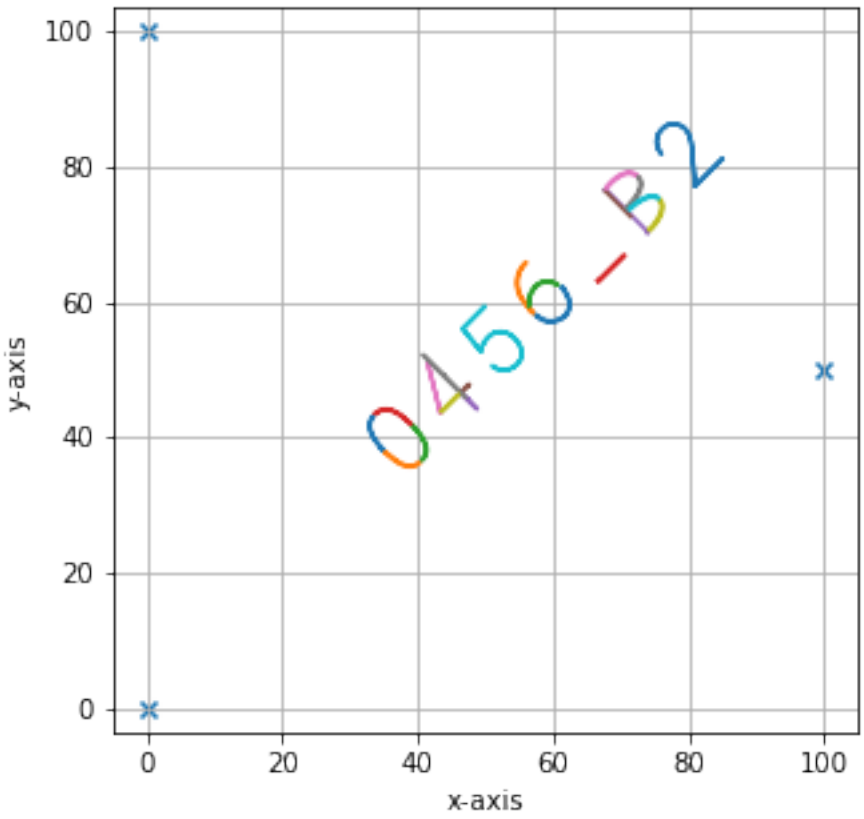
- **text** (*str*) – Text as string.
- **divisor** (*int*) – Parameter which determines the resolution when creating a polygon from a SVG. A larger divisor will lead to fewer datapoints for the glyph. Default value: 10
- **offset** (*np.ndarray*) – Location of the text based on the top left corner. Default value: np.array((0, 0))
- **multiplier** (*float*) – Scaling parameter for defining the size of the text. The default height of a glyph is 10 units. Default value: 1

Returns Uncalibrated Collection which contains the Shapes for the text.

Return type lmd.lib.Collection

Example

```
import numpy as np
from lmd.lib import Collection, Shape
from lmd import tools

calibration = np.array([[0, 0], [0, 100], [100, 50]])
my_first_collection = Collection(calibration_points = calibration)

identifier_1 = tools.text('0456_B2', offset=np.array([30, 40]), rotation = -np.
↪pi/4)
my_first_collection.join(identifier_1)
my_first_collection.plot(calibration = True)
```


100
80
60
40
20
0
y-axis
0
20
40
60
80
100
x-axis
0456-B2

INDEX