florin

Release 0.0.1a

Jun 11, 2019

Contents:

FLoRIN, the Flexible Learning-free Reconstruction of Neural Volumes pipeline, is a pipeline for large-scale parallel and distributed computer vision. Offering easy setup and access to hierarchical parallelism, FLorIN is ideal for scaling computer vision to HPC systems.

Originally, this project was our response to the question of how to segment and reconstruct neural microscopy (e.g., micro-CT tomography, low-resolution electron microscopy, fluorescence microscopy etc.) without large amounts of training data available to train a neural network. We tackled this problem by revisiting classical computer vision methods, eventually developing the N-Dimensional Neighborhood Thresholding (NDNT) algorithm as a modern update to integral image-based thresholding. FLoRIN has since been shown to be a fast, robust segmentation and reconstruction engine across different imaging modalities and datasets.

This package implements the NDNT algorithm, as well as a straightforward API for mixed serial, parallel, and distributed computer vision. These docs provide examples of how to use FLoRIN with various mixtures of serial and parallel processing and how to customize the FLoRIN pipeline with new functions and features.

CHAPTER 1

Installation

pip

```
pip install florin
```

anaconda

```
conda install -c jeffkinnison florin
```

CHAPTER 2

Publications

1. Shahbazi, Ali, Jeffery Kinnison, Rafael Vescovi, Ming Du, Robert Hill, Maximilian Jösch, Marc Takeno et al. "Flexible Learning-Free Segmentation and Reconstruction of Neural Volumes." Scientific reports 8, no. 1 (2018): 14247.

2.1 Installation

FLoRIN can be installed with all of its Python dependencies through the Python Package Index or Anaconda.

2.1.1 PyPI

```
pip install florin
```

2.1.2 Anaconda

```
conda install -c jeffkinnison florin
```

2.1.3 Python Dependencies

- Python 3.4+
- numpy
- scipy
- scikit-image
- pathos
- mpi4py

- h5py

2.2 Examples using FLoRIN

2.2.1 A First Example

This example will walk through basic FLoRIN usage segmenting and reconstructing a small X-Ray volume.

Segmenting

The following code sets up a serial pipeline to segment the image:

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    # Threshold with NDNT
    thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

    # Clean up a little bit
    morphology.binary_opening(),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff')
)

# Run the pipeline
segmented = pipeline()
```

At the end of the pipeline, a TIFF stack with the binary segmentation will be output.

Weak Classification

After we have the binary mask, we want to determine what type of structure each object is. The previous pipeline can be extended to perform weak classification by user-defined bounds on the segmented objects:

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
```

(continues on next page)

(continued from previous page)

```
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    # Threshold with NDNT
    thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

    # Clean up a little bit
    morphology.binary_opening(),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff'),

    # Find connected components
    conncomp.label(),
    morphology.remove_small_holes(min_size=20),
    conncomp.regionprops(),

    # Classify the connected components by their volume and dimensions
    florin.classify(
        florin.bounds_classifier(
            'cell',
            area=(100, 300),
            depth=(10, 25),
            width=(50, 100),
            height=(50, 100)
        ),
        florin.bounds_classifier('vasculature')
    ),

    # Reconstruct the labeled volume
    florin.reconstruct(),

    # Write out the labeled volume
    florin.save('labeled.tiff')
)

# Run the pipeline
segmented = pipeline()
```

This pipeline save both the binary segmentation and the labeled volume where each class is represented by a different color.

Closing Remarks

Rolling out a basic FLoRIN pipeline is relatively easy (20 lines of code without the comments and whitespace). This example runs everything on a single cores, but the next example demonstrates parallel processing, which is just as easy to set up.

2.2.2 Parallel Processing Pipelines

This example will show how to convert the previous example to perform multiprocessing on the tiles and connected components created during segmentation and weak classification, respectively.

Parallelism

Parallel processing can be invoked by creating sub-pipelines around commands that will receive multiple inputs.

Multithreading

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    florin.Multithread(
        # Threshold with NDNT
        thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

        # Clean up a little bit
        morphology.binary_opening()
    ),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff'),

    # Find connected components
    conncomp.label(),
    morphology.remove_small_holes(min_size=20),
    conncomp.regionprops(),

    # Classify the connected components by their volume and dimensions
    florin.Multithread(
        florin.classify(
            florin.bounds_classifier(
                'cell',
                area=(100, 300),
                depth=(10, 25),
                width=(50, 100),
                height=(50, 100)
            ),
            florin.bounds_classifier('vasculature')
        )
    )

    # Reconstruct the labeled volume
    florin.reconstruct(),

    # Write out the labeled volume
    florin.save('labeled.tiff')
)

# Run the pipeline
segmented = pipeline()
```

Multiprocessing

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    florin.Multiprocess(
        # Threshold with NDNT
        thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

        # Clean up a little bit
        morphology.binary_opening()
    ),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff'),

    # Find connected components
    conncomp.label(),
    morphology.remove_small_holes(min_size=20),
    conncomp.regionprops(),

    # Classify the connected components by their volume and dimensions
    florin.Multiprocess(
        florin.classify(
            florin.bounds_classifier(
                'cell',
                area=(100, 300),
                depth=(10, 25),
                width=(50, 100),
                height=(50, 100)
            ),
            florin.bounds_classifier('vasculature')
        )
    )

    # Reconstruct the labeled volume
    florin.reconstruct(),

    # Write out the labeled volume
    florin.save('labeled.tiff')
)

# Run the pipeline
segmented = pipeline()
```

MPI

```python
import florin
```

(continues on next page)

(continued from previous page)

```
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    florin.MPI(
        # Threshold with NDNT
        thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

        # Clean up a little bit
        morphology.binary_opening()
    ),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff'),

    # Find connected components
    conncomp.label(),
    morphology.remove_small_holes(min_size=20),
    conncomp.regionprops(),

    # Classify the connected components by their volume and dimensions
    florin.MPI(
        florin.classify(
            florin.bounds_classifier(
                'cell',
                area=(100, 300),
                depth=(10, 25),
                width=(50, 100),
                height=(50, 100)
            ),
            florin.bounds_classifier('vasculature')
        )
    )

    # Reconstruct the labeled volume
    florin.reconstruct(),

    # Write out the labeled volume
    florin.save('labeled.tiff')
)

# Run the pipeline
segmented = pipeline()
```

All of these examples scale to the number of availble cores (or MPI ranks in the MPI version), and can be parameterized to use a specific number when the sub-pipelines are created.

Mixed Parallelism

Using the sub-pipeline model in the above example, it is possible to mix parallel processing paradigms. For example, segmenting tiles with NDNT uses vectorized operations and may be better suited to multi-node parallelism with MPI, but classification is more lightweight and can be carried out in threads. This sort of a pipeline would look like:

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    florin.MPI(
        # Threshold with NDNT
        thresholding.ndnt(shape=(10, 64, 64), thresshold=0.3),

        # Clean up a little bit
        morphology.binary_opening()
    ),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff'),

    # Find connected components
    conncomp.label(),
    morphology.remove_small_holes(min_size=20),
    conncomp.regionprops(),

    # Classify the connected components by their volume and dimensions
    florin.Multithread(
        florin.classify(
            florin.bounds_classifier(
                'cell',
                area=(100, 300),
                depth=(10, 25),
                width=(50, 100),
                height=(50, 100)
            ),
            florin.bounds_classifier('vasculature')
        )
    )

    # Reconstruct the labeled volume
    florin.reconstruct(),

    # Write out the labeled volume
    florin.save('labeled.tiff')
)

# Run the pipeline
segmented = pipeline()
```

In this case, an implicit join after the MPI pipeline converts merges the segmented tiles into a single volume. Connected components are then computed over the whole volume and classified concurrently using a multithreading model.

Closing Remarks

Parallel processing with FLoRIN is as easy as specifying the type of parallel pipeline to use, and they are roughly interchangeable (MPI requires using the standard `mpirun` or `mpiexec` invocations, or an equivalent).

2.2.3 Using Custom Functions in FLoRIN

Because of the wide array of computer vision methods, FLoRIN comes with utilities to prepare functions. This section will go over the two cases for preparing functions: without parameters, and with parameters.

Single-Argument Functions

Functions with a single argument (e.g., those taking a single image or a single numpy array and no other arguments) require no additional preparation. This example shows how to incorporate `np.squeeze` into a pipeline:

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

import numpy as np

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    # Remove any axes with shape 1. Simply pass np.squeeze without invoking
    np.squeeze,

    # Threshold with NDNT
    thresholding.ndnt(shape=(10, 64, 64), threshold=0.3),

    # Clean up a little bit
    morphology.binary_opening(),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff')
)

# Run the pipeline
segmented = pipeline()
```

Note that `np.squeeze` is not invoked. The function is just passed to the pipeline as-is, and FLoRIN will call it later.

Parameterizing Functions with `florinate`

Functions with parameters can also be used within FLoRIN by wrapping them with `florin.florinate`. This function records any parameters passed while setting up the pipeline and then automatically applies them when the data comes through (i.e. partial function application):

```
.. content-tabs::
```

Decorator

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Create the custom function and decorate it with ``florinate``
@florin.florinate
def scale(image, scalar=1):
    """Scale an images values by some number.

    Parameters
    ----------
    image : array_like
    scale : int or float

    Returns
    -------
    image * scale
    """
    return image * scale

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    # Add the custom function to the pipeline
    scale(scalar=2.0),

    # Threshold with NDNT
    thresholding.ndnt(shape=(10, 64, 64), threshold=0.3),

    # Clean up a little bit
    morphology.binary_opening(),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff')
)

# Run the pipeline
segmented = pipeline()
```

In-Line

```python
import florin
import florin.conncomp as conncomp
import florin.morphology as morphology
import florin.thresholding as thresholding

# Create the custom function
def scale(image, scalar=1):
    """Scale an images values by some number.

    Parameters
    ----------
    image : array_like
    scale : int or float

    Returns
    -------
    image * scale
    """
    return image * scale

# Set up a serial pipeline
pipeline = florin.Serial(
    # Load in the volume from file
    florin.load(),

    # Add the custom function to the pipeline and wrap it in ``florinate``
    florin.florinate(scale)(scalar=2.0),

    # Tile the volume into overlapping 64 x 64 x 10 subvolumes
    florin.tile(shape=(10, 64, 64), stride=(10, 32, 32)),

    # Add the custom function to the pipeline
    scale(scalar=2.0),

    # Threshold with NDNT
    thresholding.ndnt(shape=(10, 64, 64), threshold=0.3),

    # Clean up a little bit
    morphology.binary_opening(),

    # Save the output to a TIFF stack
    florin.save('segmented.tiff')
)

# Run the pipeline
segmented = pipeline()
```

`florinate` will handle any number of arguments and keyword arguments passed to it, applying them every time the function is called during the pipeline.

Why `florinate`?

The `functools` module already has an implementation of partial functions (`functools.partial`), the the natural question is: why reinvent the wheel? When building FLoRIN, we noticed that most computer vision functions take the image as the *first* argument; `functools.partial`, however will only *append* arguments when called. `florinate` solves this by *prepending* the argument(s) when called, lining up with the norm for computer vision APIs.

If a custom function takes the image as the *last* argument, `functools.partial` can be used in place of `florinate` with no changes.

2.2.4 Adding New Pipeline Types to FLoRIN

FLoRIN offers a number of pipeline options (Serial, Multithread, Multiprocess, etc.) out of the box, but what if you need a different model? This example will show how to create a custom pipeline class with a different style of execution.

SLURMPipeline

Suppose you work on a cluster that uses SLURM and want to submit a job to a queue. This requires a pipeline that

1. Accepts parameters to configure `sbatch`
2. Sets up a job script
3. Submits the job script for processing
4. Blocks until all jobs are finished

Such a pipeline may look like this

```python
import re
import subprocess
import time

import dill  # dill is installed with florin

from florin.pipelines import Pipeline

class SLURMPipeline(Pipeline):
    """Pipeline that sets up and runs a SLURM job.

    Parameters
    ----------
    operations : callables
        The functions of the pipeline.

    Other Parameters
    ----------------
    Keyword arguments corresponding to SLURM directives, e.g. qos='debug',
    time=60, etc. These are dynamically added to the jobscript before
    submission.
    """

    def __init__(self, *operations, **kwargs):
        super(SLURMPipeline, self).__init__(*operations)
        self.slurm_directives = kwargs

    def run(self, data):
        """Submit and run a pipeline on SLURM.

        Parameters
        ----------
        data : list
            The input to the first function in the pipeline, e.g. a
```

(continues on next page)

(continued from previous page)

```
            filepath for florin.load().
        """
        # Serialize this current pipeline
        pipeline_path = 'my_pipeline.pkl'
        self.dump(pipeline_path)

        # Set up the job script. This sets up the shebang header, then
        # iterates over the provided #SBATCH disrectives and sets each one
        # up on its own line, then finally invokes srun to deserialize the
        # pipeline and run it on the data.
        jobscript = "#/usr/bin/env bash"
        jobscript = '\n'.join(
            ['#!/usr/bin/env bash'] +
            ['#SBATCH --{}={}'.format(key, val) for key, val in self.slurm_directives.
↪items()] +
            ['srun python -m florin.run {} $1'.format(pipeline_path)])

        # Dump the jobscript to file
        with open('my_jobscript.job', 'w') as f:
            f.write(jobscript)

        jobids = []

        # Submit one job for each data item.
        for item in data:
            out = subprocess.check_output(['sbatch', my_jobscript, item])
            jobids.append(re.search(r'([\d]+)', out).group())

        # Wait until all jobs have completed to exit.
        while len(jobids) > 0:
            time.sleep(10)
            completed = []

            for jid in jobids:
                out = subprocess.check_output(['sacct', '-j', jid])
                if re.search(r'(COMPLETE)', out):
                    completed.add(jid)

            for jid in completed:
                jobids.remove(jid)
```

Note that this code is untested and by no means guaranteed to work, it is only meant to be a non-trivial example of what a custom pipeline may look like.

Other Examples

Another great source of examples for setting up custom pipelines is the `florin.pipelines` module, where the source code for the officially supported pipelines.

2.3 API Documentation

`florin`	The FLoRIN pipeline for large-scale learning-free computer vision.
`florin.classification`	Utilities for classifying connected components.
`florin.closure`	Closure decorator for delayed processing.
`florin.compose`	Deferred function composition with functools.
`florin.conncomp`	Convenience functions for image connected components operations.
`florin.io`	I/O functions for loading and saving data in a variety of formats.
`florin.morphology`	Convenience functions for image morphological operations.
`florin.ndnt`	N-Dimensional Neighborhood Thresholding for any-dimensional data.
`florin.pipelines`	Deferred execution pipelines with different computational models.
`florin.reconstruction`	Reconstruct connected components as an array of pixel-wise class labels.
`florin.thresholding`	Convenience functions for image thresholding operations.
`florin.tiling`	Utilities for tiling images and volumes.

2.3.1 florin

The FLoRIN pipeline for large-scale learning-free computer vision.

Classes

Balsam Distributed computation using the Balsam job submission database.

MPI Multicore/multi-node parallel computation with MPI.

Multiprocessing Multiprocessing using the standard fork/join model.

Multithreading Multithreading using the Python multithreading library.

Serial Single-core serial deferred computation.

WorkQueue Distributed computing using Work Queue to manage tasks.

Functions

bounds_classifier Classify connected components based on boundary conditions.

classify Classify connected components into multiple classes.

florinate Prepare a function for use in the FLoRIN pipeline.

join Join one or more tiles into a single array.

load Load image data into FLoRIN.

reconstruct Create a label array from connected component classification labels.

save Save image data from FLoRIN.

tile Split a single array into sub-arrays.

2.3.2 florin.classification

Utilities for classifying connected components.

Classes

FlorinClassifier Classification unit for weak object grouping.

Functions

classify Classify a segmented object.

Functions

classify(obj, *classes)	Multiclass classificaiton based on human-tuned boundaries.

Classes

FlorinClassifier(label, **kwargs)	Classify connected components based on boundary conditions.

class `florin.classification.`**`FlorinClassifier`**(*label*, ***kwargs*)

Classify connected components based on boundary conditions.

Parameters

- **label** – The class label identifying this class. Can be any arbitrary label.
- **boundaries** – Pairs of values (2-tuples) passed as keyword arguments defining the boundaries to classify along. For example, passing `area=(5, 10)` tells this class that the objects it contains have an area/volume of 5 <= obj.area <= 10.

Methods

classify(self, obj)	Determine if an object is in this class.

`classify`(*self*, *obj*)

Determine if an object is in this class.

Parameters **obj** (*skimage.measure._regionprops.RegionProperties*) – The object to classify.

Returns True if the object is within all defined boundaries else False. If no boundaries were provided, return True (e.g., the default class).

Return type bool

`florin.classification.`**`classify`**(*obj*, **classes*)

Multiclass classificaiton based on human-tuned boundaries.

Parameters

- **obj** (*skimage.measure._regionprops.RegionProperties*) – The object to classify.
- **classes** (*florin.classify.FlorinClassifiers*) – The classes to select from.

Returns Updates `obj` with a class label (`obj.class_label`) and passes it on for further processing.

Return type obj

Notes

In a typical FLoRIN pipeline, florin.reconstruct() will be called immediately after florin.classify().

2.3.3 florin.closure

Closure decorator for delayed processing.

Functions

florinate Decorator to wrap arbitrary functions and enable delayed evaluation.

Functions

`florinate`(func)	Decorator to wrap arbitrary functions and enable delayed evaluation.

`florin.closure.`**`florinate`**(*func*)

Decorator to wrap arbitrary functions and enable delayed evaluation.

Parameters **func** (*callable*) – The function/Python callable to wrap.

Returns **wrapper** – The wrapped `func` which stores any arguments passed to `func` and may be called on new data at a future time.

Return type callable

Notes

`florinate` is essentially a rebranding of functools.partial to allow passing the deferred arguments at the front of the call instead of the tail. This conforms with the signatures of many computer vision API functions, which tend to accept image data as the first argument.

Examples

`florinate` may be appplied as a decorator to standard function definitions to then make subsequent calls return the deferred function.

```
>>> @florinate
... def add(x, y):
...     return x + y
>>> plus_one = add(1)
```

(continues on next page)

(continued from previous page)

```
>>> plus_one(5)
6
```

Functions may also be florinated on the fly by using it as a standard function:

```
>>> def concat(str1, str2):
...     return ' '.join([str1, str2])
>>> worlder = florinate(concat)('World')
>>> worlder('Hello')
'Hello World'
```

2.3.4 florin.compose

Deferred function composition with functools.

Functions

compose Compose a chain of functions on an initial input.

Functions

compose(*functions)	Compose a chain of functions on an initial input.

`florin.compose.`**`compose`**`(`**functions*`)`

Compose a chain of functions on an initial input.

Applies a sequence of functions in order to some initial data, performing a reduce over the entire function chain.

Parameters functions (*list of callable*) – The functions to execute. List contents may be any callable, including functools.partial objects to enable parameterizing deferred functions.

Returns

Return type A partial function to be applied to data at a later time.

2.3.5 florin.conncomp

Convenience functions for image connected components operations.

Functions

label Integer labeling for binary connected components.

regionprops Compute various properties of labeled connected components.

Functions

label(image, *args, **kwargs)	Wrapper that casts arrays to integers before labeling

Continued on next page

Table 7 – continued from previous page

regionprops(image, **kwargs)	Compute the properties of connected components.

`florin.conncomp.`**`label`**(*image*, **args*, ***kwargs*)

Wrapper that casts arrays to integers before labeling

`florin.conncomp.`**`regionprops`**(*image*, ***kwargs*)

Compute the properties of connected components.

Parameters

- **image** (*array_like*) – The labeled image to process for connected components.
- **intensity_image** (*array_like*) – The original image from which `image` was computed. Passing this enables computing summary statistics about the image pixel intensities.

Notes

This function wraps skimage.measure.regionprops to allow for additional bookkeeping and feature computation.

2.3.6 florin.io

I/O functions for loading and saving data in a variety of formats.

Functions

load Load image(s) from a file.

load_hdf5 Load data from an HDF5 file.

load_image Load an image file.

load_images Load a directory of image files.

load_npy Load data from a numpy array file.

load_tiff Load a TIFF stack.

save Save image(s) in a variety of formats.

save_hdf5 Save an image to HDF5 format.

save_image Save an image.

save_images Save a sequence of images.

save_npy Save an image to a numpy array file.

save_tiff Save an image to TIFF format.

Functions

load(path, **kwargs)	Load images from a file.
load_hdf5(path[, key])	Load data from an HDF5 file.
load_image(path)	Load an image file.
load_images(path[, ext])	Load a directory of image files.

Continued on next page

Table 8 – continued from previous page

`load_npy`(path)	Load data from a numpy array file.
`load_tiff`(path)	Load a TIFF stack.
`save`(img, path, **kwargs)	Save image(s) in a variety of formats.
`save_hdf5`(img, path[, key])	Save an image to HDF5 format.
`save_image`(img, path)	Save an image.
`save_images`(img, path[, ext])	Save a sequence of images.
`save_npy`(img, path)	Save an image to a numpy array file.
`save_tiff`(img, path)	Save an image to TIFF format.

`florin.io.`**`load`**(*path*, ***kwargs*)

Load images from a file.

Generic loader function that uses the file extension to determine how to load the data.

Parameters **path** (*str*) – Path to the image file(s) to load.

Other Parameters **key** – Key to load data from when working with key/value stores (e.g. HDF5, npz, etc.)

Returns **data**

Return type numpy.ndarray

`florin.io.`**`load_hdf5`**(*path*, *key='stack'*)

Load data from an HDF5 file.

Parameters

- **path** (*str*) – Path to the HDF5 file to load.
- **key** – Key to load data from.

Returns **data**

Return type h5py.Dataset

`florin.io.`**`load_image`**(*path*)

Load an image file.

Parameters **path** (*str*) – Path to the image file to load.

Returns **data**

Return type numpy.ndarray

`florin.io.`**`load_images`**(*path*, *ext='png'*)

Load a directory of image files.

Parameters

- **path** (*str*) – Path to the image file(s) to load.
- **ext** (*str*) – The file extension to match. Only files with this extension will be loaded. Default: 'png'

Returns **data**

Return type numpy.ndarray

`florin.io.`**`load_npy`**(*path*)

Load data from a numpy array file.

Parameters **path** (*str*) – Path to the array file to load.

Returns data

Return type numpy.ndarray

`florin.io.`**`load_tiff`**(*path*)

Load a TIFF stack.

Parameters **path** (*str*) – Path to the TIFF stack to load.

Returns data

Return type numpy.ndarray

`florin.io.`**`save`**(*img*, *path*, ***kwargs*)

Save image(s) in a variety of formats.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to. This path determines which format the data will be saved as.

Returns The unaltered image/volume.

Return type img

Other Parameters

- **See ‘‘save_hdf5‘‘, ‘‘save_image‘‘, ‘‘save_images‘‘, ‘‘save_npy‘‘, and**
- **‘‘save_tiff‘‘ for filetype-specific arguments.**

Notes

The filetype passed as `path` will determine the format of the saved file. If no extension is found, 3D arrays will automatically be saved as numbered PNG files in a directory created at `path` and 2D arrays will be saved to `path` directly as a PNG.

`florin.io.`**`save_hdf5`**(*img*, *path*, *key='stack'*)

Save an image to HDF5 format.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to.

`florin.io.`**`save_image`**(*img*, *path*)

Save an image.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to.

`florin.io.`**`save_images`**(*img*, *path*, *ext='png'*)

Save a sequence of images.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to.
- **ext** (*str*) – The file extension to save each image with.

`florin.io.`**`save_npy`**`(`*img*, *path*`)`
Save an image to a numpy array file.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to.

`florin.io.`**`save_tiff`**`(`*img*, *path*`)`
Save an image to TIFF format.

Parameters

- **img** (*array_like*) – The image/volume to save.
- **path** (*str*) – The filepath to save the data to.

2.3.7 florin.morphology

Convenience functions for image morphological operations.

Functions

closing	Perform a grayscale morphological closing on an image.
dilation	Perform a grayscale morphological dilation on an image.
erosion	Perform a grayscale morphological erosion on an image.
opening	Perform a grayscale morphological opening on an image.
binary_closing	Perform a binary morphological closing on an image.
binary_dilation	Perform a binary morphological dilation on an image.
binary_erosion	Perform a binary morphological erosion on an image.
binary_opening	Perform a binary morphological opening on an image.
remove_small_holes	Fill in contiguous holes smaller than the specified size.
remove_small_objects	Remove contiguous objects smaller than the specified size.

2.3.8 florin.ndnt

N-Dimensional Neighborhood Thresholding for any-dimensional data.

Functions

ndnt	Binarize data with N-Dimensional Neighborhood Thresholding.
integral_image	Compute the integral image of a n image or volume.
integral_image_sum	Compute the neighborhood sum of an integral image.

Classes

InvalidThresholdError

Functions

integral_image(img[, inplace])	Compute the integral image of an image or image volume.
integral_image_sum(int_img[, shape, . . .])	Compute pixel neighborhood statistics.
ndnt(img[, shape, threshold, inplace])	Compute an n-dimensional Bradley thresholding of an image or volume.

Exceptions

InvalidThresholdError(t)	Raised when the NDNT threshold value is out of domain.

exception `florin.ndnt.`**InvalidThresholdError**(*t*)

Raised when the NDNT threshold value is out of domain.

with_traceback()

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

`florin.ndnt.`**integral_image**(*img*, *inplace=False*)

Compute the integral image of an image or image volume.

Parameters

- **img** (*array-like*) – The original 2D image or 3D volume.
- **inplace** (*bool, optional*) – If True, compute the integral image in the same array as the original image.

Returns **int_img** – The integral image of the original image or volume.

Return type array-like

Notes

This function extends the integral image to *n* dimensions as described in [1].

References

`florin.ndnt.`**integral_image_sum**(*int_img*, *shape=None*, *return_counts=True*)

Compute pixel neighborhood statistics.

Parameters

- **int_image** (*array_like*) – The integral image.
- **shape** (*tuple of int*) – The shape of the neighborhood around each pixel.
- **return_counts** (*bool*) – If True, in addition to neighborhood pixel sums, return the number of pixels used to compute each sum.

Returns

- **sums** (*array_like*) – An array where each entry is the sum of pixel values in a neighborhood. The same shape as `int_img`.

- **counts** (*array_like*) – An array where each entry is the number of pixels used to compute each entry in `sums`. The same shape as `int_img`.

`florin.ndnt.`**`ndnt`**(*img*, *shape=None*, *threshold=0.25*, *inplace=False*)

Compute an n-dimensional Bradley thresholding of an image or volume.

The Bradley thresholding, also called Local Adaptive Thresholding, uses the integral image of an image or volume to threshold an image based on local mean greyscale intensities. The underlying assumption is that while the mean intensity may shift, the distribution of intensities will remain roughly constant across an entire image or volume.

Parameters

- **img** (*array-like*) – The image to threshold.
- **shape** (*array-like, optional*) – The dimensions of the local neighborhood around each pixel/voxel.
- **threshold** (*float*) – The threshold value as the percentage of greyscale value to keep. Must be in [0, 1].

Notes

The original Bradley thresholding was introduced in [1] as a means for quickly thresholding images or video. Shahbazi *et al.* [2] extended this method to operate on data of arbitrary dimensionality using the method described by Tapia [3].

References

2.3.9 florin.pipelines

Deferred execution pipelines with different computational models.

Classes

BalsamPipeline Distributed computation using the Balsam job submission database.

MPIPipeline Multicore/multi-node parallel computation with MPI.

MultiprocessingPipeline Multiprocessing using the standard fork/join model.

MultithreadingPipeline Multithreading using the Python multithreading library.

SerialPipeline Single-core serial deferred computation.

WorkQueuePipeline Distributed computing using Work Queue to manage tasks.

class `florin.pipelines.`**`BalsamPipeline`**(**operations*)

Methods

`__call__`(self, data)	Call self as a function.
`add`(self, func)	Append a callable to this pipeline.
`dump`(self, path)	Save this pipeline to file.
`dumps`(self)	Serialize this pipeline as a string.

Continued on next page

Table 11 – continued from previous page

run(self, data)	Run data through the pipeline.

add (*self*, *func*)
Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

dump (*self*, *path*)
Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

dumps (*self*)
Serialize this pipeline as a string.

run (*self*, *data*)
Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

class `florin.pipelines.`**MPIPipeline** (**operations*)
MPI-based multiprocessing pipeline.

Parameters **operations** (*callables*) – Sequence of operations to run in the pipeline.

Notes

MPI is configured by wrapping Python in an `mpiexec` or `mpirun` call at runtime.

Methods

`__call__`(self, data)	Call self as a function.
add(self, func)	Append a callable to this pipeline.
dump(self, path)	Save this pipeline to file.
dumps(self)	Serialize this pipeline as a string.
run(self, data)	Run data through the pipeline.

add (*self*, *func*)
Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

dump (*self*, *path*)
Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

dumps (*self*)
Serialize this pipeline as a string.

run (*self*, *data*)
Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

class `florin.pipelines.`**`MultiprocessingPipeline`**(**operations*, *processes=None*)

Pipeline for multi-core parallel processing on a single machine.

Parameters

- **operations** (*callables*) – The operations/functions/callable classes to run in this pipeline.
- **processes** (*int, optional*) – The number of processes to use. Setting None will attempt to use as many as can be supported.

Methods

`__call__`(self, data)	Call self as a function.
`add`(self, func)	Append a callable to this pipeline.
`dump`(self, path)	Save this pipeline to file.
`dumps`(self)	Serialize this pipeline as a string.
`run`(self, data)	Run data through the pipeline.

`add`(*self*, *func*)

Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

`dump`(*self*, *path*)

Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

`dumps`(*self*)

Serialize this pipeline as a string.

`run`(*self*, *data*)

Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

class `florin.pipelines.`**`MultithreadingPipeline`**(**operations*, *threads=None*)

Pipeline for multithreaded parallel processing on a single machine.

Parameters

- **operations** (*callables*) – Sequence of operations to run in the pipeline.
- **threads** (*int, optional*) – The number of threads to use. Setting None will attempt to use as many as can be supported.

Methods

`__call__`(self, data)	Call self as a function.
`add`(self, func)	Append a callable to this pipeline.

Continued on next page

Table 14 – continued from previous page

dump(self, path)	Save this pipeline to file.
dumps(self)	Serialize this pipeline as a string.
run(self, data)	Run data through the pipeline.

add (*self*, *func*)
Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

dump (*self*, *path*)
Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

dumps (*self*)
Serialize this pipeline as a string.

run (*self*, *data*)
Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

class `florin.pipelines.`**SerialPipeline** (**operations*)
Pipeline for single-core serial computation.

Parameters **operations** (*callables*) – The operations/functions/callable classes to run in this pipeline.

Methods

`__call__`(self, data)	Call self as a function.
add(self, func)	Append a callable to this pipeline.
dump(self, path)	Save this pipeline to file.
dumps(self)	Serialize this pipeline as a string.
run(self, data)	Run data through the pipeline.

add (*self*, *func*)
Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

dump (*self*, *path*)
Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

dumps (*self*)
Serialize this pipeline as a string.

run (*self*, *data*)
Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

`class florin.pipelines.`**`WorkQueuePipeline`**`(`**operations*`)`

Methods

`__call__`(self, data)	Call self as a function.
`add`(self, func)	Append a callable to this pipeline.
`dump`(self, path)	Save this pipeline to file.
`dumps`(self)	Serialize this pipeline as a string.
`run`(self, data)	Run data through the pipeline.

`add``(`*self, func*`)`
Append a callable to this pipeline.

Parameters **func** (*callable*) – New function to add.

`dump``(`*self, path*`)`
Save this pipeline to file.

Parameters **path** (*str*) – Path to the file to write this pipeline to.

`dumps``(`*self*`)`
Serialize this pipeline as a string.

`run``(`*self, data*`)`
Run data through the pipeline.

Parameters **data** – Input to the first function in the pipeline.

Returns The result of applying the pipeline to `data`.

Return type result

2.3.10 florin.reconstruction

Reconstruct connected components as an array of pixel-wise class labels.

Functions

reconstruct Create a labeled image or volume from classified connected components.

Functions

`reconstruct`(objs)	Create a labeled image or volume from classified connected components.

`florin.reconstruction.`**`reconstruct`**`(`*objs*`)`
Create a labeled image or volume from classified connected components.

Parameters **objs** (*list of obj : skimage.measure._regionprops.RegionProperties*) – Classified objects to be labeled.

2.3.11 florin.thresholding

Convenience functions for image thresholding operations.

Functions

ndnt Binarize data with N-Dimensional Neighborhood Thresholding.

2.3.12 florin.tiling

Utilities for tiling images and volumes.

Functions

tile_generator Subdivide an array into equally-sized tiles.

join_tiles Join a sequence of tiles into a single array.

Functions

join(tiles)	Join a set of tiles into a single array.
join_tiles(tiles)	Join a set of tiles into a single array.
tile(img[, shape, stride, offset, tile_store])	Tile data into n-dimensional subdivisions.
tile_generator(img[, shape, stride, offset, . . .])	Tile data into n-dimensional subdivisions.

Exceptions

DimensionMismatchError
InvalidTileShapeError
InvalidTileStepError
ShapeStepMismatchError

exception `florin.tiling.`**DimensionMismatchError**

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

exception `florin.tiling.`**InvalidTileShapeError**

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

exception `florin.tiling.`**InvalidTileStepError**

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

exception `florin.tiling.`**ShapeStepMismatchError**

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

`florin.tiling.`**join**(*tiles*)
Join a set of tiles into a single array.

Parameters

- **tiles** (*collection of FlorinArray*) – The collection of tiles to join.
- **shape** (*tuple of int*) – The shape of the joined array.

Returns **joined** – The array created by joining the tiles and inserting them into the correct positions.

Return type array_like

`florin.tiling.`**join_tiles**(*tiles*)
Join a set of tiles into a single array.

Parameters

- **tiles** (*collection of FlorinArray*) – The collection of tiles to join.
- **shape** (*tuple of int*) – The shape of the joined array.

Returns **joined** – The array created by joining the tiles and inserting them into the correct positions.

Return type array_like

`florin.tiling.`**tile**(*img*, *shape=None*, *stride=None*, *offset=None*, *tile_store=None*)
Tile data into n-dimensional subdivisions.

Parameters

- **img** (*array_like*) – The data to subdivide.
- **shape** (*tuple of int*) – The shape of the subdivisions.
- **stride** (*tuple of int*) – The stride between subdivisions.

Yields

- **tile** (*florin.FlorinVolume*) – A subdivision of `img`. Subdivisions are yielded in sequence from the start of `img`.
- **metadata** (*dictionary*) – Key/value store of metadata, e.g. for joining tiles.

Notes

Everything up to the for loop will be run exactly once when the first tile is requested.

`florin.tiling.`**tile_generator**(*img*, *shape=None*, *stride=None*, *offset=None*, *tile_store=None*)
Tile data into n-dimensional subdivisions.

Parameters

- **img** (*array_like*) – The data to subdivide.
- **shape** (*tuple of int*) – The shape of the subdivisions.
- **stride** (*tuple of int*) – The stride between subdivisions.

Yields

- **tile** (*florin.FlorinVolume*) – A subdivision of `img`. Subdivisions are yielded in sequence from the start of `img`.

- **metadata** (*dictionary*) – Key/value store of metadata, e.g. for joining tiles.

Notes

Everything up to the for loop will be run exactly once when the first tile is requested.

Bibliography

[1] Tapia, E., 2011. A note on the computation of high-dimensional integral images. Pattern Recognition Letters, 32(2), pp.197-201.

[1] Bradley, D. and Roth, G., 2007. Adaptive thresholding using the integral image. Journal of Graphics Tools, 12(2), pp.13-21.

[2] Shahbazi, E., Kinnison, J., et al.

[3] Tapia, E., 2011. A note on the computation of high-dimensional integral images. Pattern Recognition Letters, 32(2), pp.197-201.

Python Module Index

Index