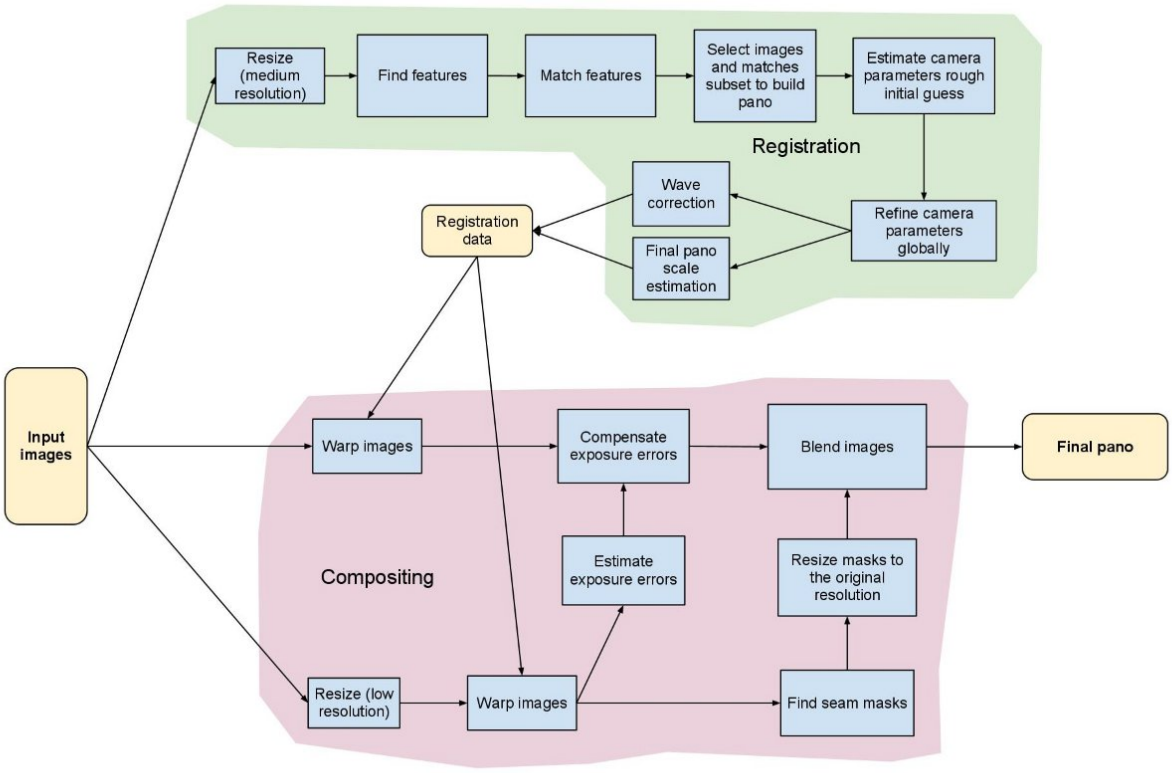
Stitching Tutorial

The Workflow of the Stitching Pipeline can be seen in the following. Note that the image comes from the OpenCV Documentation (https://docs.opencv.org/3.4/d1/d46/group__stitching.html).



With the following block, we allow displaying resulting images within the notebook:

In [1]:
```
from matplotlib import pyplot as plt
import cv2 as cv
import numpy as np

def plot_image(img, figsize_in_inches=(5,5)):
    fig, ax = plt.subplots(figsize=figsize_in_inches)
    ax.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
    plt.show()

def plot_images(imgs, figsize_in_inches=(5,5)):
    fig, axs = plt.subplots(1, len(imgs), figsize=figsize_in_inches)
    for col, img in enumerate(imgs):
        axs[col].imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
    plt.show()
```

With the following block, we load the correct img paths to the used image sets:

In [2]:
```
from pathlib import Path
def get_image_paths(img_set):
    return [str(path.relative_to('.')) for path in Path('imgs').rglob(f'{img_set}*')]

weir_imgs = get_image_paths('weir')
budapest_imgs = get_image_paths('buda')
exposure_error_imgs = get_image_paths('exp')
barcode_imgs = get_image_paths('barc')
barcode_masks = get_image_paths('mask')
```

Resize Images

The first step is to resize the images to medium (and later to low) resolution. The class which can be used is the `Images` class. If the images should not be stitched on full resolution, this can be achieved by setting the `final_megapix` parameter to a number above 0.

```
Images.of(images, medium_megapix=0.6, low_megapix=0.1, final_megapix=-1)
```

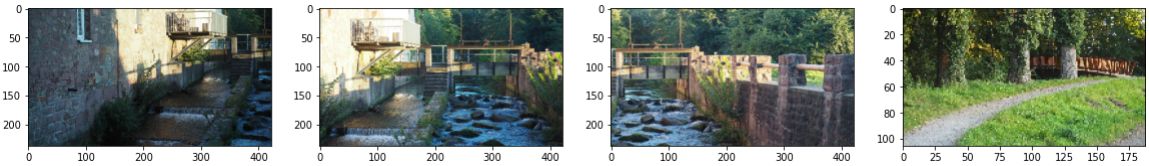
In [3]:
```
from stitching.images import Images

images = Images.of(weir_imgs)

medium_imgs = list(images.resize(Images.Resolution.MEDIUM))
low_imgs = list(images.resize(Images.Resolution.LOW))
final_imgs = list(images.resize(Images.Resolution.FINAL))
```

NOTE: Everytime `list()` is called in this notebook means that the function returns a generator (generators improve the overall stitching performance). To get all elements at once we use `list(generator_object)`

In [4]:
```
plot_images(low_imgs, (20,20))
```



In [5]:
```
original_size = images.sizes[0]
medium_size = images.get_image_size(medium_imgs[0])
low_size = images.get_image_size(low_imgs[0])
final_size = images.get_image_size(final_imgs[0])

print(f"Original Size: {original_size}  -> {'{:,}'.format(np.prod(original_size))} px ~ 1 MP")
print(f"Medium Size:   {medium_size}  -> {'{:,}'.format(np.prod(medium_size))} px ~ 0.6 MP")
print(f"Low Size:      {low_size}   -> {'{:,}'.format(np.prod(low_size))} px ~ 0.1 MP")
print(f"Final Size:    {final_size}  -> {'{:,}'.format(np.prod(final_size))} px ~ 1 MP")
```

```
Original Size: (1333, 750)  -> 999,750 px ~ 1 MP
Medium Size:   (1033, 581)  -> 600,173 px ~ 0.6 MP
Low Size:      (422, 237)   -> 100,014 px ~ 0.1 MP
Final Size:    (1333, 750)  -> 999,750 px ~ 1 MP
```

Find Features

On the medium images, we now want to find features that can describe conspicuous elements within the images which might be found in other images as well. The class which can be used is the `FeatureDetector` class.

`FeatureDetector(detector='orb', nfeatures=500)`

In [6]:
```
from stitching.feature_detector import FeatureDetector

finder = FeatureDetector()
features = [finder.detect_features(img) for img in medium_imgs]
keypoints_center_img = finder.draw_keypoints(medium_imgs[1], features[1])
```

In [7]:
```
plot_image(keypoints_center_img, (15,10))
```

Match Features

Now we can match the features of the pairwise images. The class which can be used is the FeatureMatcher class.

`FeatureMatcher(matcher_type='homography', range_width=-1)`

In [8]:
```
from stitching.feature_matcher import FeatureMatcher

matcher = FeatureMatcher()
matches = matcher.match_features(features)
```

We can look at the confidences, which are calculated by:

`confidence = number of inliers / (8 + 0.3 * number of matches)` (Lines 435-7 of this file (https://github.com/opencv/opencv/blob/68d15fc62edad980f1ffa15ee478438335f39cc3/modules/stitching/src/matchers.cpp))

The inliers are calculated using the random sample consensus (RANSAC) method, e.g. in this file (https://github.com/opencv/opencv/blob/68d15fc62edad980f1ffa15ee478438335f39cc3/modules/stitching/src/matchers.cpp) in Line 425. We can plot the inliers which is shown later.

In [9]:
```
matcher.get_confidence_matrix(matches)
```

Out[9]:
```
array([[0.        , 2.45009074, 0.56      , 0.44247788],
       [2.45009074, 0.        , 2.01729107, 0.42016807],
       [0.56      , 2.01729107, 0.        , 0.38709677],
       [0.44247788, 0.42016807, 0.38709677, 0.        ]])
```

It can be seen that:

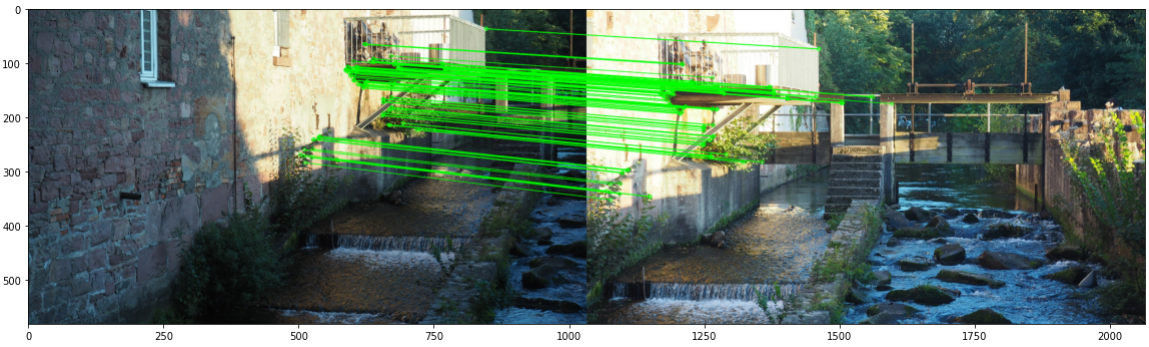
- image 1 has a high matching confidence with image 2 and low confidences with image 3 and 4
- image 2 has a high matching confidence with image 1 and image 3 and low confidences with image 4
- image 3 has a high matching confidence with image 2 and low confidences with image 1 and 4
- image 4 has low matching confidences with image 1, 2 and 3

With a `confidence_threshold`, which is introduced in detail in the next step, we can plot the relevant matches with the inliers:

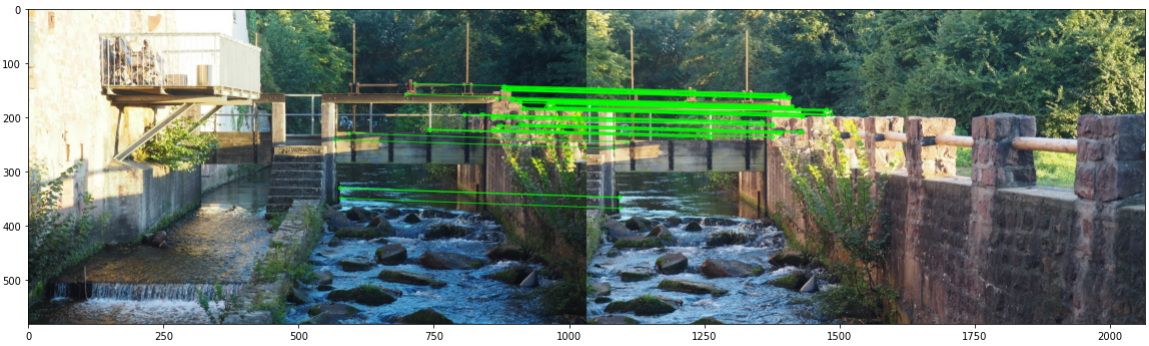
In [10]:
```
all_relevant_matches = matcher.draw_matches_matrix(medium_imgs, features, matches, conf_thresh=1,
                                                   inliers=True, matchColor=(0, 255, 0))

for idx1, idx2, img in all_relevant_matches:
    print(f"Matches Image {idx1+1} to Image {idx2+1}")
    plot_image(img, (20,10))
```

```
Matches Image 1 to Image 2
```




```
Matches Image 2 to Image 3
```



Subset

Above we saw that the noise image has no connection to the other images which are part of the panorama. We now want to create a subset with only the relevant images. The class which can be used is the `Subsetter` class. We can specify the `confidence_threshold` from when a match is regarded as good match. We saw that in our case `1` is sufficient. For the parameter `matches_graph_dot_file` a file name can be passed, in which a matches graph in dot notation is saved.

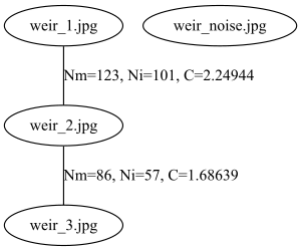
```
Subsetter(confidence_threshold=1, matches_graph_dot_file=None)
```

In [11]:
```
from stitching.subsetter import Subsetter

subsetter = Subsetter()
dot_notation = subsetter.get_matches_graph(images.names, matches)
print(dot_notation)
```

```
graph matches_graph{
"weir_1.jpg" -- "weir_2.jpg"[label="Nm=157, Ni=135, C=2.45009"];
"weir_2.jpg" -- "weir_3.jpg"[label="Nm=89, Ni=70, C=2.01729"];
"weir_noise.jpg";
}
```

The matches graph visualizes what we've saw in the confidence matrix: image 1 conneced to image 2 conneced to image 3. Image 4 is not part of the panorama (note that the confidences can vary since this is a static image).



GraphvizOnline (https://dreampuf.github.io/GraphvizOnline) is used to plot the graph

We now want to subset all variables we've created till here, incl. the attributes `img_names` and `img_sizes` of the `ImageHandler`

In [12]:
```
indices = subsetter.get_indices_to_keep(features, matches)

medium_imgs = subsetter.subset_list(medium_imgs, indices)
low_imgs = subsetter.subset_list(low_imgs, indices)
final_imgs = subsetter.subset_list(final_imgs, indices)
features = subsetter.subset_list(features, indices)
matches = subsetter.subset_matches(matches, indices)

images.subset(indices)

print(images.names)
print(matcher.get_confidence_matrix(matches))
```

```
['imgs\\weir_1.jpg', 'imgs\\weir_2.jpg', 'imgs\\weir_3.jpg']
[[0.         2.45009074 0.56      ]
 [2.45009074 0.         2.01729107]
 [0.56       2.01729107 0.        ]]
```

Camera Estimation, Adjustion and Correction

With the features and matches we now want to calibrate cameras which can be used to warp the images so they can be composed correctly. The classes which can be used are `CameraEstimator`, `CameraAdjuster` and `WaveCorrector`:

```
CameraEstimator(estimator='homography')
CameraAdjuster(adjuster='ray', refinement_mask='xxxxx')
WaveCorrector(wave_correct_kind='horiz')
```

In [13]:
```
from stitching.camera_estimator import CameraEstimator
from stitching.camera_adjuster import CameraAdjuster
from stitching.camera_wave_corrector import WaveCorrector

camera_estimator = CameraEstimator()
camera_adjuster = CameraAdjuster()
wave_corrector = WaveCorrector()

cameras = camera_estimator.estimate(features, matches)
cameras = camera_adjuster.adjust(features, matches, cameras)
cameras = wave_corrector.correct(cameras)
```

Warp Images

With the obtained cameras we now want to warp the images itself into the final plane. The class which can be used is the `Warper` class:

```
Warper(warper_type='spherical', scale=1)
```

In [14]:
```
from stitching.warper import Warper

warper = Warper()
```

At first, we set the the medium focal length of the cameras as scale:

In [15]:
```
warper.set_scale(cameras)
```

Warp low resolution images

In [16]:
```
low_sizes = images.get_scaled_img_sizes(Images.Resolution.LOW)
camera_aspect = images.get_ratio(Images.Resolution.MEDIUM, Images.Resolution.LOW)  # since cameras were obtained on medium imgs

warped_low_imgs = list(warper.warp_images(low_imgs, cameras, camera_aspect))
warped_low_masks = list(warper.create_and_warp_masks(low_sizes, cameras, camera_aspect))
low_corners, low_sizes = warper.warp_rois(low_sizes, cameras, camera_aspect)
```

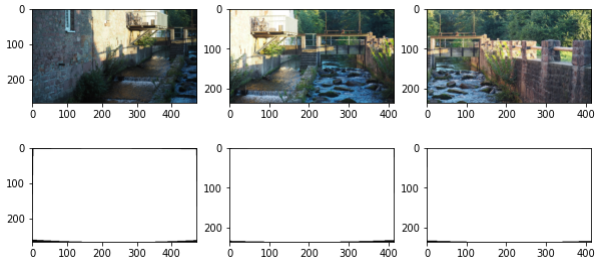
Warp final resolution images

In [17]:
```
final_sizes = images.get_scaled_img_sizes(Images.Resolution.FINAL)
camera_aspect = images.get_ratio(Images.Resolution.MEDIUM, Images.Resolution.FINAL)

warped_final_imgs = list(warper.warp_images(final_imgs, cameras, camera_aspect))
warped_final_masks = list(warper.create_and_warp_masks(final_sizes, cameras, camera_aspect))
final_corners, final_sizes = warper.warp_rois(final_sizes, cameras, camera_aspect)
```

We can plot the results. Not much scaling and rotating is needed to align the images. Thus, the images are only slightly adjusted in this example

In [18]:
```
plot_images(warped_low_imgs, (10,10))
plot_images(warped_low_masks, (10,10))
```



With the warped corners and sizes we know where the images will be placed on the final plane:

In [19]:
```
print(final_corners)
print(final_sizes)
```

```
[(-1362, 4152), (-677, 4114), (-21, 4094)]
[(1496, 847), (1310, 744), (1312, 746)]
```

Excursion: Timelapser

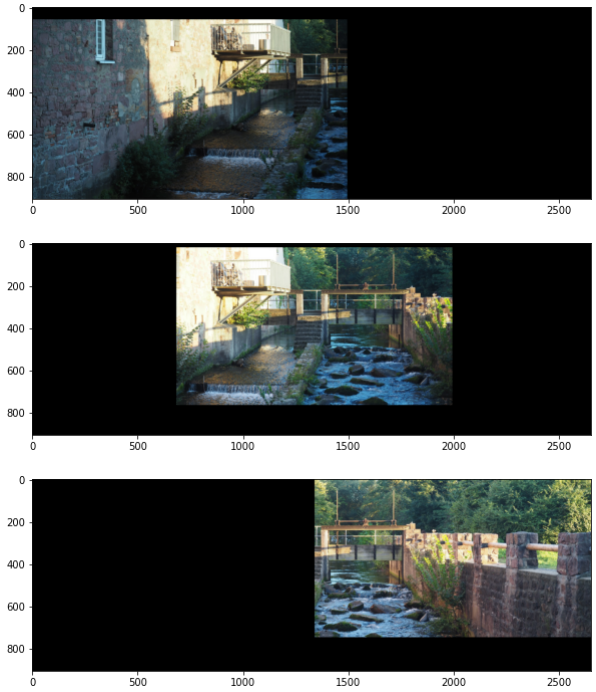
The Timelapser functionality is a nice way to grasp how the images are warped into a final plane. The class which can be used is the `Timelapser` class:

```
Timelapser(timelapse='no')
```

In [20]:
```
from stitching.timelapser import Timelapser

timelapser = Timelapser('as_is')
timelapser.initialize(final_corners, final_sizes)

for img, corner in zip(warped_final_imgs, final_corners):
    timelapser.process_frame(img, corner)
    frame = timelapser.get_frame()
    plot_image(frame, (10,10))
```



Crop

We can see that none of the images have the full height of the final plane. To get a panorama without black borders we can now estimate the largest joint interior rectangle and crop the single images accordingly. The class which can be used is the `Cropper` class:

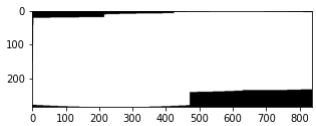
```
Cropper(crop=True)
```

In [21]:
```
from stitching.cropper import Cropper

cropper = Cropper()
```

We can estimate a panorama mask of the potential final panorama (using a Blender which will be introduced later)

In [22]:
```
mask = cropper.estimate_panorama_mask(warped_low_imgs, warped_low_masks, low_corners, low_sizes)
plot_image(mask, (5,5))
mask.shape
```



Out[22]:
```
(285, 838)
```

The estimation of the largest interior rectangle is not yet implemented in OpenCV, but a Numba (https://numba.pydata.org/) Implementation by my own. You check out the details here (https://github.com/lukasalexanderweber/lir). Compiling the Code takes a bit (only once, the compiled code is then cached (https://numba.pydata.org/numba-doc/latest/developer/caching.html) for future function calls)

In [23]:
```
lir = cropper.estimate_largest_interior_rectangle(mask)
```

```
---------------------------------------------------------------------------
TypingError                               Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_57272/2745942039.py in <module>
----> 1 lir = cropper.estimate_largest_interior_rectangle(mask)

~\anaconda3\lib\site-packages\stitching\cropper.py in estimate_largest_interior_rectangle(self, mask)
     91         # largestinteriorrectangle is only imported if cropping
     92         # is explicitly desired (needs some time to compile at the first run!)
---> 93         import largestinteriorrectangle
     94
     95         contours, hierarchy = cv.findContours(mask, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

~\anaconda3\lib\site-packages\largestinteriorrectangle\__init__.py in <module>
----> 1 from .lir import lir, pt1, pt2
      2
      3 __version__ = "0.2.0"

~\anaconda3\lib\site-packages\largestinteriorrectangle\lir.py in <module>
      1 from .lir_basis import largest_interior_rectangle as lir_basis
----> 2 from .lir_within_contour import largest_interior_rectangle \
      3     as lir_within_contour
      4 from .lir_within_polygon import largest_interior_rectangle \
      5     as lir_within_polygon

~\anaconda3\lib\site-packages\largestinteriorrectangle\lir_within_contour.py in <module>
     61
     62
---> 63 @nb.njit('uint32[:](uint32[:,::1], uint32, uint32)', cache=True)
     64 def h_vector_bottom2top(h_adjacency, x, y):
     65     vector_size = predict_vector_size(np.flip(h_adjacency[:y+1, x]))

~\anaconda3\lib\site-packages\numba\decorators.py in wrapper(func)
    198             with typeinfer.register_dispatcher(disp):
    199                 for sig in sigs:
--> 200                     disp.compile(sig)
    201                 disp.disable_compile()
    202         return disp

~\anaconda3\lib\site-packages\numba\compiler_lock.py in _acquire_compile_lock(*args, **kwargs)
     30         def _acquire_compile_lock(*args, **kwargs):
     31             with self:
---> 32                 return func(*args, **kwargs)
     33         return _acquire_compile_lock
     34

~\anaconda3\lib\site-packages\numba\dispatcher.py in compile(self, sig)
    766             self._cache_misses[sig] += 1
    767             try:
--> 768                 cres = self._compiler.compile(args, return_type)
    769             except errors.ForceLiteralArg as e:
    770                 def folded(args, kws):

~\anaconda3\lib\site-packages\numba\dispatcher.py in compile(self, args, return_type)
     79             return retval
     80         else:
---> 81             raise retval
     82
     83     def _compile_cached(self, args, return_type):

~\anaconda3\lib\site-packages\numba\dispatcher.py in _compile_cached(self, args, return_type)
     89
     90         try:
---> 91             retval = self._compile_core(args, return_type)
     92         except errors.TypingError as e:
     93             self._failed_cache[key] = e

~\anaconda3\lib\site-packages\numba\dispatcher.py in _compile_core(self, args, return_type)
    107                                       args=args, return_type=return_type,
    108                                       flags=flags, locals=self.locals,
--> 109                                       pipeline_class=self.pipeline_class)
    110         # Check typing error if object mode is used
    111         if cres.typing_error is not None and not flags.enable_pyobject:

~\anaconda3\lib\site-packages\numba\compiler.py in compile_extra(typingctx, targetctx, func, args, return_type, flags, locals, library, pipeline_class)
    549     pipeline = pipeline_class(typingctx, targetctx, library,
    550                               args, return_type, flags, locals)
--> 551     return pipeline.compile_extra(func)
    552
    553

~\anaconda3\lib\site-packages\numba\compiler.py in compile_extra(self, func)
    329         self.state.lifted = ()
    330         self.state.lifted_from = None
--> 331         return self._compile_bytecode()
    332
    333     def compile_ir(self, func_ir, lifted=(), lifted_from=None):

~\anaconda3\lib\site-packages\numba\compiler.py in _compile_bytecode(self)
    391         """
    392         assert self.state.func_ir is None
--> 393         return self._compile_core()
    394
    395     def _compile_ir(self):

~\anaconda3\lib\site-packages\numba\compiler.py in _compile_core(self)
    371                 self.state.status.fail_reason = e
    372                 if is_final_pipeline:
--> 373                     raise e
    374         else:
    375             raise CompilerError("All available pipelines exhausted")

~\anaconda3\lib\site-packages\numba\compiler.py in _compile_core(self)
    362             res = None
    363             try:
--> 364                 pm.run(self.state)
    365                 if self.state.cr is not None:
    366                     break

~\anaconda3\lib\site-packages\numba\compiler_machinery.py in run(self, state)
    345                     (self.pipeline_name, pass_desc)
    346                 patched_exception = self._patch_error(msg, e)
--> 347                 raise patched_exception
    348
    349     def dependency_analysis(self):

~\anaconda3\lib\site-packages\numba\compiler_machinery.py in run(self, state)
    336                 pass_inst = _pass_registry.get(pss).pass_inst
    337                 if isinstance(pass_inst, CompilerPass):
--> 338                     self._runPass(idx, pass_inst, state)
    339                 else:
    340                     raise BaseException("Legacy pass in use")
```

```
~\anaconda3\lib\site-packages\numba\compiler_lock.py in _acquire_compile_lock(*args, **kwargs)
     30         def _acquire_compile_lock(*args, **kwargs):
     31             with self:
---> 32                 return func(*args, **kwargs)
     33         return _acquire_compile_lock
     34

~\anaconda3\lib\site-packages\numba\compiler_machinery.py in _runPass(self, index, pss, internal_state)
    300             mutated |= check(pss.run_initialization, internal_state)
    301         with SimpleTimer() as pass_time:
--> 302             mutated |= check(pss.run_pass, internal_state)
    303         with SimpleTimer() as finalize_time:
    304             mutated |= check(pss.run_finalizer, internal_state)

~\anaconda3\lib\site-packages\numba\compiler_machinery.py in check(func, compiler_state)
    273
    274         def check(func, compiler_state):
--> 275             mangled = func(compiler_state)
    276             if mangled not in (True, False):
    277                 msg = ("CompilerPass implementations should return True/False. "

~\anaconda3\lib\site-packages\numba\typed_passes.py in run_pass(self, state)
     93                 state.return_type,
     94                 state.locals,
---> 95                 raise_errors=self._raise_errors)
     96             state.typemap = typemap
     97             if self._raise_errors:

~\anaconda3\lib\site-packages\numba\typed_passes.py in type_inference_stage(typingctx, interp, args, return_type, locals, raise_errors)
     65
     66         infer.build_constraint()
---> 67         infer.propagate(raise_errors=raise_errors)
     68         typemap, restype, calltypes = infer.unify(raise_errors=raise_errors)
     69

~\anaconda3\lib\site-packages\numba\typeinfer.py in propagate(self, raise_errors)
    983                                     if isinstance(e, ForceLiteralArg)]
    984                 if not force_lit_args:
--> 985                     raise errors[0]
    986                 else:
    987                     raise reduce(operator.or_, force_lit_args)

TypingError: Failed in nopython mode pipeline (step: nopython frontend)
Use of unsupported NumPy function 'numpy.flip' or unsupported use of the function.

File "C:\Users\user\anaconda3\lib\site-packages\largestinteriorrectangle\lir_within_contour.py", line 65:
def h_vector_bottom2top(h_adjacency, x, y):
    vector_size = predict_vector_size(np.flip(h_adjacency[:y+1, x]))
    ^

[1] During: typing of get attribute at C:\Users\user\anaconda3\lib\site-packages\largestinteriorrectangle\lir_within_contour.py (65)

File "C:\Users\user\anaconda3\lib\site-packages\largestinteriorrectangle\lir_within_contour.py", line 65:
def h_vector_bottom2top(h_adjacency, x, y):
    vector_size = predict_vector_size(np.flip(h_adjacency[:y+1, x]))
    ^
```

After compilation the estimation is really fast:

In [ ]:
```
lir = cropper.estimate_largest_interior_rectangle(mask)
print(lir)
```

In [ ]:
```
plot = lir.draw_on(mask, size=2)
plot_image(plot, (5,5))
```

By zero centering the the warped corners, the rectangle of the images within the final plane can be determined. Here the center image is shown:

In [ ]:
```
low_corners = cropper.get_zero_center_corners(low_corners)
rectangles = cropper.get_rectangles(low_corners, low_sizes)

plot = rectangles[1].draw_on(plot, (0, 255, 0), 2)  # The rectangle of the center img
plot_image(plot, (5,5))
```

Using the overlap new corners and sizes can be determined:

In [ ]:
```
overlap = cropper.get_overlap(rectangles[1], lir)
plot = overlap.draw_on(plot, (255, 0, 0), 2)
plot_image(plot, (5,5))
```

With the blue Rectangle in the coordinate system of the original image (green) we are able to crop it

In [ ]:
```
intersection = cropper.get_intersection(rectangles[1], overlap)
plot = intersection.draw_on(warped_low_masks[1], (255, 0, 0), 2)
plot_image(plot, (2.5,2.5))
```

Using all this information we can crop the images and masks and obtain new corners and sizes

In [ ]:
```
cropper.prepare(warped_low_imgs, warped_low_masks, low_corners, low_sizes)

cropped_low_masks = list(cropper.crop_images(warped_low_masks))
cropped_low_imgs = list(cropper.crop_images(warped_low_imgs))
low_corners, low_sizes = cropper.crop_rois(low_corners, low_sizes)

lir_aspect = images.get_ratio(Images.Resolution.LOW, Images.Resolution.FINAL)  # since lir was obtained on low imgs
cropped_final_masks = list(cropper.crop_images(warped_final_masks, lir_aspect))
cropped_final_imgs = list(cropper.crop_images(warped_final_imgs, lir_aspect))
final_corners, final_sizes = cropper.crop_rois(final_corners, final_sizes, lir_aspect)
```

Redo the timelapse with cropped Images:

In [ ]:
```
timelapser = Timelapser('as_is')
timelapser.initialize(final_corners, final_sizes)

for img, corner in zip(cropped_final_imgs, final_corners):
    timelapser.process_frame(img, corner)
    frame = timelapser.get_frame()
    plot_image(frame, (10,10))
```

Now we need stategies how to compose the already overlaying images into one panorama image. Strategies are:

- Seam Masks

- Exposure Error Compensation
- Blending

Seam Masks

Seam masks find a transition line between images with the least amount of interference. The class which can be used is the `SeamFinder` class:

```
SeamFinder(finder='dp_color')
```

The Seams are obtained on the warped low resolution images and then resized to the warped final resolution images. The Seam Masks can be used in the Blending step to specify how the images should be composed.

In [ ]:
```
from stitching.seam_finder import SeamFinder

seam_finder = SeamFinder()

seam_masks = seam_finder.find(cropped_low_imgs, low_corners, cropped_low_masks)
seam_masks = [seam_finder.resize(seam_mask, mask) for seam_mask, mask in zip(seam_masks, cropped_final_masks)]

seam_masks_plots = [SeamFinder.draw_seam_mask(img, seam_mask) for img, seam_mask in zip(cropped_final_imgs, seam_masks)]
plot_images(seam_masks_plots, (15,10))
```

Exposure Error Compensation

Frequently exposure errors respectively exposure differences between images occur which lead to artefacts in the final panorama. The class which can be used is the `ExposureErrorCompensator` class:

```
ExposureErrorCompensator(compensator='gain_blocks', nr_feeds=1, block_size=32)
```

The Exposure Error are estimated on the warped low resolution images and then applied on the warped final resolution images.

Note: In this example the compensation has nearly no effect and the result is not shown. To understand the stitching pipeline they are compensated anyway. A fitting example for images where Exposure Error Compensation is important can be found at the end of the notebook.

In [ ]:
```
from stitching.exposure_error_compensator import ExposureErrorCompensator

compensator = ExposureErrorCompensator()

compensator.feed(low_corners, cropped_low_imgs, cropped_low_masks)

compensated_imgs = [compensator.apply(idx, corner, img, mask)
                    for idx, (img, mask, corner)
                    in enumerate(zip(cropped_final_imgs, cropped_final_masks, final_corners))]
```

Blending

With all the previous processing the images can finally be blended to a whole panorama. The class which can be used is the `Blender` class:

```
Blender(blender_type='multiband', blend_strength=5)
```

The blend strength thereby specifies on which overlap the images should be overlayed along the transitions of the masks. This is also visualized at the end of the notebook.

In [ ]:
```
from stitching.blender import Blender

blender = Blender()
blender.prepare(final_corners, final_sizes)
for img, mask, corner in zip(compensated_imgs, seam_masks, final_corners):
    blender.feed(img, mask, corner)
panorama, _ = blender.blend()
```

In [ ]:
```
plot_image(panorama, (20,20))
```

There is the functionality to plot the seams as lines or polygons onto the final panorama to see which part of the panorama is from which image. The basis is to blend single colored dummy images with the obtained seam masks and panorama dimensions:

In [ ]:
```
blended_seam_masks = seam_finder.blend_seam_masks(seam_masks, final_corners, final_sizes)
plot_image(blended_seam_masks, (5,5))
```

This blend can be converted into lines or weighted on top of the resulting panorama:

In [ ]:
```
plot_image(seam_finder.draw_seam_lines(panorama, blended_seam_masks, linesize=3), (15,10))
plot_image(seam_finder.draw_seam_polygons(panorama, blended_seam_masks), (15,10))
```

Stitcher

All the functionality above is automated within the `Stitcher` class:

```
Stitcher(**kwargs)
```

In [ ]:
```
from stitching import Stitcher

stitcher = Stitcher()
panorama = stitcher.stitch(weir_imgs)  # the user is warned that only a subset of input images is stitched
```

In [ ]:
```
plot_image(panorama, (20,20))
```

Affine Stitcher

For images that were obtained on e.g. a flatbed scanner affine transformations are sufficient. The `AffineStitcher` convenience class sets the needed parameters as default:

```
AffineStitcher(**kwargs)
```

In [ ]:
```
from stitching import AffineStitcher

print(AffineStitcher.AFFINE_DEFAULTS)
# Comparison:
# print(Stitcher.DEFAULT_SETTINGS)
# print(AffineStitcher.DEFAULT_SETTINGS)
```

We can now process the Budapest example (with two additional parameters)

In [ ]:
```
settings = {# The whole plan should be considered
            "crop": False,
            # The matches confidences aren't that good
            "confidence_threshold": 0.5}

stitcher = AffineStitcher(**settings)
panorama = stitcher.stitch(budapest_imgs)

plot_image(panorama, (20,20))
```

Exposure Error and Blend Strenght Example

In [ ]:
```
imgs = exposure_error_imgs

stitcher = Stitcher(compensator="no", blender_type="no")
panorama1 = stitcher.stitch(imgs)

stitcher = Stitcher(compensator="no")
panorama2 = stitcher.stitch(imgs)

stitcher = Stitcher(compensator="no", blend_strength=20)
panorama3 = stitcher.stitch(imgs)

stitcher = Stitcher(blender_type="no")
panorama4 = stitcher.stitch(imgs)

stitcher = Stitcher(blend_strength=20)
panorama5 = stitcher.stitch(imgs)
```

In [ ]:
```
fig, axs = plt.subplots(3, 2, figsize=(20,20))
axs[0, 0].imshow(cv.cvtColor(panorama1, cv.COLOR_BGR2RGB))
axs[0, 0].set_title('Along Seam Masks with Exposure Errors')
axs[0, 1].axis('off')
axs[1, 0].imshow(cv.cvtColor(panorama2, cv.COLOR_BGR2RGB))
axs[1, 0].set_title('Blended with the default blend strenght of 5')
axs[1, 1].imshow(cv.cvtColor(panorama3, cv.COLOR_BGR2RGB))
axs[1, 1].set_title('Blended with a bigger blend strenght of 20')
axs[2, 0].imshow(cv.cvtColor(panorama4, cv.COLOR_BGR2RGB))
axs[2, 0].set_title('Along Seam Masks with Exposure Error Compensation')
axs[2, 1].imshow(cv.cvtColor(panorama5, cv.COLOR_BGR2RGB))
axs[2, 1].set_title('Blended with Exposure Compensation and bigger blend strenght of 20')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

Feature Masking

Sometimes specific parts of your images disrupt the stitching workflow or you only want to match at specific parts of your images. See e.g. how the QR-Code manipulates the stitching:

In [ ]:
```
stitcher = Stitcher(crop=False)
panorama = stitcher.stitch(barcode_imgs)

plot_image(panorama, (20,20))
```

With the feature masks you can exclude the QR-Code areas of your images from feature detection and matching:

In [ ]:
```
stitcher = Stitcher()
panorama = stitcher.stitch(barcode_imgs, barcode_masks)

plot_image(panorama, (20,20))
```

See the Pull Request (https://github.com/OpenStitching/stitching/pull/130) for more detailed intermediate images

In [ ]: