valis

Release "1.0.4"

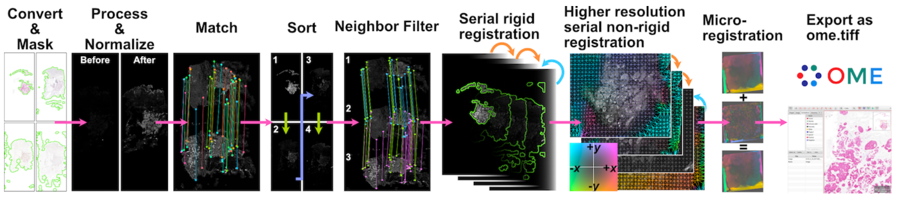
Chandler Gatenbee

Feb 02, 2024

CONTENTS

VALIS, which stands for Virtual Alignment of pathoLogy Image Series, is a fully automated pipeline to register whole slide images (WSI) using rigid and/or non-rigid transformtions. A full description of the method is described in the paper by Gatenbee et al. 2023. VALIS uses Bio-Formats, OpenSlide, libvips, and scikit-image to read images and slides, and so is able to work with over 300 image formats. Registered images can be saved as ome.tiff slides that can be used in downstream analyses. ome.tiff format is opensource and widely supported, being readable in several different programming languages (Python, Java, Matlab, etc…) and software, such as QuPath, HALO by Idica Labs, etc…

The registration pipeline is fully automated and goes as follows:



1. Images/slides are converted to numpy arrays. As WSI are often too large to fit into memory, these images are usually lower resolution images from different pyramid levels.

2. Images are processed to single channel images. They are then normalized to make them look as similar as possible. Masks are then created to focus registration on the tissue.

3. Image features are detected and then matched between all pairs of image.

4. If the order of images is unknown, they will be optimally ordered based on their feature similarity. This increases the chances of successful registration because each image will be aligned to one that looks very similar.

5. Images will be aligned *towards* (not to) a reference image. If the reference image is not specified, it will automatically be set to the image at the center of the stack.

6. Rigid registration is performed serially, with each image being rigidly aligned towards the reference image. That is, if the reference image is the 5th in the stack, image 4 will be aligned to 5 (the reference), and then 3 will be aligned to the now registered version of 4, and so on. Only features found in both neighboring slides are used to align the image to the next one in the stack. VALIS uses feature detection to match and align images, but one can optionally perform a final step that maximizes the mutual information between each pair of images. This rigid registration can optionally be updated by matching features in higher resolution versions of the images (see `micro_rigid_registrar.MicroRigidRegistrar`).

7. The registered rigid masks are combined to create a non-rigid registration mask. The bounding box of this mask is then used to extract higher resolution versions of the tissue from each slide. These higher resolution images are then processed as above and used for non-rigid registration, which is performed either by:

    - aligning each image towards the reference image following the same sequence used during rigid registration.
    - using groupwise registration that non-rigidly aligns the images to a common frame of reference. Currently this is only possible if SimpleElastix is installed.

8. One can optionally perform a second non-rigid registration using an even higher resolution versions of each image. This is intended to better align micro-features not visible in the original images, and so is referred to as micro-registration. A mask can also be used to indicate where registration should take place.

9. Error is estimated by calculating the distance between registered matched features in the full resolution images.

The transformations found by VALIS can then be used to warp the full resolution slides. It is also possible to merge non-RGB registered slides to create a highly multiplexed image. These aligned and/or merged slides can then be saved as ome.tiff images. The transformations can also be use to warp point data, such as cell centroids, polygon vertices, etc…

In addition to registering images, VALIS provides tools to read slides using Bio-Formats and OpenSlide, which can be read at multiple resolutions and converted to numpy arrays or pyvips.Image objects. One can also slice regions of interest from these slides and warp annotated images. VALIS also provides functions to convert slides to the ome.tiff format, preserving the original metadata. Please see examples and documentation for more details.

Full documentation with installation instructions and examples can be found at ReadTheDocs.

CHAPTER ONE

LICENSE

1.1 Docmentation

1.1.1 Installation

Note: Currently, VALIS requires Python 3.9 or 3.10

DockerHub

VALIS is available as a Docker image and can be downloaded from DockerHub. Starting a container will launch an Ubuntu shell, and so Python needs to be called when executing the script. In this example, the user has a file called "register.py" that takes `src_dir` and `dst_dir` arguments, which registers all of the images in `src_dir` and saves the results in `dst_dir`. This example bind mounts the home directory, and thus the full paths need to be specified.

```
$ docker run --memory=20g  -v "$HOME:$HOME" cdgatenbee/valis-wsi python3 full/path/to/
↪register.py -src_dir full/path/to/images_to_align -dst_dir full/path/to/where_to_save_
↪results
```

Important: To avoid the container from shutting down prematurely, be sure to set appropriately high memory limits (including in Docker Desktop).

Pip install

VALIS can be downloaded from PyPI as the valis-wsi package using the pip command. However, also VALIS requires several system level packages, which will need to be installed first (see *Prerequisites* below).

```
$ pip install valis-wsi
```

One can also use pip to install directly from Github

```
$ pip install git+https://github.com/MathOnco/valis.git
```

Prerequisites

VALIS uses Bioformats to read many slide formats. Bioformats is written in Java, and VALIS uses the Python package jpype to access the Bioformats jar. Therefore, the user will need to have installed a Java Development Kit (JDK) containing the Java Runtime Environment (JRE):

1. Download appropriate JDK from java downloads
2. Edit your system and environment variables to update the Java home

   ```
   $ export JAVA_HOME=/usr/libexec/java_home
   ```

3. Verify the path has been added:

   ```
   $ echo $JAVA_HOME
   ```

   should print something like `usr/libexec/java_home`

4. Install *Maven <https://maven.apache.org/index.html>*_, which is also required to use Bioformats
5. (optional) If you will be working with files that have extensions: '.vmu', '.mrxs' '.svslide', you will also need to install OpenSlide. Note that this is not the same as openslide-python, which contains Python wrappers for OpenSlide.

   Important: OpenSlide requires pixman, which must be version 0.40.0. If pixman is a different version, then the slides may be distorted when reading from any pyramid level other than 0.

6. VALIS uses pyvips to warp and save the whole slide images (WSI) as ome.tiffs. Pyvips requires libvips (not a Python package) to be on your library search path, and so libvips must be installed separately. See the pyvips installation notes for instructions on how to do this for your operating system. If you already have libvips installed, please make sure it's version is >= 8.11.

Install

Once the above prerequisites have been satistifed, valis can be installed using pip, idealy within a virtual environment

```
$ python3 -m venv venv_valis
$ source ./venv_valis/bin/activate
$ python3 -m pip install --upgrade pip
$ python3 pip install valis-wsi
```

SimpleElastix (optional)

The defaults used by VALIS work well, but VALIS also provides optional classes that require SimpleElastix. In particular, these classes are:

1. affine_optimizer.AffineOptimizerMattesMI, which uses sitk.ElastixImageFilter to simultaneously maximize Mattes Mutual Information and minimize the spatial distance between matched features.
2. non_rigid_registrars.SimpleElastixWarper, which uses sitk.ElastixImageFilter to find non-rigid transformations between pairs of images.
3. non_rigid_registrars.SimpleElastixGroupwiseWarper, which uses sitk.ElastixImageFilter to find non-rigid transformations using groupwise registration.

To install SimpleElastix, you should probably uninstall the current version of SimpleITK in your environment, and then install SimpleElastix as described in the SimpleElastix docs.

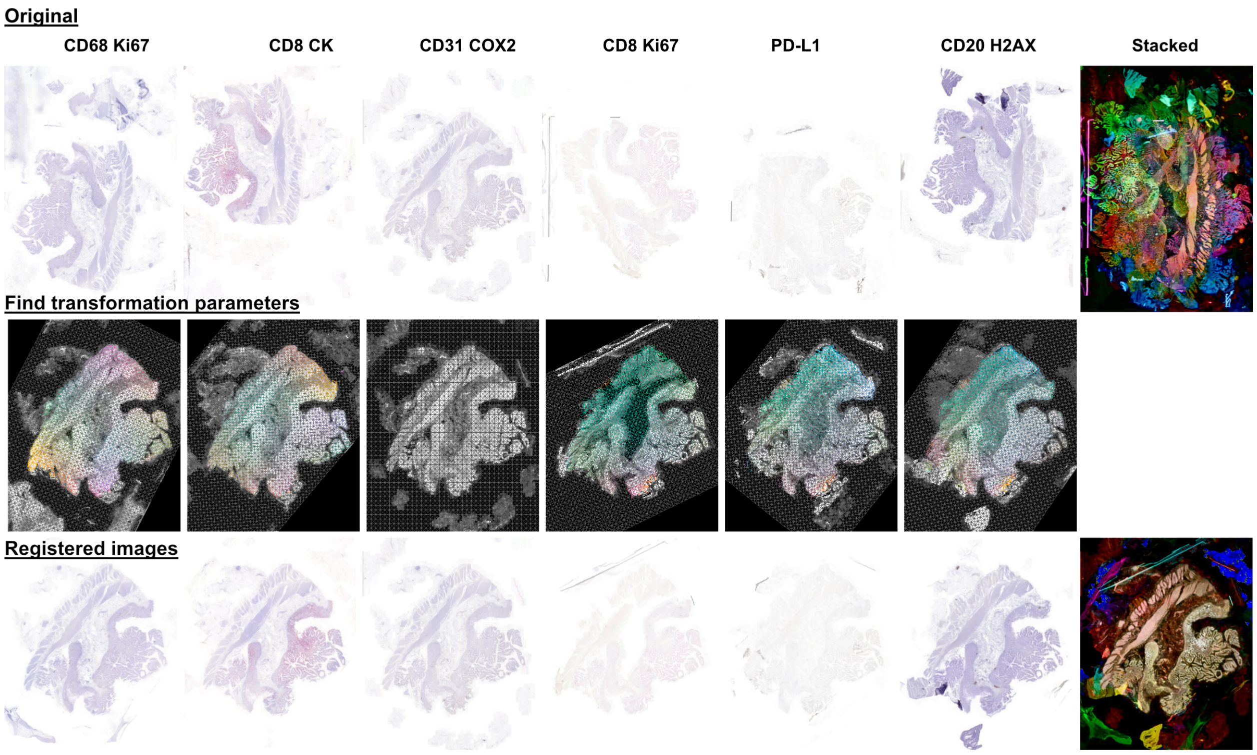
From source

One will need to install and use Poetry to install VALIS from the source code. As Poetry only installs the Python dependencies, one will also need to follow the steps above to install the JDK, Maven, libvips, and openslide. Note that the poetry lock file is included in the repository, which can be deleted before installation if so desired.

1.1.2 Examples

Important: Always be sure to always kill the JVM at the end of your script. Not doing so can prevent the software from closing. This can be accomplished by calling either `registration.kill_jvm()` or `slide_io.kill_jvm()`

Slide registration



Important: One of the most important parameters used to initialize a Valis object is `max_processed_image_dim_px`, which determines the size of the image used to find the rigid registration parameters. The default value is 850, but if registration fails or is poor, try adjusting that value. Generally speaking, values between 500-2000 work well. In cases where there is little empty space, around the tissue, smaller values may be better. However, if there is a large amount of empty space/slide (as in the images above), larger values may be needed so that the tissue is at a high enough resolution. To imporove alingment of the finer details in the images, larger images can be used in the non-rigid or micro-registration steps (set via the max_non_rigid_registration_dim_px parameter).

Important: If the order of slices is known and needs to be preserved, such as building a 3D image, set imgs_ordered=True when initializing the VALIS object. Otherwise, VALIS will sort the images based on similarity, which may or may not correspond on the sliced order. If using this option, ensure that the names of the files allow them to be sorted properly, e.g. 01.tiff, 02.tiff … 10.tiff, etc…

In this example, the slides that need to be registered are located in /path/to/slides. This process involves creating a Valis object, which is what conducts the registration. In this example no reference image is specified, and so all images will be aligned towards the center of the image stack. In this case, the resulting images will be cropped to the region where all of the images overlap. However, one can specify the reference image when initializing the Valis object, by setting reference_img_f to the filename of the image the others should be aligned towards. When the reference image is specified, the images will be cropped such that only the regions which overlap with the reference image will be saved. While this is the default behavior, one can also specify the cropping method by setting the crop parameter value when initializing the Valis object. The cropping method can also be changed when saving the registered images (see below).

```
from valis import registration
slide_src_dir = "/path/to/slides"
results_dst_dir = "./slide_registration_example"
registered_slide_dst_dir = "./slide_registration_example/registered_slides"

# Create a Valis object and use it to register the slides in slide_src_dir
registrar = registration.Valis(slide_src_dir, results_dst_dir)
rigid_registrar, non_rigid_registrar, error_df = registrar.register()
```

Important: It is also possible to register a subset of images in src_dir, or combinations of images located in different directories. This can be done by passing a list of the image paths to img_list when initializing the Valis object.

The next example shows how align each image to a reference image, followed up by micro-registration. The reference image the others should be aligned towards is set with the reference_img_f argument when initializing the Valis object. This initial registration is followed up by micro-registration in order to better align features that were not present in the smaller images used for the first registration (The size of the images used for micro-registration can is set with the max_non_rigid_registartion_dim_px argument in Valis.register_micro). Setting align_to_reference to *True* will align each image directly *to* the reference image, as opposed to *towards* it.

```
from valis import registration
slide_src_dir = "/path/to/slides"
results_dst_dir = "./slide_registration_example"
registered_slide_dst_dir = "./slide_registration_example/registered_slides"
reference_slide = "HE.tiff"

# Create a Valis object and use it to register the slides in slide_src_dir, aligning␣
↪*towards* the reference slide.
registrar = registration.Valis(slide_src_dir, results_dst_dir, reference_img_f=reference_
↪slide)
rigid_registrar, non_rigid_registrar, error_df = registrar.register()

# Perform micro-registration on higher resolution images, aligning *directly to* the␣
```

(continues on next page)

(continued from previous page)

```
↪reference image
registrar.register_micro(max_non_rigid_registration_dim_px=2000, align_to_reference=True)
```

After registration is complete, one can view the results to determine if they are acceptable. In this example, the results are located in `./slide_registration_example`. Inside this folder will be 6 subfolders:

1. **data** contains 2 files:
   - a summary spreadsheet of the alignment results, such as the registration error between each pair of slides, their dimensions, physical units, etc…
   - a pickled version of the registrar. This can be reloaded (unpickled) and used later. For example, one could perform the registration locally, but then use the pickled object to warp and save the slides on an HPC. Or, one could perform the registration and use the registrar later to warp points found in the (un-registered) slide.
2. **overlaps** contains thumbnails showing the how the images would look if stacked without being registered, how they look after rigid registration, and how they look after non-rigid registration. The rightmost images in the figure above provide examples of these overlap images.
3. **rigid_registration** shows thumbnails of how each image looks after performing rigid registration. These would be similar to the bottom row in the figure above.
4. **non_rigid_registration** shows thumbnails of how each image looks after non-rigid registration. These would be similar to the bottom row in the figure above.
5. **deformation_fields** contains images showing what the non-rigid deformation would do to a triangular mesh. These can be used to get a sense of how the images were altered by non-rigid warping. In these images, the color indicates the direction of the displacement, while brightness indicates it's magnitude. These would be similar to those in the middle row in the figure above.
6. **processed** shows thumbnails of the processed images. These are thumbnails of the images that were actually used to perform the registration. The pre-processing and normalization methods should try to make these images look as similar as possible.
7. **masks** show the images with outlines of their rigid registration mask drawn around them. If non-rigid registration is being performed, there will also be an image of the reference image with the non-rigid registration mask drawn around it.

If the results look good, then one can warp and save all of the slides as ome.tiffs. When saving the images, there are three cropping options:

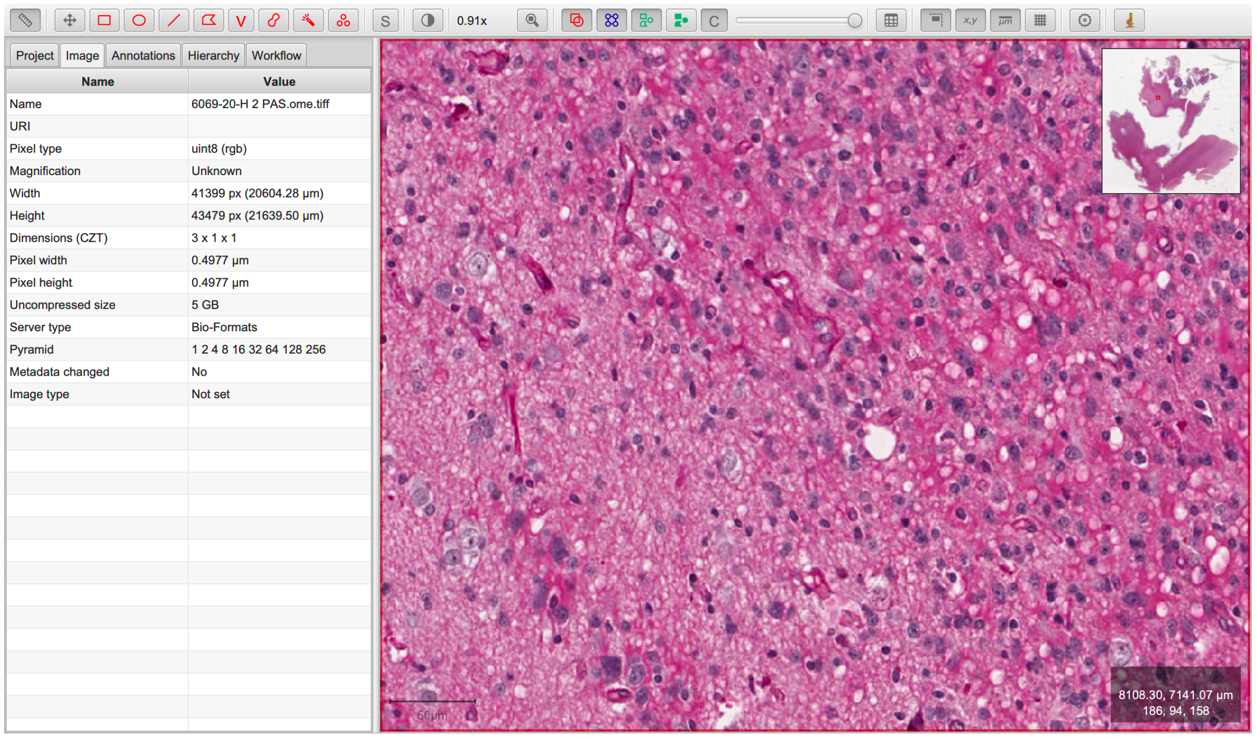
1. `crop="overlap"` will crop the images to the region where all of the images overlap.
2. `crop="reference"` will crop the images to the region where they overlap with the reference image.
3. `crop="all"` will not perform any cropping. While this keep the all of the image, the dimensions of the registered image can be substantially larger than one that was cropped, as it will need to be large enough accommodate all of the other images.

While the cropping setting can also be set when initializing the `Valis` object, any of the above cropping methods can be used when saving the images.

Important: By default, images are saved using lossless LZW compression. While this maintains the image's original quality, it may also generate files with very large sizes. One can reduce the file size by setting `compression` to `jpeg` or `jp2k`, and/or control how lossy the compression is by setting `Q` to a value less than `100`. Do note that currently `jpeg` or `jp2k` can only write uint8 images, and so may not be suitable for immunofluorescence (or similar) images that have a different datatype.

```
# Save all registered slides as ome.tiff
registrar.warp_and_save_slides(registered_slide_dst_dir, crop="overlap")

# Kill the JVM
registration.kill_jvm()
```

The ome.tiff images can subsequently be used for downstream analysis, such as QuPath



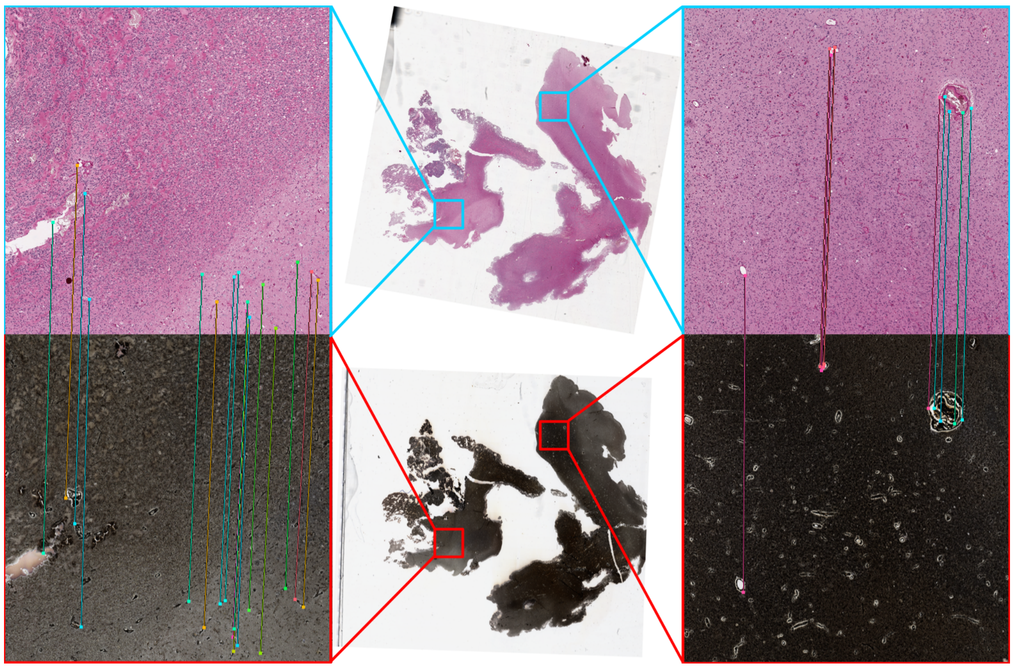
Name	Value
Name	6069-20-H 2 PAS.ome.tiff
URI	
Pixel type	uint8 (rgb)
Magnification	Unknown
Width	41399 px (20604.28 µm)
Height	43479 px (21639.50 µm)
Dimensions (CZT)	3 x 1 x 1
Pixel width	0.4977 µm
Pixel height	0.4977 µm
Uncompressed size	5 GB
Server type	Bio-Formats
Pyramid	1 2 4 8 16 32 64 128 256
Metadata changed	No
Image type	Not set

One can also choose to save individual slides. This is accomplished by accessing the Slide object associated with a particular file, `slide_f` and then "telling" it to save the slide as `out_f.ome.tiff`.

```
slide_obj = registrar.get_slide(slide_f)
slide_obj.warp_and_save_slide("out_f.ome.tiff")
```

Finally, if the non-rigid registration is deemed to have distorted the image too much, one can apply only the rigid transformation by setting `non_rigid=False` in `slide_obj.warp_and_save_slide` or `registrar.warp_and_save_slides`.

High resolution registration



The default pipeline, which uses low resolution images, can provide high quality results in a short amount of time. However, the results can be sometimes be improved by using higher resolution images, albeit at the cost of much higher computation times. This example shows how to perform registration using higher resolution images for both rigid and non-rigid registration. The `micro_rigid_registrar.MicroRigidRegistrar` is used in the main pipeline to update the rigid registration by finding matches in higher resolution versions of the registered images. Keyword arguments used to initialize `micro_rigid_registrar.MicroRigidRegistrar` can be passed in as the `micro_rigid_registrar_params` argument when initializing the `Valis` object. After the main pipeline is complete, one can also perform the 2nd high-resolution non-rigid registration using `Valis.register_micro`. In this example, we perform the micro-registration using images that are 25% of the full resolution.

```
import time
import os
import numpy as np
from valis import registration
from valis.micro_rigid_registrar import MicroRigidRegistrar # For high resolution rigid␣
↪registration

slide_src_dir = "./example_datasets/ihc"
results_dst_dir = "./expected_results/registration_hi_rez"
micro_reg_fraction = 0.25 # Fraction full resolution used for non-rigid registration

# Perform high resolution rigid registration using the MicroRigidRegistrar
start = time.time()
registrar = registration.Valis(slide_src_dir, results_dst_dir, micro_rigid_registrar_
↪cls=MicroRigidRegistrar)
rigid_registrar, non_rigid_registrar, error_df = registrar.register()

# Calculate what `max_non_rigid_registration_dim_px` needs to be to do non-rigid␣
↪registration on an image that is 25% full resolution.
img_dims = np.array([slide_obj.slide_dimensions_wh[0] for slide_obj in registrar.slide_
```

(continues on next page)

(continued from previous page)

```
↪dict.values()])
min_max_size = np.min([np.max(d) for d in img_dims])
img_areas = [np.multiply(*d) for d in img_dims]
max_img_w, max_img_h = tuple(img_dims[np.argmax(img_areas)])
micro_reg_size = np.floor(min_max_size*micro_reg_fraction).astype(int)

# Perform high resolution non-rigid registration using 25% full resolution
micro_reg, micro_error = registrar.register_micro(max_non_rigid_registration_dim_
↪px=micro_reg_size)
```

Create multiplex image from immunofluorescence images

Following registration, VALIS can merge the slides to create a single composite image. However, this should only be done for non-RGB images, such as multi/single-channel immunofluorescence images. An example would be slides of multiple CyCIF rounds. The user also has the option to provide channel names, but if not provided the channel names will become the “channel (filename)” given the channel name in the metadata. For example, if the file name is round1.ndpis then the DAPI channel name will be “DAPI (round1)”). In this example, the channel names are taken from the filename, which have the form “Tris CD20 FOXP3 CD3.ndpis”, “Tris CD4 CD68 CD3 1in25 ON.ndpis”, etc… The channel names need to be in a dictionary, where key=filename, value = list of channel names.

Important: By default, if a channel occurs in more than 1 image, only the 1st instance will be merged. For example, if DAPI is in all images, then only the DAPI channel of the 1st image will be in the resulting slide. This can be disabled by setting `drop_duplicates=False` in `warp_and_merge_slides`

First, create a VALIS object and use it to register slides located in `slide_src_dir`

```
from valis import registration
slide_src_dir = "/path/to/slides"
results_dst_dir = "./slide_merging_example"  # Registration results saved here
merged_slide_dst_f = "./slide_merging_example/merged_slides.ome.tiff"  # Where to save␣
↪merged slide

registrar = registration.Valis(slide_src_dir, results_dst_dir)
rigid_registrar, non_rigid_registrar, error_df = registrar.register()
```

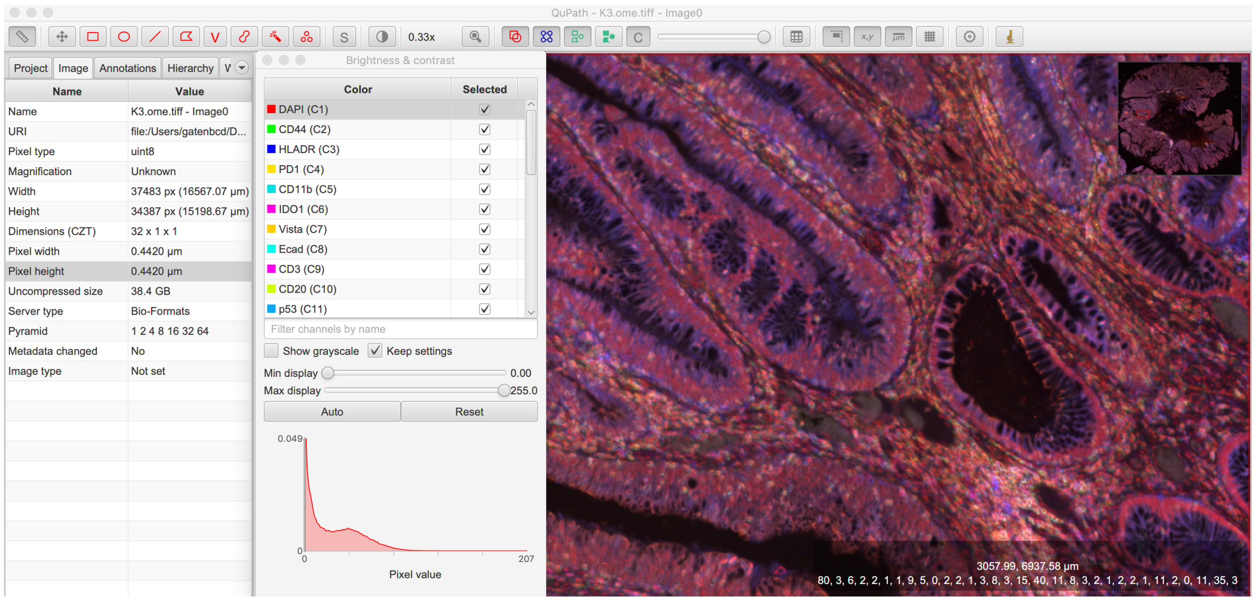
Check the results in `results_dst_dir`, and if the look good merge and save the slide. Once complete, be sure to kill the JVM.

```
# Create function to extract channel names from the image.
def cnames_from_filename(src_f):
    """Get channel names from file name
    Note that the DAPI channel is not part of the filename
    but is always the first channel.
    """

    f = valtils.get_name(src_f)
    return ["DAPI"] + f.split(" ")[1:4]

channel_name_dict = {f:cnames_from_filename(f) for f in registrar.original_img_list}
merged_img, channel_names, ome_xml = \
```

(continues on next page)

(continued from previous page)

```
    registrar.warp_and_merge_slides(merged_slide_dst_f,
                                    channel_name_dict=channel_name_dict,
                                    drop_duplicates=True)

registration.kill_jvm() # Kill the JVM
```



Name	Value
Name	K3.ome.tiff - Image0
URI	file:/Users/gatenbcd/D...
Pixel type	uint8
Magnification	Unknown
Width	37483 px (16567.07 μm)
Height	34387 px (15198.67 μm)
Dimensions (CZT)	32 x 1 x 1
Pixel width	0.4420 μm
Pixel height	0.4420 μm
Uncompressed size	38.4 GB
Server type	Bio-Formats
Pyramid	1 2 4 8 16 32 64
Metadata changed	No
Image type	Not set

Color	Selected
🟥 DAPI (C1)	✓
🟩 CD44 (C2)	✓
🟦 HLADR (C3)	✓
🟨 PD1 (C4)	✓
🟦 CD11b (C5)	✓
🟪 IDO1 (C6)	✓
🟨 Vista (C7)	✓
🟦 Ecad (C8)	✓
🟪 CD3 (C9)	✓
🟩 CD20 (C10)	✓
🟦 p53 (C11)	✓

Show grayscale [ ] Keep settings [✓]

Min display 0.00
Max display 255.0

3057.99, 6937.58 μm
80, 3, 6, 2, 2, 1, 1, 9, 5, 0, 2, 2, 1, 3, 8, 3, 15, 40, 11, 8, 3, 2, 1, 2, 2, 1, 11, 2, 0, 11, 35, 3

Warping points

Once the registration parameters have been found, VALIS can be used to warp point data, such as cell coordinates, mask polygon vertices, etc… In this example, slides will be registered, and the registration parameters will then be used warp cell positions located in a separate .csv. This accomplished by accessing the `Slide` object associated with each registered slide. This is done by passing the slide's filename (with or without the extension) to `registrar.get_slide`. This `Slide` object can the be used to warp the individual slide and/or points associated with the un-registered slide. This can be useful in cases where one has already performed an analysis on the un-registered slides, as one can just warp the point data, as opposed to warping each slide and re-conducting the analysis.

Important: It is essential that the image from which the coordinates are derived has the same aspect ratio as the image used for registration. That is, the images used for registration must be scaled up/down versions of the image from which the coordinates are taken. For example, registration may be performed on lower resolution images (an upper image pyramid level), and applied to cell coordinates found by performing cell segmentation on the full resolution (pyramid level 0) image. The default is to assume that the points came from the highest resolution image, but this can be changed by setting `pt_level` to either the pyramid level of the image the points originated, or its dimensions (width, height, in pixels). Also, the coordinates need to be in pixel units, not physical units. Finally, be sure that the coordinates are X,Y (column, row), with the origin being the top left corner of the image.

In this first example, cell segmentation and phenotyping has already been performed on the unregistered images. We can now use the `Valis` object that performed the registration to warp the cell positions to their location in the registered images.

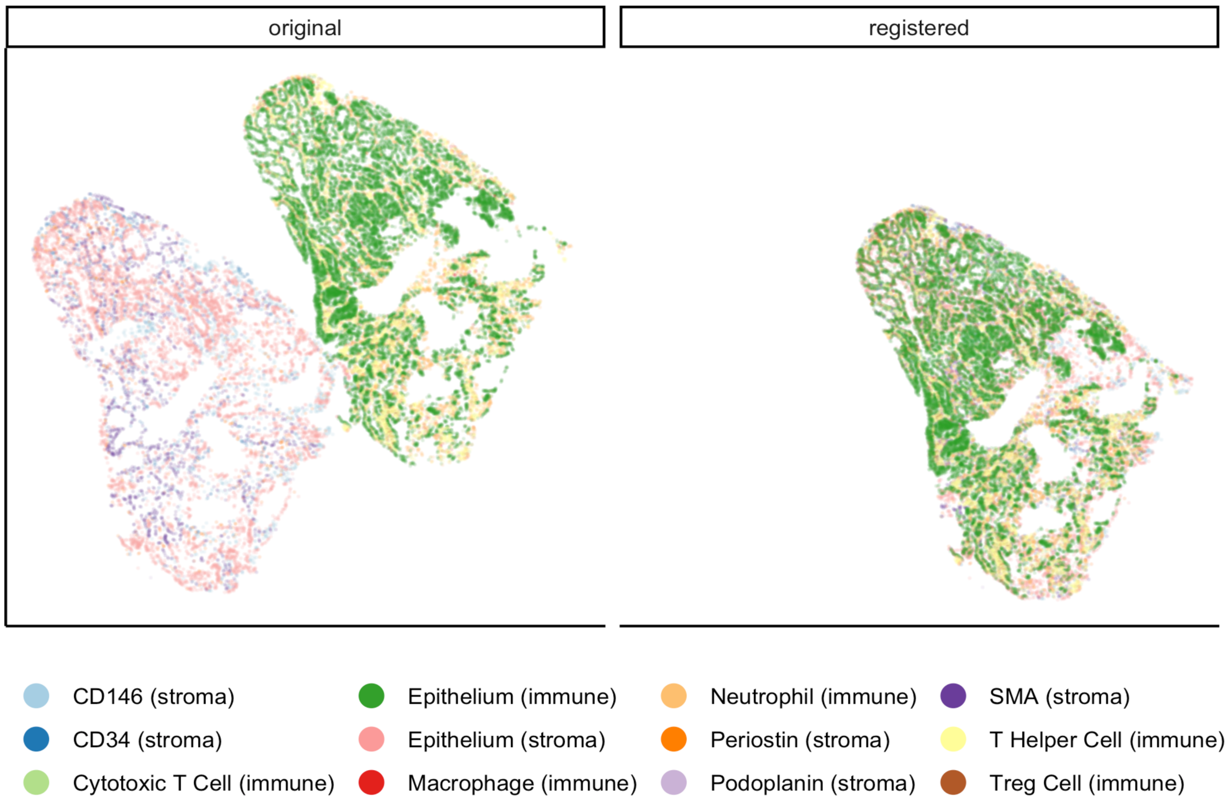
```
import os
import numpy as np
```

(continues on next page)

(continued from previous page)

```python
import pandas as pd
import pathlib
import pickle
from valis import registration

slide_src_dir = "path/to/slides"
point_data_dir = "path/to/cell_positions"
results_dst_dir = "./point_warping_example"

# Load a Valis object that has already registered the images.
registrar_f = "path/to/results/data/registrar.pickle"
registrar = registration.load_registrar(registrar_f)

# Get .csv files containing cell coordinates
point_data_list = list(pathlib.Path(point_data_dir).rglob("*.csv"))

# Go through each file and warp the cell positions
for f in point_data_list:
    # Get Slide object associated with the slide from which the point data originated
    # Point data and image have similar file names
    fname = os.path.split(f)[1]
    corresponding_img = fname.split(".tif")[0]
    slide_obj = registrar.get_slide(corresponding_img)

    # Read data and calculate cell centroids (x, y)
    points_df = pd.read_csv(f)
    x = np.mean(points_df[["XMin", "XMax"]], axis=1).values
    y = np.mean(points_df[["YMin", "YMax"]], axis=1).values
    xy = np.dstack([x, y])[0]

    # Use Slide to warp the coordinates
    warped_xy = slide_obj.warp_xy(xy)

    # Update dataframe with registered cell centroids
    points_df[["registered_x", "registered_y"]] = warped_xy

    # Save updated dataframe
    pt_f_out = os.path.split(f)[1].replace(".csv", "_registered.csv")
    full_pt_f_out = os.path.join(results_dst_dir, pt_f_out)
    points_df.to_csv(full_pt_f_out, index=False)

registration.kill_jvm() # Kill the JVM
```

Here is a comparison of before and after applying registration to cell positions found in the original un-aligned images:



In this second example, a region of interest (ROI) was marked in one of the unregistered images, in this case “ihc_2.ome.tiff” . Using the `Slide` object associated with “ihc_2.ome.tiff”, we can warp those ROI coordinates to their position in the registered images, and then use those to slice the registered ROI from each slide. Because VALIS uses pyvips to read and warp the slides, this process does not require the whole image to be loaded into memory and warped. As such, this is fast and does not require much memory. It’s also worth noting that because the points are being warped to the registered coordinate system, the slide that is the source of the ROI coordinates does not have to be the same slide that was treated as the reference image during registration.

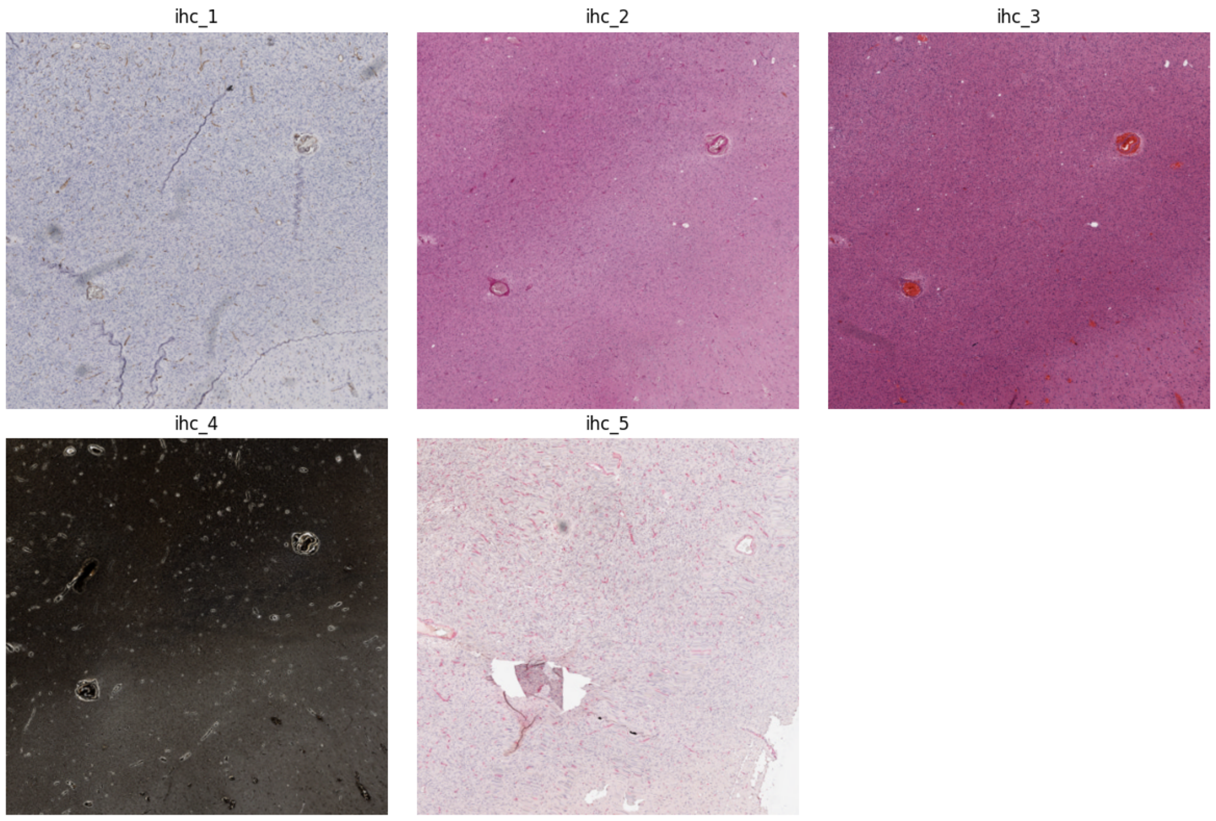
```
import os
import pickle
import numpy as np
import matplotlib.pyplot as plt
import pathlib
from valis import registration, warp_tools

# Load a registrar that has already registered the images.
registrar_f = "./expected_results/registration/ihc/data/ihc_registrar.pickle"
registrar = registration.load_registrar(registrar_f)
# Set the pyramid level from which the ROI coordinates originated. Usually 0 when␣
↪working with slides.
COORD_LEVEL = 0

# ROI coordinates, in microns. These came from the unregistered slide, "ihc_2.ome.tiff"
bbox_xywh_um = [14314, 13601, 3000, 3000]
bbox_xy_um = warp_tools.bbox2xy(bbox_xywh_um)
```

(continues on next page)

(continued from previous page)

```python
# Get slide from which the ROI coordinates originated
pt_source_img_f = "ihc_2.ome.tiff"
pt_source_slide = registrar.get_slide(pt_source_img_f)

# Convert coordinates to pixel units
um_per_px = pt_source_slide.reader.scale_physical_size(COORD_LEVEL)[0:2]
bbox_xy_px = bbox_xy_um/np.array(um_per_px)

# Warp coordinates to position in registered slides
bbox_xy_in_registered_img = pt_source_slide.warp_xy(bbox_xy_px,
                                                    slide_level=COORD_LEVEL,
                                                    pt_level=COORD_LEVEL)

bbox_xywh_in_registered_img = warp_tools.xy2bbox(bbox_xy_in_registered_img)
bbox_xywh_in_registered_img = np.round(bbox_xywh_in_registered_img).astype(int)

# Create directory where images will be saved
dst_dir = "./expected_results/roi"
pathlib.Path(dst_dir).mkdir(exist_ok=True, parents=True)

# Warp each slide and slice the ROI from it using each pyips.Image's "extract_area"␣
↪method.
fig, axes = plt.subplots(2, 3, figsize=(12, 8), sharex=True, sharey=True)
ax = axes.ravel()
for i, slide in enumerate(registrar.slide_dict.values()):
    warped_slide = slide.warp_slide(level=COORD_LEVEL)
    roi_vips = warped_slide.extract_area(*bbox_xywh_in_registered_img)
    roi_img = warp_tools.vips2numpy(roi_vips)
    ax[i].imshow(roi_img)
    ax[i].set_title(slide.name)
    ax[i].set_axis_off()

fig.delaxes(ax[5]) # Only 5 images, so remove 6th subplot
out_f = os.path.join(dst_dir, f"{registrar.name}_roi.png")
plt.tight_layout()
plt.savefig(out_f)
plt.close()

# Opening the slide initialized the JVM, so it needs to be killed
registration.kill_jvm()
```

The extracted and registered ROI are shown below:



Transferring annotations

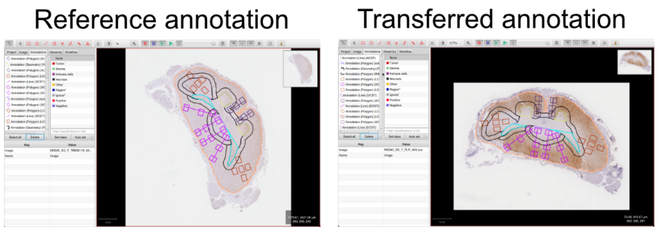
In this example, VALIS uses the registration parameters to transfer annotations found from one image to another. In this case, the annotation were performed in QuPath and exported as a geojson file. Given the geojson file, VALIS can then warp each shape in the file from the reference slide to its position on the un-registered target slide. The registered annotations can then be saved and loaded into QuPath along with the target image. Below, `annotation_img_f` refers to the filename associated with the image on which the original annotation was performed, `target_img_f` is the filename of the image associated with the image the annotations will be transferred to, `annotation_geojson_f` is the name of the file with the annotation shapes, and `warped_geojson_annotation_f` is the name of geojson file the registered annotations will be saved to.

```
import json
from valis import registration

# Perform registration
registrar = registration.Valis(slide_src_dir, results_dst_dir)
rigid_registrar, non_rigid_registrar, error_df = registrar.register()

# Transfer annotation from image associated with annotation_img_f and image associated␣
↪with target_img_f
annotation_source_slide = registrar.get_slide(annotation_img_f)
target_slide = registrar.get_slide(target_img_f)


warped_geojson_from_to = annotation_source_slide.warp_geojson_from_to(annotation_geojson_
```

(continues on next page)

(continued from previous page)

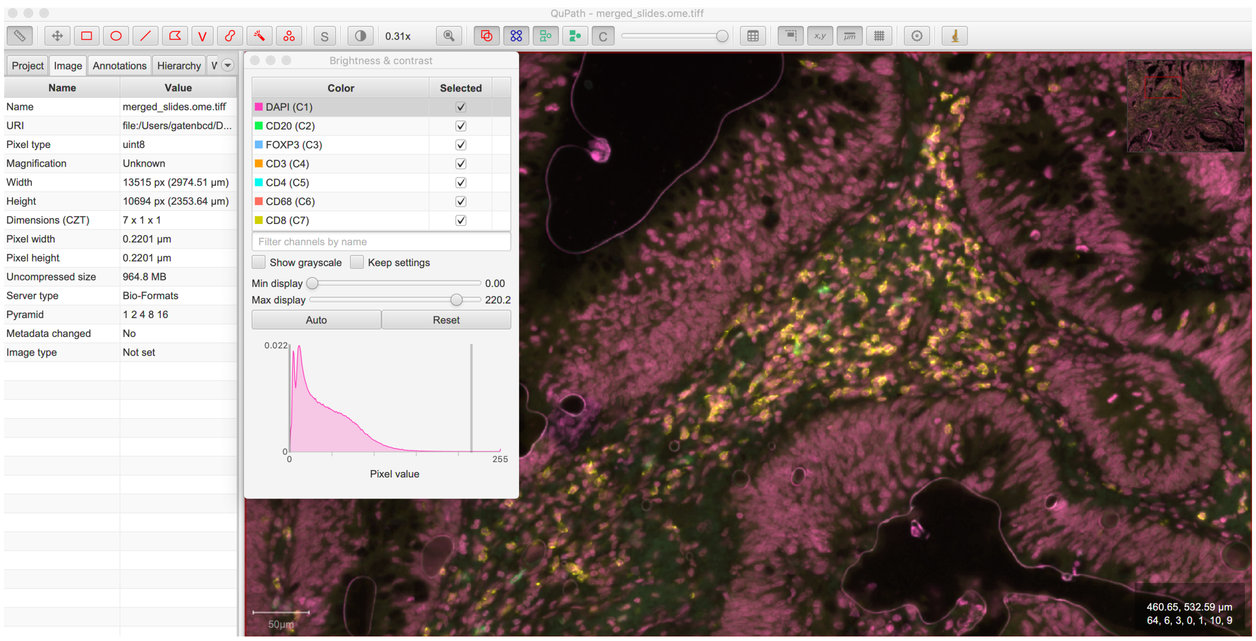
```
↪f, target_slide)
warped_geojson = annotation_source_slide.warp_geojson(annotation_geojson_f)

# Save annotation as warped_geojson_annotation_f, which can be dragged and dropped into␣
↪QuPath
with open(warped_geojson_annotation_f, 'w') as f:
    json.dump(warped_geojson, f)
```



Converting slides to ome.tiff

In addition to registering slide, VALIS can convert slides to ome.tiff, maintaining the original metadata. If the original is image is not RGB, the option `colormap` can be used to give each channel a specific color using a dictionary, where the key is the channel name, and the value is the RGB tuple (0-255). If `colormap` is not provided, the original channel colors will be used.

```
from valis import slide_io
slide_src_f = "path/to/slide
converted_slide_f = "converted.ome.tiff"
slide_io.convert_to_ome_tiff(slide_src_f,
                             converted_slide_f,
                             level=0)
slide_io.kill_jvm()
```



Name	Value
Name	merged_slides.ome.tiff
URI	file:/Users/gatenbcd/D...
Pixel type	uint8
Magnification	Unknown
Width	13515 px (2974.51 µm)
Height	10694 px (2353.64 µm)
Dimensions (CZT)	7 x 1 x 1
Pixel width	0.2201 µm
Pixel height	0.2201 µm
Uncompressed size	964.8 MB
Server type	Bio-Formats
Pyramid	1 2 4 8 16
Metadata changed	No
Image type	Not set

Color	Selected
DAPI (C1)	✓
CD20 (C2)	✓
FOXP3 (C3)	✓
CD3 (C4)	✓
CD4 (C5)	✓
CD68 (C6)	✓
CD8 (C7)	✓

Filter channels by name
Show grayscale Keep settings
Min display 0.00
Max display 220.2
Auto Reset
0.022
0
0 255
Pixel value

Reading slides

VALIS also provides functions to read images/slides using libvips, Bio-Formats, or Openslide. These reader objects also contain some of the slide's metadata. The `slide2image` method will return a numpy array of the slide, while `slide2vips` will return a `pyvips.Image`, which is ideal when working with very large images. The user can specify the pyramid level, series, and bounding box, but the default is level 0, series 0, and the whole image. See `slide_io.SlideReader` and `slide_io.MetaData` for more details.

```
from valis import slide_io
slide_src_f = "path/to/slide.svs
series = 0

# Get reader for slide format
reader_cls = slide_io.get_slide_reader(slide_src_f, series=series) #Get appropriate␣
↪slide reader class
reader = reader_cls(slide_src_f, series=series) # Instantiate reader

#Get size of images in each pyramid level (width, height)
pyramid_level_sizes_wh = reader.metadata.slide_dimensions

# Get physical units per pixel
pixel_physical_size_xyu = reader.metadata.pixel_physical_size_xyu

# Get channel names (None if image is RGB)
channel_names = reader.metadata.channel_names

# Get original xml metadata
original_xml = reader.metadata.original_xml

# Get smaller pyramid level 3 as a numpy array
img = reader.slide2image(level=3)

# Get full resolution image as a pyvips.Image
full_rez_vips = reader.slide2vips(level=0)

# Slice region of interest from level 0 and return as numpy array
roi_img = reader.slide2image(level=0, xywh=(100, 100, 500, 500))

slide_io.kill_jvm()
```

Warping slides with custom transforms

VALIS provides the functions to apply transformations to slides and then save the registered slide, meaning the user can provide their own transformation parameters. In this example, *src_f* is the path to the file associated with the slide, *M* is the inverse rigid registration matrix, and *bk_dxdy* is a list of the backwards non-rigid displacement fields (i.e. [dx, dy]), each found by aligning the fixed/target image to the moving/source image.

Important: The transformations will need to be inverted if they were found the other way around, i.e. aligning the moving/source image to the fixed/target image. Transformation matrices can be inverted using `np.linalg.inv`, while displacement fields can be inverted using `warp_tools.get_inverse_field`.

One may also need to provide the shape of the image (row, col) used to find the rigid transformation (if applicable), which is the *transformation_src_shape_rc* argument. In this case, it is the shape of the processed image that was used during feature detection. Similarly, *transformation_dst_shape_rc* is the shape of the registered image, in this case the shape of the processed image after being warped. Finally, *aligned_slide_shape_rc* is the shape of the warped slide. Please see `slide_io.warp_and_save_slide` for more information and options, like defining background color, crop area, etc..

```
from valis import slide_io

# Read and warp the slide #
slide_src_f = "path/to/slide
dst_f = "path/to/write/slide.ome.tiff"
series = 0
pyramid_level=0

slide_io.warp_and_save_slide(src_f=slide_src_f,
                             dst_f=dst_f,
                             transformation_src_shape_rc=processed_img_shape_rc,
                             transformation_dst_shape_rc=small_registered_img_shape_rc,
                             aligned_slide_shape_rc=aligned_slide_shape_rc,
                             level=pyramid_level,
                             series=series,
                             M=M,
                             dxdy=dxdy)


slide_io.kill_jvm()
```

Using non-defaults

The defaults used by VALIS work well, but one may wish to try some other values/class, and/or create their own affine optimizer, feature detector, non-rigid registrar, etc… This examples shows how to conduct registration using non-default values

Note: This example assumes that SimpleElastix has been installed.

```
from valis import registration, feature_detectors, non_rigid_registrars, affine_optimizer
slide_src_dir = "path/to/slides"
results_dst_dir = "./slide_registration_example_non_defaults"
registered_slide_dst_dir = "./slide_registration_example/registered_slides"


# Select feature detector, affine optimizer, and non-rigid registration method.
# Will use KAZE for feature detection and description
# SimpleElastix will be used for non-rigid warping and affine optimization
feature_detector_cls = feature_detectors.KazeFD
non_rigid_registrar_cls = non_rigid_registrars.SimpleElastixWarper
affine_optimizer_cls = affine_optimizer.AffineOptimizerMattesMI

# Create a Valis object and use it to register the slides in slide_src_dir
registrar = registration.Valis(slide_src_dir, results_dst_dir,
                               feature_detector_cls=feature_detector_cls,
```

(continues on next page)

(continued from previous page)

```
                                affine_optimizer_cls=affine_optimizer_cls,
                                non_rigid_registrar_cls=non_rigid_registrar_cls)


rigid_registrar, non_rigid_registrar, error_df = registrar.register()

registration.kill_jvm() # Kill the JVM
```

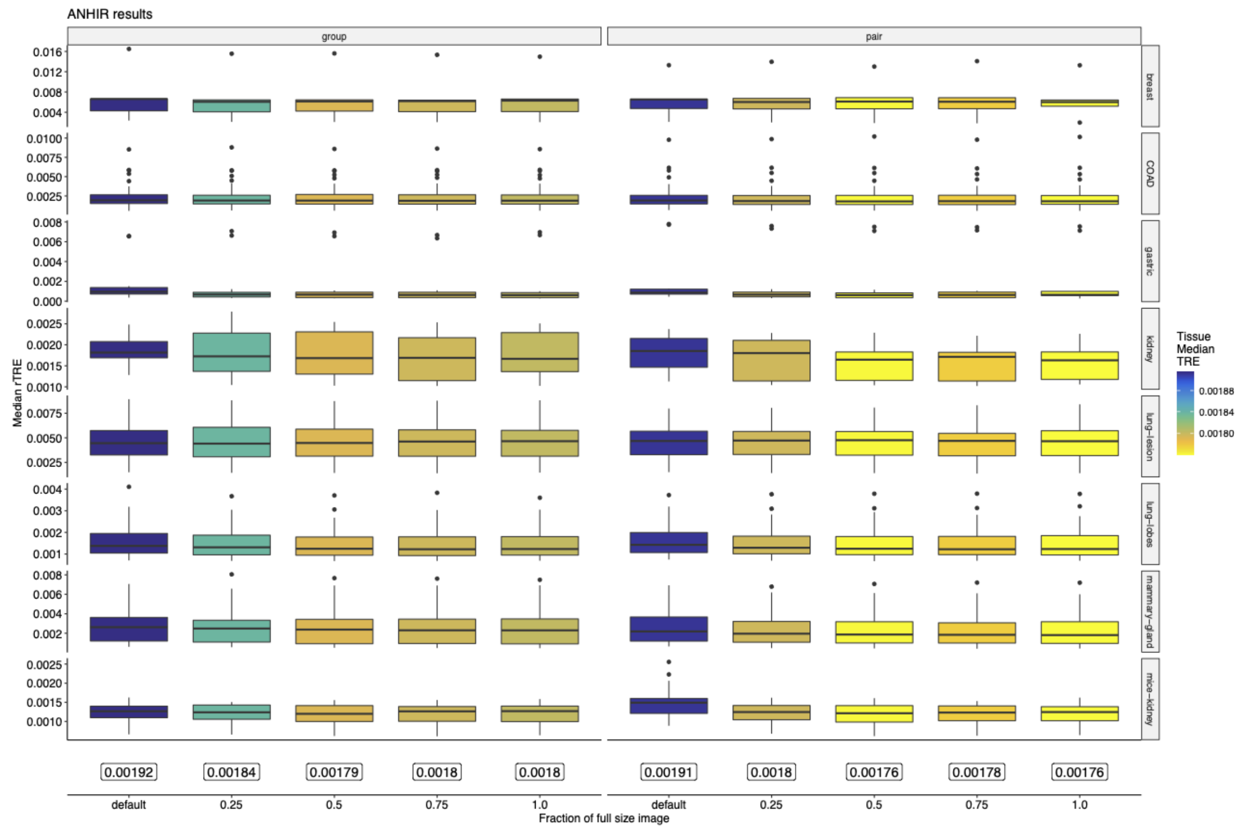
1.1.3 Datasets

This section is intended to provide information about datasets related to WSI registration and benchmarking.

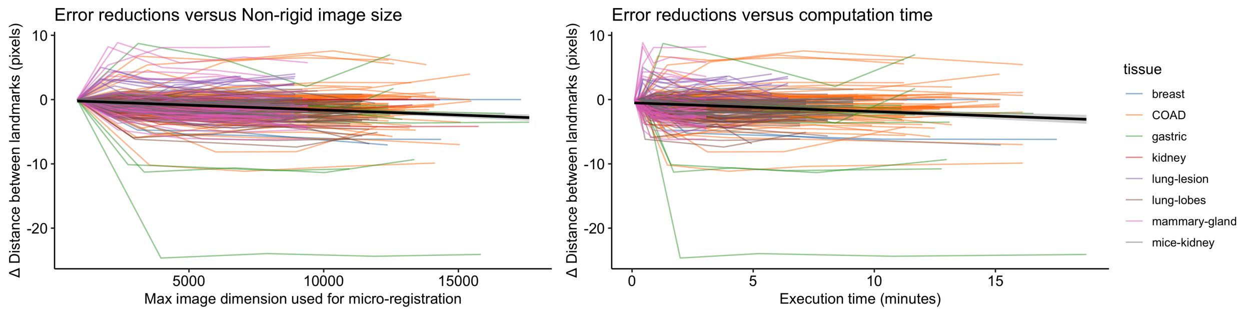
ANHIR

The Automatic Non-rigid Histological Image Registration (ANHIR) was part of 2019 the IEEE International Symposium on Biomedical Imaging (ISBI). The aim of the challenge is to automatically perform non-rigid registration of 481 brightfield WSI pairs stained with different dyes. The high resolution WSI (up to 40x) came from 49 unique samples (multiple images per sample) that a variety of source tissues, including lung lesions, lung lobes, mammary glands, colon adenoma carcinomas (COAD), mice kidneys, gastric mucosa, breast tissue, and kidney tissue. Hand annotated landmarks are provided for each “moving” image, with corresponding landmarks provided for 230 “fixed” images. These 230 images can therefore be used to train/tune the algorithm. The remaining 251 fixed landmarks are not available, but are used to calculate registration accuracy when the results are submitted. Registration accuracy is primarily measured by the median relative target registration error (rTRE), which is the distance between registered landmarks (i.e. warped moving landmarks and fixed landmarks), divided by the fixed image’s diagonal, thus providing a normalized measure of accuracy.

Benchmarking indicates that VALIS performs competitively using the default parameters, being one of the more accurate opensource methods. Below is a figure showing the rTRE scores across a range of image sizes used for registration.



This next plot shows the effect of increasing the size of the image used in the micro-registration, where the y-axis is the difference in landmark distance compared to the results when not using micro-registration (i.e. only using the lower resolution image to perform the registration). Interestingly, increasing the image size used to perform registration did not always increase accuracy, and when there were improvements, they were not always substantial.

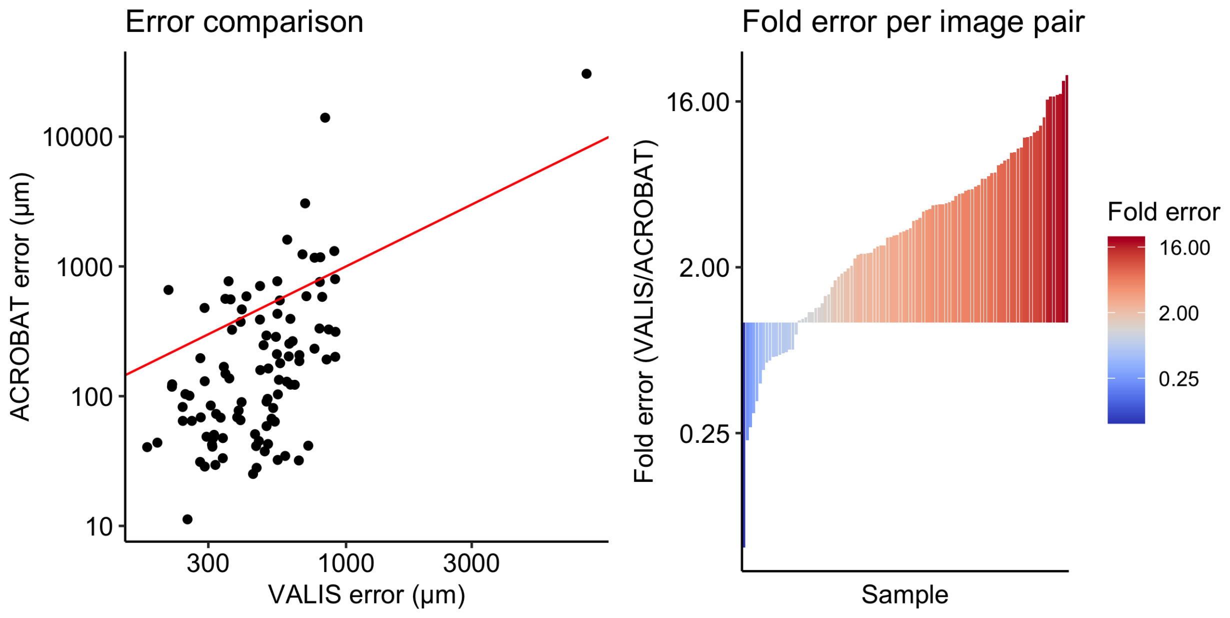


ACROBAT

The AutomatiC Registration Of Breast cAncer Tissue (ACROBAT) was part of MICCAI 2022 and, similar to ANHIR, the goal is to automatically register pairs of WSI. However, unlike ANHIR, the images were collected from routine diagnostic workflows, and so often contained artifacts common to real world datasets, such as cracks, streaks, pen marks, bubbles, etc… that increase the difficulty of image registration. There were are total of 400 unique samples used to assess registration accuracy, with 100 images pairs used for a validation leaderboard (used to aid in developing the algorithm), and 300 for a test leaderboard. Unlike ANHIR, fixed landmarks were not provided, meaning that it is not possible to train/tune the method using matched landmarks. Scores can only be calculated by upload the registered landmarks to the submission system. The primary score used to measure accuracy was the 90th percentile of physical distances between registered moving and fixed landmarks, in m.

Due to the challenging nature of the images, a custom `ImageProcesser` class was created to clean up and process the images. All image pairs were then registered using the defaults, followed by micro-registration using an image that was 25% of the full image's size. The script used to create this `ImageProcesser` and conduct the registration can be found here. Using this approach, VALIS placed second overall, and first among the opensource methods.

As ACROBAT measures error in m, and VALIS estimates error based on matched features, this dataset makes it possible to determine how well VALIS' error estimates match up with reality. The plot below shows the relationship between the estimated (VALIS) and true (ACROBAT) errors, with VALIS estimated error on the x-axis, error based on ACROBAT's hand annotations on the y-axis, and the identity line in red. These results indicate that VALIS tends to overestimate the error, with the actual accuracy being much better. This discrepancy may be due to the fact that the features used by VALIS to estimate error are based on much smaller versions of the images, and so their position is not as precise as those detected by hand.

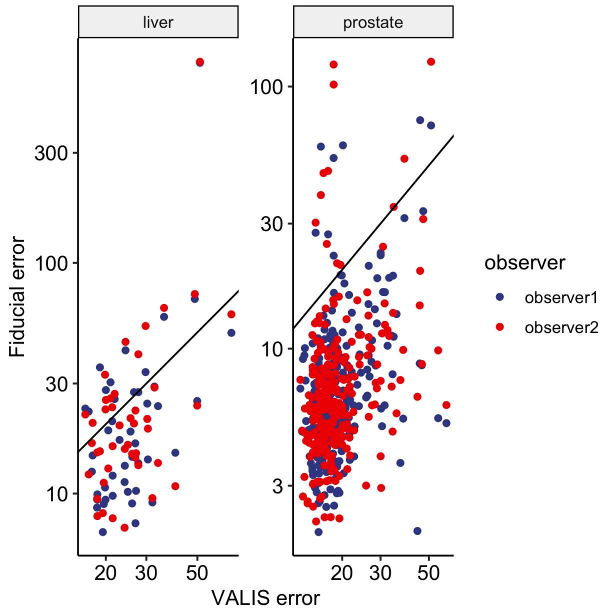


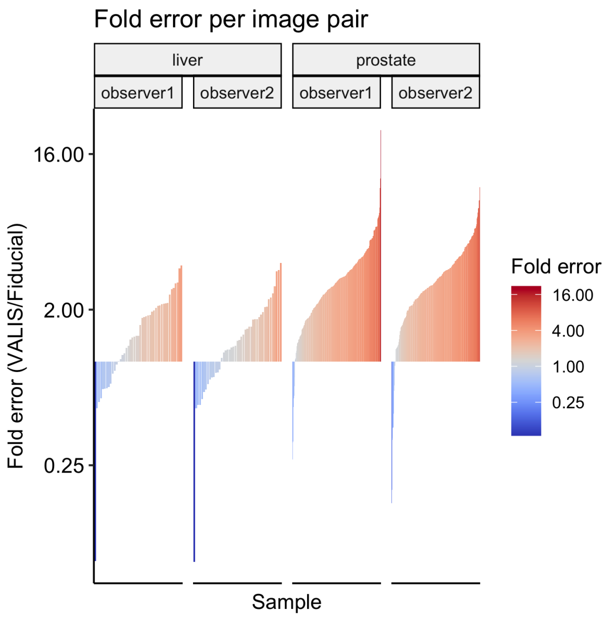
Kartasalo et al 2018

Another potential use of image registration is to construct a 3D tissue from serial slices. In 2018, Kartasalo et al. (2018) performed a study in which they compared several different frameworks for constructing 3D images, using both free and commercial software. They peformed the analysis using two datasets: one murine prostate to be reconstructed from 260 serially sliced 20x H&E images (0.46m/pixel, 5m in depth), and one murine liver to be reconstructed from 47 serial slices (0.46m/pixel, 5m in depth). Accuracy of the alignment of the liver can be determined using the positions of laser cut holes that pass through the whole tissue, and should in theory form a straight line. In the case of the prostate, for each pair of images, human operators determined the location of structures visible on both slices, preferably nuclei split by the sectioning blade. The authors refer to these landmarks as "fiducial points".

VALIS was used to register both datasets, and error was measured as the physical distance (m), i.e. TRE, between the fiducial points. These values can then be compared to those presented in Tables 1 and 2 of the manuscript, which provide the mean TRE using observer 1's landmarks (i.e. the "TRE " column). Benchmarking using the liver dataset indicates that VALIS produces a mean TRE of 52.98, compared to the compared to the baseline reference value of 27.3 (LS 1). In the case of prostate, VALIS scored 11.41, compared to the baseline reference value of 15.6 (LS 1). According to the authors, methods with scores approaching the LS 1 value can be considered "highly accurate", indicating that VALIS is suitable for 3D reconstruction. Below is a picture of the prostate tumor reconstructed from all 260 serial slices.

Similar to ACROBAT, this dataset provides the opportunity to compare VALIS' error estimate to those based on manual measurements (e.g. the fiducial points). For the same reasons as before, it appears that VALIS over-estimates the error, as shown in the plots below.





1.1.4 Registration

Functions

Classes and functions to register a collection of images

`valis.registration.`**`init_jvm`**(*jar=None*, *mem_gb=10*)

Initialize JVM for BioFormats

`valis.registration.`**`kill_jvm`**()

Kill JVM for BioFormats

`valis.registration.`**`load_registrar`**(*src_f*)

Load a Valis object

Parameters
`src_f` (*`string`*) – Path to pickled Valis object

Returns
registrar – Valis object used for registration

Return type
Valis

Classes

Valis

`class valis.registration.Valis`(*src_dir*, *dst_dir*, *series=None*, *name=None*, *image_type=None*, *feature_detector_cls=<class 'valis.feature_detectors.VggFD'>*, *transformer_cls=<class 'skimage.transform._geometric.SimilarityTransform'>*, *affine_optimizer_cls=None*, *similarity_metric='n_matches'*, *matcher=<valis.feature_matcher.Matcher object>*, *imgs_ordered=False*, *non_rigid_registrar_cls=<class 'valis.non_rigid_registrars.OpticalFlowWarper'>*, *non_rigid_reg_params={}*, *compose_non_rigid=False*, *img_list=None*, *reference_img_f=None*, *align_to_reference=False*, *do_rigid=True*, *crop=None*, *create_masks=True*, *denoise_rigid=True*, *check_for_reflections=False*, *resolution_xyu=None*, *slide_dims_dict_wh=None*, *max_image_dim_px=850*, *max_processed_image_dim_px=850*, *max_non_rigid_registration_dim_px=850*, *thumbnail_size=500*, *norm_method='img_stats'*, *micro_rigid_registrar_cls=None*, *micro_rigid_registrar_params={}*, *qt_emitter=None*)

Reads, registers, and saves a series of slides/images

Implements the registration pipeline described in "VALIS: Virtual Alignment of pathoLogy Image Series" by Gatenbee et al. This pipeline will read images and whole slide images (WSI) using pyvips, bioformats, or openslide, and so should work with a wide variety of formats. VALIS can perform both rigid and non-rigid registration. The registered slides can be saved as ome.tiff slides that can be used in downstream analyses. The ome.tiff format is opensource and widely supported, being readable in several different programming languages (Python, Java, Matlab, etc…) and software, such as QuPath or HALO.

The pipeline is fully automated and goes as follows:

1. Images/slides are converted to numpy arrays. As WSI are often too large to fit into memory, these images are usually lower resolution images from different pyramid levels.

2. Images are processed to single channel images. They are then normalized to make them look as similar as possible.

3. Image features are detected and then matched between all pairs of image.

4. If the order of images is unknown, they will be optimally ordered based on their feature similarity

5. Rigid registration is performed serially, with each image being rigidly aligned to the previous image in the stack.

6. Non-rigid registration is then performed either by 1) aliging each image towards the center of the stack, composing the deformation fields along the way, or 2) using groupwise registration that non-rigidly aligns the images to a common frame of reference.

7. Error is measured by calculating the distance between registered matched features.

The transformations found by VALIS can then be used to warp the full resolution slides. It is also possible to merge non-RGB registered slides to create a highly multiplexed image. These aligned and/or merged slides can then be saved as ome.tiff images using pyvips.

In addition to warping images and slides, VALIS can also warp point data, such as cell centoids or ROI coordinates.

`name`
Descriptive name of registrar, such as the sample's name.

Type
str

`src_dir`
Path to directory containing the slides that will be registered.

Type
str

`dst_dir`
Path to where the results should be saved.

Type
str

`original_img_list`
List of images converted from the slides in *src_dir*

Type
list of ndarray

`name_dict`
Key=full path to image, value = name used to look up *Slide* in *Valis.slide_dict*

Type
dictionary

`slide_dims_dict_wh`
Dictionary of slide dimensions. Only needed if dimensions not available in the slide/image's metadata.

`resolution_xyu`
Physical size per pixel and the unit.

Type
tuple

`image_type`
Type of image, i.e. "brightfield" or "fluorescence"

Type
str

`series`
Slide series to that was read.

Type
int

`size`
Number of images to align

Type
int

`aligned_img_shape_rc`
Shape (row, col) of aligned images

Type
tuple of int

`aligned_slide_shape_rc`

Shape (row, col) of the aligned slides

Type
tuple of int

`slide_dict`

Dictionary of Slide objects, each of which contains information about a slide, and methods to warp it.

Type
dict of *Slide*

`brightfield_procsseing_fxn_str`

Name of function used to process brightfield images.

Type
str

`if_procsseing_fxn_str`

Name of function used to process fluorescence images.

Type
str

`max_image_dim_px`

Maximum width or height of images that will be saved. This limit is mostly to keep memory in check.

Type
int

`max_processed_image_dim_px`

Maximum width or height of processed images. An important parameter, as it determines the size of of the image in which features will be detected and displacement fields computed.

Type
int

`reference_img_f`

Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be the reference.

Type
str

`reference_img_idx`

Index of slide that corresponds to *reference_img_f*, after the *img_obj_list* has been sorted during rigid registration.

Type
int

`align_to_reference`

Whether or not images should be aligne to a reference image specified by *reference_img_f*. Will be set to True if *reference_img_f* is provided.

Type
bool

`crop`

How to crop the registered images.

Type
str, optional

rigid_registrar

SerialRigidRegistrar object that performs the rigid registration.

Type
SerialRigidRegistrar

rigid_reg_kwargs

Dictionary of keyward arguments passed to *serial_rigid.register_images*.

Type
dict

feature_descriptor_str

Name of feature descriptor.

Type
str

feature_detector_str

Name of feature detector.

Type
str

transform_str

Name of rigid transform

Type
str

similarity_metric

Name of similarity metric used to order slides.

Type
str

match_filter_method

Name of method used to filter out poor feature matches.

Type
str

non_rigid_registrar

SerialNonRigidRegistrar object that performs serial non-rigid registration.

Type
SerialNonRigidRegistrar

non_rigid_reg_kwargs

Dictionary of keyward arguments passed to *serial_non_rigid.register_images*.

Type
dict

non_rigid_registrar_cls

Uninstantiated NonRigidRegistrar class that will be used by *non_rigid_registrar* to calculate the deformation fields between images.

Type
NonRigidRegistrar

`non_rigid_reg_class_str`
Name of the of class *non_rigid_registrar_cls* belongs to.

Type
str

`thumbnail_size`
Maximum width or height of thumbnails that show results

Type
int

`original_overlap_img`
Image showing how original images overlap before registration. Created by merging coloring the inverted greyscale copies of each image, and then merging those images.

Type
ndarray

`rigid_overlap_img`
Image showing how images overlap after rigid registration.

Type
ndarray

`non_rigid_overlap_img`
Image showing how images overlap after rigid + non-rigid registration.

Type
ndarray

`has_rounds`
Whether or not the contents of *src_dir* contain subdirectories that have single images spread across multiple files. An example would be .ndpis images.

Type
bool

`norm_method`
Name of method used to normalize the processed images

Type
str

`target_processing_stats`
Array of processed images' stats used to normalize all images

Type
ndarray

`summary_df`
Pandas dataframe containing information about the results, such as the error, shape of aligned slides, time to completion, etc…

Type
pd.Dataframe

`start_time`

The time at which registation was initiated.

Type
float

`end_rigid_time`

The time at which rigid registation was completed.

Type
float

`end_non_rigid_time`

The time at which non-rigid registation was completed.

Type
float

`qt_emitter`

Used to emit signals that update the GUI's progress bars

Type
PySide2.QtCore.Signal

`_non_rigid_bbox`

Bounding box of area in which non-rigid registration was conducted

Type
list

`_full_displacement_shape_rc`

Shape of full displacement field. Would be larger than *_non_rigid_bbox* if non-rigid registration only performed in a masked region

Type
tuple

`_dup_names_dict`

Dictionary describing which images would have been assigned duplicate names. Key= duplicated name, value=list of paths to images which would have been assigned the same name

Type
dictionary

`_empty_slides`

Dictionary of *Slide* objects that have empty images. Ignored during registration but added back at the end

Type
dictionary

Examples

Basic example using default parameters

```
>>> from valis import registration, data
>>> slide_src_dir = data.dcis_src_dir
>>> results_dst_dir = "./slide_registration_example"
>>> registered_slide_dst_dir = "./slide_registration_example/registered_slides"
```

Perform registration

```
>>> rigid_registrar, non_rigid_registrar, error_df = registrar.register()
```

View results in "./slide_registration_example". If they look good, warp and save the slides as ome.tiff

```
>>> registrar.warp_and_save_slides(registered_slide_dst_dir)
```

This example shows how to register CyCIF images and then merge to create a high dimensional ome.tiff slide

```
>>> registrar = registration.Valis(slide_src_dir, results_dst_dir)
>>> rigid_registrar, non_rigid_registrar, error_df = registrar.register()
```

Create function to get marker names from each slides' filename

```
>>> def cnames_from_filename(src_f):
...     f = valtils.get_name(src_f)
...     return ["DAPI"] + f.split(" ")[1:4]
...
>>> channel_name_dict = {f:cnames_from_filename(f) for f in  registrar.original_img_
↪list}
>>> merged_img, channel_names, ome_xml = registrar.warp_and_merge_slides(merged_
↪slide_dst_f, channel_name_dict=channel_name_dict)
```

View ome.tiff, located at merged_slide_dst_f

`__init__`(*src_dir*, *dst_dir*, *series=None*, *name=None*, *image_type=None*, *feature_detector_cls=<class 'valis.feature_detectors.VggFD'>*, *transformer_cls=<class 'skimage.transform._geometric.SimilarityTransform'>*, *affine_optimizer_cls=None*, *similarity_metric='n_matches'*, *matcher=<valis.feature_matcher.Matcher object>*, *imgs_ordered=False*, *non_rigid_registrar_cls=<class 'valis.non_rigid_registrars.OpticalFlowWarper'>*, *non_rigid_reg_params={}*, *compose_non_rigid=False*, *img_list=None*, *reference_img_f=None*, *align_to_reference=False*, *do_rigid=True*, *crop=None*, *create_masks=True*, *denoise_rigid=True*, *check_for_reflections=False*, *resolution_xyu=None*, *slide_dims_dict_wh=None*, *max_image_dim_px=850*, *max_processed_image_dim_px=850*, *max_non_rigid_registration_dim_px=850*, *thumbnail_size=500*, *norm_method='img_stats'*, *micro_rigid_registrar_cls=None*, *micro_rigid_registrar_params={}*, *qt_emitter=None*)

src_dir: str
Path to directory containing the slides that will be registered.

dst_dir
[str] Path to where the results should be saved.

name
[str, optional] Descriptive name of registrar, such as the sample's name

series
[int, optional] Slide series to that was read. If None, series will be set to 0.

image_type
[str, optional] The type of image, either “brightfield”, “fluorescence”, or “multi”. If None, VALIS will guess *image_type* of each image, based on the number of channels and datatype. Will assume that RGB = “brightfield”, otherwise *image_type* will be set to “fluorescence”.

feature_detector_cls
[FeatureDD, optional] Uninstantiated FeatureDD object that detects and computes image features. Default is VggFD. The available feature_detectors are found in the *feature_detectors* module. If a desired feature detector is not available, one can be created by subclassing *feature_detectors.FeatureDD*.

transformer_cls
[scikit-image Transform class, optional] Uninstantiated scikit-image transformer used to find transformation matrix that will warp each image to the target image. Default is SimilarityTransform

affine_optimizer_cls
[AffineOptimzer class, optional] Uninstantiated AffineOptimzer that will minimize a cost function to find the optimal affine transformations. If a desired affine optimization is not available, one can be created by subclassing *affine_optimizer.AffineOptimizer*.

similarity_metric
[str, optional] Metric used to calculate similarity between images, which is in turn used to build the distance matrix used to sort the images. Can be “n_matches”, or a string to used as distance in spatial.distance.cdist. “n_matches” is the number of matching features between image pairs.

match_filter_method: str, optional
“GMS” will use filter_matches_gms() to remove poor matches. This uses the Grid-based Motion Statistics (GMS) or RANSAC.

imgs_ordered
[bool, optional] Boolean defining whether or not the order of images in img_dir are already in the correct order. If True, then each filename should begin with the number that indicates its position in the z-stack. If False, then the images will be sorted by ordering a feature distance matix. Default is False.

reference_img_f
[str, optional] Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be the reference.

align_to_reference
[bool, optional] If *False*, images will be non-rigidly aligned serially towards the reference image. If *True*, images will be non-rigidly aligned directly to the reference image. If *reference_img_f* is None, then the reference image will be the one in the middle of the stack.

non_rigid_registrar_cls
[NonRigidRegistrar, optional] Uninstantiated NonRigidRegistrar class that will be used to calculate the deformation fields between images. See the *non_rigid_registrars* module for a desciption of available methods. If a desired non-rigid registration method is not available, one can be implemented by subclassing.NonRigidRegistrar. If None, then only rigid registration will be performed

non_rigid_reg_params: dictionary, optional
Dictionary containing key, value pairs to be used to initialize *non_rigid_registrar_cls*. In the case where simple ITK is used by the, params should be a SimpleITK.ParameterMap. Note that numeric values nedd to be converted to strings. See the NonRigidRegistrar classes in *non_rigid_registrars* for the available non-rigid registration methods and arguments.

compose_non_rigid
[bool, optional] Whether or not to compose non-rigid transformations. If *True*, then an image is non-

rigidly warped before aligning to the adjacent non-rigidly aligned image. This allows the transformations to accumulate, which may bring distant features together but could also result in un-wanted deformations, particularly around the edges. If *False*, the image not warped before being aaligned to the adjacent non-rigidly aligned image. This can reduce unwanted deformations, but may not bring distant features together.

img_list
[list, dictionary, optional] List of images to be registered. However, it can also be a dictionary, in which case the key: value pairs are full_path_to_image: name_of_image, where name_of_image is the key that can be used to access the image from Valis.slide_dict.

do_rigid: bool, dictionary, optional
Whether or not to perform rigid registration. If *False*, rigid registration will be skipped.

If *do_rigid* is a dictionary, it should contain inverse transformation matrices to rigidly align images to the specificed by *reference_img_f*. M will be estimated for images that are not in the dictionary. Each key is the filename of the image associated with the transformation matrix, and value is a dictionary containing the following values:

M
[(required) a 3x3 inverse transformation matrix as a numpy array.] Found by determining how to align fixed to moving. If *M* was found by determining how to align moving to fixed, then *M* will need to be inverted first.

transformation_src_shape_rc
[(optional) shape (row, col) of image used to find the rigid transformation.] If not provided, then it is assumed to be the shape of the level 0 slide

transformation_dst_shape_rc
[(optional) shape of registered image.] If not provided, this is assumed to be the shape of the level 0 reference slide.

crop: str, optional
How to crop the registered images. "overlap" will crop to include only areas where all images overlapped. "reference" crops to the area that overlaps with a reference image, defined by *reference_img_f*. This option can be used even if *reference_img_f* is *None* because the reference image will be set as the one at the center of the stack.

If both *crop* and *reference_img_f* are *None*, *crop* will be set to "overlap". If *crop* is None, but *reference_img_f* is defined, then *crop* will be set to "reference".

create_masks
[bool, optional] Whether or not to create and apply masks for registration. Can help focus alignment on the tissue, but can sometimes mask too much if there is a lot of variation in the image.

denoise_rigid
[bool, optional] Whether or not to denoise processed images before rigid regristion. Note that un-denoised images are used in the non-rigid registration

check_for_reflections
[bool, optional] Determine if alignments are improved by relfecting/mirroring/flipping images. Optional because it requires re-detecting features in each version of the images and then re-matching features, and so can be time consuming and not always necessary.

resolution_xyu: tuple, optional
Physical size per pixel and the unit. If None (the default), these values will be determined for each slide using the slides' metadata. If provided, this physical pixel sizes will be used for all of the slides. This option is available in case one cannot easily access to the original slides, but does have the information on pixel's physical units.

slide_dims_dict_wh
[dict, optional] Key= slide/image file name, value= dimensions = [(width, height), (width, height), . . .] for each level. If None (the default), the slide dimensions will be pulled from the slides' metadata. If provided, those values will be overwritten. This option is available in case one cannot easily access to the original slides, but does have the information on the slide dimensions.

max_image_dim_px
[int, optional] Maximum width or height of images that will be saved. This limit is mostly to keep memory in check.

max_processed_image_dim_px
[int, optional] Maximum width or height of processed images. An important parameter, as it determines the size of of the image in which features will be detected and displacement fields computed.

max_non_rigid_registration_dim_px
[int, optional] Maximum width or height of images used for non-rigid registration. Larger values may yeild more accurate results, at the expense of speed and memory. There is also a practical limit, as the specified size may be too large to fit in memory.

mask_dict
[dictionary] Dictionary where key = overlap type (all, overlap, or reference), and value = (mask, mask_bbox_xywh)

thumbnail_size
[int, optional] Maximum width or height of thumbnails that show results

norm_method
[str] Name of method used to normalize the processed images. Options are None when normalization is not desired, "histo_match" for histogram matching and "img_stats" for normalizing by image statistics. See preprocessing.match_histograms and preprocessing.norm_khan for details.

iter_order
[list of tuples] Each element of *iter_order* contains a tuple of stack indices. The first value is the index of the moving/current/from image, while the second value is the index of the moving/next/to image.

micro_rigid_registrar_cls
[MicroRigidRegistrar, optional] Class used to perform higher resolution rigid registration. If *None*, this step is skipped.

micro_rigid_registrar_params
[dictionary] Dictionary of keyword arguments used intialize the *MicroRigidRegistrar*

qt_emitter
[PySide2.QtCore.Signal, optional] Used to emit signals that update the GUI's progress bars

`get_slide`(*src_f*)

Get Slide

Get the Slide associated with *src_f*. Slide store registration parameters and other metadata about the slide associated with *src_f*. Slide can also:

- Convert the slide to a numpy array (Slide.slide2image)
- Convert the slide to a pyvips.Image (Slide.slide2vips)
- Warp the slide (Slide.warp_slide)
- Save the warped slide as an ome.tiff (Slide.warp_and_save_slide)
- Warp an image of the slide (Slide.warp_img)
- Warp points (Slide.warp_xy)

- Warp points in one slide to their position in another unwarped slide (Slide.warp_xy_from_to)
- Access slide ome-xml (Slide.original_xml)

See Slide for more details.

Parameters
`src_f` (*str*) – Path to the slide, or name assigned to slide (see Valis.name_dict)

Returns
slide_obj – Slide associated with src_f

Return type
Slide

`register`(*brightfield_processing_cls=<class 'valis.preprocessing.ColorfulStandardizer'>, brightfield_processing_kwargs={'c': 0.2, 'h': 0}, if_processing_cls=<class 'valis.preprocessing.ChannelGetter'>, if_processing_kwargs={'adaptive_eq': True, 'channel': 'dapi'}, processor_dict=None, reader_cls=None, reader_dict=None*)

Register a collection of images

This function will convert the slides to images, pre-process and normalize them, and then conduct rigid registration. Non-rigid registration will then be performed if the *non_rigid_registrar_cls* argument used to initialize the Valis object was not None.

In addition to the objects returned, the desination directory (i.e. *dst_dir*) will contain thumbnails so that one can visualize the results: converted image thumbnails will be in "images/"; processed images in "processed/"; rigidly aligned images in "rigid_registration/"; non-rigidly aligned images in "non_rigid_registration/"; non-rigid deformation field images (i.e. warped grids colored by the direction and magntidue) of the deformation) will be in ""deformation_fields/". The size of these thumbnails is determined by the *thumbnail_size* argument used to initialze this object.

One can get a sense of how well the registration worked by looking in the "overlaps/", which shows how the images overlap before registration, after rigid registration, and after non-rigid registration. Each image is created by coloring an inverted greyscale version of the processed images, and then blending those images.

The "data/" directory will contain a pickled copy of this registrar, which can be later be opened (unpickled) and used to warp slides and/or point data.

"data/" will also contain the *summary_df* saved as a csv file.

Parameters

- **`brightfield_processing_cls`** (`preprocessing.ImageProcesser`) – preprocessing.ImageProcesser used to pre-process brightfield images to make them look as similar as possible.
- **`brightfield_processing_kwargs`** (*dict*) – Dictionary of keyward arguments to be passed to *brightfield_processing_cls*
- **`if_processing_cls`** (`preprocessing.ImageProcesser`) – preprocessing.ImageProcesser used to pre-process immunofluorescent images to make them look as similar as possible.
- **`if_processing_kwargs`** (*dict*) – Dictionary of keyward arguments to be passed to *if_processing_cls*
- **`processor_dict`** (*dict, optional*) – Each key should be the filename of the image, and the value either a subclassed preprocessing.ImageProcessor, or a list, where the 1st element is the processor, and the second element a dictionary of keyword arguments passed to the processor. If *None*, then a default processor will be used for each image based on the inferred modality.

- **reader_cls** (SlideReader, *optional*) – Uninstantiated SlideReader class that will convert the slide to an image, and also collect metadata. If None (the default), the appropriate SlideReader will be found by *slide_io.get_slide_reader*. This option is provided in case the slides cannot be opened by a current SlideReader class. In this case, the user should create a subclass of SlideReader. See slide_io.SlideReader for details.
- **reader_dict** (dict, *optional*) – Dictionary specifying which readers to use for individual images. The keys should be the image's filename, and the values the instantiated slide_io.SlideReader to use to read that file. Valis will try to find an appropritate reader for any omitted files, or will use *reader_cls* as the default.

Returns

- **rigid_registrar** (*SerialRigidRegistrar*) – SerialRigidRegistrar object that performed the rigid registration. This object can be pickled if so desired
- **non_rigid_registrar** (*SerialNonRigidRegistrar*) – SerialNonRigidRegistrar object that performed serial non-rigid registration. This object can be pickled if so desired.
- **summary_df** (*Dataframe*) – *summary_df* contains various information about the registration.

  The "from" column is the name of the image, while the "to" column name of the image it was aligned to. "from" is analagous to "moving" or "current", while "to" is analgous to "fixed" or "previous".

  Columns begining with "original" refer to error measurements of the unregistered images. Those beginning with "rigid" or "non_rigid" refer to measurements related to rigid or non-rigid registration, respectively.

  Columns beginning with "mean" are averages of error measurements. In the case of errors based on feature distances (i.e. those ending in "D"), the mean is weighted by the number of feature matches between "from" and "to".

  Columns endining in "D" indicate the median distance between matched features in "from" and "to".

  Columns ending in "TRE" indicate the target registration error between "from" and "to".

  Columns ending in "mattesMI" contain measurements of the Mattes mutual information between "from" and "to".

  "processed_img_shape" indicates the shape (row, column) of the processed image actually used to conduct the registration

  "shape" is the shape of the slide at full resolution

  "aligned_shape" is the shape of the registered full resolution slide

  "physical_units" are the names of the pixels physcial unit, e.g. u'µm'

  "resolution" is the physical unit per pixel

  "name" is the name assigned to the Valis instance

  "rigid_time_minutes" is the total number of minutes it took to convert the images and then rigidly align them.

  "non_rigid_time_minutes" is the total number of minutes it took to convert the images, and then perform rigid -> non-rigid registration.

warp_and_merge_slides(*dst_f=None*, *level=0*, *non_rigid=True*, *crop=True*, *channel_name_dict=None*, *src_f_list=None*, *colormap='auto'*, *drop_duplicates=True*, *tile_wh=None*, *interp_method='bicubic'*, *compression='lzw'*, *Q=100*, *pyramid=True*)

Warp and merge registered slides

Parameters

- **dst_f** (*str, optional*) – Path to were the warped slide will be saved. If None, then the slides will be merged but not saved.
- **level** (*int, optional*) – Pyramid level to be warped. Default is 0, which means the highest resolution image will be warped and saved.
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied. Default is True
- **crop** (*bool, str*) – How to crop the registered images. If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initialzing the *Valis object*.
- **channel_name_dict** (*dict of lists, optional.*) – key = slide file name, value = list of channel names for that slide. If None, the the channel names found in each slide will be used.
- **src_f_list** (*list of str, optionaal*) – List of paths to slide to be warped. If None (the default), Valis.original_img_list will be used. Otherwise, the paths to which *src_f_list* points to should be an alternative copy of the slides, such as ones that have undergone processing (e.g. stain segmentation), had a mask applied, etc…
- **colormap** (*list*) – List of RGB colors (0-255) to use for channel colors
- **drop_duplicates** (*bool, optional*) – Whether or not to drop duplicate channels that might be found in multiple slides. For example, if DAPI is in multiple slides, then the only the DAPI channel in the first slide will be kept.
- **tile_wh** (*int, optional*) – Tile width and height used to save image
- **interp_method** (*str*) – Interpolation method used when warping slide. Default is "bicubic"
- **compression** (*str*) – Compression method used to save ome.tiff . Default is lzw, but can also be jpeg or jp2k. See pyips for more details.
- **Q** (*int*) – Q factor for lossy compression
- **pyramid** (*bool*) – Whether or not to save an image pyramid.

Returns

- **merged_slide** (*pyvips.Image*) – Image with all channels merged. If *drop_duplicates* is True, then this will only contain unique channels.
- **all_channel_names** (*list of str*) – Name of each channel in the image
- **ome_xml** (*str*) – OME-XML string containing the slide's metadata

Slide

`class valis.registration.Slide(`*src_f*, *image*, *val_obj*, *reader*, *name=None*`)`

Stores registration info and warps slides/points

Slide is a class that stores registration parameters and other metadata about a slide. Once registration has been completed, *Slide* is also able warp the slide and/or points using the same registration parameters. Warped slides can be saved as ome.tiff images with valid ome-xml.

`src_f`

Path to slide.

Type
str

`image`

Image to registered. Taken from a level in the image pyramid. However, image may be resized to fit within the *max_image_dim_px* argument specified when creating a *Valis* object.

Type
ndarray

`val_obj`

The "parent" object that registers all of the slide.

Type
Valis

`reader`

Object that can read slides and collect metadata.

Type
SlideReader

`original_xml`

Xml string created by bio-formats

Type
str

`img_type`

Whether the image is "brightfield" or "fluorescence"

Type
str

`is_rgb`

Whether or not the slide is RGB.

Type
bool

`slide_shape_rc`

Dimensions of the largest resolution in the slide, in the form of (row, col).

Type
tuple of int

`series`
Slide series to be read

Type
int

`slide_dimensions_wh`
Dimensions of all images in the pyramid (width, height).

Type
ndarray

`resolution`
Physical size of each pixel.

Type
float

`units`
Physical unit of each pixel.

Type
str

`name`
Name of the image. Usually *img_f* but with the extension removed.

Type
str

`processed_img`
Image used to perform registration

Type
ndarray

`rigid_reg_mask`
Mask of convex hulls covering tissue in unregistered image. Could be used to mask *processed_img* before rigid registration

Type
ndarray

`non_rigid_reg_mask`
Created by combining rigidly warped *rigid_reg_mask* in all other slides.

Type
ndarray

`stack_idx`
Position of image in sorted Z-stack

Type
int

`processed_img_f`
Path to thumbnail of the processed *image*.

Type
str

`rigid_reg_img_f`

Path to thumbnail of rigidly aligned *image*.

Type
str

`non_rigid_reg_img_f`

Path to thumbnail of non-rigidly aligned *image*.

Type
str

`processed_img_shape_rc`

Shape (row, col) of the processed image used to find the transformation parameters. Maximum dimension will be less or equal to the *max_processed_image_dim_px* specified when creating a *Valis* object. As such, this may be smaller than the image's shape.

Type
tuple of int

`aligned_slide_shape_rc`

Shape (row, col) of aligned slide, based on the dimensions in the 0th level of they pyramid. In

Type
tuple of int

`reg_img_shape_rc`

Shape (row, col) of the registered image

Type
tuple of int

`M`

Rigid transformation matrix that aligns *image* to the previous image in the stack. Found using the processed copy of *image*.

Type
ndarray

`bk_dxdy`

(2, N, M) numpy array of pixel displacements in the x and y directions. dx = bk_dxdy[0], and dy=bk_dxdy[1]. Used to warp images. Found using the rigidly aligned version of the processed image.

Type
ndarray

`fwd_dxdy`

Inverse of *bk_dxdy*. Used to warp points.

Type
ndarray

`_bk_dxdy_f`

Path to file containing bk_dxdy, if saved

Type
str

`_fwd_dxdy_f`

Path to file containing fwd_dxdy, if saved

Type
str

`_bk_dxdy_np`

bk_dxdy as a numpy array. Only not None if *bk_dxdy* becomes associated with a file

Type
ndarray

`_fwd_dxdy_np`

fwd_dxdy as a numpy array. Only not None if *fwd_dxdy* becomes associated with a file

Type
ndarray

`stored_dxdy`

Whether or not the non-rigid displacements are saved in a file Should only occur if image is very large.

Type
bool

`fixed_slide`

Slide object to which this one was aligned.

Type
Slide

`xy_matched_to_prev`

Coordinates (x, y) of features in *image* that had matches in the previous image. Will have shape (N, 2)

Type
ndarray

`xy_in_prev`

Coordinates (x, y) of features in the previous that had matches to those in *image*. Will have shape (N, 2)

Type
ndarray

`xy_matched_to_prev_in_bbox`

Subset of *xy_matched_to_prev* that were within *overlap_mask_bbox_xywh*. Will either have shape (N, 2) or (M, 2), with M < N.

Type
ndarray

`xy_in_prev_in_bbox`

Subset of *xy_in_prev* that were within *overlap_mask_bbox_xywh*. Will either have shape (N, 2) or (M, 2), with M < N.

Type
ndarray

`crop`

Crop method

Type
str

bg_px_pos_rc

Position of pixel that has the background color

Type
tuple

bg_color

Color of background pixels

Type
list, optional

is_empty

True if the image is empty (i.e. contains only 1 value)

Type
bool

__init__(*src_f*, *image*, *val_obj*, *reader*, *name=None*)

Parameters

- **src_f** (*str*) – Path to slide.
- **image** (*ndarray*) – Image to registered. Taken from a level in the image pyramid. However, image may be resized to fit within the *max_image_dim_px* argument specified when creating a *Valis* object.
- **val_obj** (Valis) – The "parent" object that registers all of the slide.
- **reader** (SlideReader) – Object that can read slides and collect metadata.
- **name** (*str, optional*) – Name of slide. If None, it will be *src_f* with the extension removed

slide2image(*level*, *series=None*, *xywh=None*)

Convert slide to image

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int, optional*) – Series number. Defaults to 0
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
img – An image of the slide or the region defined by xywh

Return type
ndarray

slide2vips(*level*, *series=None*, *xywh=None*)

Convert slide to pyvips.Image

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int, optional*) – Series number. Defaults to 0

- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
vips_slide – An of the slide or the region defined by xywh

Return type
pyvips.Image

warp_and_save_slide(*dst_f*, *level=0*, *non_rigid=True*, *crop=True*, *src_f=None*, *channel_names=None*, *colormap='auto'*, *interp_method='bicubic'*, *tile_wh=None*, *compression='lzw'*, *Q=100*, *pyramid=True*, *reader=None*)

Warp and save a slide

Slides will be saved in the ome.tiff format.

Parameters

- **dst_f** (*str*) – Path to were the warped slide will be saved.
- **level** (*int*) – Pyramid level to be warped
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied. Default is True
- **crop** (*bool, str*) – How to crop the registered images. If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initializing the *Valis object*.
- **channel_names** (*list, optional*) – List of channel names. If None, then Slide.reader will attempt to find the channel names associated with *src_f*.
- **colormap** (*dict, optional*) – Dictionary of channel colors, where the key is the channel name, and the value the color as rgb255. If None (default), the channel colors from *current_ome_xml_str* will be used, if available. If None, and there are no channel colors in the *current_ome_xml_str*, then no colors will be added
- **src_f** (*str, optional*) – Path of slide to be warped. If None (the default), Slide.src_f will be used. Otherwise, the file to which *src_f* points to should be an alternative copy of the slide, such as one that has undergone processing (e.g. stain segmentation), has a mask applied, etc…
- **interp_method** (*str*) – Interpolation method used when warping slide. Default is "bicubic"
- **tile_wh** (*int, optional*) – Tile width and height used to save image
- **compression** (*str*) – Compression method used to save ome.tiff . Default is lzw, but can also be jpeg or jp2k. See pyips for more details.
- **Q** (*int*) – Q factor for lossy compression
- **pyramid** (*bool*) – Whether or not to save an image pyramid.

warp_geojson(*geojson_f*, *M=None*, *slide_level=0*, *pt_level=0*, *non_rigid=True*, *crop=True*)

Warp geometry using registration parameters

Warps geometries to their location in the registered slide/image

Parameters

- **geojson_f** (*str*) – Path to geojson file containing the annotation geometries. Assumes coordinates are in pixels.
- **slide_level** (*int, tuple, optional*) – Pyramid level of the slide. Used to scale transformation matrices. Can also be the shape of the warped image (row, col) into which the points should be warped. Default is 0.
- **pt_level** (*int, tuple, optional*) – Pyramid level from which the points origingated. For example, if *xy* are from the centroids of cell segmentation performed on the full resolution image, this should be 0. Alternatively, the value can be a tuple of the image's shape (row, col) from which the points came. For example, if *xy* are bounding box coordinates from an analysis on a lower resolution image, then pt_level is that lower resolution image's shape (row, col). Default is 0.
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied. Default is True.
- **crop** (*bool, str*) – Apply crop to warped points by shifting points to the mask's origin. Note that this can result in negative coordinates, but might be useful if wanting to draw the coordinates on the registered slide, such as annotation coordinates.

  If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initialzing the *Valis object*.

warp_geojson_from_to(*geojson_f*, *to_slide_obj*, *src_slide_level=0*, *src_pt_level=0*, *dst_slide_level=0*, *non_rigid=True*)

Warp geoms in geojson file from annotation slide to another unwarped slide

Takes a set of geometries found in this annotation slide, and warps them to their position in the unwarped "to" slide.

Parameters

- **geojson_f** (*str*) – Path to geojson file containing the annotation geometries. Assumes coordinates are in pixels.
- **to_slide_obj** (Slide) – Slide to which the points will be warped. I.e. *xy* will be warped from this Slide to their position in the unwarped slide associated with *to_slide_obj*.
- **src_pt_level** (*int, tuple, optional*) – Pyramid level of the slide/image in which *xy* originated. For example, if *xy* are from the centroids of cell segmentation performed on the unwarped full resolution image, this should be 0. Alternatively, the value can be a tuple of the image's shape (row, col) from which the points came. For example, if *xy* are bounding box coordinates from an analysis on a lower resolution image, then pt_level is that lower resolution image's shape (row, col).
- **dst_slide_level** (*int, tuple, optional*) – Pyramid level of the slide/image in to *xy* will be warped. Similar to *src_pt_level*, if *dst_slide_level* is an int then the points will be warped to that pyramid level. If *dst_slide_level* is the "to" image's shape (row, col), then the points will be warped to their location in an image with that same shape.
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied.

Returns

warped_geojson – Dictionry of warped geojson geometries

Return type

dict

warp_img(*img=None*, *non_rigid=True*, *crop=True*, *interp_method='bicubic'*)

Warp an image using the registration parameters

img
[ndarray, optional] The image to be warped. If None, then Slide.image will be warped.

non_rigid
[bool] Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied.

crop: bool, str
How to crop the registered images. If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initialzing the *Valis object*.

interp_method
[str] Interpolation method used when warping slide. Default is "bicubic"

Returns
warped_img – Warped copy of *img*

Return type
ndarray

warp_slide(*level*, *non_rigid=True*, *crop=True*, *src_f=None*, *interp_method='bicubic'*, *reader=None*)

Warp a slide using registration parameters

Parameters

- **level** (*int*) – Pyramid level to be warped
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied. Default is True
- **crop** (*bool, str*) – How to crop the registered images. If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initialzing the *Valis object*.
- **src_f** (*str, optional*) – Path of slide to be warped. If None (the default), Slide.src_f will be used. Otherwise, the file to which *src_f* points to should be an alternative copy of the slide, such as one that has undergone processing (e.g. stain segmentation), has a mask applied, etc…
- **interp_method** (*str*) – Interpolation method used when warping slide. Default is "bicubic"

warp_xy(*xy*, *M=None*, *slide_level=0*, *pt_level=0*, *non_rigid=True*, *crop=True*)

Warp points using registration parameters

Warps *xy* to their location in the registered slide/image

Parameters

- **xy** (*ndarray*) – (N, 2) array of points to be warped. Must be x,y coordinates
- **slide_level** (*int, tuple, optional*) – Pyramid level of the slide. Used to scale transformation matrices. Can also be the shape of the warped image (row, col) into which the points should be warped. Default is 0.

- **pt_level** (*int, tuple, optional*) – Pyramid level from which the points origingated. For example, if *xy* are from the centroids of cell segmentation performed on the full resolution image, this should be 0. Alternatively, the value can be a tuple of the image's shape (row, col) from which the points came. For example, if *xy* are bounding box coordinates from an analysis on a lower resolution image, then pt_level is that lower resolution image's shape (row, col). Default is 0.
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied. Default is True.
- **crop** (*bool, str*) – Apply crop to warped points by shifting points to the mask's origin. Note that this can result in negative coordinates, but might be useful if wanting to draw the coordinates on the registered slide, such as annotation coordinates.

  If *True*, then the same crop used when initializing the *Valis* object will be used. If *False*, the image will not be cropped. If "overlap", the warped slide will be cropped to include only areas where all images overlapped. "reference" crops to the area that overlaps with the reference image, defined by *reference_img_f* when initialzing the *Valis object*.

warp_xy_from_to(*xy*, *to_slide_obj*, *src_slide_level=0*, *src_pt_level=0*, *dst_slide_level=0*, *non_rigid=True*)

Warp points from this slide to another unwarped slide

Takes a set of points found in this unwarped slide, and warps them to their position in the unwarped "to" slide.

Parameters

- **xy** (*ndarray*) – (N, 2) array of points to be warped. Must be x,y coordinates
- **to_slide_obj** (Slide) – Slide to which the points will be warped. I.e. *xy* will be warped from this Slide to their position in the unwarped slide associated with *to_slide_obj*.
- **src_pt_level** (*int, tuple, optional*) – Pyramid level of the slide/image in which *xy* originated. For example, if *xy* are from the centroids of cell segmentation performed on the unwarped full resolution image, this should be 0. Alternatively, the value can be a tuple of the image's shape (row, col) from which the points came. For example, if *xy* are bounding box coordinates from an analysis on a lower resolution image, then pt_level is that lower resolution image's shape (row, col).
- **dst_slide_level** (*int, tuple, optional*) – Pyramid level of the slide/image in to *xy* will be warped. Similar to *src_pt_level*, if *dst_slide_level* is an int then the points will be warped to that pyramid level. If *dst_slide_level* is the "to" image's shape (row, col), then the points will be warped to their location in an image with that same shape.
- **non_rigid** (*bool, optional*) – Whether or not to conduct non-rigid warping. If False, then only a rigid transformation will be applied.

1.1.5 Slide I/O

Functions

Methods and classes to read and write slides in the .ome.tiff format

valis.slide_io.**convert_to_ome_tiff**(*src_f*, *dst_f*, *level*, *series=None*, *xywh=None*, *colormap='auto'*, *tile_wh=None*, *compression='lzw'*, *Q=100*, *pyramid=True*, *reader=None*)

Convert an image to an ome.tiff image

Saves a new copy of the image as a tiled pyramid ome.tiff with valid ome-xml. Uses pyvips to save the image. Currently only writes a single series.

Parameters

- **src_f** (*str*) – Path to image to be converted
- **dst_f** (*str*) – Path indicating where the image should be saved.
- **level** (*int*) – Pyramid level to be converted.
- **series** (*str*) – Series to be converted.
- **xywh** (*tuple of int, optional*) – The region of the slide to be converted. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.
- **colormap** (*dict, optional*) – Dictionary of channel colors, where the key is the channel name, and the value the color as rgb255. If None (default), the channel colors from *current_ome_xml_str* will be used, if available. If None, and there are no channel colors in the *current_ome_xml_str*, then no colors will be added
- **tile_wh** (*int*) – Tile shape used to save the image. Used to create a square tile, so *tile_wh* is both the width and height.
- **compression** (*str*) – Compression method used to save ome.tiff . Default is lzw, but can also be jpeg or jp2k. See pyips for more details.
- **Q** (*int*) – Q factor for lossy compression
- **pyramid** (*bool*) – Whether or not to save an image pyramid.

valis.slide_io.**create_ome_xml**(*shape_xyzct*, *bf_dtype*, *is_rgb*, *pixel_physical_size_xyu=None*, *channel_names=None*, *colormap='auto'*)

Create new ome-xmml object

Parameters

- **shape_xyzct** (*tuple of int*) – XYZCT shape of image
- **bf_dtype** (*str*) – String format of Bioformats datatype
- **is_rgb** (*bool*) – Whether or not the image is RGB
- **pixel_physical_size_xyu** (*tuple, optional*) – Physical size per pixel and the unit.
- **channel_names** (*list, optional*) – List of channel names.
- **colormap** (*dict, str, optional*) – Dictionary of channel colors, where the key is the channel name, and the value the color as rgb255. If "auto" (default), the channel colors from *current_ome_xml_str* will be used, if available. If *None*, channel colors will not be assigned.

Returns

new_ome – ome_types.model.OME object containing ome-xml metadata

Return type

ome_types.model.OME

valis.slide_io.**get_slide_reader**(*src_f*, *series=None*)

Get appropriate SlideReader

If a slide can be read by openslide and bioformats, VipsSlideReader will be used because it can be opened as a pyvips.Image.

Parameters

- **src_f** (*str*) – Path to slide
- **series** (*int, optional*) – The series to be read. If *series* is None, the the *series* will be set to the series associated with the largest image. In cases where there is only 1 image in the file, *series* will be 0.

Returns
reader – SlideReader class that can read the slide and and convert them to images or pyvips.Images at the specified level and series. They also contain a *MetaData* object that contains information about the slide, like dimensions at each level, physical units, etc…

Return type
SlideReader

Notes

pyvips will be used to open ome-tiff images when *series* is 0

`valis.slide_io.`**`init_jvm`**(*jar=None*, *mem_gb=10*)

Initialize JVM for BioFormats

Parameters
mem_gb (*int*) – Amount of memory, in GB, for JVM

`valis.slide_io.`**`kill_jvm`**()

Kill JVM for BioFormats

`valis.slide_io.`**`save_ome_tiff`**(*img*, *dst_f*, *ome_xml=None*, *tile_wh=1024*, *compression='lzw'*, *Q=100*, *pyramid=True*)

Save an image in the ome.tiff format using pyvips

Parameters

- **img** (*pyvips.Image, ndarray*) – Image to be saved. If a numpy array is provided, it will be converted to a pyvips.Image.
- **ome_xml** (*str, optional*) – ome-xml string describing image's metadata. If None, it will be createdd
- **tile_wh** (*int*) – Tile shape used to save *img*. Used to create a square tile, so *tile_wh* is both the width and height.
- **compression** (*str*) – Compression method used to save ome.tiff . Default is lzw, but can also be jpeg or jp2k. See pyips for more details.
- **Q** (*int*) – Q factor for lossy compression
- **pyramid** (*bool*) – Whether or not to save an image pyramid.

`valis.slide_io.`**`update_xml_for_new_img`**(*img*, *reader*, *level=0*, *channel_names=None*, *colormap='auto'*)

Update dimensions ome-xml metadata

Used to create a new ome-xmlthat reflects changes in an image, such as its shape

If *current_ome_xml_str* is invalid or None, a new ome-xml will be created

Parameters

- **img** (*ndarry or pyvips.Image*) – Image for which xml will be generated. Used to determine shape and datatype.

- **reader** (SlideReader) – SlideReader used to open *img*. Will use this to extract other metadata, including the original xml.
- **channel_names** (*list, optional*) – List of channel names.
- **colormap** (*dict, optional*) – Dictionary of channel colors, where the key is the channel name, and the value the color as rgb255. If "auto" (the default), the channel colors from *current_ome_xml_str* will be used, if available. If None, and there are no channel colors in the *current_ome_xml_str*, then no colors will be added

Returns
new_ome – ome_types.model.OME object containing ome-xml metadata

Return type
ome_types.model.OME

Classes

MetaData

`class valis.slide_io.MetaData(name, server, series=0)`

Store slide metadata

To be filled in by a SlideReader object

name
Name of slide.

Type
str

series
Series number.

Type
int

server
String indicating what was used to read the metadata.

Type
str

slide_dimensions
Dimensions of all images in the pyramid (width, height).

is_rgb
Whether or not the image is RGB.

Type
bool

pixel_physical_size_xyu
Physical size per pixel and the unit.

channel_names
List of channel names. None if image is RGB

Type
list

n_channels

Number of channels.

Type

int

original_xml

Xml string created by bio-formats

Type

str

bf_datatype

String indicating bioformats image datatype

Type

str

optimal_tile_wh

Tile width and height used to open and/or save image

Type

int

__init__(*name*, *server*, *series=0*)

Parameters

- **name** (*str*) – Name of slide.
- **server** (*str, optional*) – String indicating what was used to read the metadata.
- **series** (*int, optional*) – Series number.

SlideReader

class valis.slide_io.**SlideReader**(*src_f*, **args*, ***kwargs*)

Read slides and get metadata

slide_f

Path to slide

Type

str

metadata

MetaData containing some basic metadata about the slide

Type

MetaData

series

Image series

Type

int

`__init__`(*src_f*, **args*, ***kwargs*)

Parameters
: **`src_f`** (*`str`*) – Path to slide

`create_metadata`()

Create and fill in a MetaData object

Returns
: **metadata** – MetaData object containing metadata about slide

Return type
: *MetaData*

`slide2image`(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to image

Parameters

- **`level`** (*`int`*) – Pyramid level
- **`xywh`** (*`tuple of int, optional`*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
: **img** – An image of the slide or the region defined by xywh

Return type
: ndarray

`slide2vips`(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to pyvips.Image

Parameters

- **`level`** (*`int`*) – Pyramid level
- **`xywh`** (*`tuple of int, optional`*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
: **vips_slide** – An of the slide or the region defined by xywh

Return type
: pyvips.Image

BioFormatsSlideReader

`class valis.slide_io.`**`BioFormatsSlideReader`**(*src_f*, *series=None*, **args*, ***kwargs*)

Bases: *`SlideReader`*

Read slides using BioFormats

Uses the packages jpype and bioformats-jar

`create_metadata`()

Create and fill in a MetaData object

Returns
: **metadata** – MetaData object containing metadata about slide

Return type
MetaData

`get_channel`(*level*, *series*, *channel*)
Get channel from slide

Parameters

- **`level`** (*`int`*) – Pyramid level
- **`series`** (*`int`*) – Series number
- **`channel`** (*`str, int`*) – Either the name of the channel (string), or the index of the channel (int)

Returns
img_channel – Specified channel sliced from the slide/image

Return type
ndarray

`get_tiles_parallel`(*level*, *tile_bbox_list*, *pixel_type*, *series=0*, *z=0*, *t=0*)
Get tiles to slice from the slide

`scale_physical_size`(*level*)
Get resolution pyramid level

Scale resolution to be for requested pyramid level

Parameters
`level` (*`int`*) – Pyramid level

Returns
level_xy_per_px

Return type
tuple

`property series`
Slide series

`slide2image`(*level*, *series=None*, *xywh=None*, *z=0*, *t=0*, **args*, ***kwargs*)
Convert slide to image

Parameters

- **`level`** (*`int`*) – Pyramid level
- **`series`** (*`int, optional`*) – Series number. Defaults to 1
- **`xywh`** (*`tuple of int, optional`*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
img – An image of the slide or the region defined by xywh

Return type
ndarray

`slide2vips`(*level*, *series=None*, *xywh=None*, *tile_wh=None*, *z=0*, *t=0*, **args*, ***kwargs*)
Convert slide to pyvips.Image

This method uses Bioformats to slice tiles from the slides, and then stitch them together using pyvips.

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int, optional*) – Series number. Defaults to 0
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.
- **tile_wh** (*int, optional*) – Size of tiles used to contstruct *vips_slide*

Returns
vips_slide – An of the slide or the region defined by xywh

Return type
pyvips.Image

VipsSlideReader

class valis.slide_io.**VipsSlideReader**(*src_f*, **args*, ***kwargs*)

Bases: *SlideReader*

Read slides using pyvips Pyvips includes OpenSlide and so can read those formats as well.

use_openslide

Whether or not openslide can be used to read this slide.

Type
bool

is_ome

Whether ot not the side is an ome.tiff.

Type
bool

Notes

When using openslide, lower levels can only be read without distortion, if pixman version 0.40.0 is installed. As of Oct 7, 2021, Macports only has pixman version 0.38, which produces distorted lower level images. If using macports may need to install from source do "./configure –prefix=/opt/local/" when installing from source.

create_metadata()

Create and fill in a MetaData object

Returns
metadata – MetaData object containing metadata about slide

Return type
MetaData

get_channel(*level*, *series*, *channel*)

Get channel from slide

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int*) – Series number

- **channel** (*str, int*) – Either the name of the channel (string), or the index of the channel (int)

Returns
img_channel – Specified channel sliced from the slide/image

Return type
ndarray

`scale_physical_size`(*level*)

Get resolution pyramid level

Scale resolution to be for requested pyramid level

Parameters
level (*int*) – Pyramid level

Returns
level_xy_per_px

Return type
tuple

`slide2image`(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to image

Parameters

- **level** (*int*) – Pyramid level.
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
img – An image of the slide or the region defined by xywh

Return type
ndarray

`slide2vips`(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to pyvips.Image

Parameters

- **level** (*int*) – Pyramid level
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
vips_slide – An of the slide or the region defined by xywh

Return type
pyvips.Image

FlattenedPyramidReader

class valis.slide_io.**FlattenedPyramidReader**(*src_f*, **args*, ***kwargs*)

Bases: *VipsSlideReader*

Read flattened pyramid using pyvips Read slide pyramids where each page/plane is a channel in the pyramid. An example would be one where the plane dimensions are something like [(600, 600), (600, 600), (600, 600), (300, 300), (300, 300), (300, 300)] for a 3 channel image with 2 pyramid levels. It seems that bioformats does not recognize these as pyramid images.

create_metadata()

Create and fill in a MetaData object

Returns
metadata – MetaData object containing metadata about slide

Return type
MetaData

get_channel(*level*, *series*, *channel*)

Get channel from slide

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int*) – Series number
- **channel** (*str, int*) – Either the name of the channel (string), or the index of the channel (int)

Returns
img_channel – Specified channel sliced from the slide/image

Return type
ndarray

scale_physical_size(*level*)

Get resolution pyramid level

Scale resolution to be for requested pyramid level

Parameters
level (*int*) – Pyramid level

Returns
level_xy_per_px

Return type
tuple

slide2image(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to image

Parameters

- **level** (*int*) – Pyramid level.
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
img – An image of the slide or the region defined by xywh

Return type
ndarray

`slide2vips`(*level*, *xywh=None*, **args*, ***kwargs*)

Convert slide to pyvips.Image

Parameters

- **level** (*int*) – Pyramid level
- **xywh** (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
vips_slide – An of the slide or the region defined by xywh

Return type
pyvips.Image

ImageReader

`class valis.slide_io.ImageReader`(*src_f*, **args*, ***kwargs*)

Bases: *SlideReader*

Read image using scikit-image

`create_metadata`()

Create and fill in a MetaData object

Returns
metadata – MetaData object containing metadata about slide

Return type
MetaData

`get_channel`(*level*, *series*, *channel*)

Get channel from slide

Parameters

- **level** (*int*) – Pyramid level
- **series** (*int*) – Series number
- **channel** (*str, int*) – Either the name of the channel (string), or the index of the channel (int)

Returns
img_channel – Specified channel sliced from the slide/image

Return type
ndarray

`scale_physical_size`(*level*)

Get resolution pyramid level

Scale resolution to be for requested pyramid level

Parameters
`level` (*int*) – Pyramid level

Returns
level_xy_per_px

Return type
tuple

`slide2image`(*xywh=None*, **args*, ***kwargs*)

Convert slide to image

Parameters

- `level` (*int*) – Pyramid level
- `xywh` (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
img – An image of the slide or the region defined by xywh

Return type
ndarray

`slide2vips`(*xywh=None*, **args*, ***kwargs*)

Convert slide to pyvips.Image

Parameters

- `level` (*int*) – Pyramid level
- `xywh` (*tuple of int, optional*) – The region to be sliced from the slide. If None, then the entire slide will be converted. Otherwise xywh is the (top left x, top left y, width, height) of the region to be sliced.

Returns
vips_slide – An of the slide or the region defined by xywh

Return type
pyvips.Image

1.1.6 Image pre-processing

Functions

Collection of pre-processing methods for aligning images

`valis.preprocessing.create_tissue_mask_from_multichannel`(*img*, *kernel_size=3*)

Get foreground of multichannel imaage

`valis.preprocessing.create_tissue_mask_from_rgb`(*img*, *brightness_q=0.99*, *kernel_size=3*, *gray_thresh=0.075*, *light_gray_thresh=0.875*, *dark_gray_thresh=0.7*)

Create mask that only covers tissue

Also remove dark regions on the edge of the slide, which could be artifacts

Parameters

- **grey_thresh** (*float*) – Colorfulness values (from JCH) below this are considered "grey", and thus possibly dirt, hair, coverslip edges, etc…
- **light_gray_thresh** (*float*) – Upper limit for light gray
- **dark_gray_thresh** (*float*) – Upper limit for dark gray

Returns

- **tissue_mask** (*ndarray*) – Mask covering tissue
- **concave_tissue_mask** (*ndarray*) – Similar to *tissue_mask*, but each region is replaced by a concave hull. Covers more area

`valis.preprocessing.`**`get_luminosity`**`(`*img*, ***kwargs*`)`

Get luminosity of an RGB image
Converts and RGB image to the CAM16-UCS colorspace, extracts the luminosity, and then scales it between 0-255

Parameters
img (*ndarray*) – RGB image

Returns
lum – CAM16-UCS luminosity

Return type
ndarray

`valis.preprocessing.`**`match_histograms`**`(`*src_image*, *ref_histogram*, *bins=256*`)`

Source: https://automaticaddison.com/how-to-do-histogram-matching-using-opencv/

This method matches the source image histogram to the reference signal :param image src_image: The original source image :param image ref_image: The reference image :return: image_after_matching :rtype: image (array)

`valis.preprocessing.`**`match_histograms`**`(`*src_image*, *ref_histogram*, *bins=256*`)`

Source: https://automaticaddison.com/how-to-do-histogram-matching-using-opencv/

This method matches the source image histogram to the reference signal :param image src_image: The original source image :param image ref_image: The reference image :return: image_after_matching :rtype: image (array)

`valis.preprocessing.`**`norm_img_stats`**`(`*img*, *target_stats*, *mask=None*`)`

Normalize an image

Image will be normalized to have same stats as *target_stats*

Based on method in "A nonlinear mapping approach to stain normalization in digital histopathology images using image-specific color deconvolution.", Khan et al. 2014

Assumes that *img* values range between 0-255

`valis.preprocessing.`**`norm_img_stats`**`(`*img*, *target_stats*, *mask=None*`)`

Normalize an image

Image will be normalized to have same stats as *target_stats*

Based on method in "A nonlinear mapping approach to stain normalization in digital histopathology images using image-specific color deconvolution.", Khan et al. 2014

Assumes that *img* values range between 0-255

`valis.preprocessing.standardize_colorfulness(img, c=0.2, h=0)`

Give image constant colorfulness and hue

Image is converted to cylindrical CAM-16UCS assigned a constant hue and colorfulness, and then coverted back to RGB.

Parameters

- **img** (*ndarray*) – Image to be processed
- **c** (*int*) – Colorfulness
- **h** (*int*) – Hue, in radians (-pi to pi)

Returns

rgb2 – *img* with constant hue and colorfulness

Return type

ndarray

Classes

Base ImageProcesser

`class valis.preprocessing.ImageProcesser(image, src_f, level, series, reader=None)`

Process images for registration

ImageProcesser sub-classes processes images to single channel images which are then used in image registration.

Each *ImageProcesser* is initialized with an image, the path to the image, the pyramid level, and the series number. These values will be set during the registration process.

ImageProcesser must also have a *process_image* method, which is called during registration. As *ImageProcesser* has the image and and its relevant information (filename, level, series) as attributes, it should be able to access and modify the image as needed. However, one can also pass extra args and kwargs to *process_image*. As such, *process_image* will also need to accept args and kwargs.

image

Image to be processed

Type

ndarray

src_f

Path to slide/image.

Type

str

level

Pyramid level to be read.

Type

int

series

The series to be read.

Type

int

__init__(*image*, *src_f*, *level*, *series*, *reader=None*)

Parameters

- **image** (*ndarray*) – Image to be processed
- **src_f** (*str*) – Path to slide/image.
- **level** (*int*) – Pyramid level to be read.
- **series** (*int*) – The series to be read.

process_image(**args*, ***kwargs*)

Pre-process image for registration

Pre-process image for registration. Processed image should be a single channel uint8 image.

Returns
processed_img – Single channel processed copy of *image*

Return type
ndarray

ChannelGetter

class valis.preprocessing.**ChannelGetter**(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: *ImageProcesser*

Select channel from image

__init__(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **image** (*ndarray*) – Image to be processed
- **src_f** (*str*) – Path to slide/image.
- **level** (*int*) – Pyramid level to be read.
- **series** (*int*) – The series to be read.

process_image(*channel='dapi'*, *adaptive_eq=True*, **args*, ***kwaargs*)

Pre-process image for registration

Pre-process image for registration. Processed image should be a single channel uint8 image.

Returns
processed_img – Single channel processed copy of *image*

Return type
ndarray

ColorfulStandardizer

`class valis.preprocessing.ColorfulStandardizer`(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: `ImageProcesser`

Standardize the colorfulness of the image

`__init__`(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **`image`** (*ndarray*) – Image to be processed
- **`src_f`** (*str*) – Path to slide/image.
- **`level`** (*int*) – Pyramid level to be read.
- **`series`** (*int*) – The series to be read.

`process_image`(*c=0.2*, *invert=True*, *adaptive_eq=False*, **args*, ***kwargs*)

Pre-process image for registration

Pre-process image for registration. Processed image should be a single channel uint8 image.

Returns
processed_img – Single channel processed copy of *image*

Return type
ndarray

StainFlattener

`class valis.preprocessing.StainFlattener`(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: `ImageProcesser`

`__init__`(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **`image`** (*ndarray*) – Image to be processed
- **`src_f`** (*str*) – Path to slide/image.
- **`level`** (*int*) – Pyramid level to be read.
- **`series`** (*int*) – The series to be read.

`process_image`(*n_colors=100*, *q=95*, *with_mask=True*, *adaptive_eq=True*, *max_colors=100*)

Parameters

- **`n_colors`** (*int*) – Number of colors to use for deconvolution. If *n_stains = -1*, then the number of colors will be estimated using the K-means "elbow method".
- **`max_colors`** (*int*) – If *n_colors = -1*, this value sets the maximum number of color clusters

BgColorDistance

class valis.preprocessing.**BgColorDistance**(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: *ImageProcesser*

Calculate distance between each pixel and the background color

__init__(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **image** (*ndarray*) – Image to be processed
- **src_f** (*str*) – Path to slide/image.
- **level** (*int*) – Pyramid level to be read.
- **series** (*int*) – The series to be read.

process_image(*brightness_q=0.99*, **args*, ***kwargs*)

Pre-process image for registration

Pre-process image for registration. Processed image should be a single channel uint8 image.

Returns
processed_img – Single channel processed copy of *image*

Return type
ndarray

Luminosity

class valis.preprocessing.**Luminosity**(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: *ImageProcesser*

Get luminosity of an RGB image

__init__(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **image** (*ndarray*) – Image to be processed
- **src_f** (*str*) – Path to slide/image.
- **level** (*int*) – Pyramid level to be read.
- **series** (*int*) – The series to be read.

process_image(**args*, ***kwaargs*)

Pre-process image for registration

Pre-process image for registration. Processed image should be a single channel uint8 image.

Returns
processed_img – Single channel processed copy of *image*

Return type
ndarray

H&E deconvolution

class valis.preprocessing.**HEDeconvolution**(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Bases: *ImageProcesser*

Normalize staining appearence of hematoxylin and eosin (H&E) stained image and get the H or E deconvolution image.

Reference

A method for normalizing histology slides for quantitative analysis. M. Macenko et al., ISBI 2009.

__init__(*image*, *src_f*, *level*, *series*, **args*, ***kwargs*)

Parameters

- **image** (*ndarray*) – Image to be processed
- **src_f** (*str*) – Path to slide/image.
- **level** (*int*) – Pyramid level to be read.
- **series** (*int*) – The series to be read.

process_image(*stain='hem'*, *Io=240*, *alpha=1*, *beta=0.15*, **args*, ***kwargs*)

Reference

A method for normalizing histology slides for quantitative analysis. M. Macenko et al., ISBI 2009.

Note: Adaptation of the code from https://github.com/schaugf/HEnorm_python.

1.1.7 Feature detectors and descriptors

Functions

Functions and classes to detect and describe image features

Bundles OpenCV feature detectors and descriptors into the FeatureDD class

Also makes it easier to mix and match feature detectors and descriptors from different pacakges (e.g. skimage and OpenCV). See CensureVggFD for an example

valis.feature_detectors.**filter_features**(*kp*, *desc*, *n_keep=20000*)

Get keypoints with highest response

Parameters

- **kp** (*list*) – List of cv2.KeyPoint detected by an OpenCV feature detector.
- **desc** (*ndarray*) – 2D numpy array of keypoint descriptors, where each row is a keypoint and each column a feature.
- **n_keep** (*int*) – Maximum number of features that are retained.

Returns

- *Keypoints and and corresponding descriptors that the the n_keep highest*
- *responses.*

Classes

Base feature detector

`class valis.feature_detectors.`**`FeatureDD`**`(`*kp_detector=None*, *kp_descriptor=None*`)`

Abstract class for feature detection and description.

User can create other feature detectors as subclasses, but each must return keypoint positions in xy coordinates along with the descriptors for each keypoint.

Note that in some cases, such as KAZE, kp_detector can also detect features. However, in other cases, there may need to be a separate feature detector (like BRISK or ORB) and feature descriptor (like VGG).

`kp_detector`

Keypoint detetor, by default from OpenCV

Type

object

`kp_descriptor`

Keypoint descriptor, by default from OpenCV

Type

object

`kp_detector_name`

Name of keypoint detector

Type

str

`kp_descriptor`

Name of keypoint descriptor

Type

str

`detectAndCompute``(`*image*, *mask=None*`)`

Detects and describes keypoints in image

`__init__``(`*kp_detector=None*, *kp_descriptor=None*`)`

Parameters

- **`kp_detector`** (`object`) – Keypoint detetor, by default from OpenCV
- **`kp_descriptor`** (`object`) – Keypoint descriptor, by default from OpenCV

`detect_and_compute``(`*image*, *mask=None*`)`

Detect the features in the image

Detect the features in the image using the defined kp_detector, then describe the features using the kp_descriptor. The user can override this method so they don't have to use OpenCV's Keypoint class.

Parameters

- **image** (*ndarray*) – Image in which the features will be detected. Should be a 2D uint8 image if using OpenCV
- **mask** (*ndarray, optional*) – Binary image with same shape as image, where foreground > 0, and background = 0. If provided, feature detection will only be performed on the foreground.

Returns

- **kp** (*ndarry*) – (N, 2) array positions of keypoints in xy corrdinates for N keypoints
- **desc** (*ndarry*) – (N, M) array containing M features for each of the N keypoints

BRISK

class valis.feature_detectors.**BriskFD**(*kp_descriptor=< cv2.BRISK 0x7f455a45ce50>*)

Bases: *FeatureDD*

Uses BRISK for feature detection and description

KAZE

class valis.feature_detectors.**KazeFD**(*kp_descriptor=< cv2.KAZE 0x7f455a45ce30>*)

Bases: *FeatureDD*

Uses KAZE for feature detection and description

AKAZE

class valis.feature_detectors.**AkazeFD**(*kp_descriptor=< cv2.AKAZE 0x7f455a45ced0>*)

Bases: *FeatureDD*

Uses AKAZE for feature detection and description

DAISY

class valis.feature_detectors.**DaisyFD**(*kp_detector=< cv2.BRISK 0x7f455a45ce10>, kp_descriptor=< cv2.xfeatures2d.DAISY 0x7f455a394cf0>*)

Bases: *FeatureDD*

Uses BRISK for feature detection and DAISY for feature description

LATCH

class valis.feature_detectors.**LatchFD**(*kp_detector=< cv2.BRISK 0x7f455a45ce10>, kp_descriptor=< cv2.xfeatures2d.LATCH 0x7f455a394310>*)

Bases: *FeatureDD*

Uses BRISK for feature detection and LATCH for feature description

BOOST

class valis.feature_detectors.**BoostFD**(*kp_detector=< cv2.BRISK 0x7f455a45ce10>*, *kp_descriptor=< cv2.xfeatures2d.BoostDesc 0x7f455a394430>*)

Bases: `FeatureDD`

Uses BRISK for feature detection and Boost for feature description

VGG

class valis.feature_detectors.**VggFD**(*kp_detector=< cv2.BRISK 0x7f455a45ce10>*, *kp_descriptor=< cv2.xfeatures2d.VGG 0x7f455a3945b0>*)

Bases: `FeatureDD`

Uses BRISK for feature detection and VGG for feature description

Orb + Vgg

class valis.feature_detectors.**OrbVggFD**(*kp_detector=< cv2.ORB 0x7f455a394690>*, *kp_descriptor=< cv2.xfeatures2d.VGG 0x7f455a394590>*)

Bases: `FeatureDD`

Uses ORB for feature detection and VGG for feature description

SuperPoint

class valis.feature_detectors.**SuperPointFD**(*keypoint_threshold=0.005*, *nms_radius=4*, *force_cpu=False*, *kp_descriptor=None*, *kp_detector=None*)

Bases: `FeatureDD`

SuperPoint *FeatureDD*

Use SuperPoint to detect and describe features (*detect_and_compute*) Adapted from https://github.com/magicleap/SuperGluePretrainedNetwork/blob/master/match_pairs.py

References

Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. SuperGlue: Learning Feature Matching with Graph Neural Networks. In CVPR, 2020. https://arxiv.org/abs/1911.11763

1.1.8 Feature matching

Functions

Functions and classes to match and filter image features

valis.feature_matcher.**filter_matches**(*kp1_xy*, *kp2_xy*, *method='RANSAC'*, *filtering_kwargs=None*)

Use RANSAC or GMS to remove poor matches

Parameters

- **kp1_xy** (*ndarray*) – (N, 2) array containing image 1s keypoint positions, in xy coordinates.
- **kp2_xy** (*ndarray*) – (N, 2) array containing image 2s keypoint positions, in xy coordinates.
- **method** (*str*) – *method* = "GMS" will use filter_matches_gms() to remove poor matches. This uses the Grid-based Motion Statistics. *method* = "RANSAC" will use RANSAC to remove poor matches
- **filtering_kwargs** (*dict*) – Extra arguments passed to filtering function

  If *method* == "GMS", these need to include: img1_shape, img2_shape, scaling, thresholdFactor. See filter_matches_gms for details

  If *method* == "RANSAC", this can be None, since the ransac value is a class attribute

Returns

- **filtered_src_points** (*ndarray*) – (M, 2) ndarray of inlier keypoints from kp1_xy
- **filtered_dst_points** (*(N, 2) array*) – (M, 2) ndarray of inlier keypoints from kp2_xy
- **good_idx** (*ndarray*) – (M, 1) array containing ndices of inliers

valis.feature_matcher.**match_desc_and_kp**(*desc1*, *kp1_xy*, *desc2*, *kp2_xy*, *metric=None*, *metric_type=None*, *metric_kwargs=None*, *max_ratio=1.0*, *filter_method='RANSAC'*, *filtering_kwargs=None*)

Match the descriptors of image 1 with those of image 2 and remove outliers.

Metric can be a string to use a distance in scipy.distnce.cdist(), or a custom distance function

Parameters

- **desc1** (*ndarray*) – (N, P) array of image 1's descriptions for N keypoints, which each keypoint having P features
- **kp1_xy** (*ndarray*) – (N, 2) array containing image 1's keypoint positions (xy)
- **desc2** (*ndarray*) – (M, P) array of image 2's descriptions for M keypoints, which each keypoint having P features
- **kp2_xy** (*(M, 2) array*) – (M, 2) array containing image 2's keypoint positions (xy)
- **metric** (*string, or callable*) – Metric to calculate distance between each pair of features in desc1 and desc2. Can be a string to use as distance in spatial.distance.cdist, or a custom distance function
- **metric_kwargs** (*dict*) – Optionl keyword arguments to be passed into pairwise_distances() or pairwise_kernels() from the sklearn.metrics.pairwise module
- **max_ratio** (*float, optional*) – Maximum ratio of distances between first and second closest descriptor in the second set of descriptors. This threshold is useful to filter ambiguous matches between the two descriptor sets. The choice of this value depends on the statistics of the chosen descriptor, e.g., for SIFT descriptors a value of 0.8 is usually chosen, see D.G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints", International Journal of Computer Vision, 2004.
- **filter_method** (*str*) – "GMS" will use uses the Grid-based Motion Statistics "RANSAC" will use RANSAC
- **filtering_kwargs** (*dict*) – Dictionary containing extra arguments for the filtering method. kp1_xy, kp2_xy, feature_d are calculated here, and don't need to be in filtering_kwargs. If filter_method == "GMS", then the required arguments are: img1_shape, img2_shape, scaling, thresholdFactor. See filter_matches_gms for details.

If filter_method == "RANSAC", then the required arguments are: ransac_val. See filter_matches_ransac for details.

- **Returns** –
- **--------** –
- **match_info12** (MatchInfo) – Contains information regarding the matches between image 1 and image 2. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info12** (MatchInfo) – Contains information regarding the matches between image 1 and image 2. These results have undergone filtering, and so contain good matches
- **match_info21** (MatchInfo) – Contains information regarding the matches between image 2 and image 1. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info21** (MatchInfo) – Contains information regarding the matches between image 2 and image 1. These results have undergone filtering, and so contain good matches

Classes

MatchInfo

class valis.feature_matcher.**MatchInfo**(*matched_kp1_xy*, *matched_desc1*, *matches12*, *matched_kp2_xy*, *matched_desc2*, *matches21*, *match_distances*, *distance*, *similarity*, *metric_name*, *metric_type*, *img1_name=None*, *img2_name=None*)

Class that stores information related to matches. One per pair of images

All attributes are all set as parameters during initialization

__init__(*matched_kp1_xy*, *matched_desc1*, *matches12*, *matched_kp2_xy*, *matched_desc2*, *matches21*, *match_distances*, *distance*, *similarity*, *metric_name*, *metric_type*, *img1_name=None*, *img2_name=None*)

Stores information about matches and features

Parameters

- **matched_kp1_xy** (*ndarray*) – (Q, 2) array of image 1 keypoint xy coordinates after filtering
- **matched_desc1** (*ndarray*) – (Q, P) array of matched descriptors for image 1, each of which has P features
- **matches12** (*ndarray*) – (1, Q) array of indices of featiures in image 1 that matched those in image 2
- **matched_kp2_xy** (*ndarray*) – (Q, 2) array containing Q matched image 2 keypoint xy coordinates after filtering
- **matched_desc2** (*ndarray*) – (Q, P) containing Q matched descriptors for image 2, each of which has P features
- **matches21** (*ndarray*) – (1, Q) containing indices of featiures in image 2 that matched those in image 1
- **match_distances** (*ndarray*) – Distances between each of the Q pairs of matched descriptors
- **n_matches** (*int*) – Number of good matches (i.e. the number of inlier keypoints)

- **distance** (*float*) – Mean distance of features
- **similarity** (*float*) – Mean similarity of features
- **metric_name** (*str*) – Name of metric
- **metric_type** (*str*) – "distsnce" or "similarity"
- **img1_name** (*str*) – Name of the image that kp1 and desc1 belong to
- **img2_name** (*str*) – Name of the image that kp2 and desc2 belong to

Matcher

class valis.feature_matcher.**Matcher**(*metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Class that matchs the descriptors of image 1 with those of image 2

Outliers removed using RANSAC or GMS

metric

Metric to calculate distance between each pair of features in desc1 and desc2. Can be a string to use as distance in spatial.distance.cdist, or a custom distance function

Type
str, or callable

metric_name

Name metric used. Will be the same as metric if metric is string. If metric is function, this will be the name of the function.

Type
str

metric_type

String describing what the custom metric function returns, e.g. 'similarity' or 'distance'. If None, and metric is a function it is assumed to be a distance, but there will be a warning that this variable should be provided to either define that it is a similarity, or to avoid the warning by having metric_type='distance' In the case of similarity, the number of features will be used to convert distances

Type
str, or callable

ransac

The residual threshold to determine if a match is an inlier. Only used if filter_method == {RANSAC_NAME}. Default is "RANSAC"

Type
int

gms_threshold

Used when filter_method is "GMS". The higher, the fewer matches.

Type
int

scaling

Whether or not image scaling should be considered when filter_method is "GMS"

Type
bool

metric_kwargs

Keyword arguments passed into the metric when calling spatial.distance.cdist

Type
dict

match_filter_method

"GMS" will use filter_matches_gms() to remove poor matches. This uses the Grid-based Motion Statistics (GMS) or RANSAC.

Type
str

__init__(*metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Parameters

- **metric** (*str, or callable*) – Metric to calculate distance between each pair of features in desc1 and desc2. Can be a string to use as distance in spatial.distance.cdist, or a custom distance function
- **metric_type** (*str, or callable*) – String describing what the custom metric function returns, e.g. 'similarity' or 'distance'. If None, and metric is a function it is assumed to be a distance, but there will be a warning that this variable should be provided to either define that it is a similarity, or to avoid the warning by having metric_type='distance' In the case of similarity, the number of features will be used to convert distances
- **metric_kwargs** (*dict*) – Keyword arguments passed into the metric when calling spatial.distance.cdist
- **filter_method** (*str*) – "GMS" will use filter_matches_gms() to remove poor matches. This uses the Grid-based Motion Statistics (GMS) or RANSAC.
- **ransac_val** (*int*) – The residual threshold to determine if a match is an inlier. Only used if filter_method is "RANSAC".
- **gms_threshold** (*int*) – Used when filter_method is "GMS". The higher, the fewer matches.
- **scaling** (*bool*) – Whether or not image scaling should be considered when filter_method is "GMS".

match_images(*desc1*, *kp1_xy*, *desc2*, *kp2_xy*, *additional_filtering_kwargs=None*, **args*, ***kwargs*)

Match the descriptors of image 1 with those of image 2, Outliers removed using match_filter_method. Metric can be a string to use a distance in scipy.distnce.cdist(), or a custom distance function. Sets atttributes for Matcher object

Parameters

- **desc1** (*(N, P) array*) – Image 1s 2D array containinng N keypoints, each of which has P features
- **kp1_xy** (*(N, 2) array*) – Image 1s keypoint positions, in xy coordinates, for each of the N descriptors in desc1
- **desc2** (*(M, P) array*) – Image 2s 2D array containinng M keypoints, each of which has P features

- **kp2_xy** (*(M, 2) array*) – Image 1s keypoint positions, in xy coordinates, for each of the M descriptors in desc2
- **additional_filtering_kwargs** (*dict, optional*) – Extra arguments passed to filtering function If self.match_filter_method == "GMS", these need to include: img1_shape, img2_shape. See filter_matches_gms for details If If self.match_filter_method == "RANSAC", this can be None, since the ransac value is class attribute

Returns

- **match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results have undergone filtering, and so contain good matches
- **match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1.

SuperPointAndGlue

class valis.feature_matcher.**SuperPointAndGlue**(*weights='indoor'*, *keypoint_threshold=0.005*, *nms_radius=4*, *sinkhorn_iterations=100*, *match_threshold=0.2*, *force_cpu=False*, *metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Use SuperPoint SuperPoint + SuperGlue to match images (*match_images*)

Implementation adapted from https://github.com/magicleap/SuperGluePretrainedNetwork/blob/master/match_pairs.py

References

Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. SuperGlue: Learning Feature Matching with Graph Neural Networks. In CVPR, 2020. https://arxiv.org/abs/1911.11763

__init__(*weights='indoor'*, *keypoint_threshold=0.005*, *nms_radius=4*, *sinkhorn_iterations=100*, *match_threshold=0.2*, *force_cpu=False*, *metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Parameters

- **weights** (*str*) – SuperGlue weights. Options= ["indoor", "outdoor"]
- **keypoint_threshold** (*float*) – SuperPoint keypoint detector confidence threshold
- **nms_radius** (*int*) – SuperPoint Non Maximum Suppression (NMS) radius (must be positive)
- **sinkhorn_iterations** (*int*) – Number of Sinkhorn iterations performed by SuperGlue
- **match_threshold** (*float*) – SuperGlue match threshold
- **force_cpu** (*bool*) – Force pytorch to run in CPU mode

- **scaling** (*bool*) – Whether or not image scaling should be considered when filter_method is “GMS”.

match_images(*img1=None*, *desc1=None*, *kp1_xy=None*, *img2=None*, *desc2=None*, *kp2_xy=None*, *additional_filtering_kwargs=None*)

Match the descriptors of image 1 with those of image 2, Outliers removed using match_filter_method. Metric can be a string to use a distance in scipy.distnce.cdist(), or a custom distance function. Sets atttributes for Matcher object

Parameters

- **desc1** (*(N, P) array*) – Image 1s 2D array containinng N keypoints, each of which has P features
- **kp1_xy** (*(N, 2) array*) – Image 1s keypoint positions, in xy coordinates, for each of the N descriptors in desc1
- **desc2** (*(M, P) array*) – Image 2s 2D array containinng M keypoints, each of which has P features
- **kp2_xy** (*(M, 2) array*) – Image 1s keypoint positions, in xy coordinates, for each of the M descriptors in desc2
- **additional_filtering_kwargs** (*dict, optional*) – Extra arguments passed to filtering function If self.match_filter_method == “GMS”, these need to include: img1_shape, img2_shape. See filter_matches_gms for details If If self.match_filter_method == “RANSAC”, this can be None, since the ransac value is class attribute

Returns

- **match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results haven’t undergone filtering, so contain many poor matches.
- **filtered_match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results have undergone filtering, and so contain good matches
- **match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1. These results haven’t undergone filtering, so contain many poor matches.
- **filtered_match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1.

SuperGlueMatcher

class valis.feature_matcher.**SuperGlueMatcher**(*weights='indoor'*, *keypoint_threshold=0.005*, *nms_radius=4*, *sinkhorn_iterations=100*, *match_threshold=0.2*, *force_cpu=False*, *metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Use SuperGlue to match images (*match_images*)

Implementation adapted from https://github.com/magicleap/SuperGluePretrainedNetwork/blob/master/match_pairs.py

References

Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. SuperGlue: Learning Feature Matching with Graph Neural Networks. In CVPR, 2020. https://arxiv.org/abs/1911.11763

`__init__`(*weights='indoor'*, *keypoint_threshold=0.005*, *nms_radius=4*, *sinkhorn_iterations=100*, *match_threshold=0.2*, *force_cpu=False*, *metric=None*, *metric_type=None*, *metric_kwargs=None*, *match_filter_method='RANSAC'*, *ransac_thresh=7*, *gms_threshold=15*, *scaling=False*)

Use SuperGlue to match images (*match_images*)

Adapted from https://github.com/magicleap/SuperGluePretrainedNetwork/blob/master/match_pairs.py

Parameters

- **`weights`** (*`str`*) – SuperGlue weights. Options= ["indoor", "outdoor"]
- **`keypoint_threshold`** (*`float`*) – SuperPoint keypoint detector confidence threshold
- **`nms_radius`** (*`int`*) – SuperPoint Non Maximum Suppression (NMS) radius (must be positive)
- **`sinkhorn_iterations`** (*`int`*) – Number of Sinkhorn iterations performed by SuperGlue
- **`match_threshold`** (*`float`*) – SuperGlue match threshold
- **`force_cpu`** (*`bool`*) – Force pytorch to run in CPU mode
- **`scaling`** (*`bool`*) – Whether or not image scaling should be considered when filter_method is "GMS".

`match_images`(*img1=None*, *desc1=None*, *kp1_xy=None*, *img2=None*, *desc2=None*, *kp2_xy=None*, *additional_filtering_kwargs=None*)

Match the descriptors of image 1 with those of image 2, Outliers removed using match_filter_method. Metric can be a string to use a distance in scipy.distnce.cdist(), or a custom distance function. Sets atttributes for Matcher object

Parameters

- **`desc1`** (*`(N, P) array`*) – Image 1s 2D array containinng N keypoints, each of which has P features
- **`kp1_xy`** (*`(N, 2) array`*) – Image 1s keypoint positions, in xy coordinates, for each of the N descriptors in desc1
- **`desc2`** (*`(M, P) array`*) – Image 2s 2D array containinng M keypoints, each of which has P features
- **`kp2_xy`** (*`(M, 2) array`*) – Image 1s keypoint positions, in xy coordinates, for each of the M descriptors in desc2
- **`additional_filtering_kwargs`** (*`dict, optional`*) – Extra arguments passed to filtering function If self.match_filter_method == "GMS", these need to include: img1_shape, img2_shape. See filter_matches_gms for details If If self.match_filter_method == "RANSAC", this can be None, since the ransac value is class attribute

Returns

- **match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info12** (*MatchInfo*) – Contains information regarding the matches between image 1 and image 2. These results have undergone filtering, and so contain good matches

- **match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1. These results haven't undergone filtering, so contain many poor matches.
- **filtered_match_info21** (*MatchInfo*) – Contains information regarding the matches between image 2 and image 1.

1.1.9 Affine optimization

AffineOptimizer

`class valis.affine_optimizer.`**`AffineOptimizer`**(*nlevels=1*, *nbins=256*, *optimization='Powell'*, *transformation='EuclideanTransform'*)

Class that optimizes ridid registration

`nlevels`

Number of levels in the Gaussian pyramid

Type
int

`nbins`

Number of bins to have in histograms used to estimate mutual information

Type
int

`optimization`

Optimization method. Can be any method from scipy.optimize "FuzzyPSO" for Fuzzy Self-Tuning PSO in the fst-pso package (https://pypi.org/project/fst-pso/) "gp_minimize", "forest_minimize", "gbrt_minimize" from scikit-opt

Type
str

`transformation`

Type of transformation, "EuclideanTransform" or "SimilarityTransform"

Type
str

`current_level`

Current level of the Guassian pyramid that is being registered

Type
int

`accepts_xy`

Bool declaring whether or not the optimizer will use corresponding points to optimize the registration

Type
bool

`setup`(*moving*, *fixed*, *mask*, *initial_M=None*)

Gets images ready for alignment

`cost_fxn`(*fixed_image*, *transformed*, *mask*)

Calculates metric that is to be minimized `align`(*moving*, *fixed*, *mask*, *initial_M=None*, *moving_xy=None*, *fixed_xy=None*)

Align images by minimizing cost_fxn

Notes

All AffineOptimizer subclasses need to have the method align(moving, fixed, mask, initial_M, moving_xy, fixed_xy) that returns the aligned image, optimal_M, cost_list

AffineOptimizer subclasses must also have a cost_fxn(fixed_image, transformed, mask) method that returns the registration metric value

If one wants to use the same optimization methods, but a different cost function, then the subclass only needs to have a new cost_fxn method. See AffineOptimizerDisplacement for an example implementing a new cost function

Major overhauls are possible too. See AffineOptimizerMattesMI for an example on using SimpleITK's optimization methods inside of an AffineOptimizer subclass

If the optimizer uses corressponding points, then the class attribute accepts_xy needs to be set to True. The default is False.

`__init__`(*nlevels=1*, *nbins=256*, *optimization='Powell'*, *transformation='EuclideanTransform'*)

AffineOptimizer registers moving and fixed images by minimizing a cost function

Parameters

- **nlevels** (*int*) – Number of levels in the Gaussian pyramid
- **nbins** (*int*) – Number of bins to have in histograms used to estimate mutual information
- **optimization** (*str*) – Optimization method. Can be any method from scipy.optimize
- **transformation** (*str*) – Type of transformation, "EuclideanTransform" or "SimilarityTransform"

`align`(*moving*, *fixed*, *mask*, *initial_M=None*, *moving_xy=None*, *fixed_xy=None*)

Align images by minimizing self.cost_fxn. Aligns each level of the Gaussian pyramid, and uses previous transform as the initial guess in the next round of optimization. Also uses other "good" estimates to define the parameter boundaries.

Parameters

- **moving** (*ndarray*) – Image to warp to align with fixed
- **fixed** (*ndarray*) – Image moving is warped to align with
- **mask** (*ndarray*) – 2D array having non-zero pixel values, where values of 0 are ignnored during registration
- **initial_M** (*(3x3) array*) – Initial transformation matrix
- **moving_xy** (*ndarray, optional*) – (N, 2) array containing points in the moving image that correspond to those in the fixed image
- **fixed_xy** (*ndarray, optional*) – (N, 2) array containing points in the fixed image that correspond to those in the moving image

Returns

- **aligned** (*(N,M) array*) – Moving image warped to align with the fixed image
- **M** (*(3,3) array*) – Optimal transformation matrix
- **cost_list** (*list*) – list containing the minimized cost for each level in the pyramid

AffineOptimizerMattesMI

`class valis.affine_optimizer.AffineOptimizerMattesMI`(*nlevels=4.0*, *nbins=32*, *optimization='AdaptiveStochasticGradientDescent'*, *transform='EuclideanTransform'*)

Bases: *AffineOptimizer*

Optimize rigid registration using Simple ITK

AffineOptimizerMattesMI is an AffineOptimizer subclass that uses simple ITK's AdvancedMattesMutualInformation. If moving_xy and fixed_xy are also provided, then Mattes mutual information will be maximized, while the distance between moving_xy and fixed_xy will be minimized (the CorrespondingPointsEuclideanDistanceMetric in Simple ITK).

nlevels

Number of levels in the Gaussian pyramid

Type
int

nbins

Number of bins to have in histograms used to estimate mutual information

Type
int

transformation

Type of transformation, "EuclideanTransform" or "SimilarityTransform"

Type
str

Reg

sitk.ElastixImageFilter object that will perform the optimization

Type
sitk.ElastixImageFilter

fixed_kp_fname

Name of file where to fixed_xy will be temporarily be written. Eventually deleted

Type
str

moving_kp_fname

Name of file where to moving_xy will be temporarily be written. Eventually deleted

Type
str

setup(*moving*, *fixed*, *mask*, *initial_M=None*, *moving_xy=None*, *fixed_xy=None*)

Create parameter map and initialize Reg

calc_cost(*p*)

Inherited but not used, returns None

write_elastix_kp(*kp*, *fname*)

Temporarily write fixed_xy and moving_xy to file `align`(*moving*, *fixed*, *mask*, *initial_M=None*, *moving_xy=None*, *fixed_xy=None*)

Align images by minimizing cost_fxn

`__init__`(*nlevels=4.0*, *nbins=32*, *optimization='AdaptiveStochasticGradientDescent'*, *transform='EuclideanTransform'*)

AffineOptimizer registers moving and fixed images by minimizing a cost function

Parameters

- **nlevels** (*int*) – Number of levels in the Gaussian pyramid
- **nbins** (*int*) – Number of bins to have in histograms used to estimate mutual information
- **optimization** (*str*) – Optimization method. Can be any method from scipy.optimize
- **transformation** (*str*) – Type of transformation, "EuclideanTransform" or "SimilarityTransform"

`align`(*moving*, *fixed*, *mask*, *initial_M=None*, *moving_xy=None*, *fixed_xy=None*)

Optimize rigid registration

Parameters

- **moving** (*ndarray*) – Image to warp to align with fixed
- **fixed** (*ndarray*) – Image moving is warped to align with
- **mask** (*ndarray*) – 2D array having non-zero pixel values, where values of 0 are ignnored during registration
- **initial_M** (*(3x3) array*) – Initial transformation matrix
- **moving_xy** (*ndarray, optional*) – (N, 2) array containing points in the moving image that correspond to those in the fixed image
- **fixed_xy** (*ndarray, optional*) – (N, 2) array containing points in the fixed image that correspond to those in the moving image

Returns

- **aligned** (*(N,M) array*) – Moving image warped to align with the fixed image
- **M** (*(3,3) array*) – Optimal transformation matrix
- **cost_list** (*None*) – None is returned because costs are not recorded

1.1.10 Micro-rigid registration

Classes

MicroRigidRegistrar

`class valis.micro_rigid_registrar.MicroRigidRegistrar`(*val_obj*, *feature_detector_cls=<class 'valis.feature_detectors.SuperPointFD'>*, *matcher=<class 'valis.feature_matcher.SuperPointAndGlue'>*, *processor_dict=None*, *scale=0.125*, *tile_wh=512*, *roi='mask'*)

Refine rigid registration using higher resolution images

Rigid transforms found during lower resolution images are applied to the WSI and then downsampled. The higher resolution registered images are then divided into tiles, which are processed and normalized. Next, features are detected and matched for each tile, the results of which are combined into a common keypoint list. These higher resolution keypoints are then used to estimate a new rigid transform. Replaces thumbnails in the rigid registration folder.

`val_obj`

The "parent" object that registers all of the slides.

Type

Valis

`feature_detector_cls`

Uninstantiated FeatureDD object that detects and computes image features. Default is SuperPointFD. The available feature_detectors are found in the *feature_detectors* module. If a desired feature detector is not available, one can be created by subclassing *feature_detectors.FeatureDD*.

Type

FeatureDD, optional

`matcher`

Matcher object that will be used to match image features

Type

Matcher

`scale`

Degree of downsampling to use for the reigistration, based on the registered WSI shape (i.e. Slide.aligned_slide_shape_rc)

Type

float

`tile_wh`

Width and height of tiles extracted from registered WSI

Type

int

`roi`

Determines how the region of interest is defined. *roi="mask"* will use the bounding box of non-rigid registration mask to define the search area. *roi=matches* will use the bounding box of the previously matched features to define the search area.

Type

string

`iter_order`

Determines the order in which images are aligned. Goes from reference image to the edges of the stack.

Type

list of tuples

`__init__`(*val_obj*, *feature_detector_cls=<class 'valis.feature_detectors.SuperPointFD'>*, *matcher=<class 'valis.feature_matcher.SuperPointAndGlue'>*, *processor_dict=None*, *scale=0.125*, *tile_wh=512*, *roi='mask'*)

Parameters

- **`val_obj`** (`Valis`) – The "parent" object that registers all of the slides.

- **feature_detector_cls** (`FeatureDD, optional`) – Uninstantiated FeatureDD object that detects and computes image features. Default is SuperPointFD. The available feature_detectors are found in the *feature_detectors* module. If a desired feature detector is not available, one can be created by subclassing *feature_detectors.FeatureDD*.
- **matcher** (`Matcher`) – Matcher object that will be used to match image features
- **processor_dict** (`dict, optional`) – Each key should be the filename of the image, and the value either a subclassed preprocessing.ImageProcessor, or a list, where the 1st element is the processor, and the second element a dictionary of keyword arguments passed to the processor. If *None*, a default processor will be assigned to each image based on its modality.
- **scale** (`float`) – Degree of downsampling to use for the reigistration, based on the registered WSI shape (i.e. Slide.aligned_slide_shape_rc)
- **tile_wh** (`int`) – Width and height of tiles extracted from registered WSI
- **roi** (`string`) – Determines how the region of interest is defined. *roi="mask"* will use the bounding box of non-rigid registration mask to define the search area. *roi=matches* will use the bo

register(*brightfield_processing_cls=<class 'valis.preprocessing.StainFlattener'>, brightfield_processing_kwargs={'adaptive_eq': False, 'with_mask': False}, if_processing_cls=<class 'valis.preprocessing.ChannelGetter'>, if_processing_kwargs={'adaptive_eq': True, 'channel': 'dapi'}*)

Parameters

- **brightfield_processing_cls** (`ImageProcesser`) – ImageProcesser to pre-process brightfield images to make them look as similar as possible. Should return a single channel uint8 image.
- **brightfield_processing_kwargs** (`dict`) – Dictionary of keyward arguments to be passed to *brightfield_processing_cls*
- **if_processing_cls** (`ImageProcesser`) – ImageProcesser to pre-process immunofluorescent images to make them look as similar as possible. Should return a single channel uint8 image.
- **if_processing_kwargs** (`dict`) – Dictionary of keyward arguments to be passed to *if_processing_cls*

1.1.11 Non-rigid registration

Perform non-rigid registration

Base NonRigidRegistrar

class `valis.non_rigid_registrars.`**NonRigidRegistrar**(*params=None*)

Abstract class for non-rigid registration using displacement fields

Warps moving_img to align with fixed_img using backwards transformations VALIS offers 3 implementations: dense optical flow (OpenCV), SimpleElastix, and groupwise SimpleElastix. Displacement fields can come from other packages, indluding SimpleITK, PIRT, DIPY, etc… Those other methods can be used by subclassing the NonRigidRegistrar classes in VALIS.

`moving_img`

Image with shape (N,M) thata is warp to align with *fixed_img*.

Type
ndarray

`fixed_img`

Image with shape (N,M) that *moving_img* is warped to align with.

Type
ndarray

`mask`

2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Type
ndarray

`shape`

Number of rows and columns in each image. Will be (N,M).

Type
tuple

`grid_spacing`

Number of pixels between deformation grid points.

Type
int

`warped_image`

Registered copy of *moving_img*.

Type
ndarray

`deformation_field_img`

Image showing deformation applied to a regular grid.

Type
ndarray

`backward_dx`

(N,M) array defining the displacements in the x-dimension.

Type
ndarray

`backward_dy`

(N,M) array defining the displacements in the y-dimension.

Type
ndarray

`method`

Name of registration method.

Type
str

Note: All NonRigidRegistrar subclasses need to have a calc() method, which must at least take the following arguments: moving_img, fixed_img, mask. calc() should return the displacement field as a (2, N, M) numpy array, with the first element being an array of displacements in the x-dimension, and the second element being an array of displacements in the y-dimension.

Note that the NonRigidRegistrarXY subclass should be used if corresponding points in moving and fixed images can be used to aid the registration.

`__init__`(*params=None*)

Parameters

`params` (*dictionary*) – Keyword: value dictionary of parameters to be used in reigstration. Will get used in the calc() method.

In the case where simple ITK will be used, params should be a SimpleITK.ParameterMap. Note that numeric values needd to be converted to strings.

`calc`(*moving_img*, *fixed_img*, *mask*, **args*, ***kwargs*)

Cacluate displacement fields

Can record subclass specific atrributes here too

Parameters

- `moving_img` (*ndarray*) – Image to warp to align with *fixed_img*. Has shape (N, M).
- `fixed_img` (*ndarray*) – Image *moving_img* is warped to align with. Has shape (N, M).
- `mask` (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Returns

bk_dxdy – (2, N, M) numpy array of pixel displacements in the x and y directions. dx = bk_dxdy[0], and dy=bk_dxdy[1].

Return type

ndarray

`register`(*moving_img*, *fixed_img*, *mask=None*, ***kwargs*)

Register images, warping moving_img to align with fixed_img

Uses backwards transforms to register images (i.e. aligning fixed to moving), so the inverse transform needs to be used to warp points from moving_img. This is automatically done in warp_tools.warp_xy

Parameters

- `moving_img` (*ndarray*) – Image to warp to align with *fixed_img*.
- `fixed_img` (*ndarray*) – Image *moving_img* is warped to align with.
- `mask` (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.
- `**kwargs` (*dict, optional*) – Additional keyword arguments passed to NonRigidRegistrar.calc

Returns

- **warped_img** (*ndarray*) – Moving image registered to align with fixed image.

- **warped_grid** (*ndarray*) – Image showing deformation applied to a regular grid.
- **bk_dxdy** (*ndarray*) – (2, N, M) numpy array of pixel displacements in the x and y directions.

Base NonRigidRegistrarXY

class valis.non_rigid_registrars.**NonRigidRegistrarXY**(*params=None*)

Bases: *NonRigidRegistrar*

Abstract class for non-rigid registration using displacement fields

Subclass of NonRigidRegistrar that can (optionally) use corresponding points (xy coordinates) to aid in the registration

moving_img

Image with shape (N,M) thata is warp to align with *fixed_img*.

Type
ndarray

fixed_img

Image with shape (N,M) that *moving_img* is warped to align with.

Type
ndarray

mask

2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Type
ndarray

shape

Number of rows and columns in each image. Will be (N,M).

Type
tuple

grid_spacing

Number of pixels between deformation grid points/

Type
int

warped_image

Registered copy of *moving_img*.

Type
ndarray

deformation_field_img

Image showing deformation applied to a regular grid.

Type
ndarray

`backward_dx`

(N,M) array defining the displacements in the x-dimension.

Type
ndarray

`backward_dy`

(N,M) array defining the displacements in the y-dimension.

Type
ndarray

`method`

Name of registration method.

Type
str

`moving_xy`

(N, 2) array containing points in *moving_img* that correspond to those in the fixed image.

Type
ndarray, optional

`fixed_xy`

(N, 2) array containing points in *fixed_img* that correspond to those in the moving image.

Type
ndarray, optional

Note: All NonRigidRegistrarXY subclasses need to have a calc() method, which needs to at least take the following arguments: moving_img, fixed_img, mask, moving_xy, fixed_xy. calc() should return the warped moving image, warped regular grid, and the displacement field as an (2, N, M) numpy array.

Note that NonRigidRegistrar should be used if corresponding points in moving and fixed images can not be used to aid the registration.

`__init__`(*params=None*)

Parameters
params (*dictionary*) – Keyword: value dictionary of parameters to be used in reigstration. Will get used in the calc() method.

In the case where simple ITK will be used, params should be a SimpleITK.ParameterMap. Note that numeric values needd to be converted to strings.

`calc`(*moving_img*, *fixed_img*, *mask*, **args*, ***kwargs*)

Cacluate displacement fields

Can record subclass specific atrributes here too

Parameters

- **moving_img** (*ndarray*) – Image to warp to align with *fixed_img*. Has shape (N, M).
- **fixed_img** (*ndarray*) – Image *moving_img* is warped to align with. Has shape (N, M).
- **mask** (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Returns
bk_dxdy – (2, N, M) numpy array of pixel displacements in the x and y directions. dx = bk_dxdy[0], and dy=bk_dxdy[1].

Return type
ndarray

`register`(*moving_img*, *fixed_img*, *mask=None*, *moving_xy=None*, *fixed_xy=None*, ***kwargs*)

Register images, warping moving_img to align with fixed_img

Uses backwards transforms to register images (i.e. aligning fixed to moving), so the inverse transform needs to be used to warp points from moving_img. This is automatically done in warp_tools.warp_xy

Parameters

- **moving_img** (*ndarray*) – Image to warp to align with *fixed_img*.
- **fixed_img** (*ndarray*) – Image *moving_img* is warped to align with.
- **mask** (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.
- **moving_xy** (*ndarray, optional*) – (N, 2) array containing points in the *moving_img* that correspond to those in *fixed_img*.
- **fixed_xy** (*ndarray, optional*) – (N, 2) array containing points in the *fixed_img* that correspond to those in the *moving_img*.

Returns

- **warped_img** (*ndarray*) – *moving_img* registered to align with *fixed_img*.
- **warped_grid** (*ndarray*) – Image showing deformation applied to a regular grid.
- **bk_dxdy** (*ndarray*) – (2, N, M) numpy array of pixel displacements in the x and y directions.

Base NonRigidRegistrarGroupwise

`class valis.non_rigid_registrars.NonRigidRegistrarGroupwise`(*params=None*)

Bases: `NonRigidRegistrar`

Performs groupwise non-rigid registration

This subclass can register a collection (>= 2) of images, and so is not limited to pairs of images.

`img_list`

List of images, each with shape (N,M) that are to be co-registered

Type
list of ndarray

`mask`

2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Type
ndarray

`shape`
Number of rows and columns in each image. Will be (N,M).

Type
tuple of int

`warped_image`
Registered copy of *moving_img*.

Type
ndarray

`deformation_field_img`
Image showing deformation applied to a regular grid.

Type
ndarray

`backward_dx`
(N,M) array defining the displacements in the x-dimension.

Type
ndarray

`backward_dy`
(N,M) array defining the displacements in the y-dimension.

Type
ndarray

`grid_spacing`
Number of pixels between deformation grid points

Type
int

`method`
Name of registration method.

Type
str

`size`
Number of images that are being registered as a group

Type
int

`__init__`(*params=None*)

Parameters
`params` (*dictionary*) – Keyword: value dictionary of parameters to be used in reigstration. Will get used in the calc() method.

In the case where simple ITK will be used, params should be a SimpleITK.ParameterMap. Note that numeric values needd to be converted to strings.

`register`(*img_list*, *mask=None*)
Register images in img_list

Uses backwards transforms to register images (i.e. aligning fixed to moving), so the inverse transform needs to be used to warp points from moving_img. This is automatically done in warp_tools.warp_xy

Parameters

img_list (*list of ndarray*) – List of I images, each with shape (N,M) that are to be co-registered.

Returns

- **warped_img** (*list of ndarray*) – List of moving images registered to align with the fixed image.
- **warped_grid** (*list of ndarray*) – Image showing deformation applied to a regular grid.
- **bk_dxdy** (*list of ndarray*) – List numpy array of pixel displacements in the x and y directions for each image. Has shape (I, N, M, 2).

OpticalFlowWarper

class valis.non_rigid_registrars.**OpticalFlowWarper**(*params=None*, *optical_flow_obj=None*, *n_grid_pts=50*, *sigma_ratio=0.005*, *paint_size=5000*, *fold_penalty=1e-06*, *smoothing_method=None*)

Bases: *NonRigidRegistrar*

Use dense optical flow to register images.

Dense optical flow fields may not be diffeomorphic, and so this class provides options to smooth displacement fields.

__init__(*params=None*, *optical_flow_obj=None*, *n_grid_pts=50*, *sigma_ratio=0.005*, *paint_size=5000*, *fold_penalty=1e-06*, *smoothing_method=None*)

Parameters

- **params** (*dictionary*) – Keyword: value dictionary of parameters to be used in reigstration. Will get used in the calc() method.
- **optical_flow_obj** (*object*) – Object that will perform dense optical flow.
- **n_grid_pts** (*int*) – Number of gridpoints used to detect folds. Also the number of gridpoints to use when regularizing he mesh when *method* = "regularize".
- **paint_size** (*int*) – Used to determine how much to resize the image to have efficient inpainting. Larger values = longer processing time. Only used if *smoothing_method* = "inpaint".
- **fold_penalty** (*float*) – How much to penalize folding/stretching. Larger values will make the deformation field more uniform, which may or may not be desired, as too much can remove all displacements. Only used if *smoothing_method* = "regularize"
- **sigma_ratio** (*float*) – Determines the amount of Gaussian smoothing, as sigma = max(shape) *sigma_ratio. Larger values do more smoothing. Only used if *smoothing_method* is "gauss".
- **smoothing** (*str*) – If "gauss", then a Gaussian blur will be applied to the deformation fields, using sigma defined by sigma_ratio.

  If "inpaint", folded regions will be detected and removed. Folded regions will then be removed using inpainting.

  If "regularize", folded regions will be detected and regularized using the method fescribed in "Foldover-free maps in 50 lines of code" Garanzha et al. 2021.

  If "None" then no smoothing will be applied.

`calc`(*moving_img*, *fixed_img*, **args*, ***kwargs*)

Cacluate displacement fields

Can record subclass specific atrributes here too

Parameters

- **moving_img** (*ndarray*) – Image to warp to align with *fixed_img*. Has shape (N, M).
- **fixed_img** (*ndarray*) – Image *moving_img* is warped to align with. Has shape (N, M).
- **mask** (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Returns

bk_dxdy – (2, N, M) numpy array of pixel displacements in the x and y directions. dx = bk_dxdy[0], and dy=bk_dxdy[1].

Return type

ndarray

SimpleElastixWarper

`class valis.non_rigid_registrars.`**`SimpleElastixWarper`**(*params=None*, *ammi_weight=0.33*, *bending_penalty_weight=0.33*, *kp_weight=0.33*)

Bases: *NonRigidRegistrarXY*

Uses SimpleElastix to register images

May optionally using corresponding points

`__init__`(*params=None*, *ammi_weight=0.33*, *bending_penalty_weight=0.33*, *kp_weight=0.33*)

Parameters

- **ammi_weight** (*float*) – Weight given to the AdvancedMattesMutualInformation metric.
- **bending_penalty_weight** (*float*) – Weight given to the TransformBendingEnergyPenalty metric.
- **kp_weight** (*float*) – Weight given to the CorrespondingPointsEuclideanDistanceMetric metric. Only used if moving_xy and fixed_xy are provided as arguments to the *register()* method.

`calc`(*moving_img*, *fixed_img*, *mask=None*, *moving_xy=None*, *fixed_xy=None*, **args*, ***kwargs*)

Perform non-rigid registration using SimpleElastix.

Can include corresponding points to help in registration by providing *moving_xy* and *fixed_xy*.

`static get_default_params`(*img_shape*, *grid_spacing_ratio=0.025*)

Get default parameters for registration with sitk.ElastixImageFilter

See https://simpleelastix.readthedocs.io/Introduction.html for advice on parameter selection

SimpleElastixGroupwiseWarper

`class valis.non_rigid_registrars.SimpleElastixGroupwiseWarper(params=None)`

Bases: *NonRigidRegistrarGroupwise*

Performs groupwise non-rigid registration using SimpleElastix.

SimpleElastixGroupwiseWarper can register a collection (>= 2) of images, and so is not limited to pairs of images.

img_list

List of images, each with shape (N,M) that are to be co-registered.

Type
list

mask

2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Type
ndarray

shape

Number of rows and columns in each image. Will have shaape (N,M).

Type
tuple of int

warped_image

Registered copy of *moving_img*.

Type
ndarray

deformation_field_img

Image showing deformation applied to a regular grid.

Type
ndarray

backward_dx

(N,M) array defining the displacements in the x-dimension.

Type
ndarray

backward_dy

(N,M) array defining the displacements in the y-dimension.

Type
ndarray

grid_spacing

Number of pixels between deformation grid points.

Type
int

`method`
Name of registration method.

Type
str

`__init__`(*params=None*)

Parameters
`params` (*dictionary*) – Keyword: value dictionary of parameters to be used in reigstration. Will get used in the calc() method.

In the case where simple ITK will be used, params should be a SimpleITK.ParameterMap. Note that numeric values needd to be converted to strings.

`calc`(*img_list*, *mask=None*, **args*, ***kwargs*)
Cacluate displacement fields

Can record subclass specific atrributes here too

Parameters

- **moving_img** (*ndarray*) – Image to warp to align with *fixed_img*. Has shape (N, M).
- **fixed_img** (*ndarray*) – Image *moving_img* is warped to align with. Has shape (N, M).
- **mask** (*ndarray*) – 2D array with shape (N,M) where non-zero pixel values are foreground, and 0 is background, which is ignnored during registration. If None, then all non-zero pixels in images will be used to create the mask.

Returns
bk_dxdy – (2, N, M) numpy array of pixel displacements in the x and y directions. dx = bk_dxdy[0], and dy=bk_dxdy[1].

Return type
ndarray

`static get_default_params`(*img_shape*, *grid_spacing_ratio=0.025*)
See https://simpleelastix.readthedocs.io/Introduction.html for advice on parameter selection

1.1.12 Serial rigid registration

Functions

Classes and functions to perform serial rigid registration of a set of images

`valis.serial_rigid.register_images`(*img_dir, dst_dir=None, name='registrar', feature_detector=<valis.feature_detectors.VggFD object>, matcher=<valis.feature_matcher.Matcher object>, transformer=<EuclideanTransform(matrix= [[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])>, affine_optimizer=None, imgs_ordered=False, reference_img_f=None, similarity_metric='n_matches', check_for_reflections=False, max_scaling=3.0, align_to_reference=False, qt_emitter=None, valis_obj=None*)

Rigidly align collection of images

Parameters

- **img_dir** (*str*) – Path to directory containing the images that the user would like to be registered. These images need to be single channel, uint8 images

- **dst_dir** (*str, optional*) – Top directory where aliged images should be save. SerialRigidRegistrar will be in this folder, and aligned images in the "registered_images" subdirectory. If None, the images will not be written to file
- **name** (*str, optional*) – Descriptive name of registrar, such as the sample's name
- **feature_detector** (FeatureDD) – FeatureDD object that detects and computes image features.
- **matcher** (Matcher) – Matcher object that will be used to match image features
- **transformer** (*scikit-image Transform object*) – Transformer used to find transformation matrix that will warp each image to the target image.
- **affine_optimizer** (*AffineOptimzer object*) – Object that will minimize a cost function to find the optimal affine transoformations
- **imgs_ordered** (*bool*) – Boolean defining whether or not the order of images in img_dir are already in the correct order. If True, then each filename should begin with the number that indicates its position in the z-stack. If False, then the images will be sorted by ordering a feature distance matix.
- **reference_img_f** (*str, optional*) – Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be the reference.
- **check_for_reflections** (*bool, optional*) – Determine if alignments are improved by relfecting/mirroring/flipping images. Optional because it requires re-detecting features in each version of the images and then re-matching features, and so can be time consuming and not always necessary.
- **similarity_metric** (*str*) – Metric used to calculate similarity between images, which is in turn used to build the distance matrix used to sort the images.
- **summary** (*Dataframe*) – Pandas dataframe containing the median distance between matched features before and after registration.
- **align_to_reference** (*bool, optional*) – Whether or not images should be aligned to a reference image specified by *reference_img_f*.
- **qt_emitter** (*PySide2.QtCore.Signal, optional*) – Used to emit signals that update the GUI's progress bars

Returns

registrar – SerialRigidRegistrar object contains general information about the alginments, but also a list of Z-images. Each ZImage contains the warp information for an image in the stack, including the transformation matrices calculated at each step, keypoint poisions, image descriptors, and matches with other images. See attributes from Zimage for more information.

Return type

SerialRigidRegistrar

Classes

ZImage

`class valis.serial_rigid.ZImage(`*image*, *img_f*, *img_id*, *name*`)`

Class store info about an image, including the rigid registration parameters

`image`

Greyscale image that will be used for feature detection. This images should be greyscale and may need to have undergone preprocessing to make them look as similar as possible.

Type
ndarray

`full_img_f`

full path to the image

Type
str

`img_id`

ID of the image, based on its ordering *processed_src_dir*

Type
int

`name`

Name of the image. Usually *img_f* but with the extension removed.

Type
str

`desc`

(N, M) array of N desciptors for each keypoint, each of which has M features

Type
ndarray

`kp_pos_xy`

(N, 2) array of position for each keypoint

Type
ndarray

`match_dict`

Dictionary of image matches. Key= img_obj this ZImage is being compared to, value= MatchInfo containing information about the comparison, such as the position of matches, features for each match, number of matches, etc… The MatchInfo objects in this dictionary contain only the info for matches that were considered "good".

Type
dict

`unfiltered_match_dict`

Dictionary of image matches. Key= img_obj this ZImage is being compared to, value= MatchInfo containing inoformation about the comparison, such as the position of matches, features for each match, number of matches, etc… The MatchInfo objects in this dictionary contain info for all matches that were cross-checked.

Type
dict

`stack_idx`
Position of image in sorted Z-stack

Type
int

`fixed_obj`
ZImage to which this ZImage was aligned, i.e. this is the "moving" image, and *fixed_obj* is the "fixed" image. This is set during the *align_to_prev* method of the SerialRigidRegistrar. The *fixed_obj* will either be immediately above or immediately below this ZImage in the image stack.

Type
ZImage

`reflection_M`
Transformation to reflect the image in the x and/or y axis, before padding. Will be the first transformation performed

Type
ndarray

`T`
Transformation matrix that translates the image such that it is in a padded image that has the same shape as all other images

Type
ndarray

`to_prev_A`
Transformation matrix that warps image to align with the previous image

Type
ndarray

`optimal_M`
Transformation matrix found by minimizing a cost function. Used as final optional step to refine alignment

Type
ndarray

`crop_T`
Transformation matrix used to crop image after registration

Type
ndarray

`M`
Final transformation matrix that aligns image in the Z-stack.

Type
ndarray

`M_inv`
Inverse of final transformation matrix that aligns image in the Z-stack.

Type
ndarray

`registered_img`

image after being warped

Type

ndarray

`padded_shape_rc`

Shape of padded image. All other images will have this shape

Type

tuple

`registered_shape_rc = tuple`

Shape of aligned image. All other aligned images will have this shape

`__init__`(*image*, *img_f*, *img_id*, *name*)

Class that stores information about an image

Parameters

- **image** (*ndarray*) – Greyscale image that will be used for feature detection. This images should be single channel uint8 images, and may need to have undergone preprocessing and/or normalization to make them look as similar as possible.
- **img_f** (*str*) – full path to *image*
- **img_id** (*int*) – ID of the image, based on its ordering in the image source directory
- **name** (*str*) – Name of the image. Usually img_f but with the extension removed.

SerialRigidRegistrar

`class valis.serial_rigid.SerialRigidRegistrar`(*img_dir*, *imgs_ordered=False*, *reference_img_f=None*, *name=None*, *align_to_reference=False*)

Class that performs serial rigid registration

Registration is conducted by first detecting features in all images. Features are then matched between images, which are then used to construct a distance matrix, D. D is then sorted such that the most similar images are adjcent to one another. The rigid transformation matrics are then found to align each image with the previous image. Optionally, optimization can be performed to improve the alignments, although the "optimized" matrix will be discarded if it increases the distances between matched features.

SerialRigidRegistrar creates a list and dictionary of ZImage objects, each of which contains information related to feature matching and the rigid registration matrices.

`img_dir`

Path to directory containing the images that will be registered. The images in this folder should be single channel uint8 images. For the best registration results, they have undergone some sort of pre-processing and normalization. The preprocessing module contains methods for this, but the user may want/need to use other methods.

Type

str

`aleady_sorted`

Whether or not the order of images already known. If True, the file names should start with ascending numbers, with the first image file having the smallest number, and the last image file having the largest number. If False (the default), the order of images will be determined by ordering a distance matrix.

Type
bool, optional

`name`
Descriptive name of registrar, such as the sample's name

Type
str

`img_file_list`
List of full paths to single channel uint8 images

Type
list

`size`
Number of images to align

Type
int

`distance_metric_name`
Name of distance metric used to determine the dis/similarity between each pair of images

Type
str

`distance_metric_type`
Name of the type of metric used to determine the dis/similarity between each pair of images. Despite the name, it could be "similarity" if the Matcher object compares image feautres using a similarity metric. In that case, similarities are converted to distances.

Type
str

`img_obj_list`
List of ZImage objects. Initially unordered, but eventually be sorted

Type
list

`img_obj_dict`
Dictionary of ZImage objects. Created to conveniently access ZIimages. Key = ZImage.name, value= ZImage

Type
dict

`optimal_Z`
Ordered linkage matrix for *distance_mat*

Type
ndarray

`unsorted_distance_mat`
Distance matrix with shape (N, N), where each element is the disimilariy betweewn each pair of the N images. The order of rows and columns reflects the order in which the images were read. This matrix is used to order the images the Z-stack.

Type
ndarray

`distance_mat`

unsorted_distance_mat reorderd such that the most similar images are adjacent to one another

Type
ndarray

`unsorted_similarity_mat`

Similar to *unsorted_distance_mat*, except the elements are image similarity

Type
ndarray

`similarity_mat`

Similar to *distance_mat*, except the elements are image similarity

Type
ndarray

`features`

Name of feature detector and descriptor used

Type
str

`transform_type`

Name of scikit-image transformer class that was used

Type
str

`reference_img_f`

Filename of image that will be treated as the center of the stack.

Type
str

`reference_img_idx`

Index of ZImage that corresponds to *reference_img_f*, after the *img_obj_list* has been sorted.

Type
int

`align_to_reference`

Whether or not images should be aligne to a reference image specified by *reference_img_f*. Will be set to True if *reference_img_f* is provided.

Type
bool, optional

`iter_order`

Each element of *iter_order* contains a tuple of stack indices. The first value is the index of the moving/current/from image, while the second value is the index of the moving/next/to image.

Type
list of tuples

`summary_df`

Pandas dataframe containin the registration error of the alignment between each image and the previous one in the stack.

Type
Dataframe

`__init__`(*img_dir*, *imgs_ordered=False*, *reference_img_f=None*, *name=None*, *align_to_reference=False*)

Class that performs serial rigid registration

Parameters

- `img_dir` (*str*) – Path to directory containing the images that will be registered. The images in this folder should be single channel uint8 images. For the best registration results, they have undergone some sort of pre-processing and normalization. The preprocessing module contains methods for this, but the user may want/need to use other methods.
- `imgs_ordered` (*bool*) – Whether or not the order of images already known. If True, the file names should start with ascending numbers, with the first image file having the smallest number, and the last image file having the largest number. If False (the default), the order of images will be determined by sorting a distance matrix.
- `reference_img_f` (*str, optional*) – Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be the reference.
- `name` (*str, optional*) – Descriptive name of registrar, such as the sample's name
- `align_to_reference` (*bool, optional*) – Whether or not images should be aligne to a reference image specified by *reference_img_f*. Will be set to True if *reference_img_f* is provided.

`summarize`()

Summarize alignment error

Returns
summary_df – Pandas dataframe containin the registration error of the alignment between each image and the previous one in the stack.

Return type
Dataframe

1.1.13 Serial non-rigid registration

Functions

Classes and functions to perform serial non-rigid registration of a set of images

`valis.serial_non_rigid.register_images`(*src*, *non_rigid_reg_class=<class 'valis.non_rigid_registrars.OpticalFlowWarper'>*, *non_rigid_reg_params=None*, *dst_dir=None*, *reference_img_f=None*, *moving_to_fixed_xy=None*, *mask=None*, *name=None*, *align_to_reference=False*, *img_params=None*, *compose_transforms=True*, *qt_emitter=None*)

Parameters

- `src` (`SerialRigidRegistrar`, *str*) – Either a SerialRigidRegistrar object that was used to optimally align a series of images, or a string indicating where the images to be aligned are located. If src is a string, the images should be named such that they are read in the correct order, i.e. each starting with a number.

- **non_rigid_reg_class** (NonRigidRegistrar) – Uninstantiated NonRigidRegistrar class that will be used to calculate the deformation fields between images. By default this is an OpticalFlowWarper that uses the OpenCV implementation of DeepFlow.
- **non_rigid_reg_params** (*dictionary, optional*) – Dictionary containing parameters {name: value} to be used to initialize the NonRigidRegistrar. In the case where simple ITK is used by the, params should be a SimpleITK.ParameterMap. Note that numeric values nedd to be converted to strings.
- **dst_dir** (*str, optional*) – Top directory where aliged images should be save. SerialNonRigidRegistrar will be in this folder, and aligned images in the "registered_images" sub-directory. If None, the images will not be written to file
- **reference_img_f** (*str, optional*) – Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be returned.
- **moving_to_fixed_xy** (*dict of list, or bool*) – If *moving_to_fixed_xy* is a dict of list, then Key = image name, value = list of matched keypoints between each moving image and the fixed image. Each element in the list contains 2 arrays:

  1. Rigid registered xy in moving/current/from image
  2. Rigid registered xy in fixed/next/to image

  To deterime which pairs of images will be aligned, use *warp_tools.get_alignment_indices*. Can use *get_imgs_from_dir* to see the order inwhich the images will be read, which will correspond to the indices retuned by *warp_tools.get_alignment_indices*.

  If *src* is a SerialRigidRegistrar and *moving_to_fixed_xy* is True, then the matching features in the SerialRigidRegistrar will be used. If False, then matching features will not be used.
- **mask** (*ndarray, bool, optional*) – Mask used in non-rigid alignments.

  If an ndarray, it must have the same size as the other images.

  If True, then the *overlap_mask* in the SerialRigidRegistrar will be used.

  If False or None, no mask will be used.
- **name** (*optional*) – Optional name for this SerialNonRigidRegistrar
- **align_to_reference** (*bool, optional*) – Whether or not images should be aligne to a reference image specified by *reference_img_f*. Will be set to True if *reference_img_f* is provided.
- **img_params** (*dict, optional*) – Dictionary of parameters to be used for each particular image. Useful if images to be registered haven't been processed. Will be passed to *non_rigid_reg_class* init and register functions. key = file name, value= dictionary of keyword arguments and values
- **qt_emitter** (*PySide2.QtCore.Signal, optional*) – Used to emit signals that update the GUI's progress bars

Returns
nr_reg – SerialNonRigidRegistrar that has registeredt the images in *src*

Return type
SerialNonRigidRegistrar

Classes

NonRigidZImage

class valis.serial_non_rigid.**NonRigidZImage**(*reg_obj*, *image*, *name*, *stack_idx*, *moving_xy=None*, *fixed_xy=None*, *mask=None*)

Class that store info about an image, including both rigid and non-rigid registration parameters

image

Original, unwarped image with shape (P, Q)

Type
ndarray

name

Name of image.

Type
str

stack_idx

Position of image in the stack

Type
int

moving_xy

(V, 2) array containing points in the moving image that correspond to those in the fixed image. If these are provided, non_rigid_reg_class should be a subclass of non_rigid_registrars.NonRigidRegistrarXY

Type
ndarray, optional

fixed_xy

(V, 2) array containing points in the fixed image that correspond to those in the moving image

Type
ndarray, optional

bk_dxdy

(2, N, M) numpy array of pixel displacements in the x and y directions from the reference image. dx = bk_dxdy[0], and dy=bk_dxdy[1]. Used to warp images

Type
ndarray

fwd_dxdy

Inversion of bk_dxdy. dx = fwd_dxdy[0], and dy=fwd_dxdy[1]. Used to warp points

Type
ndarray

warped_grid

Image showing deformation applied to a regular grid.

Type
ndarray

`__init__`(*reg_obj*, *image*, *name*, *stack_idx*, *moving_xy=None*, *fixed_xy=None*, *mask=None*)

Parameters

- **image** (*ndarray*) – Original, unwarped image with shape (P, Q)
- **name** (*str*) – Name of image.
- **stack_idx** (*int*) – Position of image in the stack
- **moving_xy** (*ndarray, optional*) – (V, 2) array containing points in the moving image that correspond to those in the fixed image. If these are provided, non_rigid_reg_class should be a subclass of non_rigid_registrars.NonRigidRegistrarXY
- **fixed_xy** (*ndarray, optional*) – (V, 2) array containing points in the fixed image that correspond to those in the moving image
- **mask** (*ndarray, optional*) – Mask covering area to be registered.

SerialNonRigidRegistrar

class valis.serial_non_rigid.**SerialNonRigidRegistrar**(*src*, *reference_img_f=None*, *moving_to_fixed_xy=None*, *mask=None*, *name=None*, *align_to_reference=False*, *compose_transforms=True*)

Class that performs serial non-rigid registration, based on results SerialRigidRegistrar

A SerialNonRigidRegistrar finds the deformation fields that will non-rigidly align a series of images, using the rigid registration parameters found by a SerialRigidRegistrar object. There are two types of non-rigid registration methods:

#. Images are aligned towards a reference image, which may or may not be at the center of the stack. In this case, the image directly "above" the reference image is aligned to the reference image, after which the image 2 steps above the reference image is aligned to the 1st (now aligned) image above the reference image, and so on. The process is similar when aligning images "below" the reference image.

#. All images are aligned simultaneously, and so a reference image is not # required. An example is the SimpleElastix groupwise registration.

Similar to SerialRigidRegistrar, SerialNonRigidRegistrar creates a list and dictionary of NonRigidZImage objects each of which contains information related to the non-rigid registration, including the original rigid transformation matrices, and the calculated deformation fields.

name

Optional name of this SerialNonRigidRegistrar

Type
str, optional

from_rigid_reg

Whether or not the images are from a SerialRigidRegistrar

Type
bool

ref_image_name

Name of mage that is being treated as the "center" of the stack. For example, this may be associated with an H+E image that is the 2nd image in a stack of 7 images.

Type
str

size

Number of images to align

Type
int

shape

Shape of each image to register. Must be the same for all images

Type
tuple of int

non_rigid_obj_dict

Dictionary, where each key is the name of a NonRigidZImage, and the value is the assocatiated NonRigidZ-Image

Type
dict

non_rigid_reg_params

Dictionary containing parameters {name: value} to be used to initialize the NonRigidRegistrar. In the case where simple ITK is used by the, params should be a SimpleITK.ParameterMap. Note that numeric values nedd to be converted to strings.

Type
dictionary

mask

Mask used in non-rigid alignments, with shape (P, Q).

Type
ndarray

mask_bbox_xywh

Bounding box of *mask* (top left x, top left y, width, height)

Type
ndarray

summary

Pandas dataframe containing the median distance between matched features before and after registration.

Type
Dataframe

__init__(*src*, *reference_img_f=None*, *moving_to_fixed_xy=None*, *mask=None*, *name=None*, *align_to_reference=False*, *compose_transforms=True*)

Parameters
src (`SerialRigidRegistrar`, *str*, *dict*) – A SerialRigidRegistrar object that was used to optimally align a series of images.

If a string, it should indicating where the images to be aligned are located. If src is a string, the images should be named such that they are read in the correct order, i.e. each starting with a number.

If a dictionary, it should contain the following key, value pairs:

”img_list” : list of images to register “img_f_list” : list of filenames of each image “name_list” : list of image names. If not provided, will come from file names “mask_list” list of masks for each image

reference_img_f
[str, optional] Filename of image that will be treated as the center of the stack. If None, the index of the middle image will be returned.

moving_to_fixed_xy
[dict of list, or bool] If *moving_to_fixed_xy* is a dict of list, then Key = image name, value = list of matched keypoints between each moving image and the fixed image. Each element in the list contains 2 arrays:

1. Rigid registered xy in moving/current/from image
2. Rigid registered xy in fixed/next/to image

To deterime which pairs of images will be aligned, use *get_alignment_indices*. Can use *get_imgs_from_dir* to see the order inwhich the images will be read, which will correspond to the indices retuned by *get_alignment_indices*.

If *src* is a SerialRigidRegistrar and *moving_to_fixed_xy* is True, then the matching features in the SerialRigidRegistrar will be used. If False, then matching features will not be used.

mask
[ndarray, bool, optional] Mask used for all non-rigid alignments.

If an ndarray, it must have the same size as the other images.

If True, then the *overlap_mask* in the SerialRigidRegistrar will be used.

If False or None, no mask will be used.

name
[optional] Optional name for this SerialNonRigidRegistrar

align_to_reference
[bool, optional] Whether or not images should be aligned to a reference image specified by *reference_img_f*.

img_params
[dict, optional] Dictionary of parameters to be used for each particular image. Useful if images to be registered haven’t been processed. Will be passed to *non_rigid_reg_class* init and register functions. key = file name, value= dictionary of keyword arguments and values

`register`(*non_rigid_reg_class*, *non_rigid_reg_params*, *img_params=None*)

Non-rigidly align images, either as a group or serially

Images will be registered serially if *non_rigid_reg_class* is a subclass of NonRigidRegistrarGroupwise, then groupwise registration will be conductedd. If *non_rigid_reg_class* is a subclass of NonRigidRegistrar then images will be aligned serially.

Parameters

- **non_rigid_reg_class** (`NonRigidRegistrar`, `NonRigidRegistrarGroupwise`) – Uninstantiated NonRigidRegistrar or NonRigidRegistrarGroupwise class that will be used to calculate the deformation fields between images
- **non_rigid_reg_params** (*dictionary, optional*) – Dictionary containing parameters {name: value} to be used to initialize the NonRigidRegistrar. In the case where simple ITK is used by the, params should be a SimpleITK.ParameterMap. Note that numeric values nedd to be converted to strings.

- **img_params** (`dict`, *optional*) – Dictionary of parameters to be used for each particular image. Useful if images to be registered haven't been processed. Will be passed to *non_rigid_reg_class* init and register functions. key = file name, value= dictionary of keyword arguments and values

`summarize()`

Summarize alignment error

Returns

summary_df – Pandas dataframe containin the registration error of the alignment between each image and the previous one in the stack.

Return type

Dataframe

1.1.14 Visualization

Various functions used to visualize registration results

`valis.viz.cam16ucs_cmap`(*luminosity=0.8*, *colorfulness=0.5*, *max_h=300*)

Get colormap based on CAM16-UCS colorspace.

Parameters

- **luminosity** (`float`, *optional*) –
- **colorfulness** (`float`, *optional*) –
- **max_h** (`int`, *optional*) –

`valis.viz.color_displacement_grid`(*bk_dx*, *bk_dy*, *c_range=(0, 0.025)*, *l_range=(0.004, 0.015)*, *thickness=None*, *grid_spacing_ratio=0.02*, *cspace='JzAzBz'*)

Color a displacement grid

`valis.viz.color_displacement_tri_grid`(*bk_dx*, *bk_dy*, *img=None*, *n_grid_pts=25*, *c_range=(0, 0.025)*, *l_range=(0.004, 0.015)*, *thickness=None*, *cspace='JzAzBz'*)

View how a displacement warps a triangular mesh.

`valis.viz.color_dxdy`(*dx*, *dy*, *c_range=(0, 0.025)*, *l_range=(0.004, 0.015)*, *cspace='JzAzBz'*)

Color displacement, where larger displacements are more colorful, and, if scale_l=True, brighter.

dx: array
1D Array containing the displacement in the X (column) direction

dy: array
1D Array containing the displacement in the Y (row) direction

c_range: (float, float)
Minimum and maximum colorfulness in JzAzBz colorspace

l_range: (float, float)
Minimum and maximum luminosity in JzAzBz colorspace

scale_l: boolean
Scale the luminosity based on magnitude of displacement

displacement_rgb
[array] RGB (0, 255) color for each displacement, with the same shape as dx and dy

`valis.viz.`**`color_multichannel`**(*multichannel_img*, *marker_colors*, *rescale_channels=False*, *normalize_by='image'*, *cspace='Hunter Lab'*)

Color a multichannel image to view as RGB

Parameters

- **multichannel_img** (*ndarray*) – Image to color
- **marker_colors** (*ndarray*) – sRGB colors for each channel.
- **rescale_channels** (*bool*) – If True, then each channel will be scaled between 0 and 1 before building the composite RGB image. This will allow markers to 'pop' in areas where they are expressed in isolation, but can also make it appear more marker is expressed than there really is.
- **normalize_by** (*str, optionaal*) – "image" will produce an image where all values are scaled between 0 and the highest intensity in the composite image. This will produce an image where one can see the expression of each marker relative to the others, making it easier to compare marker expression levels.

  "channel" will first scale the intensity of each channel, and then blend all of the channels together. This will allow one to see the relative expression of each marker, but won't allow one to directly compare the expression of markers across channels.
- **cspace** (*str*) – Colorspace in which *marker_colors* will be blended. JzAzBz, Hunter Lab, and sRGB all work well. But, see colour.COLOURSPACE_MODELS for other possible colorspaces

Returns

rgb – An sRGB version of *multichannel_img*

Return type

ndarray

`valis.viz.`**`draw_features`**(*kp_xy*, *image*, *n_features=500*)

Draw keypoints on a image

`valis.viz.`**`draw_matches`**(*src_img*, *kp1_xy*, *dst_img*, *kp2_xy*, *rad=3*, *alignment='horizontal'*)

Draw feature matches between two images

Parameters

- **src_img** (*ndarray*) – Image associated with *kp1_xy*
- **kp1_xy** (*ndarray*) – xy coordinates of feature points found in *src_img*
- **dst_img** (*ndarray*) – Image associated with *kp2_xy*
- **kp2_xy** (*ndarray*) – xy coordinates of feature points found in *dst_img*
- **rad** (*int*) – Radius of circles used to draw feature points
- **alignment** (*string*) – How to stack the images, either 'veritcal' or 'horizontal'.

Returns

feature_img – Image show corresponding features of *src_img* and *dst_img*

Return type

ndarray

`valis.viz.`**`get_n_colors`**(*rgb*, *n*)

Pick n most different colors in rgb. Differences based of rgb values in the CAM16UCS colorspace Based on https://larssonjohan.com/post/2016-10-30-farthest-points/

valis.viz.**jzazbz_cmap**(*luminosity=0.012*, *colorfulness=0.02*, *max_h=260*)

Get colormap based on JzAzBz colorspace, which has good hue linearity. Already preceptually uniform.

Parameters

- **luminosity** (*float*, *optional*) –
- **colorfulness** (*float*, *optional*) –
- **max_h** (*int*, *optional*) –

valis.viz.**turbo_cmap**()

Turbo colormap https://gist.github.com/mikhailov-work/ee72ba4191942acecc03fe6da94fc73f

1.1.15 Change Log

Version 1.0.4 (February 2, 2024)

1. Added checks and unit tests to verify reference image is not being warped. Can confirm no transformations were being applied to the reference image, but values may have differed slightly due to interpolation effects, as the reference image was being padded and cropped. To avoid these interpolation effects, the original reference image is returned when "warping" the reference slide with `crop="reference"`, which occurs regardless of the `align_to_reference` setting. This avoids unnecessary computation and potential interpolation errors. There are still checks to make sure that the warped image would have been the same as the unwarped image.
2. Merge channels based on the position of each slide in the stack. This will be the same as the original order when `imgs_ordered=True`.
3. Can provide colormaps for each slide when calling `Valis.warp_and_save_slides`
4. Added `pyramid` argument to `Valis.warp_and_save_slides`
5. Ignore setting series=0 message when there is only 1 series
6. Updated openCV version in project.toml, as suggested in Github issue 76
7. Added `slide_io.check_xml_img_match` to determine if there are mismatches between the xml metadata and the image that was read. If there are mismatches, the metadata will be updated based on the image (instead of the xml) and warning messages will be printed to let the user know about the mismatch.
8. If a single channel image does not have channel names in the metadata, the channel name will be set to the image's name.
9. Added `denoise_rigid` as an argument to initialize the `Valis` object. Determines whether or not to denoise the processed images prior to rigid registration. Had been fixed as True in previous versions, but this makes it optional (default remains True).
10. Fixed issue where merged images were being interpreted as RGB and saved with extra channels (reported in Github issue 76)

Version 1.0.3 (January 25, 2024)

1. Can specify which slide readers to use for each image by passing a dictionary to the `reader_dict` argument in `Valis.register`. The keys, value pairs are image filename and instantiated `SlideReader` to use to read that file. Valis will try to find an appropriate reader for any omitted files. Can be especially useful in cases where one needs different series for different images, as the `series` argument is set when the `SlideReader` is created.
2. Each `Slide` is assigned a `SlideReader`, ensuring that the same series will always be read.
3. Added traceback messages to critical try/except blocks to help with debugging.
4. Micro-registration can now use multiple image processors, and so should be able to perform multi-modal registration
5. Now possible to save the images as non-pyarmid WSI by setting `pyramid=False` when calling the various slide saving methods (requested in github issue 56).
6. Tested the `slide_io.CziJpgxrReader` with more jpegxr compressed czi images, including 3 RGB (2 mosaic, 1 not mosaic), 1 multichannel non-RGB (mosaic), 1 single channel (mosaic). Related to github issue 76.
7. Added checks to make sure all channel names are in the colormap, addressing github issues 78 and 86 .
8. Setting `colormap=None` to the various save functions will not add any color channels, and so the slide viewer's default colormaps will be used.
9. Updated `slide_io.get_slide_reader` to give preference to reading images with libvips/openslide. Should be faster since image will not need to be constructed from tiles.
10. JVM will only be initialized if bioformats is needed to read the image.
11. Updated `slide_io.VipsSlideReader` to use the ome-types pacakge to extract metadata, instead of Bio-formats. Should avoid having to launch JVM unless Bio-formats is really needed.
12. Added checks to ensure that channels in merged image are in the correct order when `imgs_ordered=True`, addressing the comment github issue 56 .
13. Added tests for images with minimal ome-xml (i.e. no channel names, units, etc…)
14. Removed usage of `imghdr`, which is being deprecated
15. Replaced joblib with pqdm. May resolve issue posted on image.sc
16. Removed interpolation and numba packages as dependencies
17. Updated ome-types' parser to "lxml"
18. Merged github pull request 95.

Version 1.0.2 (October 11, 2023)

1. Fix issue with pip installation, where the pyproject.toml tried to get aicspylibczi from Github, not PyPi

Version 1.0.1 (October 6, 2023)

1. Bug fixes to functions related to saving slides as ome.tiff
2. Address numba deprecation warnings

Version 1.0.0 (October 4, 2023)

1. Added option for high resolution rigid registration using the `micro_rigid_registrar.MicroRigidRegistrar` class. To use this option, pass an uninstantiated `micro_rigid_registrar.MicroRigidRegistrar` to `micro_rigid_registrar_cls` when initializing the `Valis` object. This class refines the rigid registration by detecting and matching features in higher resolution images warped using the initial rigid transforms. This should result in more accurate rigid registration, error estimates, and hopefully fewer unwanted non-rigid deformations.
2. Added support for SuperPoint and SuperGlue feature detection and matching
3. Masks not applied to images for rigid registration. Instead, the masks are used to filter the matches (i.e. keep only matches inside masks).
4. Added support to extract different Z and T planes using the `slide_io.BioFormatsSlideReader`.
5. Non-rigid masks found by combining the intensities of the rigidly aligned images, as opposed to combining the rigid masks. Testing indicates these masks fit more tightly around the tissue, which will translate to having higher resolution images being used for non-rigid registration.
6. Added the `preprocessing.StainFlattener` class, which can be used with brightfield images.
7. Added option for lossy compression by setting the `Q` parameter using functions that save slides. Confirmed that RGB images saved using lossy JPEG and JPEG2000 compression open as expected in QuPath. Do note that float images saved using these compression methods will get cast to uint8 images. Addresses request made in github issue 60.
8. Added `Valis.draw_matches` method to visualize feature matches between each pair of images.
9. Fixed issue converting big-endian WSI to `pyvips.Image` (reported on *image.sc <https://forum.image.sc/t/problems-registering-fluorescence-ome-tiffs-using-valis/82685>*_)
10. Added citation information
11. Updated docker container to include pytorch

Version 1.0.0rc15 (May 10, 2023)

1. Added import for `aicspylibczi.CziFile` in `slide_io` (found in github issue 44). Also added `aicspylibczi` to the poetry lock file.
2. Added `src_f` argument in `Slide.warp_and_save_slide`. Previously would end up using the `Slide.src_f`, and preventing one from being able to warp and save other images using the same transformations (github issue 49).
3. Various bug fixes to allow the initial non-rigid registration to work with larger images (which may be `pyvips.Image` objects).
4. Fixed bug where errors that occurred while reading images would prevent Python from shutting down.
5. Updated documentation for `valis.preprocessing`
6. Added more tests
7. Fixed many typos in the documentation.

Version 1.0.0rc14 (April 24, 2023)

1. Added `max_ratio` as an argument for `feature_matcher.match_desc_and_kp` (github issue 36).
2. Added `CziJpgxrReader` to read CZI images that have JPGXR compression but can't be opened with Bioformats. It's very limited and experimental (only tested with single scence RGB), but may be an issue specific to Apple silcon?
3. Supports scenario where images might be assigned the same name (e.g. same file names, but different directories).
4. Support tiling for initial non-rigid registration, making it possible to perform non-rigid on much larger images
5. Skips empty images (github issue 44).
6. Can now specify an `ImageProcesser` for each image by passing a dicitonary to the `processor_dict` argrument of `Valis.register`. Keys should be the filename of the image, and values a list, where the first element is the `ImageProcesser` to use, and the second element is a dictionary of keyword argruments passed to `ImageProcesser.process_image`. This should make it easier to register different image modalities.
7. Added an H&E color deconvolution `ImageProcesser` using the method described in M. Macenko et al., ISBI 2009. Generously provided by Github user aelskens (Arthur Elskenson) (PR 42).
8. Small improvements in `valtils` functions, provided by Github user ajinkya-kulkarni (Ajinkya Kulkarni) (PR 46).
9. Docker Images bundled with bioformats jar file, so does not require internet connection or Maven. Also now checks for bioformats jar in valis folder
10. Fixed bug that threw error when trying to warp an empty Shapely polygon
11. Fixed bug in micro-registration, related to trying to combine numpy and pyvips images (github issues 40 and 47)
12. Fixed typo in "max_non_rigid_registration_dim_px", which was "max_non_rigid_registartion_dim_px" (github issue 39)
13. Fixed error that caused excessive memory consumption when trying to mask numpy array with pyvips image in `preprocessing.norm_img_stats`

Version 1.0.0rc13 (January 31, 2023)

1. Now available as a Docker image
2. Added methods to transfer geojson annotations, such as those generated by QuPath, from one slide to another (`Slide.warp_geojson_from_to` and `Slide.warp_geojson`). Also provide examples in documentation. Addresses github issue 13
3. Fixed bug reported in github issue 33
4. Default is to not compose non-rigid transformations, reducing accumulation of unwanted distortions, especially in 3D.
5. The `scale_factor` parameter for `feature_detectors.VggFD` is now set to 5.0, as per the OpenCV documentation
6. Installlation now uses poetry via the pyproject.toml file. Includes a poetry.lock file, but it can be deleted before installation if so desired.
7. Removed bioformats_jar as a dependency
8. Added a datasets page
9. Moved examples to separate page

Version 1.0.0rc12 (November 7, 2022)

1. Fixed bug where would get out of bounds errors when cropping with user provided transformations (github issue 14 https://github.com/MathOnco/valis/issues/14)
2. Fixed bug where feature matches not drawn in correct location in `src_img` in `viz.draw_matches`.
3. Can now check if refelcting/mirroring/flipping images improves alignment by setting `check_for_reflections=True` when initializing the `Valis` object. Addresses githib issue 22 (https://github.com/MathOnco/valis/issues/22)
4. Channel colors now transfered to registered image (github issue 23 https://github.com/MathOnco/valis/issues/23). Also option to provide a colormap when saving the slides. This replaces the `perceputally_uniform_channel_colors` argument

Version 1.0.0rc11 (August 26, 2022)

1. Fixed bug when providing rigid transformations (Issue 14, https://github.com/MathOnco/valis/issues/14).
2. Can now warp one image onto another, making it possible to transfer annotations using labeled images (Issue 13 https://github.com/MathOnco/valis/issues/13). This can be done using a Slide object's `warp_img_from_to` method. See example in examples/warp_annotation_image.py
3. `ImageProcesser` objects now have a `create_mask` function that is used to build the masks for rigid registration. These are then used to create the mask used for non-rigid registration, where they are combined such that the final mask is where they overlap and/or touch.
4. Non-rigid registration performed on higher resolution version of the image. The area inside the non-rigid mask is sliced out such that it encompasses the area inside the mask but has a maximum dimension of `Valis.max_non_rigid_registartion_dim_px`. This can improve accuracy when the tissue is only a small part of the image. If masks aren't created, this region will be where all of the slides overalp.
5. Version used to submit results to the ACROBAT Grand Challenge. Code used to perform registration can be found in examples/acrobat_grand_challenge.py. This example also shows how to use and create a custom `ImageProcesser` and perform micro-registration with a mask.

Version 1.0.0rc10 (August 11, 2022)

1. Fixed compatibility with updated interpolation package (Issue 12).

Version 1.0.0rc9 (August 4, 2022)

1. Reduced memory usage for micro-registration and warping. No longer copying memory before warping, and large displacement fields saved as .tiff images instead of .vips images.
2. Reduced unwanted accumulation of displacements
3. `viz.draw_matches` now returns an image instead of a matplotlib pyplot
4. Pull request 9-11 bug fixes (many thanks to crobbins327 and zindy): Not converting uint16 to uint8 when reading using Bio-Formats or pyvips; fixed rare error when filtering neighbor matches; `viz.get_grid` consistent on Linux and Windows; typos.

Version 1.0.0rc8 (July 1, 2022)

1. Now compatible with single channel images. These images are treated as immunofluorescent images, and so custom pre-processing classes and arguments should be passed to `if_processing_cls` and `if_processing_kwargs` of the `Valis.register` method. The current method will perform adaptive histogram equalization and scales the image to 0-255 (see `preprocessing.ChannelGetter`). Also, since it isn't possible to determine if the single channel image is a greyscale RGB (light background) or single channel immunofluorescence (or similar with dark background), the background color will not be estimated, meaning that in the registered image the area outside of the warped image will be black (as opposed to the estimated background color). Tissue masks will still be created, but if it seems they are not covering enough area then try setting `create_masks` to *False* when initializing the `Valis` object.

Version 1.0.0rc7 (June 27, 2022)

1. Can set size of image to be used for non-rigid registration, which may help improve aligment of micro-architectural structures. However this will increase the amount of time it takes to perform non-rigid registration, and will increase amount of memory used during registration, and the size of the pickled :code: *Valis* object. To change this value, set the `max_non_rigid_registartion_dim_px` parameter when initializing the `Valis` object.
2. Can now do a second non-rigid registartion on higher resolution images, including the full resolution one. This can be done with the `Valis.register_micro`. If the images are large, they will be sliced into tiles, and then each tile registered with one another. The deformation fields will be saved separately as .vips images within the data folder.
3. Added `registration.load_registrar` function to open a `Valis` object. This should be used instead of *pickle.load*.
4. Creating and applying tissue masks before registration. This improves image normalization, reduces the number of poor feature matches, and helps remove unwanted non-rigid deformations (especially around the image edges), all of which improve alignment accuracy. This step can be skipped by setting `create_masks` to *False* when initializing the `Valis` object.
5. Now possible to directly non-rigidly align to the reference image specified by `reference_img_f`. This can be done by setting `align_to_reference` to *True* when initializing the `Valis` object. The default is *False*, which means images will be aligned serially towards the reference image. This option is also available with `Valis.register_micro`, meaning that one could do a second alignment, but aligning all directly to a reference image.
6. RANSAC filtered matches found for rigid registration undergo second round of filtering, this time using Tukey's method to remove matches whose distance after being warped would be considered outliers.
7. Now have option off whether or not to compose non-rigid transformations. This can be set specifying the `compose_non_rigid` argument when initialzing the *Valis* object.
8. Can provide rigid transformation matrices by passing in a dictionary to the `do_rigid` parameter when initializing the `Valis` object. Setting `do_rigid` to *False* will completely skip the rigid registration step. See the documentation for initializing the *Valis* object for more details.
9. Added examples of how to read slides and use custom transforms
10. Benchmarked using ANHIR Grand Challenge dataset and posted results on leaderboard.
11. bioformats_jar has been deprecated, so added support for its replacement, scyjava. However, the default behavior will be to use the bioformats_jar JAR file if it's already been installed. One can also now specify the JAR file when calling `init_jvm`.

Version 1.0.0rc6 (April 18, 2022)

1. More accurate color mixing with fewer artifacts. Affects overlap images and pseudo-colored multi-channel images.
2. Initializing 'is_flattended_pyramid' with False. Pull request #6
3. Reformatting flattened pyramids to have same datatype as that in metadata.
4. Saving all images using pyvips. Should be faster.
5. Using Bio-Formats to read non-RGB ome-tiff. Addresses an issue where converting non-RGB ome-tiff to numpy was very slow.

Version 1.0.0rc5 (April 5, 2022)

1. Can provide a reference image that the others will be aligned towards. To do this, when initializinig the Valis object, set the `reference_img_f` argument to be the file name of the reference image. If not set by the user, the reference image will be set as the one at the center of the ordered image stack
2. Both non-rigid and rigid now align *towards* a reference image, meaning that reference image will have neither rigid nor non-rigid transformations applied to it.
3. Two cropping methods. First option is to crop seach registered slides to contain only the areas where all registered images overlap. The second option is to crop the registered slide to contain only the area that intersects with the reference image. It is also possible to not crop an image/slide.
4. Images are now cropped during the warp, not after, and so is now faster and requires less memory. For example, on a 2018 MacBook Pro with a 2.6 GHz Intel Core i7 processor, it takes 2-3 minutes to warp and save a 41399 x 43479 RGB image.
5. Warping of images and slides done using the same function, built around pyvips. Faster, more consistent, and should prevent excessive memory usage.
6. Fixed bug that caused a crash when warping large ome.tiff images.
7. Read slides and images using pyvips whenever possible.
8. Background color now automatically set to be same as the brightest (IHC) or darkest (IF) pixel in the image. Because of this, the "bg_color" argument in the slide warping functions was removed.
9. Reduced accumulation of unwanted non-rigid deformations
10. Displacement fields drawn on top of non-rigid registered image to help determine where the deformations occured.
11. If a slide has multiple series, and a series is not specficed, the slide reader will read the series containing the largest image.

1.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX