CrackDect

Release 0.2

Matthias Drvoderic

Sep 14, 2022

EXAMPLES

This package provides crack detection algorithms for tunneling off axis cracks in glass fiber reinforced materials.

Full paper: https://www.sciencedirect.com/science/article/pii/S2352711021001205
GitHub Repo: https://github.com/mattdrvo/CrackDect

If you use this package in publications, please cite the paper.

In this package, crack detection algorithms based on the works of Glud et al.[1] and Bender et al.[2] are implemented. This implementation is aimed to provide a modular "batteries included" package for this crack detection algorithms as well as a framework to preprocess image series to suite the prerequisites of the different crack detection algorithms.

[1] J.A. Glud, J.M. Dulieu-Barton, O.T. Thomsen, L.C.T. Overgaard Automated counting of off-axis tunnelling cracks using digital image processing Compos. Sci. Technol., 125 (2016), pp. 80-89

[2] Bender JJ, Bak BLV, Jensen SM, Lindgaard E. Effect of variable amplitude block loading on intralaminar crack initiation and propagation in multidirectional GFRP laminate Composites Part B: Engineering. 2021 Jul

CHAPTER

ONE

QUICK START

To install CrackDect, check at first the *Prerequisites* of your python installation. Upon meeting all the criteria, the package can be installed with pip, or you can clone or download the repo. If the installed python version or certain necessary packages are not compatible we recommend the use of virtual environments by virtualenv or Conda. See the conda guide for infos about creating and managing Conda environments.

Installation:

Open a command line and check if python is available

```
$ python --version
```

This displays the version of the global python environment. If this does not return the python version, something is not working and you need to fix your global python environment.

If all the prerequisites are met CrackDect can be installed in the global environment via pip

```
$ pip install crackdect
```

Quick Start shows an illustrative example of the crack detection.

Crack Detection provides a quick theoretical introduction into the crack detection algorithm.

CHAPTER

TWO

PREREQUISITES

It is recommended to use virtual environments (anaconda). This package is written and tested in Python 3.8 and relies on here listed packages.

scikit-image 0.18.1
numpy 1.18.5
scipy 1.6.0
matplotlib 3.3.4
sqlalchemy 1.3.23
numba 0.52.0
psutil 5.8.0

And if the visualization module is used PyQt5 is also needed.

CHAPTER

THREE

MOTIVATION

Most algorithms and methods for scientific research are implemented as in-house code and not accessible for other researchers. Code rarely gets published and implementation details are often not included in papers presenting the results of these algorithms. Our motivation is to provide transparent and modular code with high level functions for crack detection in composite materials and the framework to efficiently apply it to experimental evaluations.

3.1 Quick Start

Lets start with initialising an *ImageStack*. Its a container object for a stack of images. The crack detection works with this container objects to process the whole image stack at once. It is also possible to work just with a list of images if no extra functionality from *ImageStack* is needed.

Create an stack and add images to it. The images in this example can be downloaded `here`. Unpack the folder and set the working directory in the parent folder of *example_images*.

```
import numpy as np
import crackdect as cd
# read image paths from folder
paths = cd.image_paths('example_images')

# We want the dtype of the images to be np.float32 and only grayscale images.
stack = cd.ImageStack.from_paths(paths,  dtype=np.float32, as_gray=True)
```

The following image shows the last image from the stack. A lot of small and some bigger cracks are visible. Also, the region of interest is only the middle part of the specimen without the edges of the specimen and the painted black bar.

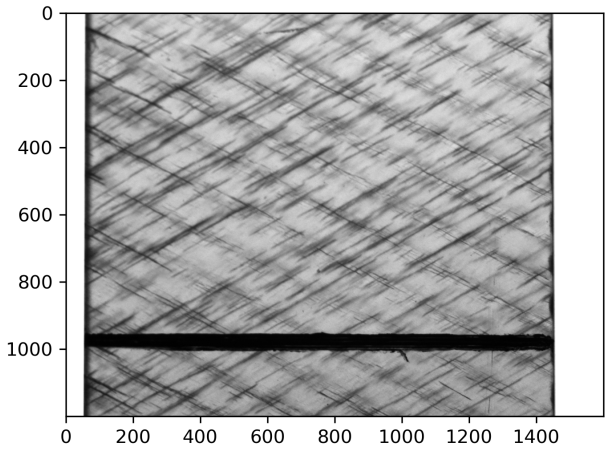
Before cutting to the desired shape a shift correction is necessary to align all images in a global coordinate system. The following image shows the last image of the stack after the shift correction and the cut to the region of interest.

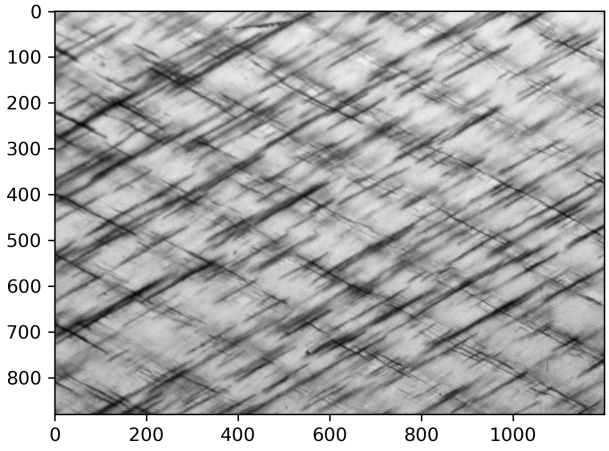
```
# shift correction to align all images in one coordinate system
cd.shift_correction(stack)

# The region of interest is form pixel 200-1400 in x and 20-900 in y
cd.region_of_interest(stack, 200, 1400, 20, 900)
```

Currently, three functions using different algorithms for crack detection are available in the package. For this tutorial, *detect_cracks()*, the simplest crack detection function with the least prerequisites in image preprocessing is used. For more information go to *Crack Detection*.

Only cracks in a set direction are detected with *detect_cracks()*. For the algorithm to work properly, the following arguments must be set.

1. **theta:** The angle between the cracks and a vertical line.


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

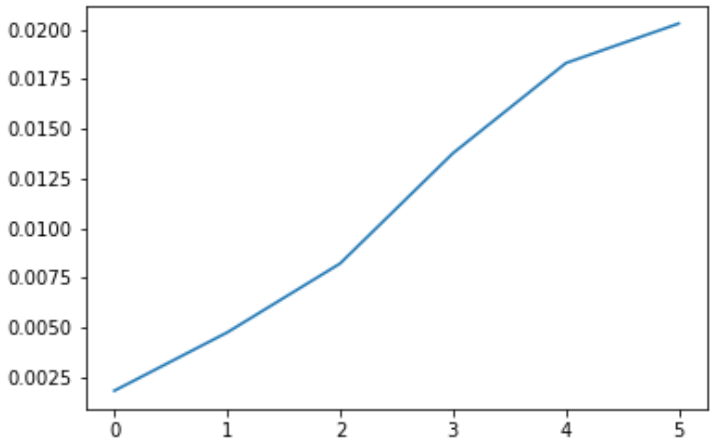

0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000

2. **crack_width:** Approximate width of the major detected cracks in pixels. This value is taken as the wavelength of the Gabor kernel.
3. **ar:** The aspect ratio of the kernel. Since cracks are usually long and thin an aspect ratio bigger than 1 should be chosen. A good compromise between speed and accuracy is 2. Too big aspect ratios can lead to false detection.
4. **min_size:** The minimum length of detected cracks in pixels. Since small artifacts or noise can lead to false detection, this parameter provides an reliable filter.

```
# crack detection
rho, cracks, thd = cd.detect_cracks(stack, theta=60, crack_width=10, ar=2, bandwidth=1,␣
↪min_size=10)
```

The results can be plotted and inspected.

```
# plot the crack density
import matplotlib.pyplot as plt
plt.plot(np.arange(len(stack)), rho)
```



The crack density is growing with each image. To look if all cracks are detected lets look at the last image in the stack.

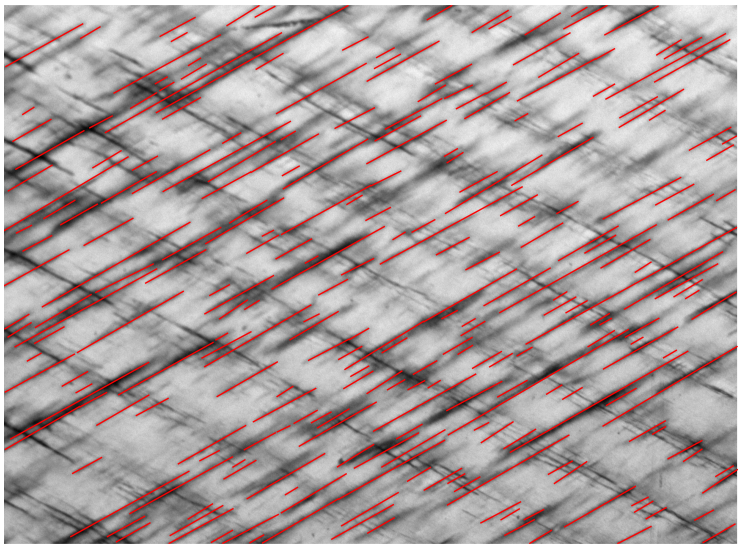
```
# plot the background image and the associated cracks
cd.plot_cracks(stack[-1], cracks[-1])
```

Nearly all cracks get detected. Some cracks are too close to each other and the crack detection can not distinguish them. Cracks in other directions are not detected. This image has low contrast so it is hard to detect all the cracks since some are quite faint compared to the background. There is also quite a lot of blur at some cracks. This are the main problems with the crack detection. This image would benefit form an histogram equalization to boost the contrast.

The full script:

```
import numpy as np
import crackdect as cd
# read image paths from folder
paths = cd.image_paths('example_images')

# We want the dtype of the images to be np.float32 and only grayscale images.
```

(continues on next page)



(continued from previous page)

```
stack = cd.ImageStack.from_paths(paths,  dtype=np.float32, as_gray=True)
# shift correction to align all images in one coordinate system
cd.shift_correction(stack)
# The region of interest is form pixel 200-1400 in x and 20-900 in y
cd.region_of_interest(stack, 200, 1400, 20, 900)
# crack detection
rho, cracks, thd = cd.detect_cracks(stack, theta=60, crack_width=10, ar=2, bandwidth=1,␣
↪min_size=10)
# plot the background image and the associated cracks
cd.plot_cracks(stack[-1], cracks[-1])
```

3.2 The Image Stack

Since this package is build for processing multiple images the efficient handling of image collections is important. The whole functionality of the package as also available for working with single images but the API of most top level functions is built for image stacks.

The image stack is the core of this package. It works as a container for collections of images. It can hold images of any size and color/grayscale images can be mixed. The only restriction is that all images are from the same data type e.g *np.float32*, *np.unit8*, etc. The data type of incoming images is checked automatically and if the types do not match the incoming image is converted.

All image stack classes have the same io structure. Images can be added, removed and altered. Single images are accessed with indices and groups of images via slicing. Currently these image stacks are available:

- *ImageStack*: The most basic image stack. It just manages the dtype checks and conversion of incoming images. All images are held in memory (RAM) at all times. When working with just a few images this is the best choice since it adds nearly zero overhead.
- *ImageStackSQL*: When working with a large number of images available RAM can become a problem. This container manages the used RAM of the stack. When exceeding set limits it automatically saves the current state of the images to an SQL database. It is built with sqlalchemy for maximum flexibility. It can als be used to save

the current state of an image stack for later usage or transferring the data to other locations. Image stacks can be constructed directly from the created databases. The creation of the database is automated and does not need user input. One database can hold multiple stacks so more than one image stack can interact with one database at once. Sqlalchemy handles all transactions with the database.

This is a quick introduction on how the image stack works. The full API documentation is *here*.

Now a few examples are given on how to use an image stack. Image stacks can be constructed from directly or via convenience methods to automatically load all images from a list of paths.

3.2.1 Basic functionality

This is an example of the basic functionality all image stacks must have to work with the preprocessing funcitons and the crack detection.

Directly construct an image stack. The dtype of the images in the stack should be set. The default is *np.float32* since all functions and the crack detection are optimised for handling float images.

```
import crackdect as cd
stack = cd.ImageStack(dtype=np.float32)
```

Adding images to the stack directly. Numpy arrays and pillow image objects can be added. PIL images will be converted to numpy arrays.

```
stack.add_image(img)
```

To access images from the stack use indices or slices if multiple images should be accessed. The return when slicing will be a new image stack.

```
stack[0]  # => first image = numpy array
stack[1:4]  # => image stack of the images with index 1-4(not included).
stack[-1]  # => last image of the stack.
```

Overriding images in a stack works also like for objects in normal lists.

```
stack[1] = np.random.rand(200,200) * np.linspace(0,1,200)
```

This overrides the 2nd image in the stack. If the dtype does not fit the image is converted. Multiple images can be overridden at once

```
stack[1:5] = ['list of 4 images']
```

But unlike lists 4 images must be given to replace 4 images in the stack. There is no thing as sub-stacks. Removing images also works like for lists.

```
del stack[4]  # removes the 5th image of the stack
del stack[-3:]  # removes the last 3 images.
stack.remove_image(4)  # the same as del stack[4] but no slicing possible
stack.remove_image()  # removes per default the last image
```

3.2.2 Advanced Features

ImageStackSQL has more functionality to it. It can be created like the normal *ImageStack* but it is it is recommended to set the name of the database and the name of the table the images will be stored in. With this it is easy to identify saved results. If no names are set, the object id is taken. The database is created in the current working directory.

```
stack = cd.ImageStackSQL()  # completely default creation
stack = cd.ImageStackSQL(database='test', stack_name='test_stack1')
```

Multiple stacks can be connected with one database

```
stack2 = cd.ImageStackSQL(database='test', stack_name='test_stack2')
stack3 = cd.ImageStackSQL(database='test', stack_name='test_stack3')
```

Saving and loading is done automatically but only when needed. So it is possible that the stack was altered but the current state is not saved jet. To save the current state call

```
stack.save_state()
```

This will save all changes and free the RAM the images used. When images are accessed after this, they are loaded form the databased again.

All stacks can be copied.

```
new_stack = stack.copy()  # works for all stacks
```

Stacks with sql connection should be named

```
new_sql_stack = sql_stack.copy(stack_name='test_stack4')
```

Copying a normal stack will not use more ram until the images in the new stack are overridden. Copying a stack with sql-connection will create a new table in the database and copy all images to the new table. For big image stacks, this is a costly operation since all images will be loaded at some point, copied to the other table and saved there. It the image stack exceeds its set RAM limits multiple rounds of loading parts of the stack and saving them in the new table may be required.

3.2.3 Convenience Creation

To avoid manually loading all images and putting them into an image stack there are several options to automatically create an image stack. Images are loaded with skimage.io.imread so a huge flexibility is provided to control the loading process which can be controlled with kwargs.

```
# create from a list of image paths
stack = cd.ImageStack.from_paths(['list of paths'])
# create image stack with database connection. Database and stack_name are optional
stack = cd.ImageStackSQL.from_paths(['list of paths'], 'database', 'stack_name')
# create from previously saved database.
stack = cd.ImageStackSQL.load_from_database('database', 'stack_name')
```

The simplest form of creating a basic *ImageStack* is

```
stack = cd.load_images(['list of paths'])
```

For more information and more control over the behaviour of the full documentation for *imagestacks*.

3.3 Preprocessing

The preprocessing for the images is a modular process. Since each user might capture the images in a slightly different way it´s impossible to just set up one preprocessing routine and expect it to work for all circumstances. Therefore, the preprocessing is modular. The preprocessing routines included in this package are defined in *stack_operations*. But these are just some predefined functions for the most important preprocessing steps. Here an example of how to use custom image processing functions with the image stack (*imagestack*) is shown.

3.3.1 Apply functions

An arbitrary function that takes one image and other arguments can be applied to the whole image stack. The function must return an image and nothing else. Applying such an function to the whole image stack will alter all the images in the stack since the images from the stack are taken as input and are replaced with the output of the function. E.g histogram equalisation for the whole image stack can be done in one line of code.

```
import crackdect as cd
from skimage import exposure

stack.execute_function(exposure.equalize_adapthist, clip_limit=0.03)
```

This performs *equalize_adapthist* on all images in the stack with a clip limit of 0.03. *clip_limit* is a keyword argument of *equalize_adapthist*.

With this functionality custom functions can be defined easily without worrying about the image stack. A cascade of different preprocessing functions can be performed on one image stack. This enables a really modular approach and the most flexibility.

```
def contrast_stretching(img):
    p5, p95 = np.percentile(img, (5, 95))
    return exposure.rescale_intensity(img, in_range=(p5, p95))

stack.execute_function(custom_stretching)
```

3.3.2 Rolling Operations

Another way to preprocess the images in stacks is to perform an rolling operation on them. A function for rolling operations takes two images as input and returns just one image. Change detection with image differencing is an example.

$I_d = I_2 - I_1$

Applying this function to the image stack would look like this:

```
def simple_differencing(img1, img2):
    return img2-img1

stack.execute_rolling_function(simple_differencing, keep_first=False)
```

This will evaluate *simple_differencing* for all images starting from the second image in the stack. The n-th image in the stack is computed with this schema.

$I^n_{new} = f(I^{n-1}, I^n)$

Since this schema can only start at the second image, the argument *keep_first* defines is the first image is deleted after the rolling operation or not. The first image will not be changed since the function is not applied on it.

3.3.3 Predefined Preprocessing Functions

The most important preprocessing for the crack detection is the change detection and shift correction. This package comes with functions for these routines. There are variations for both routines and other useful functions like cutting to the region of interest in *stack_operations*.

All the functions in *stack_operations* take an image stack and return the stack with the results of the routine. The images in the stack get changed. If the state of the image stack prior to applying a routine should be kept, copy the stack before.

For more information see the documentation from *stack_operations*.

3.4 Shift Correction

In image series taken during tensile mechanical tests, it often appears that the specimen is moving relative to the background like the specimen in this gif.

Since some crack detection algorithms work only with image series with no shift of the specimen all must be aligned in a global coordinate system for them. Otherwise, these algorithms will compute wrong results. Currently, the following functions need an aligned image stack:

1. *detect_cracks_glud()*
2. *detect_cracks_bender()*

Warning: All shift correction algorithms **only** take images with the **same** dimensionality as input. If the dimensionality of just one image differs it will result in an error. When using an *ImageStack* make sure to ether only add images with the same dimensionality or when using *from_paths()* to construct the stack, add kwargs to ensure all images are loaded the same way (parameters for reading)!

3.4.1 Global Shift Correction

There are two methods to correct the global shift of an image and one to correct the shift as well as distortion. The later can be used to correct images where the specimen show significant strain.

1. *biggest_common_sector()*
2. *shift_correction()*

Both correct the shift of the image with global phase-cross-correlation and cut the image to the biggest common sector that no black borders appear. *biggest_common_sector()* is more efficient but less accurate.

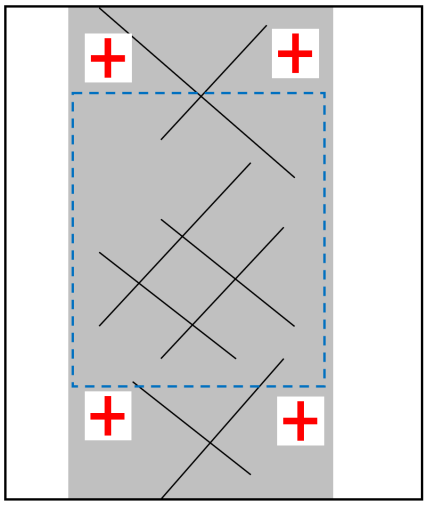
The following gif shows the corrected image series from above.

With this corrected image series, crack detection methods that incorporate the history of the image are applicable.

3.4.2 Shift-Distortion Correction

If the distortion due to strain of the specimen is significant, *shift_distortion_correction()* can be used. This function tracks for subareas of the images to compute global shift, rotation and relative movement between this subareas. The only prerequisite is that for distinct features are visible throughout the whole image stack.

A best practice example would be to mark specimen at four positions to enable reliable shift-distortion correction.



The red crosses are used to correct the shift and distortion of the image while the blue dotted rectangle would mark the usable region of interest for the crack detection.

Note: If shift and distortion are high it can be necessary to apply the correction twice in a row.

Always check if the shift correction worked properly since its reliability depends on the quality of the images!

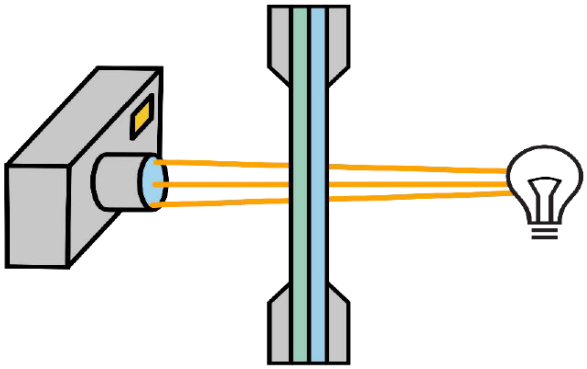
3.5 Crack Detection

imagestack and *stack_operations* are not necessary restrained for the usage of crack detection. *imagestack* provides the framework to conveniently process images stacks. The module *crack_detection* provides the algorithms for crack detection. In *CrackDect* the basic crack detection algorithms are implemented without image preprocessing since this step depends on the image quality, imaging techinque etc. Currently, three algorithms are implemented for the detection of **multiple straight cracks in a given direction** in **grayscale** images.

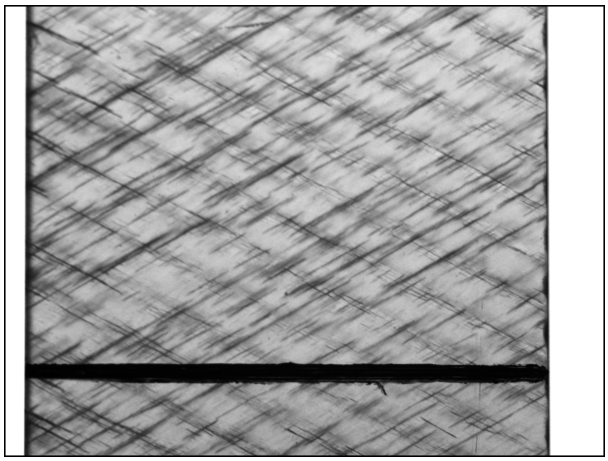
1. *detect_cracks()*
2. *detect_cracks_glud()*
3. *detect_cracks_bender()*

3.5.1 Theory

This package features algorithms for the detection of multiple straight cracks in a given direction. The algorithms were developed for computing the crack density semi-transparent composites where transilluminated white light imaging (TWLI) can be used as imaging technique.



This technique results in a bright image of the specimen with dark crack like this.



CrackDect works only on images similar to the one above, where cracks appear dark on a bright background. The cracks must be straight, since the currently implemented algorithms work with masks and filters to extract cracks in a specific direction. Therefore, it is possible to only detect one kind of cracks and ignore cracks in other directions.

Glud´s algorithm

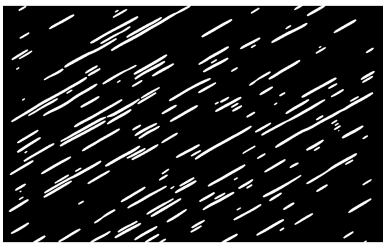
The following functions work with the basis of this algorithm:

1. `detect_cracks()`
2. `detect_cracks_glud()`

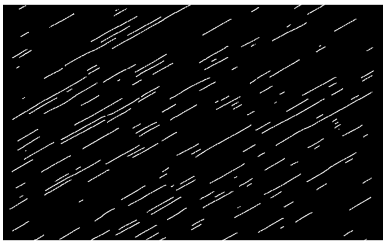
While `detect_cracks()` is a pure implementation of the filter and processing steps, `detect_cracks_glud()` incorporates cracks detected in the n-1st image to the nth image. Therefore all images must be related and without shift (see *Shift Correction*). `detect_cracks_glud()` is basically the full crack detection (without change detection) described by Glud et al. whereas `detect_cracks()` is only the "crack counting algorithm". `detect_cracks()` is more versatile and detects cracks for each image in the given stack without influence of other images. If the position of a crack changes in the image stack, use `detect_cracks()`.

This method from Glud et al. is designed to detect off axis tunneling cracks in composite materials. It works on applying the following filters on the images:

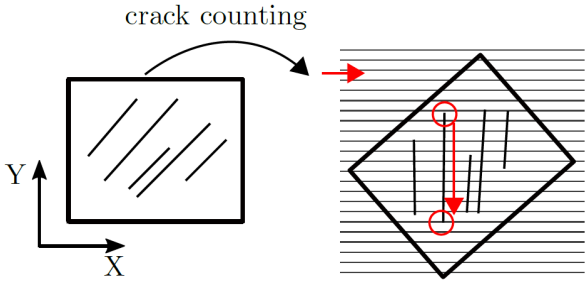
1. **Gabor Filter:** The Gabor filter is applied which detects lines in a set direction. Cracks are only detected in the given direction. This allows to separate crack densities from different layers of the laminate.
2. **Threshold:** A threshold is applied on the result of the Gabor filter. This separates foreground and background in an image. The default is Yen´s threshold. In the case of the crack detection it separates cracked and intact area. The Result of the threshold for the image is shown in the next image.



3. **Skeletonizing:** Since off axis tunneling cracks are aligned with the fibers they are straight. The white bands from the threshold are thinned to a width of one pixel. The algorithm which determines the start and end of each crack relies on only one pixel wide lines. The result of this skeletonizing for a part af the threshold image from above is shown in the next image. This The lines in this image are not continuous. The skeletonizing is done in a rotated coordinate system. This image is rotated back which creates this effect.



4. **Crack Counting:** The cracks are counted in the skeletonized image. The skeletonized image is rotated into a coordinate system where all cracks are vertical (y-direction). Then a loop scans each pixel in each line of pixels in the image. If a crack is found, it follows it down the ydirection until the end of the crack. The coordinates of the beginning and end are saved. After one crack has been detected, it is removed from the image to avoid double detection when the loop runs over the next line of pixels. The following image shows this process.



5. **Crack Density:** The crack density is computed from the detected cracks with

   $\rho_c = \frac{\sum_{i=1}^{n} L_i}{AREA}$

   with L_i as the length of the i-th crack and $AREA$ as the area of the image.

6. **Threshold Density:** The threshold density is the area which is detected as cracked divided by the total image area. It simply is the ratio of white pixels to the total number of pixels in the threshold image. For series of related images from the same specimen where the cracks grow and new cracks initiate this measure can be taken as an sanity check. If the cracks grow too close to each other the white bands in the threshold image merge. Then the crack density fails to detect two individual cracks since the skeletonizing will result in only one line for two merged bands. The crack density starts to decrease even tho the threshold density still rises. This is a sign that the crack detection reached its limit and the cracks in the images are too close to each other.

The crack density, crack coordinates (start- and endpoints) and the threshold density are the main results of the crack detection.

Disadvantages

The usage of a computed threshold for the separation between cracks and no cracks is "greedy". This means, cracks will be detected in the image. In images without cracks, artefacts will appear. This problem is dampened with the use of Yen´s threshold. Otsu´s threshold, as used in the original paper is even more "greedy" and will always detect cracks even if there are none.

Note: The results of this algorithm are sensitive to the input parameters especially to the parameters which control the gabor filter. Therefore it is a good practice to try the input parameters on a few images from the preprocessed stack before running the crack detection for the whole stack. The crack detection is resource intensive and can take a long time if lots of images are processed at once.

Bender´s algorithm

is implemented as *detect_cracks_bender()*

This method introduced by Bender JJ was also developed to detect off-axis cracks in fiber-reinforced polymers. It can only be used for a series of shift-corrected images and needs a background image as reference (the first image in the stack). Therefore, it is not as versatile as *detect_cracks()* but it has shown good results for image series up to high crack densities. It is also not "greedy" because a hard threshold is used.

The following filters and image processing steps are used to extract the cracks:

1. **Image History**: Starting form the second image in the stack, as the first is used as the background image, only darker pixels are taken form the image. This builds on the fact that cracks only result in dark pixels on the image and therefore, brighter pixels are only random noise.
2. **Image division** with background image (first image of the stack) to remove constant objects.
3. The image is divided by a blurred version of itself to remove the background.
4. A **directional Gaussian filter** is applied to diminish cracks in other than the given directions.
5. Images are **sharpened** with an unsharp_mask.
6. A **threshold** is applied to remove falsely identified cracks or artefacts with a weak signal.
7. **Morphological closing** of the image with a crack-like footprint to smooth the cracks and patch small discontinuities.
8. **Binarization** of the image. This results in an image with only cracks and background.

9. **Adding prior cracks**: The n-1st binarized image is added to add cracks already detected in prior images to the current image since cracks can only grow.
10. **Crack counting** with the skeletonizing and scanning method similar to the 4th point of *Glud´s algorithm*.

Note: This algorithm relies on the fact that cracks only grow. Also, cracks must not move between the images. Therefore, *Shift Correction* is important for this algorithm. If this prerequisites are nor met, do not use this crack detection method.

CHAPTER

FOUR

CONTRIBUTING

Clone the repository and add changes to it. Test the changes and make a pull request.

CHAPTER

FIVE

MODULES

imagestack	This module provides the core functionality for handling a stack of images at once.
image_functions	Preprocessing functions for single images
stack_operations	Routines for preprocessing image stacks.
io	IO module
crack_detection	Crack detection algorithms

5.1 imagestack

This module provides the core functionality for handling a stack of images at once.

Image stacks are objects that hold multiple images and act in many cases like python lists. They can be indexed and images in the stack can be changed. All images in one image stack must have the same dtype. If an image with another dtype is added or an image in the stack is replaced with an other image with different dtype, the incoming image is automatically converted to match the dtype of the image stack.

It is strongly recommended that *np.float32* is used when performing a crack detection. The crack detecion is tested and developed for images of dtypes *float*, *np.float64*, *np.float32* or *np.float16*.

Currently, there are two image stack objects that can be used. All image stack have the same structure. Accessing images, replacing images in the stack and adding new images works the same for all image stacks.

- *ImageStack*: A simple wrapper around a list. This container holds all images in the system memory (RAM).
- *ImageStackSQL*: Manages RAM usage of the image stack. Images are held in memory as long as the total available memory does not exceed a certain percentage of available memory or the image stack exceeds a set number of MB. If any more images are added, all current loaded images get stored in a database and only references to the images are kept in memory. The images are only loaded when directly accessed. This allows working and changing images of a stack even if the stack is too big to fit into the memory. The loaded images will be kept in memory until the stack exceeds the RAM limits again. This reduces the number loading and storing operations and therefore saves time since this can be quite time consuming for a lot of images.

The image stack is quite easy to use.

Classes

ImageStack([dtype])	This object holds multiple images.
ImageStackSQL([database, stack_name, dtype, ...])	This class works the same as ImageStack.
NumpyType()	Numpy Type for sql databases when using sqlalchemy.

5.1.1 ImageStack

class ImageStack(*dtype=<class 'numpy.float32'>*)

This object holds multiple images. All images are converted to the same datatype. This ensures that all images have the same characteristics for further processing.

All images are represented as numpy arrays. The same convention for representing images is used as in skimage.

If an image with mismatching dtype is added it is automatically converted to match the dtype. Read more about conversion details at skimage.util.dtype.

This object behaves a lot like a list. Individual images or groups of images can be retrieved with slicing. Setitem and delitem behaviour is like with normal python lists but mages can only be added with add_image.

Parameters

dtype: optional, default=np.float32
The dtype all images will be converted to. E.g. np.float32, bool, etc.

Examples

```
>>> # make an ImageStack object where all images are represented as unsigned␣
↪integer arrays [0-255]
>>> stack = ImageStack(dtype=np.uint8)
>>> # Add an image to it.
>>> img = (np.random.rand(200,200) * np.arange(200))/200  # floating point images␣
↪must be in range [-1,1]
>>> stack.add_image(img)
This ImageStack can be indexed.
>>> stack[0]  # getting the image with index 0 from the stack
Changing an image in the stack. The input will also be converted to the dtype of␣
↪the stack.
>>> stack[0] = (np.random.rand(200,200) * np.arange(200))/200[::-1]  # setting an␣
↪image in the stack
Or deleting an image form the stack
>>> del stack[0]
```

Methods

add_image(img)	Add an image to the stack.
change_dtype(dtype)	Change the dtype of all images in the stack.
copy()	Copy the current image stack.
execute_function(func, *args, **kwargs)	Perform an operation on all the images in the stack.
execute_rolling_function(func[, keep_first])	Perform an rolling operation on all the images in the stack.
from_paths(paths[, dtype])	Make an ImageStack object directly form paths of images.
remove_image([i])	Remove an image from the stack.

add_image(*img*)

Add an image to the stack. The image must be a numpy array

The input array will be converted to the dtype of the ImageStack

Parameters

img: np.ndarray

change_dtype(*dtype*)

Change the dtype of all images in the stack. All images will be converted to the new dtype.

Parameters

dtype

copy()

Copy the current image stack.

The copy is shallow until images are changed in the new stack.

Returns

out: ImageStack

execute_function(*func*, **args*, ***kwargs*)

Perform an operation on all the images in the stack.

The operation can be any function which takes one images and other arguments as input and returns only one image.

This operation changes the images in the stack. If the current state should be kept copy the stack first.

Parameters

func: function
A function which takes ONE image as first input and returns ONE image.

args:
args are forwarded to the func.

kwargs:
kwargs are forwarded to the func.

Examples

```
>>> def fun(img, to_add):
>>>     return img + to_add
```

```
>>> stack.execute_function(fun, to_add=4)
This will apply the function *fun* to all images in the stack.
```

execute_rolling_function(*func*, *keep_first=False*, **args*, ***kwargs*)

Perform an rolling operation on all the images in the stack.

The operation can be any function which takes two images and other arguments as input and returns only one image.

$I_{new} = func(I_{n-1}, I_n)$

This operation changes the images in the stack. If the current state should be kept copy the stack first.

Since the 0-th image in the stack will remain unchanged because the rolling operation starts at the 1-st image, the 0-th image is removed if *keep_first* is set to *False* (default).

Parameters

func: function
A function which takes TWO images and other arguments as input and returns ONE image. The function must have the following input structure: *fun(img1, img2, args, kwargs)*. *img1* will be the n-1st image in the calls.

keep_first: bool
If True, keeps the first image in the stack. Delete it otherwise.

args:
args are forwarded to the func.

kwargs:
kwargs are forwarded to the func.

Examples

```
>>> def fun(img1, img2):
>>>     mask = img1 > img1.max()/2
>>>     return img2[mask]
```

```
>>> stack.execute_rolling_function(fun, keep_first=False)
This will apply the function *fun* to all images in the stack.
```

img1 is always the n-1st image in the rolling operation.

classmethod from_paths(*paths*, *dtype=None*, ***kwargs*)

Make an ImageStack object directly form paths of images. The images will be loaded, converted to the dtype of the ImageStack and added.

Parameters

paths: list
paths of the images to be added

dtype: optional
The dtype all images will be converted to. E.g. np.float32, bool, etc. If this is not set, the dtype of the first image loaded will determine the dtype of the stack.

kwargs:
kwargs are forwarded to skimage.io.imread For grayscale images simply add **as_gray = True**. For the kwargs for colored images use parameters for reading. Keep in mind that some images might have alpha channels and some not even if they have the same format.

Returns

out: ImageStack
An ImageStack with all images from paths as arrays.

Examples

```
>>> paths = ['list of image paths']
>>> stack = ImageStack.from_paths(paths, as_gray=True)
```

remove_image(*i=-1*)

Remove an image from the stack.

Parameters

i: int
Index of the image to be removed

5.1.2 ImageStackSQL

class ImageStackSQL(*database=''*, *stack_name=''*, *dtype=<class 'numpy.float32'>*, *max_size_mb=None*, *cache_limit=80*)

This class works the same as ImageStack.

ImageStackSQL objects will track the amount of memory the images occupy. When the memory limit if surpassed, all data will be stored in an sqlite database and the RAM will be cleared. Only a lazy loaded object is left in the image stack. Only when directly accessing the images in the stack they will be loaded into RAM again. sqlalchemy is used to connect to the database in which all data is stored.

This makes this container suitable for long term storage and transfer of a lot of images. The images can be loaded into an ImageStackSQL object in a new python session.

Parameters

database: str, optional
Path of the database. If it does not exist, it will be created. If none is entered, the name is id(object)

stack_name: str, optional
The name of the table the images will be saved. If none is entered it will be id(object)

dtype: optional, default=np.float32
The dtype all images will be converted to. E.g. np.float32, bool, etc.

max_size_mb: float, optional
The maximal size in mb the image stack is allowed to be. If a new image is added after surpassing this size all images will be saved in the database and the occupied RAM is cleared. All images are still accessible but will be loaded only when directly accessed.

cache_limit: float, optional, default=90
The limit of the RAM usage in percent of the available system RAM. When the RAM usage of the system surpasses this limit, all images will be saved in the database and RAM is freed again even it max_size_mb is not reached. This makes sure that the system never runs out of RAM. Values over 100 will effectively deactivate this behaviour. If the total size of the image stack is too small to free enough RAM to reach the cache limit newly added images will be saved immediately in the database. This also lead to constant reads from the database as no images will be kept im RAM. Therefore it is recommended to set this well over the current RAM usage of the system when instantiating an object.

Attributes

nbytes
Sum of bytes for all currently fully loaded images.

Methods

add_image(img)	Add an image to the stack.
change_dtype(dtype)	Change the dtype of all images in the stack.
copy([stack_name])	Copy the current image stack.
execute_function(func, *args, **kwargs)	Perform an operation on all the images in the stack.
execute_rolling_function(func[, keep_first])	Perform an rolling operation on all the images in the stack.
from_paths(paths[, database, stack_name, ...])	Make an ImageStackSQL object directly form paths of images.
load_from_database([database, stack_name])	Load an image stack from a database.
relaod()	Reload the images form the table.
remove_image([i])	Remove an image from the stack.
save_state()	Saves the current state of the image stack to the database.

add_image(*img*)

Add an image to the stack. The image must be a numpy array

The input array will be converted to the dtype of the ImageStack

Parameters

img: np.ndarray

change_dtype(*dtype*)

Change the dtype of all images in the stack. All images will be converted to the new dtype.

Parameters

dtype

copy(*stack_name=''*)

Copy the current image stack.

A new table in the database is created where all images are stored.

Parameters

stack_name: str
The name of the stack. This is also the name of the new table in the database

Returns

out: ImageStack

execute_function(*func*, **args*, ***kwargs*)

Perform an operation on all the images in the stack.

The operation can be any function which takes one images and other arguments as input and returns only one image.

This operation changes the images in the stack. If the current state should be kept copy the stack first.

Parameters

func: function
A function which takes ONE image as first input and returns ONE image.

args:
args are forwarded to the func.

kwargs:
kwargs are forwarded to the func.

Examples

```
>>> def fun(img, to_add):
>>>     return img + to_add
```

```
>>> stack.execute_function(fun, to_add=4)
This will apply the function *fun* to all images in the stack.
```

execute_rolling_function(*func*, *keep_first=False*, **args*, ***kwargs*)

Perform an rolling operation on all the images in the stack.

The operation can be any function which takes two images and other arguments as input and returns only one image.

$I_{new} = func(I_{n-1}, I_n)$

This operation changes the images in the stack. If the current state should be kept copy the stack first.

Since the 0-th image in the stack will remain unchanged because the rolling operation starts at the 1-st image, the 0-th image is removed if *keep_first* is set to *False* (default).

Parameters

func: function
A function which takes TWO images and other arguments as input and returns ONE image. The function must have the following input structure: *fun(img1, img2, args, kwargs)*. *img1* will be the n-1st image in the calls.

keep_first: bool
If True, keeps the first image in the stack. Delete it otherwise.

args:
args are forwarded to the func.

kwargs:
kwargs are forwarded to the func.

Examples

```
>>> def fun(img1, img2):
>>>     mask = img1 > img1.max()/2
>>>     return img2[mask]
```

```
>>> stack.execute_rolling_function(fun, keep_first=False)
This will apply the function *fun* to all images in the stack.
```

img1 is always the n-1st image in the rolling operation.

classmethod from_paths(*paths*, *database=''*, *stack_name=''*, *dtype=None*, *max_size_mb=None*, *cache_limit=80*, ***kwargs*)

Make an ImageStackSQL object directly form paths of images. The images will be loaded, converted to the dtype of the ImageStack and added.

Parameters

paths: list
paths of the images to be added

database: str, optional
Path of the database. If it does not exist, it will be created. If none is entered, the name is id(object)

stack_name: str, optional
The name of the table the images will be saved. If none is entered it will be id(object)

dtype: optional
The dtype all images will be converted to. E.g. np.float32, bool, etc. If this is not set, the dtype of the first image loaded will determine the dtype of the stack.

max_size_mb: float, optional
ImageStackSQL for more details.

cache_limit: float, optional, default=90
:class`ImageStackSQL` for more details.

kwargs:
kwargs are forwarded to skimage.io.imread For grayscale images simply add **as_gray = True**. For the kwargs for colored images use parameters for reading. Keep in mind that some images might have alpha channels and some not even if they have the same format.

Returns

out: ImageStackSQL
A new ImageStackSQL object with connection to the database.

classmethod load_from_database(*database=''*, *stack_name=''*)

Load an image stack from a database.

A table of a database which was made with an ImageStackSQL object can be loaded and an ImageStackSQL object with all the images is made. The dtype of the images in the new object is the same as the images in the table. All images, which will be added to the object will be converted to match the dtype.

Parameters

database: str
Path of the database.

stack_name: str
Name of the table

Returns

out: ImageStackSQL
The image stack object with connection to the database.

`property nbytes`

Sum of bytes for all currently fully loaded images.

This tracks the used RAM from the images. The overhead of the used RAM from sqlalchemy is not included and will not be tracked.

`relaod()`

Reload the images form the table. All not saved changes will be lost.

`remove_image(i=-1)`

Remove an image from the stack.

Parameters

i: int
Index of the image to be removed

`save_state()`

Saves the current state of the image stack to the database.

This commits all changes and adds all new images to the table. All currently loaded images are expired. This means, that all RAM used by the images is freed.

Call this method before closing the python session if the changes made to the image stack should be saved permanently.

5.1.3 NumpyType

`class NumpyType`

Numpy Type for sql databases when using sqlalchemy.

This handles the IO with a sql database and sqlalchemy.

Inside the database, an numpy array is stored as LargeBinary. sqlalchemy handles loading and storing of entries for columns marked with this custom type. All arrays are converted to numpy arrays when loading and converted to binary when storing in the database automatically.

Methods

bind_processor(dialect)	Provide a bound value processing function for the given `Dialect`.
impl	alias of `LargeBinary`
result_processor(dialect, coltype)	Provide a result value processing function for the given `Dialect`.

bind_processor(*dialect*)

Provide a bound value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for bound value conversion which normally occurs via the `_types.TypeEngine.bind_processor()` method.

Note: User-defined subclasses of `_types.TypeDecorator` should **not** implement this method, and should instead implement `_types.TypeDecorator.process_bind_param()` so that the "inner" processing provided by the implementing type is maintained.

Parameters

dialect – Dialect instance in use.

impl

alias of `LargeBinary`

result_processor(*dialect*, *coltype*)

Provide a result value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for bound value conversion which normally occurs via the `_types.TypeEngine.result_processor()` method.

Note: User-defined subclasses of `_types.TypeDecorator` should **not** implement this method, and should instead implement `_types.TypeDecorator.process_result_value()` so that the "inner" processing provided by the implementing type is maintained.

Parameters

- **dialect** – Dialect instance in use.
- **coltype** – A SQLAlchemy data type

5.2 image_functions

Preprocessing functions for single images

In this module image processing functions for single images are grouped as well as helper functions.

Functions

detect_changes_division(img1, img2[, ...])	Detect the changes between two images by division.
detect_changes_subtraction(img1, img2[, ...])	Simple change detection by image subtracting.
scale_to(x[, x_min, x_max])	Linear scaling to new range of values.

5.2.1 detect_changes_division

detect_changes_division(*img1*, *img2*, *output_range=None*)

Detect the changes between two images by division. Areas with a lot of change will appear darker and areas with little change bright.

The change from img1 to img2 is computed with.

$I_d = \frac{img2+1}{img1+1}$

The range of the output image is scaled to the range of the datatype of the input image.

Parameters

img1: np.ndarray
img2: np.ndarray
output_range: tuple, optional
Range of the output image. This range must be within the possible range of the dtype of the input images. E.g. (0,1) for float images.

Returns

out: np.ndarray
Input dtpye is only converved when an output_range is given. Else the result will have dtype float with a maximal possible range of 0.5-2

5.2.2 detect_changes_subtraction

detect_changes_subtraction(*img1*, *img2*, *output_range=None*)

Simple change detection by image subtracting. Areas with a lot of change will appear darker and areas with little change bright.

$I_d = I_2 - I_1$

Parameters

img1: np.ndarray
img2: np.ndarray
output_range: tuple, optional
Range of the output image. This range must be within the possible range of the dtype of the input images E.g. (0,1) for float images.

Returns

out: np.ndarray
Image as input dtype

5.2.3 scale_to

scale_to(*x*, *x_min=0*, *x_max=1*)

Linear scaling to new range of values.

This function scales the input values from their current range to the given range.

Parameters

x: array-like
x_min: float
Lower boarder of the new range

x_max: float
Upper boarder of the new range

Returns

out:
Scaled values

Examples

```
>>> y = np.array((-1,2,5,7))
>>> scale_to(y, x_min=0, x_max=1)
array([0.   , 0.375, 0.75 , 1.   ])
```

5.3 stack_operations

Routines for preprocessing image stacks.

All functions in this module are designed to take an image stack and additional arguments as input.

The main functionality consists of different methods for shift correction and change detection for consecutive images.

Functions

biggest_common_sector(images)	Biggest common sector of the image stack
change_detection_division(images[, output_range])	Change detection for all images in an image stack.
change_detection_subtraction(images[, ...])	Change detection for all images in an image stack.
cut_images_to_same_shape(images)	Cuts all images in a stack to the same shape.
image_shift(images)	Compute the shift of all images in a stack.
region_of_interest(images[, x0, x1, y0, y1])	Crop all images in a stack to the desired shape.
shift_correction(images)	Shift correction of all images in a stack.
shift_distortion_correction(images[, ...])	Shift and distortion (=strain) correction for all images in a stack.

5.3.1 biggest_common_sector

biggest_common_sector(*images*)

Biggest common sector of the image stack

This function computes the relative translation between the images with the first image in the stack as the reference image. Then the biggest common sector is cropped from the images. The cropping window moves with the relative translation of the images so that the translation is corrected.

Warping of the images which could be a result of strain is not accounted for. If the warp cant be neglected do not use this method!!

Parameters

images: list or ImageStack
Images represented as np.ndarray. All images must have the same dimensionality! If the

width and height of the images is not the same, they are cut to the shape of the smallest image.

Returns

out: list, ImageStack
list or ImageStack with the corrected images.

5.3.2 change_detection_division

`change_detection_division`(*images*, *output_range=None*)

Change detection for all images in an image stack.

Change detection with image rationing is applied to an image stack. The new images are the result of the change between the n-th and the n-1st image.

The first image will be deleted from the stack.

Parameters

images: ImageStack, list
output_range: tuple, optional
The resulting images will be rescaled to the given range. E.g. (0,1).

Returns

out: ImageStack, list

5.3.3 change_detection_subtraction

`change_detection_subtraction`(*images*, *output_range=None*)

Change detection for all images in an image stack.

Change detection with image differencing is applied to an image stack. The new images are the result of the change between the n-th and the n-1st image.

The first image will be deleted from the stack.

Parameters

images: ImageStack, list
output_range: tuple, optional
The resulting images will be rescaled to the given range. E.g. (0,1).

Returns

out: ImageStack, list

5.3.4 cut_images_to_same_shape

`cut_images_to_same_shape`(*images*)

Cuts all images in a stack to the same shape.

The images are cut to the shape of the smallest image in the stack. The top left corner is 0,0 and the

Parameters

images: list, ImageStack

Returns

out: list, ImageStack
list or ImageStack with all images in the same shape.

5.3.5 image_shift

image_shift(*images*)

Compute the shift of all images in a stack.

The shift of the n+1st image relative to the n-th is computed. The commutative sum of these shifts is the shift relative to the 0th image in the stack.

All input images must have the same width and height! :param images: :type images: ImageStack, list

Returns

out: list
[(0,0), (y1, x1), ...(yn, xn)] The shift in x and y direction relative to the first image in the stack.

5.3.6 region_of_interest

region_of_interest(*images*, *x0=0*, *x1=None*, *y0=0*, *y1=None*)

Crop all images in a stack to the desired shape.

This function changes the images in the stack. If the input images should be preserved copy the input to a separate object before!

The coordinate system is the following: x0->x1 = width, y0->y1 = height from the top left corner of the image

Parameters

images: list, ImageStack
x0: int
x1: int
y0: int
y1: int

Returns

out: list, ImageStack
ImageStack or list with the cropped images

5.3.7 shift_correction

shift_correction(*images*)

Shift correction of all images in a stack. This function is more precise than *biggest_common_sector()* but more time consuming. The memory footprint is the same.

This function computes the relative translation between the images with the first image in the stack as the reference image. The images are translated into the coordinate system of the 0th image form the stack.

Warping of the images which could be a result of strain is not accounted for. If the warp cant be neglected do not use this function!

Parameters

images: list, ImageStack
Images represented as np.ndarray. All images must have the same dimensionality! If the width and height of the images is not the same, they are cut to the shape of the smallest image.

Returns

out: list, ImageStack
list or ImageStack with the corrected images.

5.3.8 shift_distortion_correction

`shift_distortion_correction`(*images*, *reg_regions=((0, 0, 0.1, 0.1), (0.9, 0, 1, 0.1), (0, 0.9, 0.1, 1), (0.9, 0.9, 1, 1))*, *absolute=False*)

Shift and distortion (=strain) correction for all images in a stack.

This function computes the relative translation and the warp between the images with the first image in the stack as the reference image. The images are translated into the coordinate system of the first image. Black areas in the resulting images are the result of the transformation into the reference coordinate system.

Four rectangular areas must be chosen from which the global shift and distortion relative to each other is computed.

Parameters

images: list, ImageStack
Images represented as np.ndarray. All images must have the same dimensionality! If the width and height of the images is not the same, they are cut to the shape of the smallest image.

reg_regions: tuple
A tuple of four tuples containing the upper left and lower right corners of the rectangles for the areas where the phase-cross-correlation is computed. E.g. ((x1, y1, x2, y2), (...), (...), (...)) where x1, y1 etc. can be relative dimensions of the image or the absolute coordinates in pixel. Default is relative. For relative coordinates enter values from 0-1. If values above 1 are given, absolute coordinate values are assumed.

absolute: bool
True if absolute and False if relative values are given in reg_regions

Returns

out: list, ImageStack
list or ImageStack with the corrected images.

Notes

For this algorithm to work properly it is advantageous to have markers on reach corner of the images (like crosses). Make sure that the rectangular areas cover the markers in each image. If no distinct features that can be tracked are given, this function can result in wrong shift and distortion corrections. In this case, make sure to check the results before further image processing steps.

5.4 io

IO module

Convenience functions for handling image paths, sorting paths and loading images.

The default dtype for the images is np.float32. This saves memory compared to np.float64 without significant losses in accuracy since these images are normally represented in a range from 0 -> 1 or -1 -> 1.

Functions

general_path_sorter(path_list, pattern)	General sorting of paths in ascending order with regex pattern.
image_paths(img_dir[, image_types])	Selects all images given in the image_types list from a directory
load_images(paths[, dtype])	Loads all images from the paths into memory and stores them in an ImageStack.
plot_cracks(image, cracks[, linewidth, ...])	Plots cracks in the foreground with an image in the background.
plot_gabor(kernel[, cmap])	Plot the real part of the Gabor kernel
save_images(images[, image_format, names, ...])	Save all images.
sort_paths(path_list[, sorting_key])	Sorts the given paths according to a sorting keyword.

5.4.1 general_path_sorter

general_path_sorter(*path_list*, *pattern*)

General sorting of paths in ascending order with regex pattern.

The regex pattern must contain a "group1" which matches any number. The paths are then sorted with this number in ascending order.

Parameters

path_list: list
List of filenames or paths

pattern: str
regex pattern that matches any number. The group must be marked as "group1". E.g. "(?P<group1>[0-9]+)cycles" would match the number 1234 in "1234cycles".

Returns

paths: array
Sorted paths in ascending order

numbers: array
The numbers from the match corresponding to the paths.

5.4.2 image_paths

image_paths(*img_dir*, *image_types=('jpg', 'png', 'bmp')*)

Selects all images given in the image_types list from a directory

Parameters

img_dir: str
path to the directory which includes the images

image_types: list
a list of strings of the image types to select

Returns

image_paths: list

5.4.3 load_images

load_images(*paths*, *dtype=None*, ***kwargs*)

Loads all images from the paths into memory and stores them in an ImageStack.

Parameters

paths: list
Paths of the images

dtype: dtype, optional
The dtype all images will be converted to.

kwargs:
All kwargs are forwarded to *crackdect.imagestack.ImageStack.from_paths()*

Returns

out: ImageStack

5.4.4 plot_cracks

plot_cracks(*image*, *cracks*, *linewidth=1*, *color='red'*, *comparison=False*, ***kwargs*)

Plots cracks in the foreground with an image in the background.

Parameters

image: np.ndarray
Background image

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

kwargs:
Forwarded to plt.figure()

Returns

fig: Figure
ax: Axes

5.4.5 plot_gabor

`plot_gabor`(*kernel*, *cmap='plasma'*, ***kwargs*)

Plot the real part of the Gabor kernel

Parameters

kernel: CrackDetectionTWLI, np.ndarray
cmap: str
Identifier for a cmap from matplotlib

kwargs:
Forwarded to plt.figure()

Returns

fig: Figure
ax: Axes

5.4.6 save_images

`save_images`(*images*, *image_format='jpg'*, *names=''*, *directory=None*)

Save all images.

This saves all images in a stack to a given directory in the given file format.

Parameters

images: iterable
An iterable like ImageStack with images represented as np.ndarray

image_format: str, optional
The format in which to save the images. Default is jpg.

names: list, optional
A list of names corresponding to the images. If none is entered the images will be saved as 1.jpg, 2.jpg, … If these names have extensions for the image format, this extension is ignored. All images are saved with image_format

directory: str, optional
Path to the directory where the images should be saved. If the path does not exist it is created. Default is the current working directory

5.4.7 sort_paths

`sort_paths`(*path_list*, *sorting_key='cycles'*)

Sorts the given paths according to a sorting keyword.

The paths must have the following structure: /xx/xx…/[NUMBER][SORTING_KEY].* E.g. test_3cycles_force1.png. This will extract the number 3 from the filename and sort other similar filenames in ascending order.

Parameters

path_list: list
list of strings of the image names

sorting_key: str
A sorting keyword. The number for sorting must be before the keyword. E.g. "cycles" will sort all paths with [NUMBER]cycles.* e.g. 123cycles, 234cycles,.. etc

Returns

paths: array
Sorted paths in ascending order

numbers: array
The numbers from the match corresponding to the paths.

Examples

```
>>> paths = ['A_1cycles.jpg', 'A_50cycles.jpg', 'A_2cycles.jpg', 'A_test.jpg']
>>> sort_paths(paths, 'cycles')
(array(['A_1cycles.jpg', 'A_2cycles.jpg', 'A_50cycles.jpg']), array([ 1,  2, 50]))
```

5.5 crack_detection

Crack detection algorithms

These module contains the different functions for the crack detection. This includes functions for different sub-algorithms which are used in the final crack detection as well as different methods for the crack detection. The different crack detection methods are available as functions with an image stack and additional arguments as input.

Classes

CrackDetectionBender([theta, crack_width, ...])	Base class for the crack detection `method by J.J.
CrackDetectionTWLI([theta, frequency, ...])	The basic method from Glud et al. for crack detection without preprocessing.

5.5.1 CrackDetectionBender

class CrackDetectionBender(*theta=0*, *crack_width=10*, *threshold=0.96*, *min_size=None*)

Base class for the crack detection method by J.J. Bender.

This crack detection algorithm only works on an image stack with consecutive images of one specimen. The first image is used as the background image. No cracks are detected in the first image. The images must be aligned for this algorithm to work correctly. Cracks can only be detected in grayscale images with the same shape.

Following filters are applied to the images: 1: Apply image history. Images must become darker with time. This subsequently reduces noise in the imagestack 2: Image division with reference image (first image of the stack) to remove constant objects. 3: The image is divided by a blurred version of itself to remove the background. 4: A directional Gaussian filter is applied to diminish cracks in other directions. 5: Images are sharpened with an unsharp_mask. 6: A threshold is applied to remove falsely identified cracks or artefacts with a weak signal. 7: Morphological closing of the image with a crack-like footprint. 8: Binarization of the image 9: The n-1st binaritzed image is added. 10: Finde the cracks with the skeletonizing and scanning method.

Parameters

theta: float
Angle of the cracks in respect to a horizontal line in degrees

crack_width: int
The approximate width of an average crack in pixel. This determines the width of the detected features.

threshold: float, optional
Threshold of what is perceived as a crack after all filters. E.g. 0.96 means that all gray values over 0.96 in the filtered image are cracks. This value should be close to 1. A lower value will detect cracks with a weak signal but more artefacts as well. Default: 5

min_size: int, optional
The minimal number of pixels a crack can be. Cracks under this size will not get counted. Default: 5

Returns

rho_c: float
Crack density [1/px]

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

rho_th: float
A measure how much of the area of the input image is detected as foreground. If the gabor filter can not distinguish between cracks with very little space in between the crack detection will break down and lead to false results. If this value is high but the crack density is low, this is an indicator that the crack detection does not work with the given input parameters and the input image.

Methods

`anisotropic_gauss_filter`(image, kernel)

detect_cracks	
make_footprint	

5.5.2 CrackDetectionTWLI

class CrackDetectionTWLI(*theta=0*, *frequency=0.1*, *bandwidth=1*, *sigma_x=None*, *sigma_y=None*, *n_stds=3*, *min_size=5*, *threshold='yen'*, *sensitivity=0*)

The basic method from Glud et al. for crack detection without preprocessing.

This is the basis for a crack detection with this method. Each object from this class can be used to detect cracks from images. The workflow of objects from this class is quite easy.

1. Object instantiation. Create an object from with the input parameter for the crack detection.
2. Call the method *detect_cracks()* with an image as input. This method will call all sub-functions of the crack detection.

1. apply the gabor filter
2. apply otsu´s threshold to split the image into foreground and background.
3. skeletonize the foreground
4. find the cracks in the skeletonized image.

Shift detection, normalization, and other preprocessing procedures are not performed! It is assumed that all the necessary preprocessing is already done for the input image. For preprocessing please use the *stack_operations* or other means.

Parameters

theta: float
Angle of the cracks in respect to a horizontal line in degrees

frequency: float, optional
Frequency of the gabor filter. Default: 0.1

bandwidth: float, optional
The bandwidth of the gabor filter, Default: 1

sigma_x: float, optional
Standard deviation of the gabor kernel in x-direction. This applies to the kernel before rotation. The kernel is then rotated *theta* degrees.

sigma_y: float, optional
Standard deviation of the gabor kernel in y-direction. This applies to the kernel before rotation. The kernel is then rotated *theta* degrees.

n_stds: int, optional
The size of the gabor kernel in standard deviations. A smaller kernel is faster but also less accurate. Default: 3

min_size: int, optional
The minimal number of pixels a crack can be. Cracks under this size will not get counted. Default: 1

threshold: str
Method of determining the threshold between foreground and background. Choose between 'otsu' or 'yen'. Generally, yen is not as sensitive as otsu. For blurry images with lots of noise yen is nearly always better than otsu.

sensitivity: float, optional
Adds or subtracts x percent of the input image range to the threshold. E.g. sensitivity=-10 will lower the threshold to determine foreground by 10 percent of the input image range. For crack detection with bad image quality or lots of artefacts it can be helpful to lower the sensitivity to avoid too much false detections.

Methods

`__call__`(image, **kwargs)	Call self as a function.
detect_cracks(image[, out_intermediate_images])	Compute all steps of the crack detection
foreground_pattern(image[, method, sensitivity])	Apply the threshold to an image do determine foreground and background of the image.

`detect_cracks`(*image*, *out_intermediate_images=False*)

Compute all steps of the crack detection

Parameters

image: np.ndarray
out_intermediate_images: bool, optional
If True the result of the gabor filter, the foreground pattern as a result of the otsu´s threshold and the skeletonized image are also included in the output. As this are three full sized images the default is False.

Returns

crack_density: float
cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

threshold_density: float
A measure how much of the area of the input image is detected as foreground. If the gabor filter can not distinguish between cracks with very little space in between the crack detection will break down and lead to false results. If this value is high but the crack density is low, this is an indicator that the crack detection does not work with the given input parameters and the input image.

gabor: np.ndarray, optional
The result of the Gabor filter.

pattern: np.ndarray, optional
A bool image the crack detection detects as cracked area.

skel_image: np.ndarray, optional
The skeletonized pattern as bool image.

`static foreground_pattern`(*image*, *method='yen'*, *sensitivity=0*)

Apply the threshold to an image do determine foreground and background of the image.

The result is a bool array with where True is foreground and False background of the image. The image can be split with image[pattern] into foreground and image[~pattern] into background.

Parameters

image: array-like
method: str
Method of determining the threshold between foreground and background. Choose between 'otsu' or 'yen'.

sensitivity: float, optional
Adds or subtracts x percent of the input image range to the threshold. E.g. sensitivity=-10 will lower the threshold to determine foreground by 10 percent of the input image range.

Returns
——-
pattern: numpy.ndarray
Bool image with True as foreground.

Functions

anisotropic_gauss_kernel(sig_x, sig_y[, ...])	Gaussian kernel with different standard deviations in x and y direction.
crack_density(cracks, area)	Compute the crack density from an array of crack coordinates.
cracks_skeletonize(pattern, theta[, min_size])	Get the cracks and the skeletonized image from a pattern.
detect_cracks(images[, theta, crack_width, ...])	Crack detection based on a simpler version of the algorithm by J.A.
detect_cracks_bender(images[, theta, ...])	Crack detection `algorithm by J.J.
detect_cracks_glud(images[, theta, ...])	Crack detection using a slightly modified version of the `algorithm from J.A.
find_cracks(skel_im, min_size)	Find the cracks in a skeletonized image.
rotation_matrix_z(phi)	Rotation matrix around the z-axis.

5.5.3 anisotropic_gauss_kernel

anisotropic_gauss_kernel(*sig_x*, *sig_y*, *theta=0*, *truncate=3*)

Gaussian kernel with different standard deviations in x and y direction.

Parameters

sig_x: int
Standard deviation in x-direction. A value of e.g. 5 means that the Gaussian kernel will reach a standard deviation of 1 after 5 pixel.

sig_y: int
Standard deviation in y-direction.

theta: float
Angle in degrees

truncate: float
Truncate the filter at this many standard deviations. Default is 4.0.

Returns

———-

kernel: ndarray
The Gaussian kernel as a 2D array.

5.5.4 crack_density

crack_density(*cracks*, *area*)

Compute the crack density from an array of crack coordinates.

The crack density is the combined length of all cracks in a given area. Therefore, its unit is m^-1.

Parameters

cracks: array-like
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1]], [[…]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

area: float
The area to which the density is refered to.

Returns

crack density: float

5.5.5 cracks_skeletonize

`cracks_skeletonize`(*pattern*, *theta*, *min_size=5*)

Get the cracks and the skeletonized image from a pattern.

Parameters

pattern: array-like
True/False array representing the white/black image

theta: float
The orientation angle of the cracks in degrees!!

min_size: int
The minimal length of pixels for which will be considered a crack

Returns

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

skeletonized: np.ndarray
skeletonized image

5.5.6 detect_cracks

`detect_cracks`(*images*, *theta=0*, *crack_width=10*, *ar=2*, *bandwidth=1*, *n_stds=3*, *min_size=5*, *threshold='yen'*, *sensitivity=0*)

Crack detection based on a simpler version of the algorithm by J.A. Glud. All images are treated separately.

Parameters

images: ImageStack, list
Image stack or list of grayscale images on which the crack detection will be performed. This algorithm treats each image separately as no image influences the results of the other images.

theta: float
Angle of the cracks in respect to a horizontal line in degrees

crack_width: int
The approximate width of an average crack in pixel. This determines the width of the detected features.

ar: float
The aspect ratio of the gabor kernel. Since cracks are a lot longer than wide a longer gabor kernel will automatically detect cracks easier and artifacts are filtered out better. A too large aspect ratio will result in an big kernel which slows down the computation. Default: 2

bandwidth: float, optional
The bandwidth of the gabor filter, Default: 1

n_stds: int, optional
The size of the gabor kernel in standard deviations. A smaller kernel is faster but also less accurate. Default: 3

min_size: int, optional
The minimal number of pixels a crack can be. Cracks under this size will not get counted. Default: 5

threshold: str
Method of determining the threshold between foreground and background. Choose between 'otsu' or 'yen'. Generally, yen is not as sensitive as otsu. For blurry images with lots of noise yen is nearly always better than otsu.

sensitivity: float, optional
Adds or subtracts x percent of the input image range to the Otsu-threshold. E.g. sensitivity=-10 will lower the threshold to determine foreground by 10 percent of the input image range. For crack detection with bad image quality or lots of artefacts it can be helpful to lower the sensitivity to avoid too much false detections.

Returns

rho_c: float
Crack density [1/px]

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

rho_th: float
A measure how much of the area of the input image is detected as foreground. If the gabor filter can not distinguish between cracks with very little space in between the crack detection will break down and lead to false results. If this value is high but the crack density is low, this is an indicator that the crack detection does not work with the given input parameters and the input image.

5.5.7 detect_cracks_bender

`detect_cracks_bender`(*images*, *theta=0*, *crack_width=10*, *threshold=0.96*, *min_size=None*)

Crack detection algorithm by J.J. Bender.

This crack detection algorithm only works on an image stack with consecutive images of one specimen. The first image is used as the background image. No cracks are detected in the first image. The images must be aligned for this algorithm to work correctly. Cracks can only be detected in grayscale images with the same shape.

Parameters

images: ImageStack, list
Image stack or list with consecutive grayscale images (np.ndarray) of the same shape and aligned.

theta: float
Angle of the cracks in respect to a horizontal line in degrees

crack_width: int
The approximate width of an average crack in pixel. This determines the width of the detected features.

threshold: float, optional
Threshold of what is perceived as a crack after all filters. E.g. 0.96 means that all gray values over 0.96 in the filtered image are cracks. This value should be close to 1. A lower value will detect cracks with a weak signal but more artefacts as well. Default: 5

min_size: int, optional
The minimal number of pixels a crack can be. Cracks under this size will not get counted. Default: 5

Returns

rho_c: float
Crack density [1/px]

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

rho_th: float
A measure how much of the area of the input image is detected as foreground. If the gabor filter can not distinguish between cracks with very little space in between the crack detection will break down and lead to false results. If this value is high but the crack density is low, this is an indicator that the crack detection does not work with the given input parameters and the input image.

5.5.8 detect_cracks_glud

`detect_cracks_glud`(*images*, *theta=0*, *crack_width=10*, *ar=2*, *bandwidth=1*, *n_stds=3*, *min_size=5*, *threshold='yen'*, *sensitivity=0*)

Crack detection using a slightly modified version of the algorithm from J.A. Glud.

This crack detection algorithm only works on an image stack with consecutive images of one specimen. In contrast to the original algorithm from J.A. Glud, no change detection is applied in this implementation since it can be easily applied as a preprocessing step if needed (see *stack_operations*)

Parameters

images: ImageStack, list
Image stack or list with consecutive grayscale images (np.ndarray) of the same shape and aligned.

theta: float
Angle of the cracks in respect to a horizontal line in degrees

crack_width: int
The approximate width of an average crack in pixel. This determines the width of the detected features.

ar: float
The aspect ratio of the gabor kernel. Since cracks are a lot longer than wide a longer gabor kernel will automatically detect cracks easier and artifacts are filtered out better. A too large aspect ratio will result in an big kernel which slows down the computation. Default: 2

bandwidth: float, optional
The bandwidth of the gabor filter, Default: 1

n_stds: int, optional
The size of the gabor kernel in standard deviations. A smaller kernel is faster but also less accurate. Default: 3

min_size: int, optional
The minimal number of pixels a crack can be. Cracks under this size will not get counted. Default: 5

threshold: str
Method of determining the threshold between foreground and background. Choose between 'otsu' or 'yen'. Generally, yen is not as sensitive as otsu. For blurry images with lots of noise yen is nearly always better than otsu.

sensitivity: float, optional
Adds or subtracts x percent of the input image range to the threshold. E.g. sensitivity=-10 will lower the threshold to determine foreground by 10 percent of the input image range. For crack detection with bad image quality or lots of artefacts it can be helpful to lower the sensitivity to avoid too much false detections.

Returns

rho_c: float
Crack density [1/px]

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

rho_th: float
A measure how much of the area of the input image is detected as foreground. If the gabor filter can not distinguish between cracks with very little space in between the crack detection will break down and lead to false results. If this value is high but the crack density is low, this is an indicator that the crack detection does not work with the given input parameters and the input image.

5.5.9 find_cracks

`find_cracks`(*skel_im*, *min_size*)

Find the cracks in a skeletonized image.

This function finds the start and end of the cracks in a skeletonized image. All cracks must be aligned approximately vertical.

Parameters

skel_im: np.ndarray
Bool-Image where False is background and True is the 1 pixel wide representation of the crack.

min_size: int
Minimal minimal crack length in pixels that is detected.

Returns

cracks: np.ndarray
Array with the coordinates of the crack with the following structure: ([[x0, y0],[x1,y1], [...]]) where x0 and y0 are the starting coordinates and x1, y1 the end of one crack. Each crack is represented by a 2x2 array stacked into a bigger array (x,2,2).

5.5.10 rotation_matrix_z

`rotation_matrix_z`(*phi*)

Rotation matrix around the z-axis.

Computes the rotation matrix for the angle phi(radiant) around the z-axis

Parameters

phi: float
rotation angle (radiant)

Returns

R: array
3x3 rotation matrix

CHAPTER

SIX

AUTHORS

- Matthias Drvoderic

CHAPTER

SEVEN

LICENSE

This project is licensed under the MIT License

CHAPTER

EIGHT

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

C

INDEX

L

M

N

P

R

S