pygorpho Documentation

Release 1.0.0

Patrick M. Jensen

Dec 20, 2020

Contents

Welcome to the documentation for pygorpho. This is a library for fast 3D mathematical morphology using CUDA.

CHAPTER 1

Features

- Dilation and erosion for grayscale 3D images.
- Support for flat or grayscale structuring elements.
- A van Herk/Gil-Werman implementation for fast dilation/erosion with flat line segments in 3D.
- Automatic block processing for 3D images which can't fit in GPU memory.

1.1 Installation

This page contains instructions on how to install pygorpho. To use the library you must have an NVIDIA GPU and install CUDA Toolkit 9.2 or later.

1.1.1 Installing with pip

Install with pip:

```
pip install pygorpho
```

1.1.2 Installing from source

First, you need a compatible C++ compiler, which supports C++14. Then, following these instructions should allow you to build and install the package:

1. Clone the repo: `git clone https://github.com/patmjen/pygorpho.git`
2. Change directory: `cd pygorpho`
3. Install the required Python packages: `pip install numpy scikit-build cmake ninja`
4. Build and install: `python setup.py install`

That should be it! To test, run `python`, and try to `import pygorpho as pg`.

1.2 API Documentation

Fast 3D mathematical morphology using CUDA.

Modules:

1.2.1 pygorpho.flat

Mathematical morphology with flat (binary) structuring elements.

pygorpho.flat.**morph**(*vol, strel, op, block_size=[256, 256, 256]*)

Morphological operation with flat structuring element.

Parameters

- **vol** – Volume to apply operation to. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **op** – Operation to perform. Must be either DILATE, ERODE, OPEN, CLOSE, TOPHAT, CLOSE from constants.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the operation.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[50, 50, 50] = 1
strel = np.ones((11, 11, 11))
res = pg.flat.morph(vol, strel, pg.DILATE)
```

pygorpho.flat.**dilate**(*vol, strel, block_size=[256, 256, 256]*)

Dilation with flat structuring element.

Parameters

- **vol** – Volume to dilate/erode. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of dilation.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[50, 50, 50] = 1
strel = np.ones((11, 11, 11))
res = pg.flat.dilate(vol, strel)
```

pygorpho.flat.**erode**(*vol, strel, block_size=[256, 256, 256]*)

Erosion with flat structuring element.

Parameters

- **vol** – Volume to dilate/erode. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of erosion.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple erosion with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[50, 50, 50] = 0
strel = np.ones((11, 11, 11))
res = pg.flat.erode(vol, strel)
```

pygorpho.flat.**open**(*vol, strel, block_size=[256, 256, 256]*)

Opening with flat structuring element.

Parameters

- **vol** – Volume to open. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of opening.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple opening with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
```

(continues on next page)

(continued from previous page)

```
vol[10:15,10:15,48:53] = 1  # Small box
vol[60:80,60:80,40:60] = 1  # Big box
strel = np.ones((11, 11, 11))
res = pg.flat.open(vol, strel)
```

pygorpho.flat.**close**(*vol, strel, block_size=[256, 256, 256]*)

Closing with flat structuring element.

Parameters

- **vol** – Volume to close. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of closing.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple closing with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
strel = np.ones((11, 11, 11))
res = pg.flat.close(vol, strel)
```

pygorpho.flat.**tophat**(*vol, strel, block_size=[256, 256, 256]*)

Top-hat transform with flat structuring element.

Also known as a white top hat transform. It is given by `tophat(x) = x - open(x)`.

Parameters

- **vol** – Volume to top-hat transform. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the top-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple top-hat with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[10:15,10:15,48:53] = 1  # Small box
```

(continues on next page)

(continued from previous page)

```
vol[60:80,60:80,40:60] = 1  # Big box
strel = np.ones((11, 11, 11))
res = pg.flat.tophat(vol, strel)
```

pygorpho.flat.**bothat**(*vol, strel, block_size=[256, 256, 256]*)

Bot-hat transform with flat structuring element.

Also known as a black top-hat transform. It is given by `bothat(x) = close(x) - x`.

Parameters

- **vol** – Volume to bot-hat transform. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the bot-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple bot-hat with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
strel = np.ones((11, 11, 11))
res = pg.flat.bothat(vol, strel)
```

pygorpho.flat.**linear_morph**(*vol, line_steps, line_lens, op, block_size=[256, 256, 512]*)

Morphological operation with flat line segment structuring elements.

Performs a morphological operation volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operation is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to apply operation to. Must be convertible to numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **op** – Operation to perform for all line segments. Must be either `DILATE` or `ERODE` from `constants`.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the operation.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 15 x 21 box structuring element
vol = np.zeros((100,100,100))
vol[50, 50, 50] = 1
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 15, 21]
res = pg.flat.linear_morph(vol, lineSteps, lineLens, pg.DILATE)
```

References

pygorpho.flat.**linear_dilate**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Dilation with flat line segment structuring elements.

Erodes volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to dilate. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of dilation.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 15 x 21 box structuring element
vol = np.zeros((100,100,100))
vol[50, 50, 50] = 1
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 15, 21]
res = pg.flat.linear_dilate(vol, lineSteps, lineLens)
```

pygorpho.flat.**linear_erode**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Erosion with flat line segment structuring elements.

Erodes volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to erode. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of erosion.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple erosion with an 11 x 15 x 21 box structuring element
vol = np.ones((100,100,100))
vol[50, 50, 50] = 0
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 15, 21]
res = pg.flat.linear_erode(vol, lineSteps, lineLens)
```

pygorpho.flat.**linear_open**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Opening with flat line segment structuring elements.

Opens volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to open. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of opening.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple opening with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[10:15,10:15,48:53] = 1  # Small box
```

(continues on next page)

(continued from previous page)

```
vol[60:80,60:80,40:60] = 1  # Big box
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 11, 11]
res = pg.flat.linear_open(vol, lineSteps, lineLens)
```

pygorpho.flat.**linear_close**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Closing with flat line segment structuring elements.

Closes volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to close. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of closing.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple closing with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 11, 11]
res = pg.flat.linear_close(vol, lineSteps, lineLens)
```

pygorpho.flat.**linear_tophat**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Top-hat transform with flat line segment structuring elements.

Top-hat transforms volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to top-hat transform. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.

- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of top-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple top-hat with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 11, 11]
res = pg.flat.linear_tophat(vol, lineSteps, lineLens)
```

pygorpho.flat.**linear_bothat**(*vol, line_steps, line_lens, block_size=[256, 256, 512]*)

Bot-hat transform with flat line segment structuring elements.

Bot-hat transforms volume with a sequence of flat line segments. Line segments are parameterized with a (integer) step vector and a length giving the number of steps. The operations is the same for all line segments.

The operations are performed using the van Herk/Gil-Werman algorithm [H92] [GW93].

Parameters

- **vol** – Volume to bot-hat transform. Must be convertible to a numpy array of at most 3 dimensions.
- **line_steps** – Step vector or sequence of step vectors. A step vector must have integer coordinates and control the direction of the line segment.
- **line_lens** – Length or sequence of lengths. Controls the length of the line segments. A length of 0 leaves the volume unchanged.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of bot-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple bot-hat with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
lineSteps = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
lineLens = [11, 11, 11]
res = pg.flat.linear_tophat(vol, lineSteps, lineLens)
```

1.2.2 pygorpho.gen

Mathematical morphology with general (grayscale) structuring elements.

`pygorpho.gen.`**`morph`**(*vol, strel, op, block_size=[256, 256, 256]*)

Morphological operation with general structuring element.

Parameters

- **`vol`** – Volume to apply operation to. Must be convertible to numpy array of at most 3 dimensions.
- **`strel`** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **`op`** – Operation to perform. Must be either `DILATE` or `ERODE` from `constants`.
- **`block_size`** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the operation.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[50, 50, 50] = 1
strel = np.ones((11, 11, 11))
res = pg.gen.morph(vol, strel, pg.DILATE)
```

`pygorpho.gen.`**`dilate`**(*vol, strel, block_size=[256, 256, 256]*)

Dilation with general structuring element.

Parameters

- **`vol`** – Volume to dilate/erode. Must be convertible to numpy array of at most 3 dimensions.
- **`strel`** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **`block_size`** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of dilation/erosion.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple dilation with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[50, 50, 50] = 1
strel = np.ones((11, 11, 11))
res = pg.gen.dilate(vol, strel)
```

pygorpho.gen.**erode**(*vol, strel, block_size=[256, 256, 256]*)
Erosion with general structuring element.

Parameters

- **vol** – Volume to dilate/erode. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of dilation/erosion.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple erosion with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[50, 50, 50] = 0
strel = np.ones((11, 11, 11))
res = pg.gen.erode(vol, strel)
```

pygorpho.gen.**open**(*vol, strel, block_size=[256, 256, 256]*)
Opening with general structuring element.

Parameters

- **vol** – Volume to open. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of opening.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple opening with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[10:15,10:15,48:53] = 1  # Small box
vol[60:80,60:80,40:60] = 1  # Big box
strel = np.ones((11, 11, 11))
res = pg.gen.open(vol, strel)
```

pygorpho.gen.**close**(*vol, strel, block_size=[256, 256, 256]*)
Closing with general structuring element.

Parameters

- **vol** – Volume to close. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.

- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of closing.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple closing with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
strel = np.ones((11, 11, 11))
res = pg.gen.close(vol, strel)
```

pygorpho.gen.**tophat**(*vol, strel, block_size=[256, 256, 256]*)

Top-hat transform with general structuring element.

Also known as a white top-hat transform. It is given by `tophat(x) = x - open(x)`.

Parameters

- **vol** – Volume to top-hat transform. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.
- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the top-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple top-hat with an 11 x 11 x 11 box structuring element
vol = np.zeros((100, 100, 100))
vol[10:15,10:15,48:53] = 1  # Small box
vol[60:80,60:80,40:60] = 1  # Big box
strel = np.ones((11, 11, 11))
res = pg.gen.tophat(vol, strel)
```

pygorpho.gen.**bothat**(*vol, strel, block_size=[256, 256, 256]*)

Bot-hat transform with general structuring element.

Also known as a black top-hat transform. It is given by `bothat(x) = close(x) - x`.

Parameters

- **vol** – Volume to bot-hat transform. Must be convertible to numpy array of at most 3 dimensions.
- **strel** – Structuring element. Must be convertible to numpy array of at most 3 dimensions.

- **block_size** – Block size for GPU processing. Volume is sent to the GPU in blocks of this size.

Returns Volume of same size as vol with the result of the bot-hat transform.

Return type numpy.array

Example

```
import numpy as np
import pygorpho as pg
# Simple bot-hat with an 11 x 11 x 11 box structuring element
vol = np.ones((100, 100, 100))
vol[10:15,10:15,48:53] = 0  # Small box
vol[60:80,60:80,40:60] = 0  # Big box
strel = np.ones((11, 11, 11))
res = pg.gen.bothat(vol, strel)
```

1.2.3 pygorpho.strel

Structuring elements for mathematical morhology

pygorpho.strel.**flat_ball_approx**(*radius*, *type=1*)

Returns approximation to flat ball using line segments.

The approximation is constructed according to [J19] and allows for constant time morphology operations.

Parameters

- **r** – Integer radius of flat ball.
- **type** – Whether to constrain the zonohedral approximation inside or outside the sphere. Must either INSIDE, BEST, or OUTSIDE from constants.

Returns Tuple with step vectors and line lengths which parameterizes the line segments.

Return type (numpy.array, numpy.array)

Example

```
import numpy as np
import pygorpho as pg
# Dilation with ball approximation of radius 25
vol = np.zeros((100,100,100))
vol[50, 50, 50] = 1
lineSteps, lineLens = pg.strel.flat_ball_approx(25)
res = pg.flat.linear_dilate(vol, lineSteps, lineLens)
```

References

1.2.4 pygorpho.constants

One stop shop for constants.

pygorpho.constants.**DILATE = 0**

Dilation

pygorpho.constants.**ERODE = 1**
Erosion

pygorpho.constants.**OPEN = 2**
Opening

pygorpho.constants.**CLOSE = 3**
Closing

pygorpho.constants.**TOPHAT = 4**
Top hat

pygorpho.constants.**BOTHAT = 5**
Bot hat

pygorpho.constants.**INSIDE = 0**
Inside

pygorpho.constants.**BEST = 1**
Best

pygorpho.constants.**OUTSIDE = 2**
Outside

1.2.5 pygorpho.cuda

Query functions to get information about available CUDA devices

pygorpho.cuda.**get_device_count**()
Returns the number of available CUDA devices.

Returns Number of available CUDA devices.

Return type int

pygorpho.cuda.**get_device_name**(*device*)
Returns the name of queried CUDA device.

Parameters **device** – ID of CUDA device.

Returns Name of queried CUDA device.

Return type str

1.3 License

MIT License

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 2

Resources

- Free software: MIT license
- Source code: https://github.com/patmjen/pygorpho
- PyPI: https://pypi.org/project/pygorpho/

Bibliography

[H92] M. Van Herk, “A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels,” Pattern Recognition Letters 13. (pp. 517-521). 1992.

[GW93] J. Gil and M Werman, “Computing 2-D min, median, and max filters,” IEEE Transactions on Pattern Analysis and Machine Intelligence 24. (pp. 504-507). 1993.

[J19] P. M. Jensen et al., “Zonohedral Approximation of Spherical Structuring Element for Volumetric Morphology,” Scandinavian Conference on Image Analysis (pp. 128-139). Springer. 2019.

Python Module Index

p

Index

P