

---

# **spatialist Documentation**

**John Truckenbrodt, Felix Cremer, Ismail Baris**

**Jan 25, 2022**



---

# Contents

---

<b>1</b>	<b>Installation</b>	<b>1</b>
1.1	Installation of dependencies . . . . .	1
1.2	Installation of spatialist . . . . .	2
<b>2</b>	<b>API Documentation</b>	<b>3</b>
2.1	Raster Class . . . . .	3
2.2	Raster Tools . . . . .	9
2.3	Vector Class . . . . .	12
2.4	Vector Tools . . . . .	16
2.5	General Spatial Tools . . . . .	19
2.6	Database Tools . . . . .	22
2.7	Ancillary Functions . . . . .	22
2.8	ENVI HDR file manipulation . . . . .	26
<b>3</b>	<b>Some general examples</b>	<b>27</b>
3.1	in-memory vector object rasterization . . . . .	27
<b>4</b>	<b>Changelog</b>	<b>29</b>
4.1	0.4   2019-12-05 . . . . .	29
4.2	0.5   2020-04-21 . . . . .	29
4.3	0.6   2020-07-17 . . . . .	30
4.4	0.7   2021-06-30 . . . . .	30
4.5	0.8   2021-09-07 . . . . .	30
4.6	0.8.1   2021-10-05 . . . . .	31
4.7	0.9.0   2022-01-25 . . . . .	31
<b>5</b>	<b>Indices and tables</b>	<b>33</b>
	<b>Python Module Index</b>	<b>35</b>
	<b>Index</b>	<b>37</b>



The most convenient way to install spatialist is by using conda:

```
conda install --channel conda-forge spatialist
```

See below for more detailed Linux installation instructions outside of the Anaconda framework.

## 1.1 Installation of dependencies

### 1.1.1 GDAL

spatialist requires GDAL version  $\geq 2.1$  built with GEOS and PROJ4 as dependency as well as the GDAL Python binding. Alternatively, one can use [pygdal](#), a virtualenv and setuptools friendly version of standard GDAL python bindings.

#### Ubuntu

Starting with release Yakkety (16.10), Ubuntu comes with GDAL  $>2.1$ . See [here](#). You can install it like this:

```
sudo apt-get install python-gdal python3-gdal gdal-bin
```

For older Ubuntu releases you can add the [ubuntugis](#) repository to apt prior to installation to install version  $>2.1$ :

```
sudo add-apt-repository ppa:ubuntugis/ppa
sudo apt-get update
```

This way the required dependencies (GEOS and PROJ4 in particular) are also installed. You can check the version by typing:

```
gdalinfo --version
```

#### Debian

Starting with Debian 9 (Stretch) GDAL is available in version  $>2.1$  in the official repository.

### Building from source

Alternatively, you can build GDAL and the dependencies from source. The script *spatialist/install/install\_deps.sh* gives specific instructions on how to do it. It is not yet intended to run this script via shell, but rather to follow the instructions step by step.

### 1.1.2 SQLite + Spatialite

#### Windows

While *sqlite3* and its Python binding are usually already installed, the *spatialite* extension needs to be added. Two packages exist, *libspatialite* and *mod\_spatialite*. Both can be used by *spatialist*. It is strongly recommended to use Ubuntu >= 16.04 (Xenial) or Debian >=9 (Stretch), which offer the package *libsqlite3-mod-spatialite*. This package is specifically intended to only serve as an extension to *sqlite3* and can be installed like this:

```
sudo apt-get install libsqlite3-mod-spatialite
```

After installation, the following can be run in Python to test the needed functionality:

```
import sqlite3
# setup an in-memory database
con = sqlite3.connect(':memory:')
# enable loading extensions and load spatialite
con.enable_load_extension(True)
try:
    con.load_extension('mod_spatialite.so')
except sqlite3.OperationalError:
    con.load_extension('libspatialite.so')
```

In case loading extensions is not permitted you might need to install the package *pysqlite2*. See the script *spatialist/install/install\_deps.sh* for instructions. There you can also find instructions on how to install *spatialite* from source. To test *pysqlite2* you can import it as follows and then run the test above:

```
from pysqlite2 import dbapi2 as sqlite3
```

Installing this package is likely to cause problems with the *sqlite3* library installed on the system. Thus, it is safer to build a static *sqlite3* library for it (see installation script).

## 1.2 Installation of spatialist

For the installation we need the Python tool *pip*.

```
sudo apt-get install python-pip
```

Once everything is set up, *spatialist* is ready to be installed. You can install stable releases like this:

```
python -m pip install spatialist
```

or the latest GitHub master branch using *git* like this:

```
sudo apt-get install git
sudo python -m pip install git+https://github.com/johntruckenbrodt/spatialist.git
```

## 2.1 Raster Class

**class** `spatialist.raster.Raster` (*filename*, *list\_separate=True*, *timestamps=None*)

This is intended as a raster meta information handler with options for reading and writing raster data in a convenient manner by simplifying the numerous options provided by the [GDAL](#) python binding. Several methods are provided along with this class to directly modify the raster object in memory or directly write a newly created file to disk (without modifying the raster object itself). Upon initializing a Raster object, only metadata is loaded. The actual data can be, for example, loaded to memory by calling methods `matrix()` or `load()`.

### Parameters

- **filename** (str or list or `gdal.Dataset`) – the raster file(s)/object to read
- **list\_separate** (*bool*) – treat a list of files as separate layers or otherwise as a single layer? The former is intended for single layers of a stack, the latter for tiles of a mosaic.
- **timestamps** (*list[str] or function or None*) – the time information for each layer or a function converting band names to a `datetime.datetime` object

**\_\_getitem\_\_** (*index*)

subset the object by slices or vector geometry. If slices are provided, one slice for each raster dimension needs to be defined. I.e., if the raster object contains several image bands, three slices are necessary. Integer slices are treated as pixel coordinates and float slices as map coordinates. If a [Vector](#) geometry is defined, it is internally projected to the raster CRS if necessary, its extent derived, and the extent converted to raster pixel slices, which are then used for subsetting.

**Parameters** **index** (tuple of *slice* or *Vector*) – the subsetting indices to be used

**Returns** a new raster object referenced through an in-memory GDAL VRT file

**Return type** *Raster*

## Examples

```
>>> filename = 'test'
>>> with Raster(filename) as ras:
>>>     print(ras)
class      : spatialist Raster object
dimensions : 2908, 2069, 115 (rows, cols, bands)
resolution : 20.0, -20.0 (x, y)
extent      : 713315.198, 754695.198, 4068985.595, 4127145.595 (xmin, xmax,
↳ymin, ymax)
coord. ref.: +proj=utm +zone=29 +datum=WGS84 +units=m +no_defs
data source: test
>>>
>>>
>>> xmin = 0
>>> xmax = 100
>>> ymin = 4068985.595
>>> ymax = 4088985.595
>>> with Raster(filename)[ymin:ymax, xmin:xmax, :] as ras:
>>>     print(ras)
class      : spatialist Raster object
dimensions : 1000, 100, 115 (rows, cols, bands)
resolution : 20.0, -20.0 (x, y)
extent      : 713315.198, 715315.198, 4068985.595, 4088985.595 (xmin, xmax,
↳ymin, ymax)
coord. ref.: +proj=utm +zone=29 +datum=WGS84 +units=m +no_defs
data source: /tmp/tmpk5weyhq.vrt
>>>
>>>
>>> ext = {'xmin': 713315.198, 'xmax': 715315.198, 'ymin': ymin, 'ymax': ymax}
>>>
>>> with bbox(ext, crs=32629) as vec:
>>>     with Raster(filename)[vec] as ras:
>>>         print(ras)
class      : spatialist Raster object
dimensions : 1000, 100, 115 (rows, cols, bands)
resolution : 20.0, -20.0 (x, y)
extent      : 713315.198, 715315.198, 4068985.595, 4088985.595 (xmin, xmax,
↳ymin, ymax)
coord. ref.: +proj=utm +zone=29 +datum=WGS84 +units=m +no_defs
data source: /tmp/tmps4rc9o09.vrt
```

### **allstats** (*approximate=False*)

Compute some basic raster statistics

**Parameters** **approximate** (*bool*) – approximate statistics from overviews or a subset of all tiles?

**Returns** a list with a dictionary of statistics for each band. Keys: *min*, *max*, *mean*, *sdev*. See [gdal.Band.ComputeStatistics](#).

**Return type** list of dicts

### **array** ()

read all raster bands into a numpy ndarray

**Returns** the array containing all raster data

**Return type** [numpy.ndarray](#)



**assign** (*array*, *band*)

assign an array to an existing Raster object

**Parameters**

- **array** (*numpy.ndarray*) – the array to be assigned to the Raster object
- **band** (*int*) – the index of the band to assign to

**bandnames**

**Returns** the names of the bands

**Return type** *list*

**bands**

**Returns** the number of image bands

**Return type** *int*

**bbox** (*outname=None*, *driver='ESRI Shapefile'*, *overwrite=True*, *source='image'*)

**Parameters**

- **outname** (*str* or *None*) – the name of the file to write; If *None*, the bounding box is returned as *Vector* object
- **driver** (*str*) – The file format to write
- **overwrite** (*bool*) – overwrite an already existing file?
- **source** (*{ 'image', 'gcp' }*) – get the bounding box of either the image or the ground control points

**Returns** the bounding box vector object

**Return type** *Vector* or *None*

**close** ()

closes the GDAL raster file connection

**cols**

**Returns** the number of image columns

**Return type** *int*

**coord\_img2map** (*x=None*, *y=None*)

convert image pixel coordinates to map coordinates in the raster CRS. Either x, y or both must be defined.

**Parameters**

- **x** (*int* or *float*) – the x coordinate
- **y** (*int* or *float*) – the y coordinate

**Returns** the converted coordinate for either x, y or both

**Return type** *float* or *tuple*

**coord\_map2img** (*x=None*, *y=None*)

convert map coordinates in the raster CRS to image pixel coordinates. Either x, y or both must be defined.

**Parameters**

- **x** (*int* or *float*) – the x coordinate
- **y** (*int* or *float*) – the y coordinate

**Returns** the converted coordinate for either x, y or both

**Return type** `int` or `tuple`

**dim**

**Returns** (rows, columns, bands)

**Return type** `tuple`

**driver**

**Returns** a GDAL raster driver object.

**Return type** `gdal.Driver`

**dtype**

**Returns** the data type description; e.g. *Float32*

**Return type** `str`

**epsg**

**Returns** the CRS EPSG code

**Return type** `int`

**extent**

**Returns** the extent of the image

**Return type** `dict`

**extract** (*px, py, radius=1, nodata=None*)

extract weighted average of pixels intersecting with a defined radius to a point.

**Parameters**

- **px** (*int* or *float*) – the x coordinate in units of the Raster SRS
- **py** (*int* or *float*) – the y coordinate in units of the Raster SRS
- **radius** (*int* or *float*) – the radius around the point to extract pixel values from; defined as multiples of the pixel resolution
- **nodata** (*int*) – a value to ignore from the computations; If *None*, the nodata value of the Raster object is used

**Returns** the the weighted average of all pixels within the defined radius

**Return type** `int` or `float`

**files**

**Returns** a list of all absolute names of files associated with this raster data set

**Return type** list of `str`

**format**

**Returns** the name of the image format

**Return type** `str`

**geo**

General image geo information.

**Returns** a dictionary with keys *xmin, xmax, xres, rotation\_x, ymin, ymax, yres, rotation\_y*

**Return type** `dict`

**geogcs**

**Returns** an identifier of the geographic coordinate system

**Return type** `str` or `None`

**is\_valid()**

Check image integrity. Tries to compute the checksum for each raster layer and returns False if this fails. See this forum entry: [How to check if image is valid?](#).

**Returns** is the file valid?

**Return type** `bool`

**layers()**

**Returns** a list containing a `gdal.Band` object for each image band

**Return type** list of `gdal.Band`

**load()**

load all raster data to internal memory arrays. This shortens the read time of other methods like `matrix()`.

**matrix** (*band=1, mask\_nan=True*)

read a raster band (subset) into a numpy ndarray

**Parameters**

- **band** (*int*) – the band to read the matrix from; 1-based indexing
- **mask\_nan** (*bool*) – convert nodata values to `numpy.nan`? As `numpy.nan` requires at least float values, any integer array is cast to float32.

**Returns** the matrix (subset) of the selected band

**Return type** `numpy.ndarray`

**nodata**

**Returns** the raster nodata value(s)

**Return type** `float` or `list`

**proj4**

**Returns** the CRS PROJ4 description

**Return type** `str`

**proj4args**

**Returns** the PROJ4 string arguments as a dictionary

**Return type** `dict`

**projcs**

**Returns** an identifier of the projected coordinate system; If the CRS is not projected `None` is returned

**Return type** `str` or `None`

**projection**

**Returns** the CRS Well Known Text (WKT) description

**Return type** `str`

**res**

the raster resolution in x and y direction

**Returns** (xres, yres)

**Return type** `tuple`

**rescale** (*fun*)

perform raster computations with custom functions and assign them to the existing raster object in memory

**Parameters** **fun** (*function*) – the custom function to compute on the data

## Examples

```
>>> with Raster('filename') as ras:
>>>     ras.rescale(lambda x: 10 * x)
```

**rows**

**Returns** the number of image rows

**Return type** `int`

**srs**

**Returns** the spatial reference system of the data set.

**Return type** `osr.SpatialReference`

**write** (*outname*, *dtype*='default', *format*='ENVI', *nodata*='default', *overwrite*=False, *cmap*=None, *update*=False, *xoff*=0, *yoff*=0, *array*=None, *options*=None, *overviews*=None)  
write the raster object to a file.

### Parameters

- **outname** (*str*) – the file to be written
- **dtype** (*str*) – the data type of the written file; data type notations of GDAL (e.g. *Float32*) and numpy (e.g. *int8*) are supported.
- **format** (*str*) – the file format; e.g. 'GTiff'
- **nodata** (*int* or *float*) – the nodata value to write to the file
- **overwrite** (*bool*) – overwrite an already existing file? Only applies if *update* is *False*.
- **cmap** (`gdal.ColorTable`) – a color map to apply to each band. Can for example be created with function `cmap_mpl2gdal()`.
- **update** (*bool*) – open the output file for update or only for writing?
- **xoff** (*int*) – the x/column offset
- **yoff** (*int*) – the y/row offset
- **array** (`numpy.ndarray`) – write different data than that associated with the Raster object
- **options** (*list* or *None*) – as list of options for creating the output dataset; see `gdal.Driver.Create`
- **overviews** (*list* or *None*) – a list of integer overview levels to be created; see `gdal.Dataset.BuildOverviews`

## 2.2 Raster Tools

<code>apply_along_time</code>	Apply a time series computation to a 3D raster stack using multiple CPUs.
<code>png</code>	convert a raster image to png.
<code>rasterize</code>	rasterize a vector object
<code>stack</code>	function for mosaicking, resampling and stacking of multiple raster files
<code>subset_tolerance</code>	this parameter can be set to increase the pixel tolerance in percent when subsetting <i>Raster</i> objects with the extent of other spatial objects.

`spatialist.raster.apply_along_time(src, dst, func1d, nodata, format, cmap=None, maxlines=None, cores=8, *args, **kwargs)`

Apply a time series computation to a 3D raster stack using multiple CPUs. The stack is read in chunks of maxlines x columns x time steps, for which the result is computed and stored in a 2D output array. After finishing the computation for all chunks, the output array is written to the specified file.

### Notes

It is intended to directly write the computation result of each chunk to the output file respectively so that no unnecessary memory is used for storing the complete result. This however first requires some restructuring of the method `spatialist.Raster.write()`.

### Parameters

- **src** (*Raster*) – the source raster data
- **dst** (*str*) – the output file in GeoTiff format
- **func1d** (*function*) – the function to be applied over a single time series 1D array
- **nodata** (*int*) – the nodata value to write to the output file
- **format** (*str*) – the output file format, e.g. ‘GTiff’
- **cmap** (*gdal.ColorTable*) – a color table to write to the resulting file; see `spatialist.auxil.cmap_mpl2gdal()` for creation options.
- **maxlines** (*int*) – the maximum number of lines to read at once. Controls the amount of memory used.
- **cores** (*int*) – the number of parallel cores
- **args** (*any*) – Additional arguments to *func1d*.
- **kwargs** (*any*) – Additional named arguments to *func1d*.

See also:

`spatialist.ancillary.parallel_apply_along_axis()`

`spatialist.raster.png(src, dst, percent=10, scale=(2, 98), worldfile=False, nodata=None)`

convert a raster image to png. The input raster must either have one or three bands to create a grey scale or RGB image respectively.

### Parameters

- **src** (*Raster*) – the input raster image to be converted

- **dst** (*str*) – the output png file name
- **percent** (*int*) – the size of the png relative to *src*
- **scale** (*tuple* or *None*) – the percentile bounds as (min, max) for scaling the values of the input image or *None* to not apply any scaling.
- **worldfile** (*bool*) – create a world file (extension .wld)?
- **nodata** (*int* or *None*) – The no data value to write to the file. All pixels with this value will be transparent.

## Notes

Currently it is not possible to control what happens with values outside of the percentile range defined by *scale*. Therefore, if e.g. *nodata* is set to 0, all values below the lower percentile will be marked as 0 and will thus be transparent in the image. On the other hand if *nodata* is 255, all values higher than the upper percentile will be transparent. This is addressed in GDAL issue [#1825](#).

## Examples

```
>>> from spatialist.raster import Raster, png
>>> src = 'src.tif'
>>> dst = 'dst.png'
>>> with Raster(src) as ras:
>>>     png(src=ras, dst=dst, percent=10, scale=(2, 98), worldfile=True)
```

`spatialist.raster.rasterize` (*vectorobject*, *reference*, *outname=None*, *burn\_values=1*, *expressions=None*, *nodata=0*, *append=False*)

rasterize a vector object

### Parameters

- **vectorobject** (*Vector*) – the vector object to be rasterized
- **reference** (*Raster*) – a reference Raster object to retrieve geo information and extent from
- **outname** (*str* or *None*) – the name of the GeoTiff output file; if *None*, an in-memory object of type *Raster* is returned and parameter *outname* is ignored
- **burn\_values** (*int* or *list*) – the values to be written to the raster file
- **expressions** (*list*) – SQL expressions to filter the vector object by attributes
- **nodata** (*int* or *float* or *None*) – the nodata value of the target raster file
- **append** (*bool*) – if the output file already exists, update this file with new rasterized values? If *True* and the output file exists, parameters *reference* and *nodata* are ignored.

**Returns** if *outname* is *None*, a raster object pointing to an in-memory dataset else *None*

**Return type** *Raster* or *None*

## Example

```

>>> from spatialist import Vector, Raster, rasterize
>>> outname1 = 'target1.tif'
>>> outname2 = 'target2.tif'
>>> with Vector('source.shp') as vec:
>>>     with Raster('reference.tif') as ref:
>>>         burn_values = [1, 2]
>>>         expressions = ['ATTRIBUTE=1', 'ATTRIBUTE=2']
>>>         rasterize(vec, reference, outname1, burn_values, expressions)
>>>         expressions = ["ATTRIBUTE2='a'", "ATTRIBUTE2='b'"]
>>>         rasterize(vec, reference, outname2, burn_values, expressions)

```

`spatialist.raster.stack` (*srcfiles*, *dstfile*, *resampling*, *targetres*, *dstnodata*, *srcnodata*=None, *shapefile*=None, *layernames*=None, *sortfun*=None, *separate*=False, *overwrite*=False, *compress*=True, *cores*=4, *pbar*=False)  
function for mosaicking, resampling and stacking of multiple raster files

#### Parameters

- **srcfiles** (*list*) – a list of file names or a list of lists; each sub-list is treated as a task to mosaic its containing files
- **dstfile** (*str*) – the destination file or a directory (if *separate* is True)
- **resampling** (*{near, bilinear, cubic, cubicspline, lanczos, average, mode, max, min, med, Q1, Q3}*) – the resampling method; see [documentation of gdalwarp](#).
- **targetres** (*tuple or list*) – two entries for x and y spatial resolution in units of the source CRS
- **srcnodata** (*int, float or None*) – the nodata value of the source files; if left at the default (None), the nodata values are read from the files
- **dstnodata** (*int or float*) – the nodata value of the destination file(s)
- **shapefile** (*str, Vector or None*) – a shapefile for defining the spatial extent of the destination files
- **layernames** (*list*) – the names of the output layers; if *None*, the basenames of the input files are used; overrides *sortfun*
- **sortfun** (*function*) – a function for sorting the input files; not used if *layernames* is not None. This is first used for sorting the items in each sub-list of *srcfiles*; the basename of the first item in a sub-list will then be used as the name for the mosaic of this group. After mosaicking, the function is again used for sorting the names in the final output (only relevant if *separate* is False)
- **separate** (*bool*) – should the files be written to a single raster stack (ENVI format) or separate files (GTiff format)?
- **overwrite** (*bool*) – overwrite the file if it already exists?
- **compress** (*bool*) – compress the geotiff files?
- **cores** (*int*) – the number of CPU threads to use
- **pbar** (*bool*) – add a progressbar? This is currently only used if *separate*==False

Raises `RuntimeError`

## Notes

This function does not reproject any raster files. Thus, the CRS must be the same for all input raster files. This is checked prior to executing `gdalwarp`. In case a shapefile is defined, it is internally reprojected to the raster CRS prior to retrieving its extent.

## Examples

```
from pyroSAR.ancillary import groupbyTime, find_datasets, seconds
from spatialist.raster import stack

# find pyroSAR files by metadata attributes
archive_sl = '../sentinel1/GRD/processed'
scenes_sl = find_datasets(archive_sl, sensor=('S1A', 'S1B'), acquisition_mode='IW
→')

# group images by acquisition time
groups = groupbyTime(images=scenes_sl, function=seconds, time=30)

# mosaic individual groups and stack the mosaics to a single ENVI file
# only files overlapping with the shapefile are selected and resampled to its
→extent
stack(srcfiles=groups, dstfile='stack', resampling='bilinear', targetres=(20, 20),
      srcnodata=-99, dstnodata=-99, shapefile='site.shp', separate=False)
```

`spatialist.raster.subset_tolerance = 0`

this parameter can be set to increase the pixel tolerance in percent when subsetting *Raster* objects with the extent of other spatial objects.

## Examples

Coordinates are in EPSG:32632, pixel resolution of the image to be subsetted is 90 m:

(subsetting extent)

```
{'xmin': 534093.341, 'xmax': 830103.341, 'ymin': 5030609.645, 'ymax': 5250929.645}
```

```
subset_tolerance = 0
```

```
{'xmin': 534003.341, 'xmax': 830103.341, 'ymin': 5030519.645, 'ymax': 5250929.645}
```

```
subset_tolerance = 0.02
```

```
{'xmin': 534093.341, 'xmax': 830103.341, 'ymin': 5030609.645, 'ymax': 5250929.645}
```

## 2.3 Vector Class

**class** `spatialist.vector.Vector` (*filename=None, driver=None*)

This is intended as a vector meta information handler with options for reading and writing vector data in a convenient manner by simplifying the numerous options provided by the OGR python binding.

### Parameters

- **filename** (*str or None*) – the vector file to read; if filename is *None*, a new in-memory Vector object is created. In this case *driver* is overridden and set to 'Memory'. The following file extensions are auto-detected:
  - .geojson (GeoJSON)



- .gpkg (GPKG)
- .kml (KML)
- .shp (ESRI Shapefile)
- **driver** (*str*) – the vector file format; needs to be defined if the format cannot be auto-detected from the filename extension

**\_\_getitem\_\_** (*expression*)

subset the vector object by index or attribute.

**Parameters** **expression** (*int* or *str*) – the key or expression to be used for subsetting.  
See [ogr.Layer.SetAttributeFilter](#) for details on the expression syntax.

**Returns** a vector object matching the specified criteria

**Return type** *Vector*

## Examples

Assuming we have a shapefile called *testsites.shp*, which has an attribute *sitename*, we can subset individual sites and write them to new files like so:

```
>>> from spatialist import Vector
>>> filename = 'testsites.shp'
>>> with Vector(filename) ["sitename='site1'"] as site1:
>>>     site1.write('site1.shp')
```

**addfeature** (*geometry, fields=None*)

add a feature to the vector object from a geometry

### Parameters

- **geometry** (*ogr.Geometry*) – the geometry to add as a feature
- **fields** (*dict* or *None*) – the field names and values to assign to the new feature

**addfield** (*name, type, width=10*)

add a field to the vector layer

### Parameters

- **name** (*str*) – the field name
- **type** (*int*) – the OGR Field Type (OFT), e.g. *ogr.OFTString*. See [Module ogr](#).
- **width** (*int*) – the width of the new field (only for *ogr.OFTString* fields)

**addlayer** (*name, srs, geomType*)

add a layer to the vector layer

### Parameters

- **name** (*str*) – the layer name
- **srs** (*int* or *str* or *osr.SpatialReference*) – the spatial reference system. See [spatialist.auxil.crsConvert\(\)](#) for options.
- **geomType** (*int*) – an OGR well-known binary data type. See [Module ogr](#).

**addvector** (*vec*)

add a vector object to the layer of the current Vector object

### Parameters

- **vec** (*Vector*) – the vector object to add
- **merge** (*bool*) – merge overlapping polygons?

**bbox** (*outname=None, driver=None, overwrite=True*)  
create a bounding box from the extent of the Vector object

**Parameters**

- **outname** (*str* or *None*) – the name of the vector file to be written; if None, a Vector object is returned
- **driver** (*str*) – the name of the file format to write
- **overwrite** (*bool*) – overwrite an already existing file?

**Returns** if outname is None, the bounding box Vector object

**Return type** *Vector* or *None*

**close** ()  
closes the OGR vector file connection

**convert2wkt** (*set3D=True*)  
export the geometry of each feature as a wkt string

**Parameters** **set3D** (*bool*) – keep the third (height) dimension?

**extent**  
the extent of the vector object

**Returns** a dictionary with keys *xmin*, *xmax*, *ymin*, *ymax*

**Return type** *dict*

**fieldDefs**

**Returns** the field definition for each field of the Vector object

**Return type** list of *ogr.FieldDefn*

**fieldnames**

**Returns** the names of the fields

**Return type** list of *str*

**geomType**

**Returns** the layer geometry type

**Return type** *int*

**geomTypes**

**Returns** the geometry type of each feature

**Return type** *list*

**getArea** ()

**Returns** the area of the vector geometries

**Return type** *float*

**getFeatureByAttribute** (*fieldname, attribute*)  
get features by field attribute

**Parameters**

- **fieldname** (*str*) – the name of the queried field
- **attribute** (*int* or *str*) – the field value of interest

**Returns** the feature(s) matching the search query

**Return type** list of `ogr.Feature` or `ogr.Feature`

**getFeatureByIndex** (*index*)

get features by numerical (positional) index

**Parameters** **index** (*int*) – the queried index

**Returns** the requested feature

**Return type** `ogr.Feature`

**getProjection** (*type*)

get the CRS of the Vector object. See `spatialist.auxil.crsConvert()`.

**Parameters** **type** (*str*) – the type of projection required.

**Returns** the output CRS

**Return type** `int`, `str` or `osr.SpatialReference`

**getUniqueAttributes** (*fieldname*)

**Parameters** **fieldname** (*str*) – the name of the field of interest

**Returns** the unique attributes of the field

**Return type** list of `str` or `int`

**getfeatures** ()

**Returns** a list of cloned features

**Return type** list of `ogr.Feature`

**init\_features** ()

delete all in-memory features

**init\_layer** ()

initialize a layer object

**layerdef**

**Returns** the layer's feature definition

**Return type** `ogr.FeatureDefn`

**layername**

**Returns** the name of the layer

**Return type** `str`

**load** ()

load all feature into memory

**nfeatures**

**Returns** the number of features

**Return type** `int`

**nfields**

**Returns** the number of fields

**Return type** `int`

**nlayers**

**Returns** the number of layers

**Return type** `int`

**proj4**

**Returns** the CRS in PROJ4 format

**Return type** `str`

**reproject** (*projection*)  
in-memory reprojection

**Parameters** **projection** (`int`, `str`, `osr.SpatialReference`) – the target CRS. See `spatialist.auxil.crsConvert()`.

**setCRS** (*crs*)

directly reset the spatial reference system of the vector object. This is not going to reproject the Vector object, see `reproject()` instead.

**Parameters** **crs** (`int`, `str`, `osr.SpatialReference`) – the input CRS

### Example

```
>>> site = Vector('shape.shp')
>>> site.setCRS('+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs')
```

**srs**

**Returns** the geometry's spatial reference system

**Return type** `osr.SpatialReference`

**write** (*outfile*, *driver=None*, *overwrite=True*)  
write the Vector object to a file

**Parameters**

- **outfile** – the name of the file to write; the following extensions are automatically detected for determining the format driver:
  - `.geojson` (GeoJSON)
  - `.gpkg` (GPKG)
  - `.kml` (KML)
  - `.shp` (ESRI Shapefile)
- **driver** (*str*) – the output file format; needs to be defined if the format cannot be auto-detected from the filename extension
- **overwrite** (*bool*) – overwrite an already existing file?

## 2.4 Vector Tools

<code>bbox</code>	create a bounding box vector object or file.
<code>boundary</code>	Get the boundary of the largest geometry as new vector object.
<code>feature2vector</code>	create a Vector object from ogr features
<code>intersect</code>	intersect two Vector objects
<code>vectorize</code>	Vectorization of an array using <code>gdal.Polygonize</code> .
<code>wkt2vector</code>	convert a well-known text string geometry to a Vector object

`spatialist.vector.bbox` (*coordinates*, *crs*, *outname=None*, *driver=None*, *overwrite=True*)  
create a bounding box vector object or file. The CRS can be in either WKT, EPSG or PROJ4 format

#### Parameters

- **coordinates** (*dict*) – a dictionary containing numerical variables with keys *xmin*, *xmax*, *ymin* and *ymax*
- **crs** (int, str, `osr.SpatialReference`) – the coordinate reference system of the *coordinates*. See `crsConvert()` for options.
- **outname** (*str*) – the file to write to. If *None*, the bounding box is returned as *Vector* object
- **driver** (*str*) –  
  
the output file format; needs to be defined if the format cannot be auto-detected from the filename extension
- **overwrite** (*bool*) – overwrite an existing file?

**Returns** the bounding box Vector object

**Return type** *Vector* or *None*

`spatialist.vector.boundary` (*vectorobject*, *expression=None*, *outname=None*)

Get the boundary of the largest geometry as new vector object. The following steps are performed:

- find the largest geometry matching the expression
- compute the geometry's boundary using `ogr.Geometry.Boundary`, returning a MULTILINE geometry
- select the longest line of the MULTILINE geometry
- create a closed linear ring from this longest line
- create a polygon from this linear ring
- create a new *Vector* object and add the newly created polygon

#### Parameters

- **vectorobject** (*Vector*) – the vector object containing multiple polygon geometries, e.g. all geometries with a certain value in one field.
- **expression** (*str* or *None*) – the SQL expression to select the candidates for the largest geometry.
- **outname** (*str* or *None*) – the name of the output vector file; if *None*, an in-memory object of type *Vector* is returned.

**Returns** if *outname* is *None*, a vector object pointing to an in-memory dataset else *None*

**Return type** *Vector* or *None*

`spatialist.vector.feature2vector (feature, ref, layername=None)`  
create a Vector object from ogr features

**Parameters**

- **feature** (list of `ogr.Feature` or `ogr.Feature`) – a single feature or a list of features
- **ref** (`Vector`) – a reference Vector object to retrieve geo information from
- **layername** (`str` or `None`) – the name of the output layer; retrieved from *ref* if *None*

**Returns** the new Vector object

**Return type** `Vector`

`spatialist.vector.intersect (obj1, obj2)`  
intersect two Vector objects

**Parameters**

- **obj1** (`Vector`) – the first vector object; this object is reprojected to the CRS of obj2 if necessary
- **obj2** (`Vector`) – the second vector object

**Returns** the intersect of obj1 and obj2 if both intersect and None otherwise

**Return type** `Vector` or `None`

`spatialist.vector.vectorize (target, reference, outname=None, layername='layer', fieldname='value', driver=None)`  
Vectorization of an array using `gdal.Polygonize`.

**Parameters**

- **target** (`numpy.ndarray`) – the input array. Each identified object of pixels with the same value will be converted into a vector feature.
- **outname** (`str` or `None`) – the name of the vector file. If *None* a vector object is returned.
- **reference** (`Raster`) – a reference Raster object to retrieve geo information and extent from.
- **layername** (`str`) – the name of the vector object layer.
- **fieldname** (`str`) – the name of the field to contain the raster value for the respective vector feature.
- **driver** (`str` or `None`) – the vector file type of *outname*. Several extensions are read automatically (see `Vector.write()`). Is ignored if *outname*=None.

`spatialist.vector.wkt2vector (wkt, srs, layername='wkt')`  
convert a well-known text string geometry to a Vector object

**Parameters**

- **wkt** (`str`) – the well-known text description
- **srs** (`int`, `str`) – the spatial reference system; see `spatialist.auxil.crsConvert()` for options.
- **layername** (`str`) – the name of the internal `ogr.Layer` object

**Returns** the vector representation

**Return type** `Vector`

## Examples

```
>>> from spatialist.vector import wkt2vector
>>> wkt = 'POLYGON ((0. 0., 0. 1., 1. 1., 1. 0., 0. 0.))'
>>> with wkt2vector(wkt, srs=4326) as vec:
>>>     print(vec.getArea())
1.0
```

## 2.5 General Spatial Tools

<code>cmap_mpl2gdal</code>	convert a matplotlib color table to a GDAL representation.
<code>coordinate_reproject</code>	reproject a coordinate from one CRS to another
<code>crsConvert</code>	convert between different types of spatial reference representations
<code>gdalbuildvrt</code>	a simple wrapper for <code>gdal.BuildVRT</code>
<code>gdal_rasterize</code>	a simple wrapper for <code>gdal.Rasterize</code>
<code>gdal_translate</code>	a simple wrapper for <code>gdal.Translate</code>
<code>gdalwarp</code>	a simple wrapper for <code>gdal.Warp</code>
<code>haversine</code>	compute the distance in meters between two points in latlon
<code>ogr2ogr</code>	a simple wrapper for <code>gdal.VectorTranslate</code> aka <code>ogr2ogr</code>
<code>utm_autodetect</code>	get the UTM CRS for a spatial object

`spatialist.auxil.cmap_mpl2gdal(mplcolor, values)`  
convert a matplotlib color table to a GDAL representation.

### Parameters

- **`mplcolor`** (*str*) – a color table code
- **`values`** (*list*) – the integer data values for which to retrieve colors

**Returns** the color table in GDAL format

**Return type** `gdal.ColorTable`

**Note:** This function is currently only developed for handling discrete integer data values in an 8 Bit file. Colors are thus scaled between 0 and 255.

## Examples

```
>>> from osgeo import gdal
>>> from spatialist.auxil import cmap_mpl2gdal
>>> values = list(range(0, 100))
>>> cmap = cmap_mpl2gdal(mplcolor='YlGnBu', values=values)
>>> print(isinstance(cmap, gdal.ColorTable))
True
```

`spatialist.auxil.coordinate_reproject(x, y, s_crs, t_crs)`  
reproject a coordinate from one CRS to another

**Parameters**

- **x** (*int* or *float*) – the X coordinate component
- **y** (*int* or *float*) – the Y coordinate component
- **s\_crs** (int, str or *osr.SpatialReference*) – the source CRS. See *crsConvert()* for options.
- **t\_crs** (int, str or *osr.SpatialReference*) – the target CRS. See *crsConvert()* for options.

**Returns****Return type** *tuple*`spatialist.auxil.crsConvert(crsIn, crsOut)`

convert between different types of spatial reference representations

**Parameters**

- **crsIn** (int, str, *osr.SpatialReference*) – the input CRS
- **crsOut** (*str*) – the output CRS type; supported options:
  - *epsg*
  - *opengis*
  - *osr*
  - *prettyWkt*
  - *proj4*
  - *wkt*

**Returns** the output CRS**Return type** int, str, *osr.SpatialReference***Examples**

convert an integer EPSG code to PROJ.4:

```
>>> crsConvert(4326, 'proj4')
'+proj=longlat +datum=WGS84 +no_defs '
```

convert the opengis URL back to EPSG:

```
>>> crsConvert('https://www.opengis.net/def/crs/EPSG/0/4326', 'epsg')
4326
```

convert an EPSG compound CRS (WGS84 horizontal + EGM96 vertical) to PROJ.4

```
>>> crsConvert('EPSG:4326+5773', 'proj4')
'+proj=longlat +datum=WGS84 +geoidgrids=egm96_15.gtx +vunits=m +no_defs '
```

`spatialist.auxil.gdal_rasterize(src, dst, options)`a simple wrapper for *gdal.Rasterize***Parameters**

- **src** (str or *ogr.DataSource*) – the input data set
- **dst** (*str*) – the output data set



- **options** (*dict*) – additional parameters passed to `gdal.Rasterize`; see `gdal.RasterizeOptions`

`spatialist.auxil.gdal_translate` (*src, dst, options*)

a simple wrapper for `gdal.Translate`

#### Parameters

- **src** (*str*, `ogr.DataSource` or `gdal.Dataset`) – the input data set
- **dst** (*str*) – the output data set
- **options** (*dict*) – additional parameters passed to `gdal.Translate`; see `gdal.TranslateOptions`

`spatialist.auxil.gdalbuildvrt` (*src, dst, options=None, void=True*)

a simple wrapper for `gdal.BuildVRT`

#### Parameters

- **src** (*str, list*, `ogr.DataSource` or `gdal.Dataset`) – the input data set(s)
- **dst** (*str*) – the output data set
- **options** (*dict*) – additional parameters passed to `gdal.BuildVRT`; see `gdal.BuildVRTOptions`
- **void** (*bool*) – just write the results and don't return anything? If not, the spatial object is returned

`spatialist.auxil.gdalwarp` (*src, dst, options, pbar=False*)

a simple wrapper for `gdal.Warp`

#### Parameters

- **src** (*str*, `ogr.DataSource` or `gdal.Dataset`) – the input data set
- **dst** (*str*) – the output data set
- **options** (*dict*) – additional parameters passed to `gdal.Warp`; see `gdal.WarpOptions`
- **pbar** (*bool*) – add a progressbar?

`spatialist.auxil.haversine` (*lat1, lon1, lat2, lon2*)

compute the distance in meters between two points in latlon

#### Parameters

- **lat1** (*int or float*) – the latitude of point 1
- **lon1** (*int or float*) – the longitude of point 1
- **lat2** (*int or float*) – the latitude of point 2
- **lon2** (*int or float*) – the longitude of point 2

**Returns** the distance between point 1 and point 2 in meters

**Return type** `float`

`spatialist.auxil.ogr2ogr` (*src, dst, options*)

a simple wrapper for `gdal.VectorTranslate` aka `ogr2ogr`

#### Parameters

- **src** (*str* or `ogr.DataSource`) – the input data set
- **dst** (*str*) – the output data set

- **options** (*dict*) – additional parameters passed to `gdal.VectorTranslate`; see `gdal.VectorTranslateOptions`

`spatialist.auxil.utm_autodetect (spatial, crsOut)`

get the UTM CRS for a spatial object

The bounding box of the object is extracted, reprojected to [EPSG:4326](#) and its center coordinate used for computing the best UTM zone fit.

**Parameters**

- **spatial** (*Raster or Vector*) – a spatial object in an arbitrary CRS
- **crsOut** (*str*) – the output CRS type; see function `crsConvert()` for options

**Returns** the output CRS

**Return type** `int` or `str` or `osr.SpatialReference`

## 2.6 Database Tools

`spatialist.sqlite_util.sqlite_setup (driver=':memory:', extensions=None, verbose=False)`

Setup a `sqlite3` connection and load extensions to it. This function intends to simplify the process of loading extensions to `sqlite3`, which can be quite difficult depending on the version used. Particularly loading `spatialite` has caused quite some trouble. In recent distributions of Ubuntu this has become much easier due to a new apt package `libsqlite3-mod-spatialite`. For use in Windows, `spatialist` comes with its own `spatialite` DLL distribution. See [here](#) for more details on loading `spatialite` as an `sqlite3` extension.

**Parameters**

- **driver** (*str*) – the database file or (by default) an in-memory database
- **extensions** (*list*) – a list of extensions to load
- **verbose** (*bool*) – print loading information?

**Returns** the database connection

**Return type** `sqlite3.Connection`

**Example**

```
>>> from spatialist.sqlite_util import sqlite_setup
>>> conn = sqlite_setup(extensions=['spatialite'])
```

## 2.7 Ancillary Functions

This script gathers central functions and classes for general applications

`spatialist.ancillary.dissolve (inlist)`

list and tuple flattening

**Parameters** `inlist` (*list*) – the list with sub-lists or tuples to be flattened

**Returns** the flattened result

**Return type** `list`

## Examples

```
>>> dissolve([[1, 2], [3, 4]])
[1, 2, 3, 4]
```

```
>>> dissolve([(1, 2, (3, 4)), [5, (6, 7)]])
[1, 2, 3, 4, 5, 6, 7]
```

`spatialist.ancillary.finder(target, matchlist, foldermode=0, regex=False, recursive=True)`  
function for finding files/folders in folders and their subdirectories

### Parameters

- **target** (*str* or *list of str*) – a directory, zip- or tar-archive or a list of them to be searched
- **matchlist** (*list*) – a list of search patterns
- **foldermode** (*int*) –
  - 0: only files
  - 1: files and folders
  - 2: only folders
- **regex** (*bool*) – are the search patterns in matchlist regular expressions or unix shell standard (default)?
- **recursive** (*bool*) – search target recursively into all subdirectories or only in the top level? This is currently only implemented for parameter *target* being a directory.

**Returns** the absolute names of files/folders matching the patterns

**Return type** list of str

**class** `spatialist.ancillary.HiddenPrints`

Bases: `object`

Suppress console stdout prints, i.e. redirect them to a temporary string object.

Adapted from <https://stackoverflow.com/questions/8391411/suppress-calls-to-print-python>

## Examples

```
>>> with HiddenPrints():
>>>     print('foobar')
>>> print('foobar')
```

`spatialist.ancillary.multicore(function, cores, multiargs, **singleargs)`  
wrapper for multicore process execution

### Parameters

- **function** – individual function to be applied to each process item
- **cores** (*int*) – the number of subprocesses started/CPU's used; this value is reduced in case the number of subprocesses is smaller

- **multiargs** (*dict*) – a dictionary containing sub-function argument names as keys and lists of arguments to be distributed among the processes as values
- **singleargs** – all remaining arguments which are invariant among the subprocesses

**Returns** the return of the function for all subprocesses

**Return type** `None` or `list`

### Notes

- all *multiargs* value lists must be of same length, i.e. all argument keys must be explicitly defined for each subprocess
- all function arguments passed via *singleargs* must be provided with the full argument name and its value (i.e. `argname=argval`); default function args are not accepted
- if the processes return anything else than `None`, this function will return a list of results
- if all processes return `None`, this function will be of type `void`

### Examples

```
>>> def add(x, y, z):
>>>     return x + y + z
>>> multicore(add, cores=2, multiargs={'x': [1, 2]}, y=5, z=9)
[15, 16]
>>> multicore(add, cores=2, multiargs={'x': [1, 2], 'y': [5, 6]}, z=9)
[15, 17]
```

**See also:**

`pathos.multiprocessing`

`spatialist.ancillary.parse_literal(x)`

return the smallest possible data type for a string or list of strings

**Parameters** *x* (*str* or *list*) – a string to be parsed

**Returns** the parsing result

**Return type** `int`, `float` or `str`

### Examples

```
>>> isinstance(parse_literal('1.5'), float)
True
```

```
>>> isinstance(parse_literal('1'), int)
True
```

```
>>> isinstance(parse_literal('foobar'), str)
True
```

`spatialist.ancillary.run(cmd, outdir=None, logfile=None, inlist=None, void=True, error-pass=False, env=None)`

wrapper for subprocess execution including logfile writing and command prompt piping  
this is a convenience wrapper around the `subprocess` module and calls its class `Popen` internally.

#### Parameters

- **cmd** (*list*) – the command arguments
- **outdir** (*str* or *None*) – the directory to execute the command in
- **logfile** (*str* or *None*) – a file to write stdout to
- **inlist** (*list* or *None*) – a list of arguments passed to stdin, i.e. arguments passed to interactive input of the program
- **void** (*bool*) – return stdout and stderr?
- **errorpass** (*bool*) – if False, a `subprocess.CalledProcessError` is raised if the command fails
- **env** (*dict* or *None*) – the environment to be passed to the subprocess

**Returns** a tuple of (stdout, stderr) if *void* is False otherwise None

**Return type** *None* or *Tuple*

`spatialist.ancillary.which(program, mode=1)`

mimics UNIX's `which`

taken from this post: <http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python>  
can be replaced by `shutil.which()` starting from Python 3.3

#### Parameters

- **program** (*str*) – the program to be found
- **mode** (*os.F\_OK* or *os.X\_OK*) – the mode of the found file, i.e. file exists or file is executable; see `os.access()`

**Returns** the full path and name of the command

**Return type** *str* or *None*

`spatialist.ancillary.parallel_apply_along_axis(func1d, axis, arr, cores=4, *args, **kwargs)`

Like `numpy.apply_along_axis()` but using multiple threads. Adapted from [here](#).

#### Parameters

- **func1d** (*function*) – the function to be applied
- **axis** (*int*) – the axis along which to apply *func1d*
- **arr** (*numpy.ndarray*) – the input array
- **cores** (*int*) – the number of parallel cores
- **args** (*any*) – Additional arguments to *func1d*.
- **kwargs** (*any*) – Additional named arguments to *func1d*.

**Returns**

**Return type** *numpy.ndarray*

## 2.8 ENVI HDR file manipulation

This module offers functionality for editing ENVI header files

**class** `spatialist.envi.HDRObject` (*data=None*)  
ENVI HDR info handler

**Parameters** *data* (*str*, *dict* or *None*) – the file or dictionary to get the info from; If *None* (default), an object with default values for an empty raster file is returned

### Examples

```
>>> from spatialist.envi import HDRObject
>>> with HDRObject('E:/test.hdr') as hdr:
>>>     hdr.band_names = ['one', 'two']
>>>     print(hdr)
>>>     hdr.write()
```

**write** (*filename='same'*)  
write object to an ENVI header file

`spatialist.envi.hdr` (*data, filename*)  
write ENVI header files

#### Parameters

- **data** (*str* or *dict*) – the file or dictionary to get the info from
- **filename** (*str*) – the HDR file to write

---

## Some general examples

---

### 3.1 in-memory vector object rasterization

Here we create a new raster data set with the same geo-information and extent as a reference data set and burn the geometries from a shapefile into it.

In this example, the shapefile contains an attribute `Site_name` and one of the geometries in the shapefile has a value of `my_testsite` for this attribute.

We use the `expressions` parameter to subset the shapefile and burn a value of 1 in the raster at all locations where the geometry selection overlaps. Multiple expressions can be defined together with multiple burn values.

Also, burn values can be appended to an already existing raster data set. In this case, the rasterization is performed in-memory to further use it for e.g. plotting. Alternatively, an `outname` can be defined to directly write the result to disk as a GeoTiff.

See `spatialist.raster.rasterize()` for further reference.

```
>>> from spatialist import Vector, Raster
>>> from spatialist.raster import rasterize
>>> import matplotlib.pyplot as plt
>>>
>>> shapefile = 'testsites.shp'
>>> rasterfile = 'extent.tif'
>>>
>>> with Raster(rasterfile) as ras:
>>>     with Vector(shapefile) as vec:
>>>         mask = rasterize(vec, reference=ras, burn_values=1, expressions=["Site_
↳ Name='my_testsite'"])
>>>         plt.imshow(mask.matrix())
>>>         plt.show()
```





### 4.1 0.4 | 2019-12-05

- `spatialist.auxil.gdalwarp()`: optional progressbar via new argument *pbar*
- `spatialist.raster.Raster`
  - enabled reading data in zip and tar.gz archives
  - `bbox()`
    - \* renamed parameter *format* to *driver*
    - \* new parameter *source* to get coordinates from the image of the GCPs
- `spatialist.raster.stack()`
  - improved parallelization
  - new parameter *pbar* to make use of the new `gdalwarp()` functionality
- bug fixes

### 4.2 0.5 | 2020-04-21

- compatibility of SpatiaLite tools with Windows10
- compatibility with GDAL 3
- new function `spatialist.ancillary.parallel_apply_along_axis()`: like `numpy.apply_along_axis()` but using multiple threads
- new function `spatialist.auxil.cmap_mpl2gdal()`: convert matplotlib color sequences to GDAL color tables
- `spatialist.raster.Raster`

- method `write()`: new argument `cmap` to write color maps to a file; can be created with e.g. `cmap_mpl2gdal()`
- subsetting: option to use map coordinates instead of just pixel coordinates
- method `array()`:
  - \* automatically reduce dimensionality of returned arrays using `numpy.squeeze()`
  - \* cast arrays to `float32` if the native data type does not support `numpy.nan` for masking missing data
- option to read image data in all `.tar*` archives, not just `tar.gz`
- new methods `coord_map2img()` and `coord_img2map()` to convert between pixel/image and map coordinates of a dataset
- `spatialist.vector.Vector`
  - better representation of the object's geometry type(s) with new method `geomTypes()` and additional info when printing the object with `print()`
- `spatialist.explorer.RasterViewer`
  - optionally pass custom functions to create additional plots using argument `custom`

### 4.3 0.6 | 2020-07-17

- method `spatialist.raster.Raster.write()`
  - optionally update an existing file with new arg `update`
  - partial writing with new args `xoff` and `yoff`
  - write external arrays with new arg `array`
- new function `spatialist.raster.png()`
- new function `spatialist.raster.apply_along_time()`
- bug fixes

### 4.4 0.7 | 2021-06-30

- `spatialist.raster.Raster`: option to subset objects by
  - band names
  - time range
- `spatialist.auxil.crsConvert()`: raise a `RuntimeError` if no corresponding EPSG code was found
- module `spatialist.explorer` and associated demo data and notebook have been outsourced to [https://github.com/johntruckenbrodt/spatialist\\_explorer](https://github.com/johntruckenbrodt/spatialist_explorer)

### 4.5 0.8 | 2021-09-07

- `spatialist.raster.Raster`:
  - method `write()`

- \* removed argument *compress\_tif*
- \* added arguments *options* and *overviews*
- subsetting support for time stamps, color tables and nodata
- option to convert band names to time stamps by passing a function as argument *timestamps*
- bug fixes
- *spatialist.envi*: enabled reading of HDR files in ZIP archives

## 4.6 0.8.1 | 2021-10-05

- *spatialist.auxil.crsConvert()*:
  - use https for *opengis* links
  - apply ESPG validity checks when output format is *opengis* (not just when *epsg*)
- *spatialist.ancillary.finder()*:
  - raise `RuntimeError` (not `TypeError`) if target is a file but is neither zip nor tar
  - raise `RuntimeError` (not `TypeError`) if target is a string but is neither directory nor file

## 4.7 0.9.0 | 2022-01-25

- *spatialist.raster.Raster*:
  - method *write()*:
    - \* remove unused argument *compress\_tif*
    - \* add support for COG driver
- *spatialist.vector.Vector*:
  - method *addlayer()*:
    - \* enable all SRS type options supported by *crsConvert()*
- *spatialist.raster.rasterize()*: allow value *None* for argument *nodata*
- new functions:
  - *spatialist.vector.vectorize()*
  - *spatialist.vector.boundary()*



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### S

- `spatialist.ancillary`, [22](#)
- `spatialist.auxil`, [19](#)
- `spatialist.envi`, [26](#)
- `spatialist.raster`, [9](#)
- `spatialist.sqlite_util`, [22](#)
- `spatialist.vector`, [16](#)





## Symbols

`__getitem__()` (*spatialist.raster.Raster method*), 3  
`__getitem__()` (*spatialist.vector.Vector method*), 13

## A

`addfeature()` (*spatialist.vector.Vector method*), 13  
`addfield()` (*spatialist.vector.Vector method*), 13  
`addlayer()` (*spatialist.vector.Vector method*), 13  
`addvector()` (*spatialist.vector.Vector method*), 13  
`allstats()` (*spatialist.raster.Raster method*), 4  
`apply_along_time()` (*in module spatialist.raster*), 9  
`array()` (*spatialist.raster.Raster method*), 4  
`assign()` (*spatialist.raster.Raster method*), 4

## B

`bandnames` (*spatialist.raster.Raster attribute*), 5  
`bands` (*spatialist.raster.Raster attribute*), 5  
`bbox()` (*in module spatialist.vector*), 17  
`bbox()` (*spatialist.raster.Raster method*), 5  
`bbox()` (*spatialist.vector.Vector method*), 14  
`boundary()` (*in module spatialist.vector*), 17

## C

`close()` (*spatialist.raster.Raster method*), 5  
`close()` (*spatialist.vector.Vector method*), 14  
`cmap_mpl2gdal()` (*in module spatialist.auxil*), 19  
`cols` (*spatialist.raster.Raster attribute*), 5  
`convert2wkt()` (*spatialist.vector.Vector method*), 14  
`coord_img2map()` (*spatialist.raster.Raster method*), 5  
`coord_map2img()` (*spatialist.raster.Raster method*), 5  
`coordinate_reproject()` (*in module spatialist.auxil*), 19  
`crsConvert()` (*in module spatialist.auxil*), 20

## D

`dim` (*spatialist.raster.Raster attribute*), 6

`dissolve()` (*in module spatialist.ancillary*), 22  
`driver` (*spatialist.raster.Raster attribute*), 6  
`dtype` (*spatialist.raster.Raster attribute*), 6

## E

`epsg` (*spatialist.raster.Raster attribute*), 6  
`extent` (*spatialist.raster.Raster attribute*), 6  
`extent` (*spatialist.vector.Vector attribute*), 14  
`extract()` (*spatialist.raster.Raster method*), 6

## F

`feature2vector()` (*in module spatialist.vector*), 17  
`fieldDefs` (*spatialist.vector.Vector attribute*), 14  
`fieldnames` (*spatialist.vector.Vector attribute*), 14  
`files` (*spatialist.raster.Raster attribute*), 6  
`finder()` (*in module spatialist.ancillary*), 23  
`format` (*spatialist.raster.Raster attribute*), 6

## G

`gdal_rasterize()` (*in module spatialist.auxil*), 20  
`gdal_translate()` (*in module spatialist.auxil*), 21  
`gdalbuildvrt()` (*in module spatialist.auxil*), 21  
`gdalwarp()` (*in module spatialist.auxil*), 21  
`geo` (*spatialist.raster.Raster attribute*), 6  
`geogcs` (*spatialist.raster.Raster attribute*), 7  
`geomType` (*spatialist.vector.Vector attribute*), 14  
`geomTypes` (*spatialist.vector.Vector attribute*), 14  
`getArea()` (*spatialist.vector.Vector method*), 14  
`getFeatureByAttribute()` (*spatialist.vector.Vector method*), 14  
`getFeatureByIndex()` (*spatialist.vector.Vector method*), 15  
`getfeatures()` (*spatialist.vector.Vector method*), 15  
`getProjection()` (*spatialist.vector.Vector method*), 15  
`getUniqueAttributes()` (*spatialist.vector.Vector method*), 15

## H

`haversine()` (*in module spatialist.auxil*), 21

`hdr()` (in module *spatialist.envi*), 26  
`HDRObject` (class in *spatialist.envi*), 26  
`HiddenPrints` (class in *spatialist.ancillary*), 23

## I

`init_features()` (*spatialist.vector.Vector* method), 15  
`init_layer()` (*spatialist.vector.Vector* method), 15  
`intersect()` (in module *spatialist.vector*), 18  
`is_valid()` (*spatialist.raster.Raster* method), 7

## L

`layerdef` (*spatialist.vector.Vector* attribute), 15  
`layername` (*spatialist.vector.Vector* attribute), 15  
`layers()` (*spatialist.raster.Raster* method), 7  
`load()` (*spatialist.raster.Raster* method), 7  
`load()` (*spatialist.vector.Vector* method), 15

## M

`matrix()` (*spatialist.raster.Raster* method), 7  
`multicore()` (in module *spatialist.ancillary*), 23

## N

`nfeatures` (*spatialist.vector.Vector* attribute), 15  
`nfields` (*spatialist.vector.Vector* attribute), 15  
`nlayers` (*spatialist.vector.Vector* attribute), 16  
`nodata` (*spatialist.raster.Raster* attribute), 7

## O

`ogr2ogr()` (in module *spatialist.auxil*), 21

## P

`parallel_apply_along_axis()` (in module *spatialist.ancillary*), 25  
`parse_literal()` (in module *spatialist.ancillary*), 24  
`png()` (in module *spatialist.raster*), 9  
`proj4` (*spatialist.raster.Raster* attribute), 7  
`proj4` (*spatialist.vector.Vector* attribute), 16  
`proj4args` (*spatialist.raster.Raster* attribute), 7  
`projcs` (*spatialist.raster.Raster* attribute), 7  
`projection` (*spatialist.raster.Raster* attribute), 7

## R

`Raster` (class in *spatialist.raster*), 3  
`rasterize()` (in module *spatialist.raster*), 10  
`reproject()` (*spatialist.vector.Vector* method), 16  
`res` (*spatialist.raster.Raster* attribute), 8  
`rescale()` (*spatialist.raster.Raster* method), 8  
`rows` (*spatialist.raster.Raster* attribute), 8  
`run()` (in module *spatialist.ancillary*), 24

## S

`setCRS()` (*spatialist.vector.Vector* method), 16

`spatialist.ancillary` (module), 22  
`spatialist.auxil` (module), 19  
`spatialist.envi` (module), 26  
`spatialist.raster` (module), 9  
`spatialist.sqlite_util` (module), 22  
`spatialist.vector` (module), 16  
`sqlite_setup()` (in module *spatialist.sqlite\_util*), 22  
`srs` (*spatialist.raster.Raster* attribute), 8  
`srs` (*spatialist.vector.Vector* attribute), 16  
`stack()` (in module *spatialist.raster*), 11  
`subset_tolerance` (in module *spatialist.raster*), 12

## U

`utm_autodetect()` (in module *spatialist.auxil*), 22

## V

`Vector` (class in *spatialist.vector*), 12  
`vectorize()` (in module *spatialist.vector*), 18

## W

`which()` (in module *spatialist.ancillary*), 25  
`wkt2vector()` (in module *spatialist.vector*), 18  
`write()` (*spatialist.envi.HDRObject* method), 26  
`write()` (*spatialist.raster.Raster* method), 8  
`write()` (*spatialist.vector.Vector* method), 16