ClearMap Documentation

Release 0.9.2

Christoph Kirst

May 16, 2016

ClearMap is a toolbox for the analysis and registration of volumetric data from cleared tissues.

ClearMap has been designed to analyze large 3D image stack datasets obtained with Light Sheet Microscopy of iDISCO+ cleared mouse brains samples immunolabeled for nuclear proteins. ClearMap can perform image registration to a 3D annotated reference (such as the Allen Institute Brain Atlases), volumetric image processing, object detection and statistical analysis. The tools in *ClearMap* have been written with the mapping of Immediate Early Genes in the brain as the primary application.

However, these tools should also be more broadly useful for data obtained with other types of microscopes, other types of markers, and other clearing techniques. Moreover, the registration and region segmentation capabilities of ClearMap are not depending on the Atlases and annotations we used in our study. Users are free to import their own reference files and annotation files, so the use of *ClearMap* can be expanded to other species, and other organs or samples.

ClearMap is written in Python 2.7, and is designed to take advantage of parallel processing capabilities of modern workstations. We hope the open structure of the code will enable in the future many new modules to be added to ClearMap to broaden the range of applications to different types of biological objects or structures.

CHAPTER

ONE

AUTHOR AND LICENSE

1.1 Authors:

1.1.1 ClearMap lead programming and design:

Christoph Kirst, *The Rockefeller University*

1.1.2 Scripts and specific applications:

Nicolas Renier and Christoph Kirst *The Rockefeller University*

1.1.3 Documentation:

Christoph Kirst and Nicolas Renier *The Rockefeller University*

1.1.4 Additional programming and consulting:

Kannan Umadevi Venkataraju *Cold Spring Harbor Laboratories*

1.2 License

CHAPTER

TWO

USING CLEARMAP

2.1 Overview of ClearMap

ClearMap is a toolbox to analyze and register microscopy images of cleared tissue. It is targeted towards cleared brain tissue using the *iDISCO+ Clearing Method* but can be used with any volumetric imaging data. ClearMap contains a large number of functions dedicated to many aspects of 3D image manipulation and object detection, which could open a lot of possibilities for advanced users. For most users however, all relevant functions are explained in the tutorial in the next section, which contains a classic application case for ClearMap.

The ClearMap code package is structured into four main modules:

- *IO* for reading and writing images and data
- *Alignment* for resampling, reorientation and registration of images onto references
- *Image Processing* for correcting and quantifying the image data
- *Analysis* for the statistical analysis of the data

2.1.1 IO

ClearMap supports a wide range of image formats with a particular focus on volumetric data packaged as stacks or individual files:

Format	Description
TIF	tif images and stacks
RAW / MHD	raw image files with optional mhd header file
NRRD	nearly raw raster data files
IMS	Imaris image files
pattern	folder, file list or file pattern of a stack of 2d images

We recommend using when possible to use the pattern format, such as `image-Zxxxx.tif` where ''xxxx" is a number, such as 0001.

Note: ClearMap can read the image data from a Bitplane's Imaris, but can't export image data as an Imaris file.

Images are represented internally as numpy arrays. ClearMap assumes images in arrays are arranged as [x,y], [x,y,z] or [x,y,z,c] where x,y,z correspond to the x,y,z coordinates as when viewed in an image viewer such as *[ImageJ]* and c to a possible color channel.

ClearMap also supports several data formats for storing data points, such as cell center coordinates or intensities:

Format	Description
CSV	comma separated values in text file, for exporting to other programs
NPY	numpy binary file, faster and more compact format for the point data
VTK	vtk point data file, for exporting to some programs
IMS	Imaris data file, for writing points onto an existing Imaris file

points files simply contain all point coordinates arranged as an array of [x,y,z] coordinates where each line is a detected cell center. *intensities* files are companion to point files (only for csv and npy formats), where each line contains informations about intensity and detected size for the corresponding center in the point file. Each line in the array of the intensities file has 4 rows organised as follows:

Row	Description
0	Max intensity of the cell center in the raw data
1	Max intensity of the cell center after the DoG filtering.
2	Max intensity of the cell center after the background subtraction
3	Cell size in voxels after the watershed detection

2.1.2 Alignment

The Alignment module provides tools to resample, reorient and register volumetric images in a fast parallel way.

Image registration is done by interfacing to the *[Elastix]* software package. This package allows it to align cleared mouse brains onto the Allen brain atlas *[ABA]*.

2.1.3 Image Processing

ClearMap provides a number of image processing tools with a focus on the processing of large 3D volumetric images in parallel. For the detection of objects in 3D, ClearMap has a modular architecture. For the user, this is hidden and handled automatically by the `detectCells` function (see the example script).

The main processing modules include:

Module	Description
BackgroundRemoval	Background estimation and removal via morphological opening
IlluminationCorrection	Correction of vignetting and other illumination errors
Filter	Filtering of the image via large set of filter kernels
GreyReconstruction	Reconstruction of images
SpotDetection	Detection of local peaks
CellDetection	Detection of cell centers
CellSizeDetection	Detection of cell shapes via watershed
IlastikClassification	Classification of voxels via interface to *[Ilastik]*

The modular structure of this sub-packages allows for fast and flexible integration of additional modules for future developments.

2.1.4 Analysis

This part of ClearMap provides a toolbox for the statistical analysis and visualization of detected cells or structures and region specific analysis of annotated data.

For cleared mouse brains aligned to the *[ABA]* a wide range of statistical analysis tools with respect to the annotated brain regions in the atlas is supported. Two types of analysis are done:

- Voxel statistics, which are based on the heat-map generated from the detected cell centers. These are usually represented as image stacks of mean, standard deviation, p-values with False Discovery Rate options.

- Region statistics, which are based on the annotated regions from the reference annotation file. They are usually represented as spreadsheets containing the statistics for each region.

The Key modules are:

Module	Description
Statistics	Statistical tools for the analysis of detected cells
Voxelization	For voxel-based statistics: voxelization of cells for visualization and analysis
Label	For region-based statistics: tools to analyse data with the annotated reference files

The use of the modules is explained in the tutorial.

2.1.5 iDISCO+ Clearing Method

Robust quantification of 3D datasets requires images as uniform as possible for the signal properties, both on each plane, and also at all imaging depths. The iDISCO+ method is an evolution of the iDISCO whole-mount labeling technique to improve the diffusion and background of staining in large samples *[Renier2014]*, and the 3DISCO clearing technique *[Erturk2012]*. The iDISCO+ staining and clearing method is combined optimally with the very large field of view enabled by light sheet microscopy, in particular the ultramicroscope optical design, which enables low magnification imaging with high speed and relatively high resolution.

The datasets used to develop ClearMap are usually composed of two channels:

- The signal channel. Typically obtained in the far-red light spectrum, where the optical properties of the cleared tissue are at their best for signal-to-noise and transparency. It is recommended when possible to use nuclear reporters or proteins to facilitate the object detection.
- The autofluorescence channel, usually collected in the blue-green light spectrum. The background tissue fluorescence highlights the major structures of the tissue to facilitate the 3D image registration. Only the contrast between regions is important here, so it doesn't matter if the relative intensities between regions are not the same as on the reference scans.

See these videos for example of light sheet imaging of cleared tissues:

- Dopaminergic system in the embryonic mouse
- Cortical and hippocampal neurons in the adult mouse brain

More info can be found on the *[iDISCO]* webpage.

2.1.6 References

2.2 Installation

2.2.1 Requirements

ClearMap is written in Python 2.7. It should run on any Python environment, but it also relies on external softwares such as Elastix which may not run optimally on Windows or Apple systems. For typical use, we recommend a workstation running Ubuntu 14 or later with at least 4 CPU cores, 64Gb of RAM and SSD disks. 128Gb of RAM and 6 cores or above will have much increased performances. The processing time however will depend greatly on the parameters set, so your experience may be different. Also, large hard drives may be needed to host the raw data, although 1 to 4Tb of storage space should be enough for most users.

2.2.2 Installation

To install ClearMap, first create a folder to contain all the files for the program. Download the scripts from the latest version of ClearMap from The iDISCO website and extract them in the ClearMap folder. You also need to download individually the following softwares:

- To do the alignement, you should download Elastix (http://elastix.isi.uu.nl)
- If you wish to use the machine learning filters, download Ilastik (http://ilastik.org). This is an optional download, only if you wish to use this more complete object detection framework for complex objects.

If you're starting from a fresh Ubuntu 16.04LTS install, for instance, here are the steps to complete the installation. Open a terminal window and type the following instructions:

- Install pip:

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

- Install Spyder:

```
$ sudo apt-get install spyder
```

- Install the necessary libraries:

```
$ sudo apt-get install python-opencv
$ sudo apt-get install cython
$ sudo -H pip install --upgrade pip
$ sudo -H pip install scikit-image
$ sudo -H pip install h5py
$ sudo apt-get install python-vigra
```

We use Spyder to run the code. Set the project explorer and working environment in the ClearMap folder.

2.2.3 Configuration

Open the file `ClearMap/Settings.py` to set the paths of installations for Ilastik and Elastix:

```
>>> IlastikPath = '/yourpath/ilastik-05-rc-final';
>>> ElastixPath = '/yourpath/elastix';
```

Note that Ilastik is optional. If you haven't installed it, you can set the path to `None`. You can also set the installation to run on multiple machines by setting a host specific path:

```
>>> if hostname == 'kagalaska.nld':  #Christoph's Laptop
>>>     IlastikPath = '/home/ckirst/programs/ilastik-05/';
>>>     ElastixPath = '/home/ckirst/programs/elastix/';
>>> elif hostname == 'mtllab-Ubuntu': #Nico's Workstation
>>>     IlastikPath = '/usr/local/ilastik-05-rc-final';
>>>     ElastixPath = '/usr/local/elastix';
```

2.2.4 Additionnal Informations:

The following libraries are used by ClearMap:

- Matplotlib 1.5.1
- Numpy 1.11.0

- Scipy 0.16.0
- Skimage 0.12.3
- Mahotas 1.3.0 (website)
- h5py 2.6.0 (for Imaris files input/output only)
- openCV 2.4.9.1
- PyQt4
- tifffile 0.6.2
- EVTK (website) (only necessary, for output to vtk files)
- Cython 0.23.1
- Vigra (used for the original Ilastik integration).

2.3 Tutorial

The goal of this tutorial is to explain the scripts we used to analyze samples. As an example, we will use a dataset from a Light Sheet imaged adult mouse brain stained for c-fos. The tutorial files also contain an autofluorescence file to enable the registration of the scan to the reference atlas. The tutorial files are found in the ClearMap/Scripts folder. They consist of :

- *The Parameter File* This sets the parameters individually for each sample
- *The Run File* This will run all the commands to process each sample individually
- *Analysis Tools* This scripts will run the analysis tools and group statistics for all samples in the batch

A project will usually contain 1 parameter file for each sample, 1 run file for the whole experiment and 1 analysis file for the whole experiment.

2.3.1 The Parameter File

The parameter is a Python script that will contain all the necessary informations to process each sample. An example script, *parameter_file_template.py* is provided in the ClearMap/Scripts folder. Open this file to follow closely the tutorial. It contains the following sections:

Section	Description
Import modules	load from ClearMap the functions used here
Data parameters	points to the files used, their resolution and orientation
Cell detection	parameters for the cell detection, and module used
Heat map generation	to generate a voxelized map of the detected cells
Config Parameters	the parameters for memory and processors management
Run Parameters	you would usually not change these. They specify how the data will be processed

Import Modules

You would usually not change these. They are all the functions that will be used later either in the parameter file or in the execution file.

Data parameters

This is where you point to the files used, their resolution and orientation. It also defines which atlas and annotation files to use.

To set the directory where all files will be read and written for this sample:

```
>>> BaseDirectory = '/home/mtllab/pharmaco/sample1';
```

To set the image files used for the processing:

```
>>> cFosFile = os.path.join(BaseDirectory, 'cfos/0_8x-cfos-Table Z\d{4}.ome.tif');
>>> AutofluoFile = os.path.join(BaseDirectory, \
>>>                             'autofluo/0_8xs3-autofluo-Table Z\d{4}.ome.tif');
```

Note the use of the command `os.path.join` to link the set `BaseDirectory` with the folder where the files are. On the LaVision ultramicroscope system, the images files are generated not as stacks, but as numbered files in the ome.tif format. Each Z stack will end by `-Table Z0000.ome.tif`. The 0000 is the plane number. To indicate **ClearMap** to read the next planes, replace the 4 digits with the command `\d{4}`. On our system, files for each channel (here, c-fos and background fluorescence) are saved in a different stack, in a different folder.

To restrict the range for the object detection:

```
>>> cFosFileRange = {'x' : all, 'y' : (180, 2600), 'z' : all};
```

This range will only affect the region used for the cell detection. It will not be taken into account for the 3D image registration to the reference Atlas, nor for the voxelization or other analysis. This is useful to limit the amount of memory used. In our example, we use the full x range, the full z range, but restrict the y range. The camera on our system, an Andor sNEO CMOS, has a sensor size of 2160 x 2660. However, the lens used on for the acquisition, an Olympus 2X 0.5NA MVPLAPO, has a strong corner deformation, so we restrict the y range because no cells can be reliably detected outside of this range.

As a reminder, in the image files, the (0, 0, 0) coordinate corresponds to the upper left corner of the first plane. To the opposite, the (2160, 2660, 2400) coordinate will be the bottom right corner of the last plane (here 2400, but can vary).

When optimizing the parameters for the object detection, you should dramatically restrict the range to speed up the detection. We recommend using 500 planes, 500 pixels on each side:

```
>>> cFosFileRange = {'x' : (500, 1000), 'y' : (500, 1000), 'z' : (500, 1000)};
```

But of course adapting the range to where the relevant objects are on your sample.

Next, to set the resolution of the original data (in µm / pixel):

```
>>> OriginalResolution = (4.0625, 4.0625, 3);
```

In this example, this is set for a zoom factor of 0.8X on the LaVision system with the 2X lens. This information can be found in the metadata of the tif file usually. If you don't know the pixel size, we recommend opening the stack with the plugin BioFormat on ImageJ, and going to « image » -> « show info » to read the metadata. On the LaVision file, this information is at the end of the list.

The orientation of the sample has to be set to match the orientation of the Atlas reference files. It is not mandatory to acquire the sample in the same orientation as the atlas. For instance, you can acquire the left side of the brain, and map it onto the right side of the atlas:

```
>>> FinalOrientation = (1,2,3);
```

The convention is as follow (examples given, any configuration is possible):

Value of the tuple	Description
(1, 2, 3)	The scan has the same orientation as the atlas reference
(-1, 2, 3)	The x axis is mirrored compared to the atlas
(1, -2, 3)	The y axis is mirrored compared to the atlas
(2, 1, 3)	Performs a rotation by exchanging the x and y axis
(3, 2, 1)	Performs a rotation by exchanging the z and x axis

For our samples, we use the following orientation to match our atlas files:

- The right side of the brain is facing the objective, lateral side up.
- The rostral side of the brain is up
- The dorsal side is facing left
- The ventral side is facing right

This means that in our scans, if we want to image the right hemisphere, we use (1, 2, 3) and if we want to image the left hemisphere, we use (-1, 2, 3).

To set the output for the voxelized heat map file:

```
>>> VoxelizationFile = os.path.join(BaseDirectory, 'points_voxelized.tif');
```

To set the resolution of the Atlas Files (in µm/ pixel):

```
>>> AtlasResolution = (25, 25, 25);
```

To choose which atlas files you would like to use:

```
>>> PathReg        = '/home/mtllab/Documents/warping';
>>> AtlasFile      = os.path.join(PathReg, 'half_template_25_right.tif');
>>> AnnotationFile = os.path.join(PathReg, 'annotation_25_right.tif');
```

It is important to make sure that the Atlas used is in the correct orientation (see above), but also don't contain too much information outside of the field of view. While the registration program can deal with a bit of extra « bleed » outside of the sample, this should be kept to a minimum. We usually prepare different crops of the atlas file to match the usual field of views we acquire.

Cell detection

At this point, two detection methods exist: the `SpotDetection` and `Ilastik`:

- `SpotDetection` is designed for globular objects, such as neuron cell bodies or nuclei. This is the fastest method, and offers a greater degree of fine controls over the sensitivity of the detection. However, it is not well suited for complex objects.
- `Ilastik` is a framework that relies on the user generating a classifier through the graphical interface of the Ilastik program, by painting over a few objects and over the background. The program will then learn to classify the pixels between objects or backgrounds based on the user indications. This is a very easy way to tune very complex filters to detect complex objects or textures. However, the classification is a black box, and very dependent of the user's classification.

In this tutorial, we will use the SpotDetection method. To choose which method to use for the cell detection, set the variable as follows:

```
>>> ImageProcessingMethod = "SpotDetection";
```

The parameters for the Spot Detection methods are then sorted in « dictionaries » by theme :

Dictionary name	Description
correctIlluminationParameter	If you have an intensity profile for your microscope, you can correct variations in illuminations here
removeBackgroundParameter	To set the background subtraction via morphological opening
filterDoGParameter	To set the parameters for the Difference of Gaussian filter
findExtendedMaximaParameter	If the object contains multiple peaks of intensity, this will collapse them into one peak
findIntensityParameter	Often, the center of the mass of an object is not the voxel of highest intensity. This is a correction for this
detectCellShapeParameter	This sets the parameters for the cell shape filling via watershed

Correcting the illumination:

Because of the Gaussian shape of the light sheet and of the objecting lens vignetting, the sample illumination is not uniform. While correcting the illumination can improve the uniformity of the cell detection, it is usually not really necessary if all samples where imaged the same way, as the same anatomical features will be positioned in the same region of the lens across samples. Nevertheless, to correct for variation in the illumination use:

```
>>> correctIlluminationParameter = {
>>>     "flatfield"  : None,
>>>     "background" : None,
>>>     "scaling"    : "Mean",
>>>     "save"       : None,
>>>     "verbose"    : True
>>> }
```

For this tutorial, we will not use the correction, so the `flatfield` parameter is set to `None`. Please note that you need to generate an intensity profile for your system if you wish to use this function.

Background Subtraction:

This is the most important pre-treatment step, usually always turned on. The background subtraction via morphological opening is not very sensitive to the size parameter used, as long as it is in the range of the size of the objects detected. The parameters for the background subtraction are as follow:

```
>>> removeBackgroundParameter = {
>>>     "size"    : (7,7),
>>>     "save"    : None,
>>>     "verbose" : True
>>> }
```

The parameter `size` is a tuple with (x,y) in pixels and correspond to an ellipsoid of this size. Of importance, you can check the result of the background subtraction by setting the `save` parameter to a filename. This will output a series of tif images you can open with ImageJ to check the results. For instance you could set `save` like this:

```
>>>     "save"    : os.path.join(BaseDirectory, 'background/background\d{4}.ome.tif');
```

You have to use the `\d{4}` notation to save the files as a series of images, otherwise only the first plane is saved!

Note: Only use the `save` function when you analyse a small subset of your data, otherwise the full stack will be written to the disk. Don't forget to turn off this parameter when you're done optimizing the filters.

Difference of Gaussians filter:

This is an optional filter to improve the contrast of objects. This filter has a "Mexican Hat" shape that weighs negatively the intensity at the border of the objects. The main parameter to set here is the `size`, as an (x,y,z) tuple, for instance `(6,6,11)` would work well for our example. However, we'll bypass this filter and set it to `None`:

```
>>> filterDoGParameter = {
>>>     "size"    : None,
>>>     "sigma"   : None,
>>>     "sigma2"  : None,
>>>     "save"    : None,
>>>     "verbose" : True
>>> }
```

For this tutorial, we will not use the DoG filter, as it is unnecessary. The signal from a c-Fos nuclear staining has enough contrast and a simple shape that do not necessitate this additionnal filtering, but it could help increase the contrast of the relevant objects for other experiments.

Note: As for the background subtraction seen above, you can save the output of the filter to files in a folder. This is important in order to check that the settings you used are effective!

Extended Maxima:

The extended maxima is another filter to use for objects that contain several peaks of intensity, like for instance a higher resolution view of a cell nucleus where you might have a more granular texture. In the case of our tutorial, the c-fos nuclear signal is imaged at low resolution, so the object only appears as a single peak, so we will turn off the extended maxima by setting the `hMax` parameter to `None`:

```
>>> findExtendedMaximaParameter = {
>>>     "hMax"      : None,
>>>     "size"      : 0,
>>>     "threshold" : 0,
>>>     "save"      : None,
>>>     "verbose"   : True
>>> }
```

Note: Saving the files for the output of this filter as explained above would overlay in red the extended maxima found on top of the DoG filtered image (we recommend using also DoG if you use the Extended Maxima).

Peak Intensity:

By default, the code will look for the center of the 3D shape painted by the Extended Maxima and attribute the x,y,z to this coordinate. This is the coordinate that will be saved in the point file. However, we noticed that often, the pixel that contains the highest intensity (the peak) is not always the center of the volume. This is likely due to potential deformations of the shape of the nucleus by the objective lens. To look for the actual peak intensity for the detected object, we'll use this function:

```
>>> findIntensityParameter = {
>>>     "method" : 'Max',
>>>     "size"   : (3,3,3)
>>> }
```

If the parameter `method` is set to `None`, then the peak intensity will be also the pixel at the center of the volume. We'll set the parameter to `Max` to look if there is a voxel around the center that has a higher intensity than the center. This will be done by looking around the center with a box of a certain `size`, that we set here to (3,3,3), which indicates by how many pixels in each direction from the center the code will be looking for a peak of higher intensity.

Cell Volume:

The cell shape is not used for the cell detection itself (which is just searching for local maxima in intensity), but measuring the cell volume will be important to later remove the local peaks detected that do not correspond to an actual cell. This is done by setting these parameters:

```
>>> detectCellShapeParameter = {
>>>     "threshold" : 700,
>>>     "save"      : None,
>>>     "verbose"   : True
>>> }
```

The shape detection is done by a watershed, which will paint the volume of the cell from the detected center outwards. The only parameter to set is the `threshold`. The threshold corresponds to the background intensity, and tells the watershed detection where to stop painting. This value is based on the background subtracted image here. If the `threshold` is set to `None`, then the cell shape detection is bypassed.

Note: Saving the files for the output of this filter as explained above would show all detected objects. The most convenient is to open the folder of images generated with ImageJ, and apply a LUT (Lookup Table) to color each object differently, for instance using the LUT 3-3-2 RGB. The code will write on the image each detected object with a increasing intensity value, so you can make sure this way that adjacent cells are not blending together.

Heat map generation

The voxelization makes it easier to look at the results of the cell detection. The voxelization function will create an image of a specified size (usually the size of the Atlas file) and plot the detected cell centers. To make them visible easily, each point will be expanded into a sphere (or cube, or gaussian) of a given diameter. The intensity value of this sphere is set to 1 by default. So if many points are detected, the overlapping spheres will create a high intensity region.

To set the output for the voxelized heat map file (you would usually not change this):

```
>>> VoxelizationFile = os.path.join(BaseDirectory, 'points_voxelized.tif');
```

And then let's survey the parameters for the voxelization:

```
>>> voxelizeParameter = {
>>>     "method" : 'Spherical',
>>>     "size" : (15,15,15),
>>>     "weights" : None
>>> };
```

The `method` here is set to `Spherical` to paint the points as expanded spheres. the `size` of those spheres is set next, and is given in pixels. Here, we'll choose `(15,15,15)`, but feel free to experiment! Note that the size is in pixels in the final resolution. So for instance, here each sphere will have a diameter of 15 x 25 = 375µm. The `weights` parameter will be changed later, but is set to `None` at this point, meaning that each sphere has an intensity value of 1. The weights allows to change this by integrating the size or intensity of the cells when drawing each sphere.

Config parameters

There are two configuration parameter you should change to match the processing power of your workstation. The first one is the number of processors to be used for the resampling/rotation operations. As this process is not very demanding for the memory, just use the max number of parallel processes your configuration can handle. For instance, we have a 6 cores machine:

```
>>> ResamplingParameter = {"processes": 12};
```

The next settings are for the cell detection. This is limited mainly by the amount of RAM memory you have, but will also change depending on how many filters you use, and their settings. Here is the setting we use on our machine, with 128Gb of RAM for this tutorial:

```
>>> StackProcessingParameter = {
>>>     "processes" : 6,
>>>     "chunkSizeMax" : 100,
>>>     "chunkSizeMin" : 50,
>>>     "chunkOverlap" : 16,
>>>     "chunkOptimization" : True,
>>>     "chunkOptimizationSize" : all,
>>>     "processMethod" : "parallel"
>>> };
```

For the cell detection, the full stack of images will be sliced in smaller chunks to be processed in parallel, and then fused back together. Although it would be tempting to use all the available parallel processing power from your machine, each chunk will take a significant amount of RAM while being analyzed, so the more chunks you process in parallel, the smaller they will be. Also, the chunks will need to overlap, so the smaller they are, the higher the number of overlaps. The size of a chunk is set by both `chunkSizeMax` and `chunkSizeMin`. This is because the code will determine what is the ideal chunk size within that range, depending on the total number of chunks to process. When running the script, keep an eye on the amount of memory used by using the system's activity monitor for instance. If too much data has to go in the swap memory (meaning the RAM is maxed out), reduce the chunk size, or reduce the number of parallel processes.

Run Parameters

You usually would not change anything in this section of the parameter file. This section defines the name of the files generated during the run, and set the parameters for the various operations of resampling and alignment. Of note, you can check these parameters for the alignment:

```
>>> RegistrationAlignmentParameter["affineParameterFile"] = \
>>>                 os.path.join(PathReg, 'Par0000affine.txt');
>>> RegistrationAlignmentParameter["bSplineParameterFile"] = \
>>>                 os.path.join(PathReg, 'Par0000bspline.txt');
```

These point to the two files that will be used as parameter files for the alignment operation with Elastix. If you create new parameter files for the alignment based on your specific need, just make sure you link to the correct parameter file here.

We're all set for the parameter file, now let's explain how the run will proceed.

2.3.2 The Run File

The run file consist of the list of operation to execute to analyze each sample. For our tutorial, it can be found in /ClearMap/Scripts/process_template.py. The run file starts by loading all the parameters from the parameter file:

```
>>> execfile('./ClearMap/Scripts/parameter_file_template.py')
```

Make sure all the modules load correctly. Otherwise, try to install the missing modules. You might get a warning from the compiler the first time you load to parameter file on a new installation. The script shown in the tutorial will execute the following operations:

- *Resampling operations*
- *Alignment operations*
- *Cell detection and thresholding*
- *Point coordinate transformation*
- *Heat map*
- *Table generation*

Resampling operations

The first set of operations to run are the resampling, which will use the parameters set previously. The resampling is executed by the `resampleData` function:

```
>>> resampleData(**CorrectionResamplingParameterCfos);
>>> resampleData(**CorrectionResamplingParameterAutoFluo);
>>> resampleData(**RegistrationResamplingParameter);
```

As you can notice, there are 3 sets of resampling operations. The first two are optional. They create files of intermediate size for the c-Fos and Autofluorescence channels to correct for eventual movements of the sample between the acquisition of those channels. The third resampling only concerns the auto-fluorescence channel and will generate the file used to align to the Atlas reference.

Alignment operations

The alignment is done via Elastix, which is interfaced by ClearMap and is executed by the `alignData` function:

```
>>> resultDirectory  = alignData(**CorrectionAlignmentParameter);
>>> resultDirectory  = alignData(**RegistrationAlignmentParameter);
```

We again do two sets of alignments. The first one here is optional, and is using the files of intermediate resolution generated by the first two resampling operations to re-align the 2 channels in case the sample moved between acquisitions. The second alignment is done onto the Atlas.

Cell detection and thresholding

The cell detection is started by this command:

```
>>> detectCells(**ImageProcessingParameter);
```

This should take a while, between 20 minutes to a few hours depending on your machine, the size of the stack, the filter used. If you're executing the cell detection for testing the parameter, consider restricting the range of the detection as described above with the `cFosFileRange` parameter.

The cell detection will create two files here: `cells-allpoints.npy` and `intensities-allpoints.npy`. These two files will contain all the detected maxima, and need to be filtered, as most local peaks do not correspond to an actual cell.

Note: The coordinates in the files are in the original coordinates of the raw data, and are not transposed yet.

Once the cell detection is done, the points detected have to be filtered to retain only the genuine cells. This is the most important step of the cell detection. First, let's open the files `cells-allpoints.npy` and `intensities-allpoints.npy`:

```
>>> points, intensities = io.readPoints(ImageProcessingParameter["sink"]);
```

Here, we use the function io.readPoints which opens the data related to points coordinates or intensities. In ClearMap scripts, the inputs are usually referred to as `source` and the output as `sink`. `ImageProcessingParameter["sink"]` is defined in the parameter file described above, and is a tuple containing the location of both files for point coordinates and intensities. So here we're opening both files at the same time.

Then, we use the function `thresholdPoints` to threshold the points based on their size and save them to 2 files (coordinates and intensities):

```
>>> points, intensities = thresholdPoints(points, intensities, \
>>>                                threshold = (20, 900), row = (3,3));
>>> io.writePoints(FilteredCellsFile, (points, intensities));
```

The way the `thresholdPoints` function work is by setting the `threshold` parameter as `(lower limit, upper limit)`. If only one value is provided, it assumes this is the lower boundary. `row` defines for the (lower, higher) boundaries which column to use from the intensities array. We presented this array in the overview, but as a reminder:

Row	Description
0	Max intensity of the cell center in the raw data
1	Max intensity of the cell center after the DoG filtering.
2	Max intensity of the cell center after the background subtraction
3	Cell size in voxels after the watershed detection

So here we use column 3 for both boundaries, which is the volume in voxel of each detected cell, which we set at 20.

Note: the size in voxels of each detected cell is defined by the watershed, which will greatly depend on the `threshold` parameter set previously for the cell detection, so if you change this parameter, you'll have to adjust threshold here as well.

Finally, you can now check that the cell detection worked as expected by plotting the result of the detection and thresholding onto the raw data. This should be disabled if you're running the detection on the whole stack, and should only be used while testing. Just delete or comment out (with #) if you don't need it. To run the cell detection check:

```
>>> import iDISCO.Visualization.Plot as plt;
>>> pointSource= os.path.join(BaseDirectory, FilteredCellsFile[0]);
>>> data = plt.overlayPoints(cFosFile, pointSource, pointColor = None, **cFosFileRange);
>>> io.writeData(os.path.join(BaseDirectory, 'cells_check.tif'), data);
```

You might get a warning that a non-standard tiff file is being written. ImageJ, with the plugin BioFormat, can open these files. The resulting `cell_check.tif` file has 2 channels, one with the original data and one with the detected cells shown as a single pixel pointing to the detected center. Browse through the stack to make sure there is only one center detected per cell, that there are no obvious false positive or false negatives.

Point coordinate transformation

The points coordinate are then resampled and transformed onto their final position in the Atlas reference space. Again, this is done twice: once for the correction between both channels, and then for the Atlas alignment.

The first step: correction (optional)

```
>>> points = io.readPoints(CorrectionResamplingPointsParameter["pointSource"]);
>>> points = resamplePoints(**CorrectionResamplingPointsParameter);
>>> points = transformPoints(points, \
>>>     transformDirectory = CorrectionAlignmentParameter["resultDirectory"], \
>>>     indices = False, resultDirectory = None);
>>> CorrectionResamplingPointsInverseParameter["pointSource"] = points;
>>> points = resamplePointsInverse(**CorrectionResamplingPointsInverseParameter);
```

The points are first resampled with the function `resamplePoints` and then their coordinates are transformed based on the results of the alignment done by Elastix with the `transformPoints` function. This function works by interfacing with the Transformix mode of Elastix. The parameter `indices`, here set to `False` indicate to keep the point coordinates with the decimals after the resampling, instead of using the integer coordinates, so prevent loosing information while sequentially re-sampling the point coordinates.

The second step: alignment of the points in the Atlas reference space

```
>>> RegistrationResamplingPointParameter["pointSource"] = points;
>>> points = resamplePoints(**RegistrationResamplingPointParameter);
>>> points = transformPoints(points, \
>>>     transformDirectory = RegistrationAlignmentParameter["resultDirectory"],\
>>>     indices = False, resultDirectory = None);
```

Then writing the final point coordinates:

```
>>> io.writePoints(TransformedCellsFile, points);
```

The points in their Atlas coordinates are written in the file `cells_transformed_to_Atlas.npy` and will be used for the region statistics and to generate the heat maps.

Heat map

First, let's open the files generated previously:

```
>>> points = io.readPoints(TransformedCellsFile)
>>> intensities = io.readPoints(FilteredCellsFile[1])
```

Then, the heat map is generated by the `voxelize` command:

```
>>> vox = voxelize(points, AtlasFile, **voxelizeParameter);
>>> io.writeData(os.path.join(BaseDirectory, 'cells_heatmap.tif'), vox.astype('int32'));
```

The heat map is generated as a 32bit float file, so it may need to be down sampled in ImageJ. The second parameter, `AtlasFile`, is only to pass the size of the Atlas file to the function.

Note: The heat map generated here will be used later for the voxel statistics.

Table generation

The table will show the number of detected points according to the region annotations. It relies on having a labeled image, which is a nrrd or tif file. The function `countPointsInRegions` will use the intensity value of each point as defining the regions:

```
>>> ids, counts = countPointsInRegions(points, labeledImage = AnnotationFile, \
>>>                                    intensities = None, collapse = None);
```

The `AnnotationFile` is set in the parameter file as shown above, and should match the dimensions and orientation of the Atlas file used. The `collapse` function is here set to `None`, but is used if you wish to group adjacent regions into larger regions if you feel that the AnnotationFile has too many subdivisions.

Then, a table of the results is generated:

```
>>> table = numpy.zeros(ids.shape, \
>>>                     dtype=[('id','int64'),('counts','f8'),('name', 'a256')])
>>> table["id"] = ids;
>>> table["counts"] = counts;
>>> table["name"] = labelToName(ids);
>>> io.writeTable(os.path.join(BaseDirectory, 'Annotated_counts.csv'), table);
```

Note: Contrary to the heat maps, the table generated here will not be used for the statistics later, so it is not necessary to execute this in most cases.

This concludes the part of the tutorial covering the settings and run parameters used to analyze the c-Fos dataset in the mouse brain. The next section will cover how to run the statistics on the results obtained after running the `process_template.py` script for all samples of the same experiment.

2.3.3 Analysis Tools

ClearMap provides different ways to analyze the results of the cell detection. In this tutorial, we will examine how to compare the c-Fos cell distribution in 2 groups of 4 brains each analyzed with the scripts shown above. The statistics package covers two sorts of statistical analysis:

- *Voxel statistics*: these are based on the heat map generated as above
- *Regions statistics*: these are using the annotation files to segment the detected cells into anatomical regions
- *Automated region isolation*: to visualize in 3D specific regions from the heat maps

In this example, we will compare a group of control mice injected with a saline solution against mice injected with Haloperidol, which is a psycho-active drug.

Voxel statistics

First, let's import the modules necessary to run the statistics, and set the working directory:

```
>>> import ClearMap.Analysis.Statistics as stat
>>> import iDISCO.Analysis.Tools.MultipleComparisonCorrection as mc
>>> import ClearMap.Analysis.Label as lbl
>>> import ClearMap.IO.IO as io
>>> import numpy, os
>>> baseDirectory = '/home/mtllab/Documents/pharmaco/'
```

Note: Here we set the working directory to the folder containing all the samples for this experiment, while we were working inside the folders of individual samples previously.

Then, let's load the heat map image stacks from each sample into two groups:

```
>>> group1 = ['/home/mtllab/Documents/pharmaco/sample1/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample2/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample3/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample4/cells_heatmap.tif'
>>>           ];
```

```
>>> group2 = ['/home/mtllab/Documents/pharmaco/sample5/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample6/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample7/cells_heatmap.tif',
>>>           '/home/mtllab/Documents/pharmaco/sample8/cells_heatmap.tif'
>>>           ];
>>> g1 = stat.readDataGroup(group1);
>>> g2 = stat.readDataGroup(group2);
```

We can then generate average heat maps for each group, as well as standard deviation maps:

```
>>> g1m = numpy.mean(g1,axis = 0);
>>> io.writeData(os.path.join(baseDirectory, 'saline_mean.mhd'), \
>>>                                          io.sagittalToCoronalData(g1m));
>>> g1s = numpy.std(g1,axis = 0);
>>> io.writeData(os.path.join(baseDirectory, 'saline_std.mhd'),
>>>                                          io.sagittalToCoronalData(g1s));
```

The same thing will be done for group 2. Instead of writing directly the result as an image (here we wrote the file as a raw `.mhd` file), we used the function `io.sagittalToCoronalData` which is a convenient way to reorient the data in coronal plane, which is a more usual way to look at anatomical data (the scans and atlases are in sagittal orientation originally). Open the `.mhd` files in ImageJ. Don't forget that the `.mhd` file is just the header, and that the actual image comes in the companion `.raw` file.

We can now generate the *p* values map:

```
>>> pvals, psign = stat.tTestVoxelization(g1.astype('float'), g2.astype('float'),\
>>>                                       signed = True, pcutoff = 0.05);
>>> pvalscolor = stat.colorPValues(pvals, psign, positive = [0,1], negative = [1,0]);
>>> io.writeData(os.path.join(baseDirectory, 'pvalues.tif'), \
>>>              io.sagittalToCoronalData(pvalscolor.astype('float32')));
```

We used here a cutoff of 5%. The first function `stat.tTestVoxelization` generates the *p* values using a T test with the unequal variance hypothesis set by default. The `stat.colorPValues` function will attribute a color to each pixel of the *p* value map depending on whether the difference of the means between group1 and group2 is significantly positive or negative. You may get a warning that a non-standard tiff file is being written. You may also get warnings from the statistics library during the test calculation, just ignore them.

You have now generated 5 image stacks: 2 average heat maps, 2 standard deviation maps (one for each group) and 1 *p* values map.

Regions statistics

The region statistics use an annotation image file that defines the anatomical regions:

```
>>> ABAlabeledImage = '/home/mtllab/atlas/annotation_25_right.tif';
```

For the region statistics, we will load the point coordinates instead. We'll use the file that contains the coordinates transformed to the Atlas:

```
>>> group1 = ['/home/mtllab/Documents/pharmaco/sample1/cells_transformed_to_Atlas.npy',
>>>           '/home/mtllab/Documents/pharmaco/sample2/cells_transformed_to_Atlas.npy',
>>>           '/home/mtllab/Documents/pharmaco/sample3/cells_transformed_to_Atlas.npy',
>>>           '/home/mtllab/Documents/pharmaco/sample4/cells_transformed_to_Atlas.npy'
>>>           ];
>>> group2 = ['/home/mtllab/Documents/pharmaco/sample5/cells_transformed_to_Atlas.npy',
>>>           '/home/mtllab/Documents/pharmaco/sample6/cells_transformed_to_Atlas.npy',
>>>           '/home/mtllab/Documents/pharmaco/sample7/cells_transformed_to_Atlas.npy',
```

```
>>>             '/home/mtllab/Documents/pharmaco/sample8/cells_transformed_to_Atlas.npy'
>>>             ];
```

Then we'll count the number of cells per region for each group:

ids, counts1 = stat.countPointsGroupInRegions(group1, intensityGroup = None, returnIds = True, labeledImage = ABAlabeledImage, returnCounts = True, collapse = None); counts2 = stat.countPointsGroupInRegions(group2, intensityGroup = None, returnIds = False, labeledImage = ABAlabeledImage, returnCounts = True, collapse = None);

The function `stat.countPointsGroupInRegions` can return 1, 2 or 3 results depending on the parameters set:

Parameter	Description
intensityGroup	You can, on top of a cell coordinate group, create groups containing the intensity data, to include the intensity information in the statistics
returnIds	To set the function to return the region identity found in the labeled file. You only have to do it for one of the two groups
collapse	You can set regions to be fused into larger regions from the table file containing the region names and hierarchy

Then you can calculate the *p* values for the significance of the difference of the mean for each region. Those tests are independent:

```
>>> pvals, psign = stat.tTestPointsInRegions(counts1, counts2, pcutoff = None, \
>>>                                          signed = True, equal_var = False);
```

Optionally, you can also attribute a "q" value to the *p* values, to estimate the false discovery rate, as we're performing a lot of tests:

```
>>> qvals = mc.estimateQValues(pvals);
```

And then you can generate a table containing those results (see the included script file for generating the table).

Automated region isolation

You can also apply the Annotation file to the heat maps instead of the detected cells. This will generate 3D crops of the heat maps to only show specific regions according to the annotation file. Start by importing the modules as shown above, as well as setting the annotation file you wish to use as shown previously. We'll generate then a variable containing the list of all the labels IDs present in the annotation file:

```
>>> label = io.readData(annotationFile);
>>> label = label.astype('int32');
>>> labelids = numpy.unique(label);
```

Then, we'll create a map, `outside` of everything we want to exclude. For example, to exclude every structure of the brain except the cortex, let's do:

```
>>> for l in labelids:
>>>     if not lbl.labelAtLevel(l, 3) == 688:
>>>         outside = numpy.logical_or(outside, label == l);
```

Here, we're using the function `lbl.labelAtLevel` which returns the identity of the parent of a region at a given level of hierarchy. The regions IDs are annotated in a table in ClearMap with for each region of the brain the identity of its parent region containing it. For instance, the layer 4 of the barrel cortex is 8 levels deep in the hierarchy and has the ID 1047, and is contained in the barrel cortex (329) which is itself in the cortex (688). `lbl.labelAtLevel(x, n)` will return the ID of the region at the level `n` containing the region `x`. So if `x` is 5 levels deep in the hierarchy of all

regions, then when `n` is larger than 5, the function will return `x`. Otherwise, if `n` is smaller than 5, it returns the ID of the region of level `n` containing `x`.

So now, the variable outside contains a boolean array for each voxel (True or False) whether that voxel belongs to the cortex or not in the reference space (True means it is not cortex). Let's then open our average heat map:

```
>>> heatmap = io.readData('/home/mtllab/pharmaco/saline_mean.mhd');
```

Then, let's set all the voxels outside of the cortex to 0 and write the result:

```
>>> heatmap[outside] = 0;
>>> io.writeData('/home/mtllab/Documents/pharmaco/saline_cortex.mhd', heatmap);
```

In the same fashion, you can include or exclude regions easily by using the annotation files. You can then open the stacks saved in ImageJ for instance and use one of the 3D viewers to look at the region isolated in 3D.

This concludes this tutorial, which should contain enough information to get you started. The next chapter will show a few examples of the effect of the various filters included in ClearMap for cell detection, and the following chapter is a reference that covers in detail all the included functions of ClearMap.

2.4 ClearMap Image Analysis Tools

Here we introduce the main image processing steps for the detection of nuclear-located signal with examples.

The data is a small region isolated from an iDISCO+ cleared mouse brain immunostained against c-fos. This small stack is included in the ClearMap package in the Test/Data/ImageAnalysis/ folder:

```
>>> import os
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath, \
>>>                         'Test/Data/ImageAnalysis/cfos-substack.tif');
```

2.4.1 Visualizing 3D Images

Large images in 3d are best viewed in specialized software, such as Imaris for 3D rendering or ImageJ to parse the stacks. For the full size data, it is recommended to open the stacks in ImageJ using the « virtual stack » mode.

In ClearMap we provide some basic visualization tools to inspect the 3d data in the module *ClearMap.Visualization.Plot*.

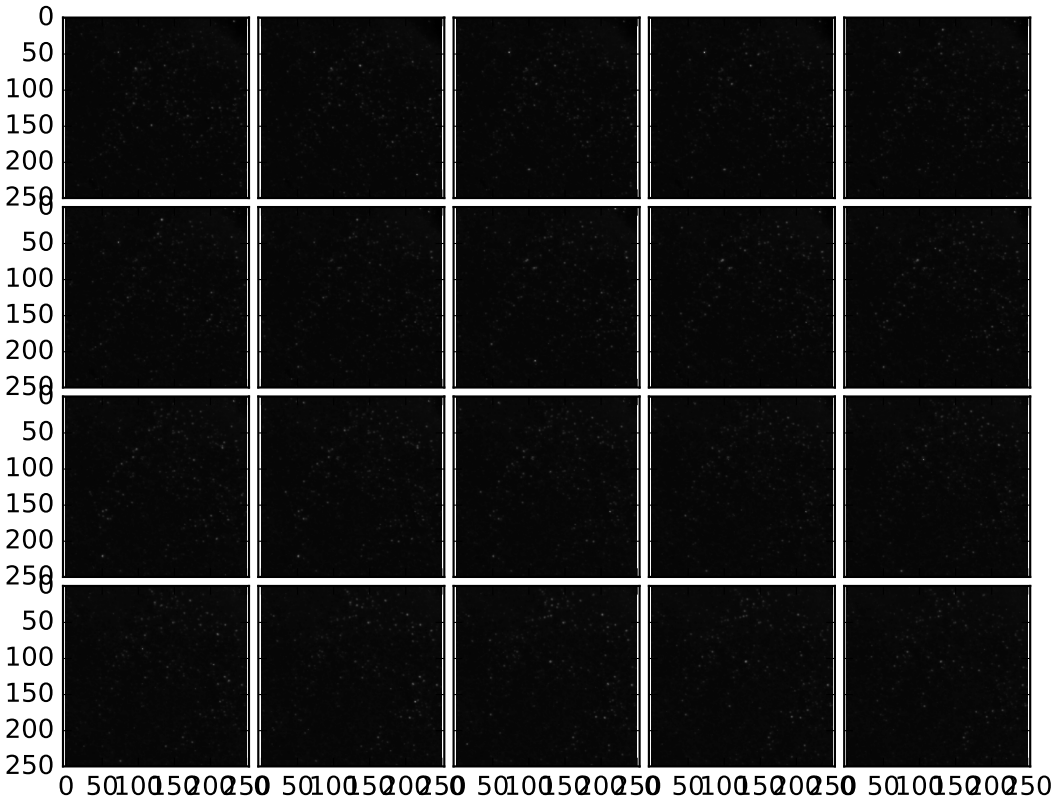
To load them run

```
>>> import ClearMap.Visualization.Plot as plt
```

Tiled Plots

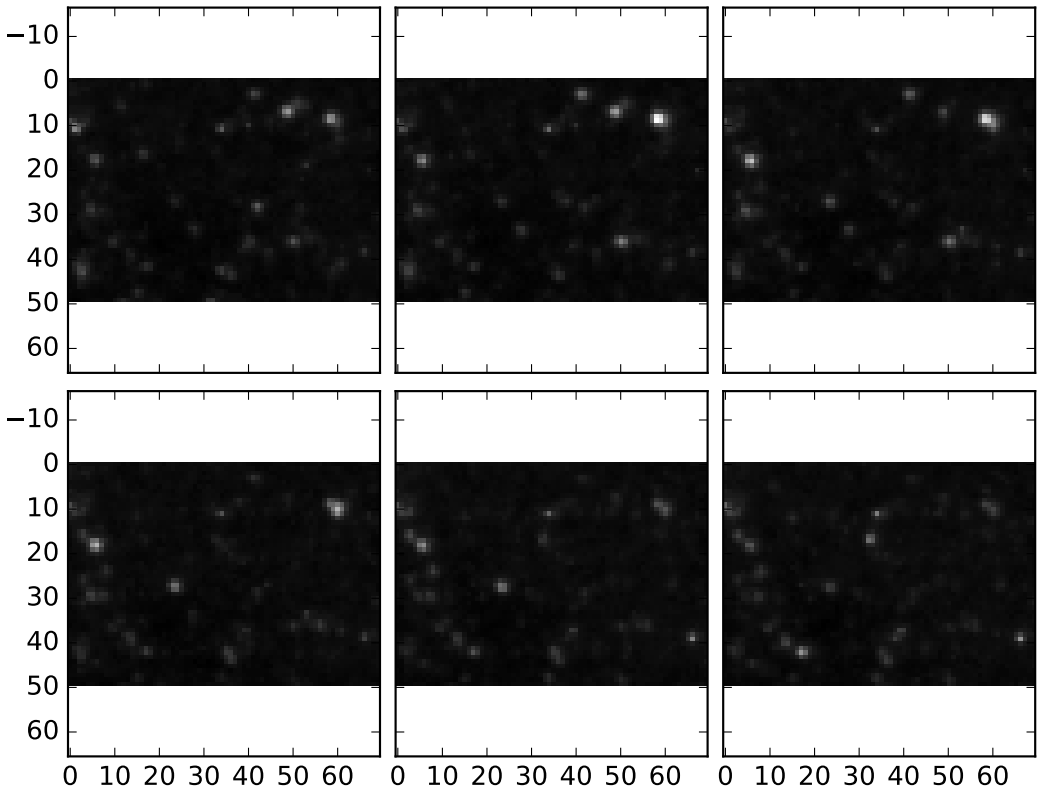
In our experience results of 3d image analysis is inspected most accurately by plotting each horizontal plane in the image in tiles that are coupled for zooming. Intermediate results from all the steps of the SpotDetection can also be written as image stacks and opened with ImageJ.

```
>>> data = io.readData(filename, z = (0,26));
>>> plt.plotTiling(data);
```



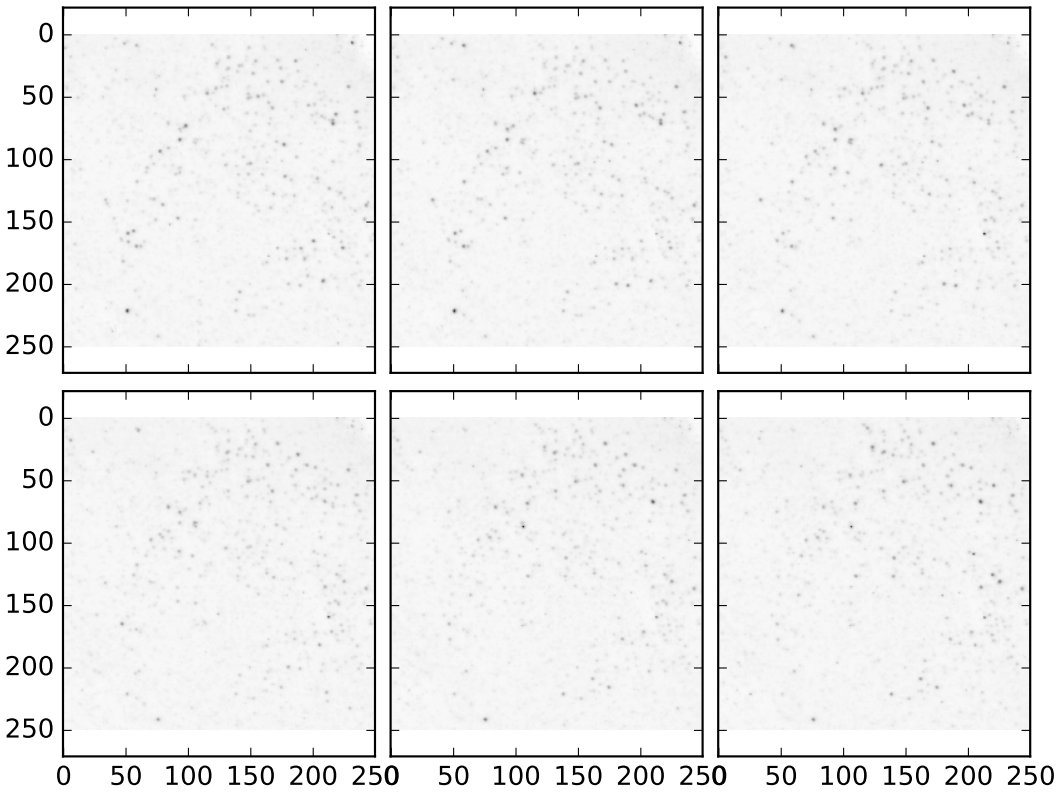
To only plot a particular subregion its possible to specify the x,y,z range.

```
>>> plt.plotTiling(data, x = (0,70), y = (0,50), z = (10,16));
```



Sometimes inverse colors may be better:

```
>>> plt.plotTiling(data, inverse = True,  z = (10,16));
```



2.4.2 Image Statistics

It is useful to gather some information about the image initially. For larger images that don't fit in memory in ClearMap certain statistics can be gathered in parallel via the module *ClearMap.ImageProcessing.ImageStatistics* module.

```
>>> import ClearMap.ImageProcessing.ImageStatistics as stat
>>> print stat.calculateStatistics(filename, method = 'mean')
2305.4042155294119
```

To get more information about the progress use the `verbose` option

```
>>> print stat.calculateStatistics(filename, method = 'mean', verbose = True)
ChunkSize: Estimated chunk size 51 in 1 chunks!
Number of SubStacks: 1
Process 0: processing substack 0/1
Process 0: file          = /ClearMap/Test/Data/ImageAnalysis/cfos-substack.tif
Process 0: segmentation  = <function calculateStatisticsOnStack at 0x7fee9c25dd70>
Process 0: ranges: x,y,z = <built-in function all>,<built-in function all>,(0, 51)
Process 0: Reading data of size (250, 250, 51): elapsed time: 0:00:00
Process 0: Processing substack of size (250, 250, 51): elapsed time: 0:00:00
Total Time Image Statistics: elapsed time: 0:00:00
2305.4042155294119
```

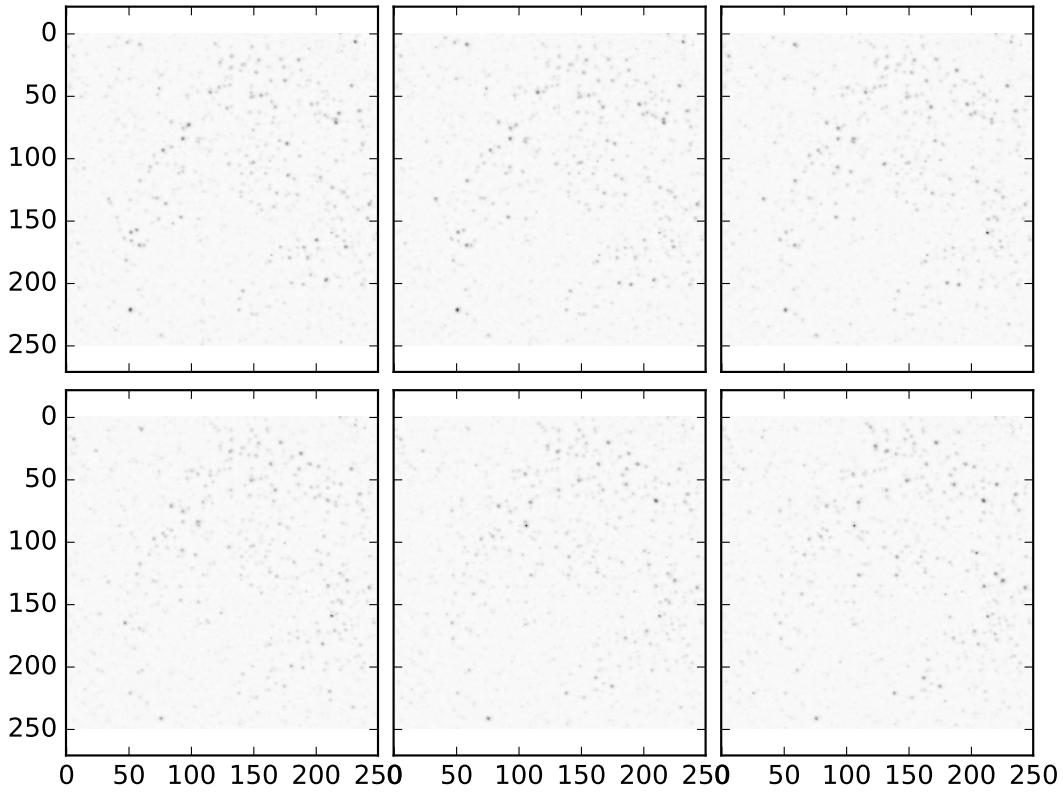
Image statistics can be very helpful for modules, such as Ilastik, requiring a different bit depth than the original 16 or

12 bits files, as it helps to determine how to resample the images to a lower bit depth.

2.4.3 Background Removal

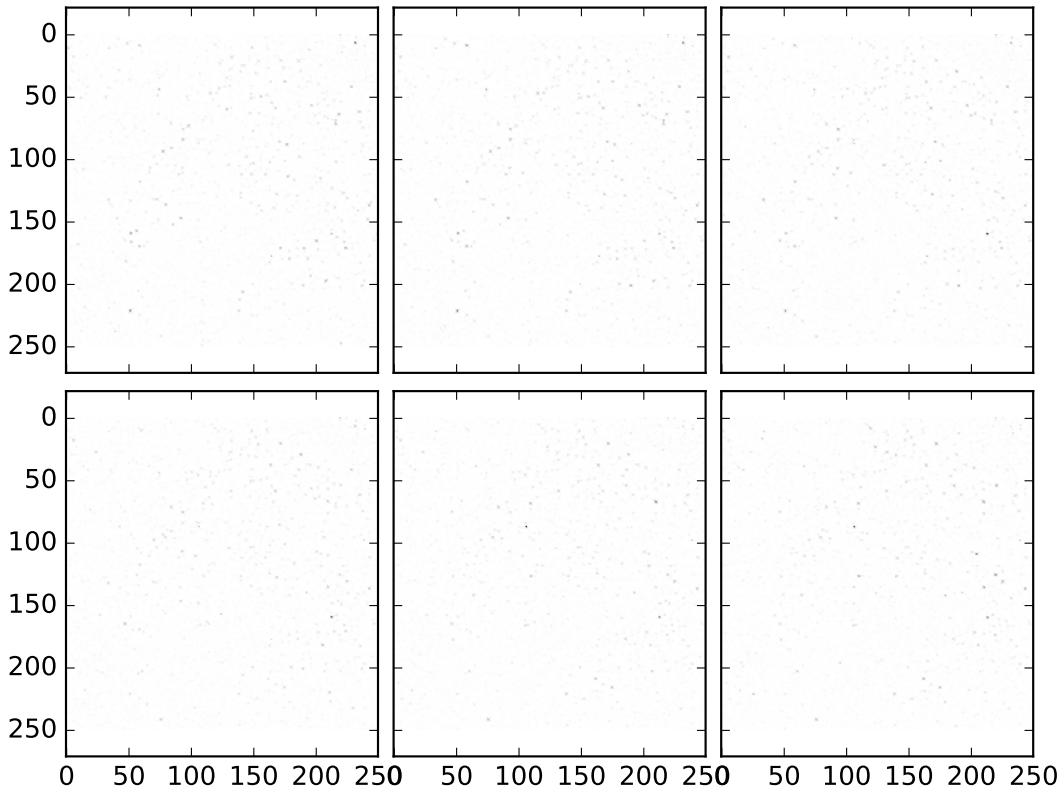
One of the first steps is often to remove background variations. The `ClearMap.Imageprocessing.BackgroundRemoval` module can be used. It performs the background subtraction by morphological opening. The parameter is set as (x,y) where x and y are the diameter of an ellipsoid of a size close to the typical object detected in pixels. The intensity of the signal is greatly reduced after the filtering, but the signal-to-noise ration is increased:

```
>>> import ClearMap.ImageProcessing.BackgroundRemoval as bgr
>>> dataBGR = bgr.removeBackground(data.asype('float'), size=(5,5), verbose = True);
>>> plt.plotTiling(dataBGR, inverse = True, z = (10,16));
```



Note that if the background feature size is chosen too small, this may result in removal of cells:

```
>>> dataBGR = bgr.removeBackground(data.astype('float'), size=(3,3), verbose = True);
>>> plt.plotTiling(dataBGR, inverse = True, z = (10,16));
```

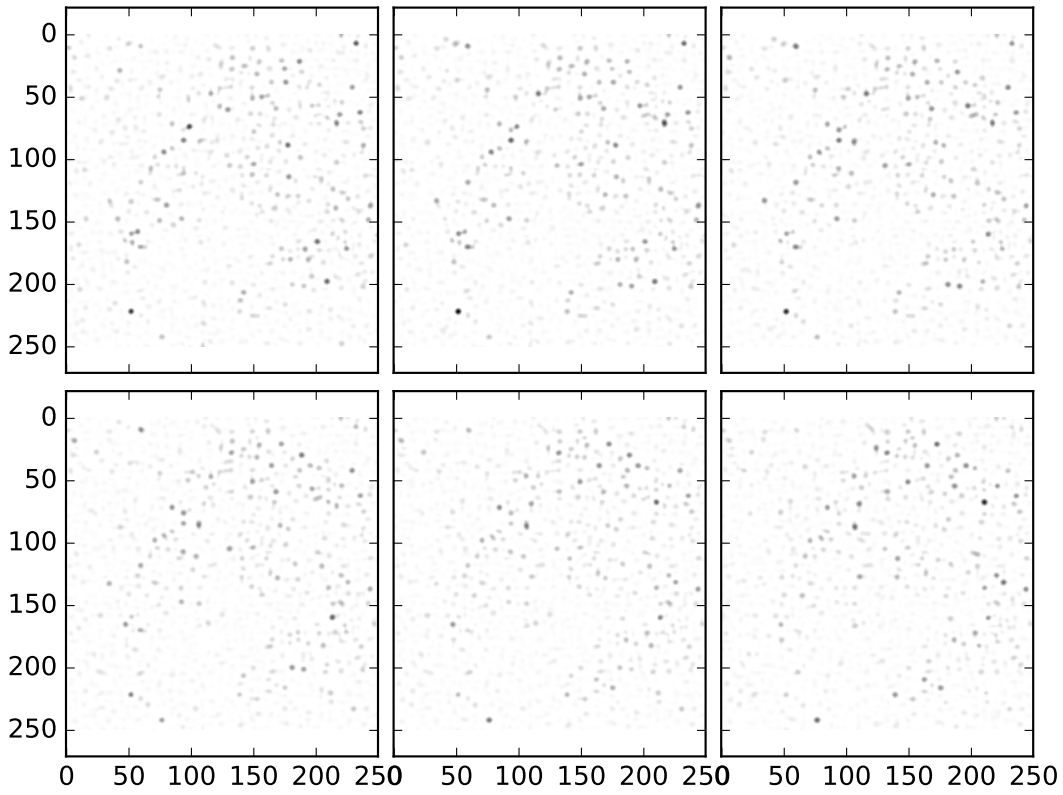


2.4.4 Image Filter

A useful feature is to filter an image. Here the `ClearMap.Imageprocessing.Filter` package can be used.

To detect cells center, the difference of Gaussians filter is a powerful way to increase the contrast between the cells and the background. The size is set as (x,y,z), and usually x, y and z are about the typical size in pixels of the detected object. As after the background subtraction, the intensity of the signal is again reduced after the filtering, but the signal-to-noise ration is dramatically increased:

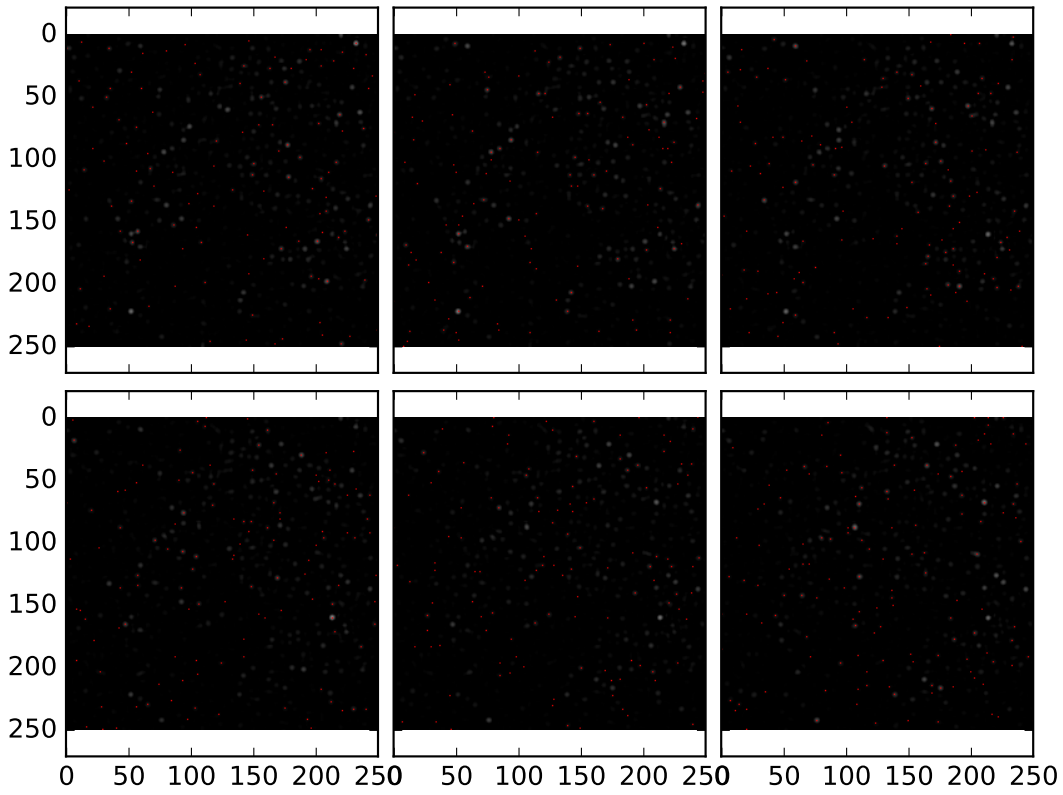
```
>>> from ClearMap.ImageProcessing.Filter.DoGFilter import filterDoG
>>> dataDoG = filterDoG(dataBGR, size=(8,8,4), verbose = True);
>>> plt.plotTiling(dataDoG, inverse = True, z = (10,16));
```



2.4.5 Maxima Detection

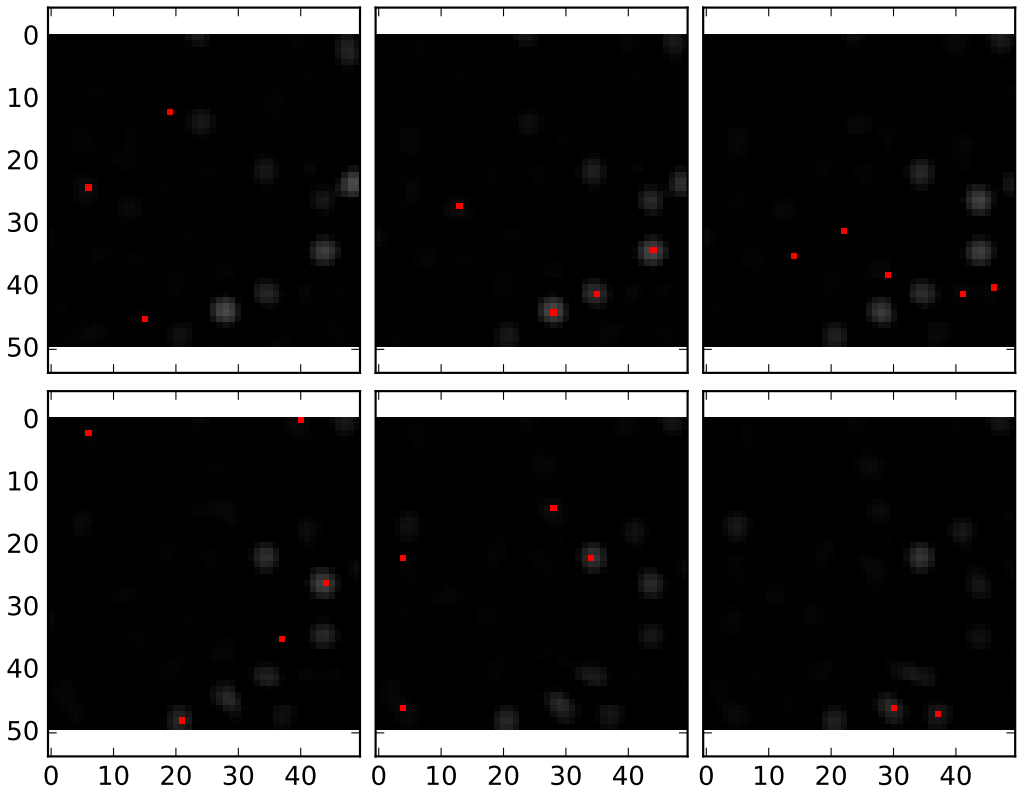
The *ClearMap.ImageProcessing.MaximaDetection* module contains a set of useful functions for the detection of local maxima. A labeled image can be visualized using the *ClearMap.Visualization.Plot.plotOverlayLabel()* routine.

```
>>> from ClearMap.ImageProcessing.MaximaDetection import findExtendedMaxima
>>> dataMax = findExtendedMaxima(dataDoG, hMax = None, verbose = True, threshold = 10);
>>> plt.plotOverlayLabel(  dataDoG / dataDoG.max(), dataMax.astype('int'), z = (10,16))
```



Its easier to see when zoomed in:

```
>>> plt.plotOverlayLabel(dataDoG / dataDoG.max(), dataMax.astype('int'), \
>>>                      z = (10,16), x = (50,100),y = (50,100))
```



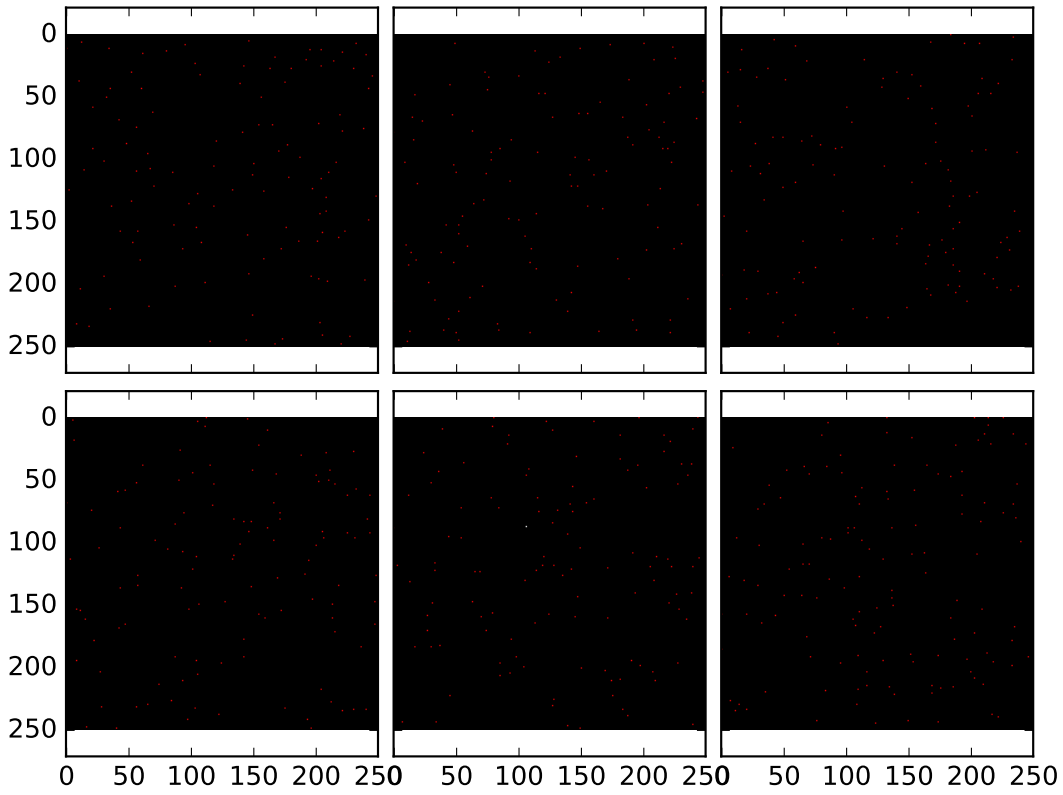
Note that for some cells, a maxima label in this subset might not be visible as maxima are detected in the entire image in 3D and the actual maxima might lie in layers not shown above or below the current planes.

Once the maxima are detected the cell coordinates can be determined:

```
>>> from ClearMap.ImageProcessing.MaximaDetection import findCenterOfMaxima
>>> cells = findCenterOfMaxima(data, dataMax);
>>> print cells.shape
(3670, 3)
```

We can also overlay the cell coordinates in an image:

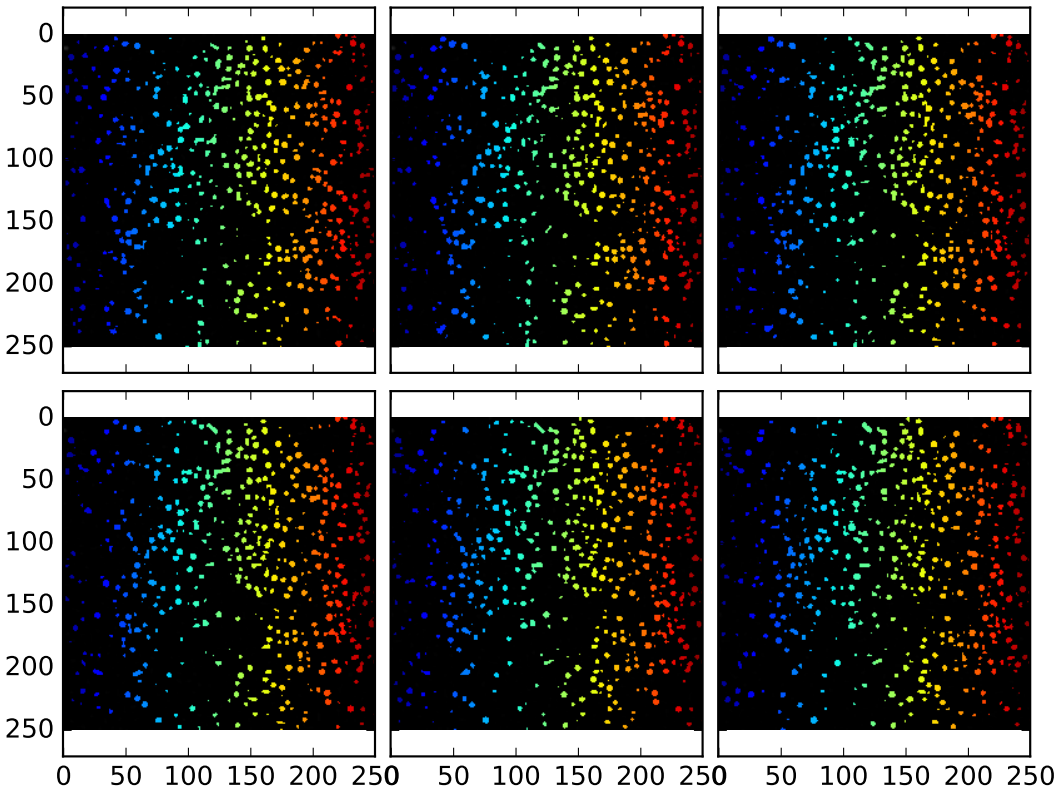
```
>>> plt.plotOverlayPoints(data, cells, z = (10,16))
```



2.4.6 Cell Shape Detection

Finally once the cell centers are detected the `ClearMap.ImageProcessing.CellShapedetection` module can be used to detect the cell shape via a watershed.

```
>>> from ClearMap.ImageProcessing.CellSizeDetection import detectCellShape
>>> dataShape = detectCellShape(dataDoG, cells, threshold = 15);
>>> plt.plotOverlayLabel(dataDoG / dataDoG.max(), dataShape, z = (10,16))
```

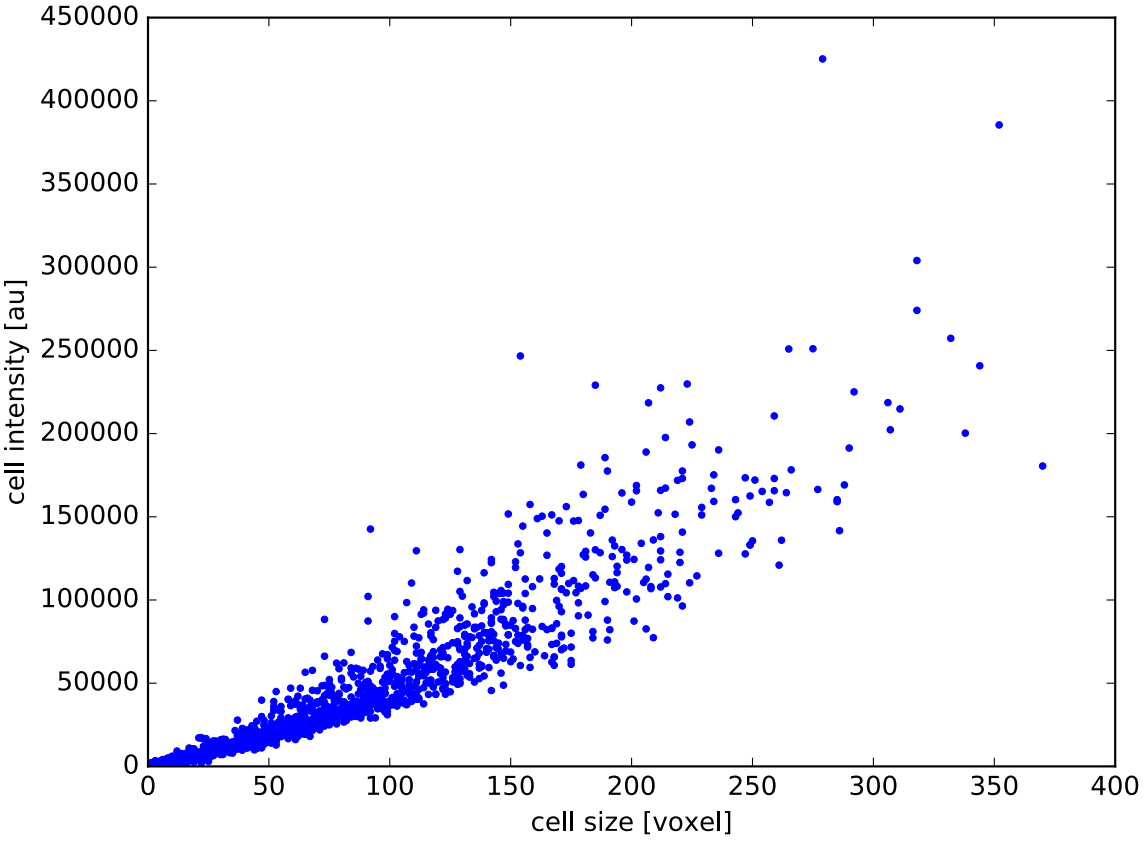


Now we can perform some measurements:

```
>>> from ClearMap.ImageProcessing.CellSizeDetection import findCellSize,\
>>>                                                     findCellIntensity
>>> cellSizes = findCellSize(dataShape, maxLabel = cells.shape[0]);
>>> cellIntensities = findCellIntensity(dataBGR, dataShape, maxLabel = cells.shape[0]);
```

and plot those:

```
>>> import matplotlib.pyplot as mpl
>>> mpl.figure()
>>> mpl.plot(cellSizes, cellIntensities, '.')
>>> mpl.xlabel('cell size [voxel]')
>>> mpl.ylabel('cell intensity [au]')
```


450000
400000
350000
300000
250000
200000
150000
100000
50000
0
cell intensity [au]
0
50
100
150
200
250
300
350
400
cell size [voxel]

CHAPTER

THREE

CLEARMAP FUNCTIONS

3.1 ClearMap package

ClearMap is a python toolbox for the analysis and registration of volumetric data from cleared tissues obtained with Light Sheet microscopy.

The ClearMap code package is structured into four main modules:

- **IO** for reading and writing images and data
- **Alignment** for resampling, reorientation and registration of images onto references
- **Image Processing** for correcting and quantifying the image data
- **Analysis** for the statistical analysis of the data

3.1.1 Subpackages

ClearMap.IO package

This sub-package provides routines to read and write data

Two types of data files are discriminated:

- *Image data*
- *Point data*

The image data are stacks from microscopes obtained by volume imaging, or the results of analysis representing the visualization of the detected objects for instance.

The point data are lists of cell coordinates or measured intensities for instance.

Image data

Images are represented internally as numpy arrays. ClearMap assumes images in arrays are arranged as [x,y], [x,y,z] or [x,y,z,c] where x,y,z correspond to the x,y,z coordinates as when viewed in an image viewer such as ImageJ. The c coordinate is a possible color channel.

Note: Many image libraries read images as [y,x,z] or [y,x] arrays!

The ClearMap toolbox supports a range of (volumetric) image formats:

Format	Descrition	Module
TIF	tif images and stacks	*TIF*
RAW / MHD	raw image files with optional mhd header file	*RAW*
NRRD	nearly raw raster data files	*NRRD*
IMS	imaris image file	*Imaris*
reg exp	folder, file list or file pattern of a stack of 2d images	*FileList*

Note: ClearMap can read the image data from a Bitplane's Imaris, but can't export image data as an Imaris file.

The image format is inferred automatically from the file name extension.

For example to read image data use *readData()*:

```
>>> import os
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath,'Test/Data/Tif/test.tif');
>>> data = io.readData(filename);
>>> print data.shape
(20, 50, 10)
```

To write image data use *writeData()*:

```
>>> import os, numpy
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath,'Test/Data/Tif/test.tif');
>>> data = numpy.random.rand(20,50,10);
>>> data[5:15, 20:45, 2:9] = 0;
>>> data = 20 * data;
>>> data = data.astype('int32');
>>> res = io.writeData(filename, data);
>>> print io.dataSize(res);
(20, 50, 10)
```

Generally, the IO module is designed to work with image sources which can be either files or already loaded numpy arrays. This is important to enable flexible parallel processing, without rewriting the data analysis routines.

For example:

```
>>> import numpy
>>> import ClearMap.IO as io
>>> data = numpy.random.rand(20,50,10);
>>> res = io.writeData(None, data);
>>> print res.shape;
(20, 50, 10)
```

Range parameter can be passed in order to only load sub sets of image data, useful when the images are very large. For example to load a sub-image:

```
>>> import os, numpy
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath,'Test/Data/Tif/test.tif');
>>> res = io.readData(filename, data, x = (0,3), y = (4,6), z = (1,4));
>>> print res.shape;
(3, 2, 3)
```

Point data

ClearMap also supports several data formats for storing arrays of points, such as cell center coordinates or intensities.

Points are assumed to be an array of coordinates where the first array index is the point number and the second the spatial dimension, i.e. [i,d] The spatial dimension can be extended with additional dimensions for intensity ,easires or other properties.

Points can also be given as tuples (coordinate arrray, property array).

ClearMap supports the following files formats for point like data:

Format	Description	Module
CSV	comma separated values in text file	*CSV*
NPY	numpy binary file	*NPY*
VTK	vtk point data file	*VTK*

Note: ClearMap can write points data to a pre-existing Bitplane's Imaris file, but can't import the points from them.

The point file format is inferred automatically from the file name extension.

For example to read point data use *readPoints()*:

```
>>> import os
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath, 'Test/ImageProcessing/points.txt');
>>> points = io.readPoints(filename);
>>> print points.shape
(5, 3)
```

and to write it use *writePoints()*:

```
>>> import os, numpy
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath, 'Test/ImageProcessing/points.txt');
>>> points = numpy.random.rand(5,3);
>>> io.writePoints(filename, points);
```

Summary

- All routines accessing data or data properties accept file name strings or numpy arrays or None
- Numerical arrays represent data and point coordinates as [x,y,z] or [x,y]

ClearMap.IO.IO module

IO interface to read microscope and point data

This is the main module to distribute the reading and writing of individual data formats to the specialized sub-modules.

See *ClearMap.IO* for details.

pointFileExtensions = ['csv', 'txt', 'npy', 'vtk', 'ims']
list of extensions supported as a point data file

pointFileTypes = ['CSV', 'NPY', 'VTK', 'Imaris']
list of point data file types

pointFileExtensionToType = {'txt': 'CSV', 'vtk': 'VTK', 'csv': 'CSV', 'npy': 'NPY', 'ims': 'Imaris'}
map from point file extensions to point file types

dataFileExtensions = ['tif', 'tiff', 'mhd', 'raw', 'ims', 'nrrd']
list of extensions supported as a image data file

dataFileTypes = ['FileList', 'TIF', 'RAW', 'NRRD', 'Imaris']
list of image data file types

dataFileExtensionToType = {'tiff': 'TIF', 'mhd': 'RAW', 'nrrd': 'NRRD', 'raw': 'RAW', 'ims': 'Imaris', 'tif': 'TIF'}
map from image file extensions to image file types

fileExtension (*filename*)
Returns file extension if exists

Parameters **filename** (*str*) – file name

Returns *str* – file extension or None

isFile (*source*)
Checks if filename is a real file, returns false if it is directory or regular expression

Parameters **source** (*str*) – source file name

Returns *bool* – true if source is a real file

isFileExpression (*source*)
Checks if filename is a regular expression denoting a file list

Parameters **source** (*str*) – source file name

Returns *bool* – true if source is regular expression with a digit label

isPointFile (*source*)
Checks if a file is a valid point data file

Parameters **source** (*str*) – source file name

Returns *bool* – true if source is a point data file

isDataFile (*source*)
Checks if a file is a valid image data file

Parameters **source** (*str*) – source file name

Returns *bool* – true if source is an image data file

createDirectory (*filename*)
Creates the directory of the file if it does not exists

Parameters **filename** (*str*) – file name

Returns *str* – directory name

pointFileNameToType (*filename*)
Returns type of a point file

Parameters **filename** (*str*) – file name

Returns *str* – point data type in *pointFileTypes*

dataFileNameToType (*filename*)
Returns type of a image data file

Parameters **filename** (*str*) – file name

Returns *str* – image data type in *dataFileTypes*

dataFileNameToModule (*filename*)
Return the module that handles io for a data file

Parameters **filename** (*str*) – file name

Returns *object* – sub-module that handles a specific data type

pointFileNameToModule (*filename*)
Return the module that handles io for a point file

Parameters **filename** (*str*) – file name

Returns *object* – sub-module that handles a specific point file type

dataSize (*source*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Returns array size of the image data needed when read from file and reduced to specified ranges

Parameters

- **source** (*array or str*) – source data
- **x,y,z** (*tuple or all*) – range specifications, `all` is full range

Returns *tuple* – size of the image data after reading and range reduction

dataZSize (*source*, *z=<built-in function all>*, ***args*)
Returns size of the array in the third dimension, None if 2D data

Parameters

- **source** (*array or str*) – source data
- **z** (*tuple or all*) – z-range specification, `all` is full range

Returns *int* – size of the image data in z after reading and range reduction

toDataRange (*size*, *r=<built-in function all>*)
Converts range r to numeric range (min,max) given the full array size

Parameters

- **size** (*tuple*) – source data size
- **r** (*tuple or all*) – range specification, `all` is full range

Returns *tuple* – absolute range as pair of integers

See also:

toDataSize(), *dataSizeFromDataRange()*

toDataSize (*size*, *r=<built-in function all>*)
Converts full size to actual size given range r

Parameters

- **size** (*tuple*) – data size
- **r** (*tuple or all*) – range specification, `all` is full range

Returns *int* – data size

See also:

toDataRange(), *dataSizeFromDataRange()*

dataSizeFromDataRange (*dataSize*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Converts full data size to actual size given ranges for x,y,z

Parameters

- **dataSize** (*tuple*) – data size
- **x,z,y** (*tuple or all*) – range specifications, `all` is full range

Returns *tuple* – data size as tuple of integers

See also:

toDataRange(), *toDataSize()*

dataToRange(*data*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Reduces data to specified ranges

Parameters

- **data** (*array*) – full data array
- **x,z,y** (*tuple or all*) – range specifications, `all` is full range

Returns *array* – reduced data

See also:

dataSizeFromDataRange()

readData(*source*, ***args*)
Read data from one of the supported formats

Parameters

- **source** (*str, array or None*) – full data array, if numpy array simply reduce its range
- **x,z,y** (*tuple or all*) – range specifications, `all` is full range
- ****args** – further arguments specific to image data format reader

Returns *array* – data as numpy array

See also:

writeData()

writeData(*sink*, *data*, ***args*)
Write data to one of the supported formats

Parameters

- **sink** (*str, array or None*) – the destination for the data, if None the data is returned directly
- **data** (*array or None*) – data to be written
- ****args** – further arguments specific to image data format writer

Returns *array, str or None* – data or file name of the written data

See also:

readData()

copyFile(*source*, *sink*)
Copy a file from source to sink

Parameters

- **source** (*str*) – file name of source
- **sink** (*str*) – file name of sink

Returns *str* – name of the copied file

See also:

copyData(), *convertData()*

copyData (*source*, *sink*)
Copy a data file from source to sink, which can consist of multiple files

Parameters

- **source** (*str*) – file name of source
- **sink** (*str*) – file name of sink

Returns *str* – name of the copied file

See also:

copyFile(), *convertData()*

convertData (*source*, *sink*, ***args*)
Transforms data from source format to sink format

Parameters

- **source** (*str*) – file name of source
- **sink** (*str*) – file name of sink

Returns *str* – name of the copied file

Warning: Not optimized for large image data sets

See also:

copyFile(), *copyData()*

toMultiChannelData (**args*)
Concatenate single channel arrays to one multi channel array

Parameters ***args** (*arrays*) – arrays to be concatenated

Returns *array* – concatenated multi-channel array

pointsToCoordinates (*points*)
Converts a (coordiantes, properties) tuple to the coordinates only

Parameters **points** (*array or tuple*) – point data to be reduced to coordinates

Returns *array* – coordiante data

Notes

Todo: Move this to a class that handles points and their meta data

pointsToProperties (*points*)
Converts a (coordiante, properties) tuple to the properties only

Parameters **points** (*array or tuple*) – point data to be reduced to properties

Returns *array* – property data

Notes

Todo: Move this to a class that handles points and their meta data

pointsToCoordinatesAndProperties (*points*)
Converts points in various formats to a (coordinates, properties) tuple

Parameters **points** (*array or tuple*) – point data to be converted to (coordinates, properties) tuple

Returns *tuple* – (coordinates, properties) tuple

Notes

Todo: Move this to a class that handles points and their meta data

pointsToCoordinatesAndPropertiesFileNames (*filename*, *propertiesPostfix='_intensities'*, ***args*)
Generates a tuple of filenames to store coordinates and properties data separately

Parameters

- **filename** (*str*) – point data file name
- **propertiesPostfix** (*str*) – postfix on file name to indicate property data

Returns *tuple* – (file name, file name for properties)

Notes

Todo: Move this to a class that handles points and their meta data

pointShiftFromRange (*dataSize*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Calculate shift of points given a specific range restriction

Parameters

- **dataSize** (*str*) – data size of the full image
- **x,y,z** (*tuples or all*) – range specifications

Returns *tuple* – shift of points from original origin of data to origin of range reduced data

pointsToRange (*points*, *dataSize=<built-in function all>*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *shift=False*, ***args*)
Restrict points to a specific range

Parameters

- **points** (*array or str*) – point source
- **dataSize** (*str*) – data size of the full image
- **x,y,z** (*tuples or all*) – range specifications
- **shift** (*bool*) – shift points to relative coordinates in the reduced image

Returns *tuple* – points reduced in range and optionally shifted to the range reduced origin

readPoints (*source*, ***args*)
Read a list of points from csv or vtk

Parameters

- **source** (*str, array, tuple or None*) – the data source file
- ****args** – further arguments specific to point data format reader

Returns *array or tuple or None* – point data of source

See also:

writePoints()

writePoints (*sink*, *points*, ***args*)
Write a list of points to csv, vtk or ims files

Parameters

- **sink** (*str or None*) – the destination for the point data
- **points** (*array or tuple or None*) – the point data, optionally as (coordinates, properties) tuple
- ****args** – further arguments specific to point data format writer

Returns *str or array or tuple or None* – point data of source

See also:

readPoints()

writeTable (*filename*, *table*)
Writes a numpy array with column names to a csv file.

Parameters

- **filename** (*str*) – filename to save table to
- **table** (*annotated array*) – table to write to file

Returns *str* – file name

ClearMap.IO.CSV module

Interface to write csv files of cell coordinates / intensities

The module utilizes the csv file writer/reader from numpy.

Example

```
>>> import os, numpy
>>> import ClearMap.IO.CSV as csv
>>> import ClearMap.Settings as settings
>>> filename = os.path.join(settings.ClearMapPath, 'Test/ImageProcessing/points.txt');
>>> points = numpy.random.rand(5,3);
>>> csv.writePoints(filename, points);
>>> points2 = csv.readPoints(filename);
>>> print points2.shape
(5, 3)
```

writePoints (*filename*, *points*, ***args*)
Write point data to csv file

Parameters

- **filename** (*str*) – file name

- **points** (*array*) – point data

Returns *str* – file name

readPoints (*filename*, ***args*)
Read point data to csv file

Parameters

- **filename** (*str*) – file name
- ****args** – arguments for `pointsToRange()`

Returns *str* – file name

test ()
Test CSV module

ClearMap.IO.FileList module

Interface to read/write image stacks saved as a list of files

The filename is given as regular expression as described here.

It is assumd that there is a single digit like regular expression in the file name, i.e. `\d{4}` indicates a placeholder for an integer with four digits using traling 0s and `\d{}` would jus asume an integer with variable size.

For example: `/test\d{4}.tif` or `/test\d{}.tif`

Examples

```
>>> import os, numpy
>>> import ClearMap.Settings as settings
>>> import ClearMap.IO.FileList as fl
>>> filename = os.path.join(settings.ClearMapPath, 'Test/Data/FileList/test\d{4}.tif')
>>> data = numpy.random.rand(20,50,10);
>>> data = data.astype('int32');
>>> fl.writeData(filename, data);
>>> img = fl.readData(filename);
>>> print img.shape
(20, 50, 10)
```

readFileList (*filename*)
Returns list of files that match the regular expression

Parameters **filename** (*str*) – file name as regular expression

Returns *str, list* – path of files, file names that match the regular expression

splitFileExpression (*filename*)
Split the regular expression at the digit place holder

Parameters **filename** (*str*) – file name as regular expression

Returns *tuple* – file header, file extension, digit format

fileExperssionToFileName (*filename*, *z*)
Insert a number into the regular expression

Parameters

- **filename** (*str*) – file name as regular expression

- **z** (*int*) – z slice index

Returns *str* – file name

dataSize (*filename*, ***args*)
Returns size of data stored as a file list

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *tuple* – data size

dataZSize (*filename*, *z=<built-in function all>*, ***args*)
Returns size of data stored as a file list

Parameters

- **filename** (*str*) – file name as regular expression
- **z** (*tuple*) – z data range specification

Returns *int* – z data size

readDataFiles (*filename*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Read data from individual images assuming they are the z slices

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *array* – image data

readData (*filename*, ***args*)
Read image stack from single or multiple images

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *array* – image data

writeData (*filename*, *data*, *startIndex=0*)
Write image stack to single or multiple image files

Parameters

- **filename** (*str*) – file name as regular expression
- **data** (*array*) – image data
- **startIndex** (*int*) – index of first z-slice

Returns *str* – file name as regular expression

copyData (*source*, *sink*)
Copy a data file from source to sink when for entire list of files

Parameters

- **source** (*str*) – file name pattern of source
- **sink** (*str*) – file name pattern of sink

Returns *str* – file name patttern of the copy

test()
Test FileList module

ClearMap.IO.Imaris module

Interface to Imaris Files

Module to read data and write points to Imaris files.

Note: To write points without errors make sure the original file has at least one spot object! You can create a fake point in Imaris, then save the file. The point will be overwritten by ClearMap.

Example

```
>>> import os, numpy
>>> import ClearMap.IO.Imaris as ims
>>> from ClearMap.Settings import ClearMapPath
>>> filename = os.path.join(ClearMapPath,'Test/Data/Imaris/test for spots added spot.ims')
>>> ims.dataSize(filename);
(256, 320, 256)
```

openFile(*filename*, *mode='a'*)
Open Imaris file as hdf5 object

Parameters

- **filename** (*str*) – file name
- **mode** (*str*) – argument to h5py.File

Returns *object* – h5py object

closeFile(*h5file*)
Close Imaris hdf5 file object

Parameters **h5file** (*object*) – h5py opject

Returns *bool* – success

readDataSet(*h5file*, *resolution=0*, *channel=0*, *timepoint=0*)
Open Imaris file and returns hdf5 image data

Parameters

- **h5file** (*object*) – h5py object
- **resolution** (*int*) – resolution level
- **channel** (*int*) – color channel
- **timepoint** (*int*) – time point

Returns *array* – image data

dataSize(*filename*, *resolution=0*, *channel=0*, *timepoint=0*, ***args*)
Read data size of the imaris image data set

Parameters

- **filename** (*str*) – imaris file name

- **resolution** (*int*) – resolution level
- **channel** (*int*) – color channel
- **timepoint** (*int*) – time point

Returns *tuple* – image data size

dataZSize (*filename*, ***args*)
Read z data size of the imaris image data set

Parameters

- **filename** (*str*) – imaris file name
- **resolution** (*int*) – resolution level
- **channel** (*int*) – color channel
- **timepoint** (*int*) – time point

Returns *int* – image z data size

readData (*filename*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *resolution=0*, *channel=0*, *timepoint=0*, ***args*)
Read data from imaris file

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications
- **resolution** (*int*) – resolution level
- **channel** (*int*) – color channel
- **timepoint** (*int*) – time point

Returns *array* – image data

getDataSize (*h5file*)
Get the full data size in pixel from h5py imaris object

Parameters **h5file** (*object*) – h5py object

Returns *tuple* – image data size

getDataExtent (*h5file*)
Get the spatial extent of data from h5py imaris object

Parameters **h5file** (*object*) – h5py object

Returns *array* – spatial extend of image

getScaleAndOffset (*h5file*)
Determine scale and offset to transform pixel to spatial coordinates as used by imaris

Parameters **h5file** (*object*) – h5py object

Returns *tuple* – image scale (length / pixel) and offset (from origin)

transformPointsToImaris (*points*, *scale=(4.0625*, *4.0625*, *3)*, *offset=(0*, *0*, *0)*)
Transform pixel coordinates of cell centers to work in Imaris

Parameters

- **points** (*array*) – point coordinate array
- **scale** (*tuple*) – spatial scale of the image data

- **offset** (*tuple*) – spatial offset of the image data

Returns *array* – scaled points

writePoints (*filename*, *points*, *mode='o'*, *radius=0.5*, *scale=<built-in function all>*, *offset=None*)
Write points to Imaris file

Parameters

- **filename** (*str*) – imaris file name
- **points** (*array*) – point data
- **mode** (*str*) – 'o'= override, 'a'=add
- **radius** (*float*) – size of each point
- **scale** (*tuple or all*) – spatial scaling of points
- **offset** (*tuple or None*) – spatial offset of points

Returns *str* – file name of imaris file

Note: This routine is still experimental !

writeData (*filename*, ***args*)
Write image data to imaris file

Note: Not implemented yet !

readPoints (*filename*)
Read points from imaris file

Note: Not implemented yet !

copyData (*source*, *sink*)
Copy a imaris file from source to sink

Parameters

- **source** (*str*) – file name pattern of source
- **sink** (*str*) – file name pattern of sink

Returns *str* – file name patttern of the copy

test ()
Test Imaris module

ClearMap.IO.NPY module

Interface to write binary files for point like data

The interface is based on the numpy library.

Example

```
>>> import os, numpy
>>> import ClearMap.Settings as settings
>>> import ClearMap.IO.NPY as npy
>>> filename = os.path.join(settings.ClearMapPath, 'Test/Data/NPY/points.npy');
>>> points = npy.readPoints(filename);
>>> print points.shape
(5, 3)
```

writePoints (*filename*, *points*, ***args*)

readPoints (*filename*, ***args*)

test ()
Test NPY module

ClearMap.IO.NRRD module

Interface to NRRD volumetric image data files.

The interface is based on nrrd.py, an all-python (and numpy) implementation for reading and writing nrrd files. See http://teem.sourceforge.net/nrrd/format.html for the specification.

Example

```
>>> import os, numpy
>>> import ClearMap.Settings as settings
>>> import ClearMap.IO.NRRD as nrrd
>>> filename = os.path.join(settings.ClearMapPath, 'Test/Data/Nrrd/test.nrrd');
>>> data = nrrd.readData(filename);
>>> print data.shape
(20, 50, 10)
```

Author

Copyright 2011 Maarten Everts and David Hammond.

Modified to integrate into ClearMap framework by Christoph Kirst, The Rockefeller University, New York City, 2015

exception NrrdError
Bases: exceptions.Exception

Exceptions for Nrrd class.

parse_nrrdvector (*inp*)
Parse a vector from a nrrd header, return a list.

parse_optional_nrrdvector (*inp*)
Parse a vector from a nrrd header that can also be none.

readHeader (*filename*)
Parse the fields in the nrrd header

nrrdfile can be any object which supports the iterator protocol and returns a string each time its next() method is called — file objects and list objects are both suitable. If csvfile is a file object, it must be opened with the 'b' flag on platforms where that makes a difference (e.g. Windows)

```
    >>> readHeader(("NRRD0005", "type: float", "dimension: 3"))
    {'type': 'float', 'dimension': 3, 'keyvaluepairs': {}}
    >>> readHeader(("NRRD0005", "my extra info:=my : colon-separated : values"))
    {'keyvaluepairs': {'my extra info': 'my : colon-separated : values'}}
```

readData (*filename*, ***args*)
Read nrrd file image data

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *array* – image data

writeData (*filename*, *data*, *options={}*, *separateHeader=False*)
Write data to nrrd file

Parameters

- **filename** (*str*) – file name as regular expression
- **data** (*array*) – image data
- **options** (*dict*) – options dictionary
- **separateHeader** (*bool*) – write a separate header file

Returns *str* – nrrd output file name

To sample date use *options['spacings'] = [s1, s2, s3]* for 3d data with sampling deltas *s1*, *s2*, and *s3* in each dimension.

dataSize (*filename*, ***args*)
Read data size from nrrd image

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *tuple* – data size

dataZSize (*filename*, *z=<built-in function all>*, ***args*)
Read data z size from nrrd image

Parameters

- **filename** (*str*) – file name as regular expression
- **z** (*tuple*) – z data range specification

Returns *int* – z data size

copyData (*source*, *sink*)
Copy an nrrd file from source to sink

Parameters

- **source** (*str*) – file name pattern of source
- **sink** (*str*) – file name pattern of sink

Returns *str* – file name of the copy

Notes

Todo: dealt with nrdh header files!

test()
Test NRRD IO module

ClearMap.IO.RAW module

Simple Interface to read RAW/MHD files e.g. created by elastix

Todo: read subsets efficiently

Example

```
>>> import os, numpy
>>> from ClearMap.Settings import ClearMapPath
>>> import ClearMap.IO.RAW as raw
>>> filename = os.path.join(ClearMapPath, 'Test/Data/Raw/test.mhd')
>>> raw.dataSize(filename);
(20, 50, 10)
```

Author

Christoph Kirst, The Rockefeller University, New York City, 2015

dataSize(*filename*, ***args*)
Read data size from raw/mhd image

Parameters

- **filename** (*str*) – imaris file name
- **x,y,z** (*tuple or all*) – range specifications

Returns *int* – raw image data size

dataZSize(*filename*, *z=<built-in function all>*, ***args*)
Read z data size from raw/mhd image

Parameters

- **filename** (*str*) – imaris file name
- **z** (*tuple or all*) – range specification

Returns *int* – raw image z data size

readData(*filename*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*)
Read data from raw/mhd image

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *array* – image data

writeHeader(*filename*, *meta_dict*)
Write raw header mhd file

Parameters

- **filename** (*str*) – file name of header
- **meta_dict** (*dict*) – dictionary of meta data

Returns *str* – header file name

writeRawData (*filename*, *data*)
Write the data into a raw format file.

Parameters

- **filename** (*str*) – file name as regular expression
- **data** (*array*) – data to write to raw file

Returns *str* – file name of raw file

writeData (*filename*, *data*, ***args*)
Write data into to raw/mhd file pair

Parameters

- **filename** (*str*) – file name as regular expression
- **data** (*array*) – data to write to raw file

Returns *str* – file name of mhd file

copyData (*source*, *sink*)
Copy a raw/mhd file pair from source to sink

Parameters

- **source** (*str*) – file name of source
- **sink** (*str*) – file name of sink

Returns *str* – file name of the copy

test ()
Test RAW io module

ClearMap.IO.TIF module

Interface to tif image files and stacks.

The interface makes use of the tifffile library.

Example

```
>>> import os, numpy
>>> import ClearMap.IO.TIF as tif
>>> from ClearMap.Settings import ClearMapPath
>>> filename = os.path.join(ClearMapPath,'Test/Data/Tif/composite.tif')
>>> data = tif.readData(filename);
>>> print data.shape
(50, 100, 30, 4)
```

dataSize (*filename*, ***args*)
Returns size of data in tif file

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *tuple* – data size

dataZSize (*filename*, *z=<built-in function all>*, ***args*)
Returns z size of data in tif file

Parameters

- **filename** (*str*) – file name as regular expression
- **z** (*tuple*) – z data range specification

Returns *int* – z data size

readData (*filename*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, ***args*)
Read data from a single tif image or stack

Parameters

- **filename** (*str*) – file name as regular expression
- **x,y,z** (*tuple*) – data range specifications

Returns *array* – image data

writeData (*filename*, *data*)
Write image data to tif file

Parameters

- **filename** (*str*) – file name
- **data** (*array*) – image data

Returns *str* – tif file name

copyData (*source*, *sink*)
Copy a data file from source to sink

Parameters

- **source** (*str*) – file name pattern of source
- **sink** (*str*) – file name pattern of sink

Returns *str* – file name of the copy

test ()
Test TIF module

ClearMap.IO.VTK module

Interface to write points to VTK files

Notes

- points are assumed to be in [x,y,z] coordinates as standard in ClearMap
- reading of points not supported at the moment!

writePoints (*filename*, *points*, *labelImage=None*)
Write point data to vtk file

Parameters

- **filename** (*str*) – file name
- **points** (*array*) – point data
- **labelImage** (*str, array or None*) – optional label image to determine point label

Returns *str* – file name

readPoints (*filename*, ***args*)
Read points form vtk file

Notes

•Not implmented yet !

ClearMap.Alignment package

This sub-package provides an interface to alignment tools in order to register cleared samples to atlases or reference samples.

Supported functionality:

- resampling and reorientation of large volumetric images in the *Resampling* module.
- registering volumetric data onto references via Elastix in the *Elastix* module.

Main routines for resampling are: *resampleData()* and *resamplePoints()*.

Main routines for elastix registration are: *alignData()*, *transformData()* and *transformPoints()*.

ClearMap.Alignment.Elastix module

Interface to Elastix for alignment of volumetric data

The elastix documentation can be found here.

In essence, a transformation $T(x)$ is sought so that for a fixed image $F(x)$ and a moving image $M(x)$:

$$F(x) = M(T(x)) \tag{3.1}$$

Once the map T is estimated via elastix, transformix maps an image $I(x)$ from the moving image frame to the fixed image frame, i.e.:

$$I(x) \rightarrow I(T(x)) \tag{3.2}$$

To register an image onto a reference image, the fixed image is typically choosed to be the image to be registered, while the moving image is the reference image. In this way an object identified in the data at position x is mapped via transformix as:

$$x \rightarrow T(x) \tag{3.3}$$

Summary

- elastix finds a transformation $T : \mathrm{fixedimage} \rightarrow \mathrm{movingimage}$
- the fixed image is image to be registered
- the moving image is typically the reference image
- the result folder may contain an image (mhd file) that is $T^{-1}(\mathrm{moving})$, i.e. has the size of the fixed image
- transformix applied to data gives $T^{-1}(\mathrm{data})$!
- transformix applied to points gives $T(\mathrm{points})$!
- point arrays are assumed to be in (x,y,z) coordinates consistent with (x,y,z) array represenation of images in ClearMap

Main routines are: *alignData()*, *transformData()* and *transformPoints()*.

See also:

Elastix documentation *Resampling*

ElastixBinary = '/home/ckirst/programs/elastix/bin/elastix'
str: the elastix executable

Notes

•setup in *initializeElastix()*

ElastixLib = '/home/ckirst/programs/elastix/lib'
str: path to the elastix library

Notes

•setup in *initializeElastix()*

TransformixBinary = '/home/ckirst/programs/elastix/bin/transformix'
str: the transformix executable

Notes

•setup in *initializeElastix()*

Initialized = True
bool: True if the elastixs binarys and paths are setup

Notes

•setup in *initializeElastix()*

printSettings()
Prints the current elastix configuration

See also:

ElastixBinary, *ElastixLib*, *TransformixBinary*, *Initialized*

setElastixLibraryPath(*path=None*)
Add elastix library path to the LD_LIBRARY_PATH variable in linux

Parameters **path** (*str or None*) – path to elastix root directory if None *ClearMap.Settings.ElastixPath* is used.

initializeElastix(*path=None*)
Initialize all paths and binaries of elastix

Parameters

- **path** (*str or None*) – path to elastix root directory, if None
- **:const:'ClearMap.Settings.ElastixPath' is used.**

See also:

ElastixBinary, *ElastixLib*, *TransformixBinary*, *Initialized*, *setElastixLibraryPath()*

checkElastixInitialized()
Checks if elastix is initialized

Returns *bool* – True if elastix paths are set.

getTransformParameterFile(*resultdir*)
Finds and returns the transformation parameter file generated by elastix

Notes

In case of multiple transformation parameter files the top level file is returned

Parameters **resultdir** (*str*) – path to directory of elastix results

Returns *str* – file name of the first transformation parameter file

setPathTransformParameterFiles(*resultdir*)
Replaces relative with abolsute path in the parameter files in the result directory

Notes

When elastix is not run in the directory of the transformation files the aboslute path needs to be given in each transformation file to point to the subsequent transformation files. This is done via this routine

Parameters **resultdir** (*str*) – path to directory of elastix results

parseElastixOutputPoints(*filename*, *indices=True*)
Parses the output points from the output file of transformix

Parameters

- **filename** (*str*) – file name of the transformix output file
- **indices** (*bool*) – if True return pixel indices otherwise float coordinates

Returns **points** (*array*) – the transformed coordinates

getTransformFileSizeAndSpacing(*transformfile*)
Parse the image size and spacing from a transformation parameter file

Parameters **transformfile** (*str*) – File name of the transformix parameter file.

Returns *(float, float)* – the image size and spacing

getResultDataFile(*resultdir*)
Returns the mhd result file in a result directory

Parameters **resultdir** (*str*) – Path to elastix result directory.

Returns *str* – The mhd file in the result directory.

setTransformFileSizeAndSpacing(*transformfile*, *size*, *spacing*)
Replaces size and scale in the transformation parameter file

Parameters

- **transformfile** (*str*) – transformation parameter file
- **size** (*tuple*) – the new image size
- **spacing** (*tuplr*) – the new image spacing

rescaleSizeAndSpacing(*size*, *spacing*, *scale*)
Rescales the size and spacing

Parameters

- **size** (*tuple*) – image size
- **spacing** (*tuple*) – image spacing
- **scale** (*tuple*) – the scale factor

Returns *(tuple, tuple)* – new size and spacing

alignData(*fixedImage*, *movingImage*, *affineParameterFile*, *bSplineParameterFile=None*, *resultDirectory=None*)
Align images using elastix, estimates a transformation T : fixed image $\rightarrow$ moving image.

Parameters

- **fixedImage** (*str*) – image source of the fixed image (typically the reference image)
- **movingImage** (*str*) – image source of the moving image (typically the image to be registered)
- **affineParameterFile** (*str or None*) – elastix parameter file for the primary affine transformation
- **bSplineParameterFile** (*str or None*) – elastix parameter file for the secondary non-linear transformation
- **resultDirectory** (*str or None*) – elastic result directory

Returns *str* – path to elastix result directory

transformData(*source*, *sink=[]*, *transformParameterFile=None*, *transformDirectory=None*, *resultDirectory=None*)
Transform a raw data set to reference using the elastix alignment results

If the map determined by elastix is $T\mathrm{fixed} \rightarrow \mathrm{moving}$, transformix on data works as $T^{-1}(\mathrm{data})$.

Parameters

- **source** (*str or array*) – image source to be transformed
- **sink** (*str, [] or None*) – image sink to save transformed image to. if [] return the default name of the data file generated by transformix.
- **transformParameterFile** (*str or None*) – parameter file for the primary transformation, if None, the file is determined from the transformDirectory.

- **transformDirectory** (*str or None*) – result directory of elastix alignment, if None the transformParameterFile has to be given.
- **resultDirectory** (*str or None*) – the directorty for the transformix results

Returns *array or str* – array or file name of the transformed data

deformationField(*sink=[]*, *transformParameterFile=None*, *transformDirectory=None*, *resultDirectory=None*)

Create the deformation field T(x) - x

The map determined by elastix is $T\text{fixed} \rightarrow \text{moving}$

Parameters

- **sink** (*str, [] or None*) – image sink to save the transformation field; if [] return the default name of the data file generated by transformix.
- **transformParameterFile** (*str or None*) – parameter file for the primary transformation, if None, the file is determined from the transformDirectory.
- **transformDirectory** (*str or None*) – result directory of elastix alignment, if None the transformParameterFile has to be given.
- **resultDirectory** (*str or None*) – the directorty for the transformix results

Returns *array or str* – array or file name of the transformed data

deformationDistance(*deformationField*, *sink=None*, *scale=None*)

Compute the distance field from a deformation vector field

Parameters

- **deformationField** (*str or array*) – source of the deformation field determined by `deformationField()`
- **sink** (*str or None*) – image sink to save the deformation field to
- **scale** (*tuple or None*) – scale factor for each dimension, if None = (1,1,1)

Returns *array or str* – array or file name of the transformed data

writePoints(*filename*, *points*, *indices=True*)

Write points as elastix/transformix point file

Parameters

- **filename** (*str*) – file name of the elastix point file.
- **points** (*array or str*) – source of the points.
- **indices** (*bool*) – write as pixel indices or physical coordiantes

Returns *str* – file name of the elastix point file

transformPoints(*source*, *sink=None*, *transformParameterFile=None*, *transformDirectory=None*, *indices=True*, *resultDirectory=None*, *tmpFile=None*)

Transform coordinates math:*x* via elastix estimated transformation to $T(x)$

Note the transformation is from the fixed image coorindates to the moving image coordiantes.

Parameters

- **source** (*str*) – source of the points
- **sink** (*str or None*) – sink for transformed points

- **transformParameterFile** (*str or None*) – parameter file for the primary transformation, if None, the file is determined from the transformDirectory.
- **transformDirectory** (*str or None*) – result directory of elastix alignment, if None the transformParameterFile has to be given.
- **indices** (*bool*) – if True use points as pixel coordinates otherwise spatial coordinates.
- **resultDirectory** (*str or None*) – elastic result directory
- **tmpFile** (*str or None*) – file name for the elastix point file.

Returns *array or str* – array or file name of transformed points

test()
Test Elastix module

ClearMap.Alignment.Resampling module

The *Resampling* module provides methods to resample and reorient volumetric and point data.

Resampling the data is usually necessary as the first step to match the resolution and orientation of the reference object.

Main routines for resampling are: *resampleData()* and *resamplePoints()*.

Image Representation and Size The module assumes that images in arrays are arranged as

- [x,y] or
- [x,y,z]

where x,y,z correspond to the x,y,z coordinates as displayed in e.g. ImageJ. For example an image of size (512,512) stored in an array `img` will have:

```
>>> img.shape
(512,512)
```

Points are assumed to be given as x,y,z coordinates

Parameters such as *resolution* or *dataSize* are assumed to be given in (x,y) or (x,y,z) format, e.g.

```
>>> dataSize = (512,512)
```

Orientation The *orientation* parameter is a tuple of d numbers from 1 to d that specifies the permutation of the axes, a minus sign infront of a numbeer indicates inversion of that axes. For exmaple

```
>>> orientation=(2,-1)
```

indicates that x and y should be exchanged and the new y axes should be reversed.

Generally a re-orientation is composed of first a permutation of the axes and then inverting the indicated axes.

A *permutation* is an orientation without signs and with numbers from 0 to d-1.

Examples

```
>>> import os
>>> import ClearMap.IO as io
>>> from ClearMap.Settings import ClearMapPath
>>> from ClearMap.Alignment.Resampling import resampleData
>>> filename = os.path.join(ClearMapPath,'Test/Data/OME/16-17-27_0_8X-s3-20HF_UltraII_C00_xyz-Table Z
>>> print io.dataSize(filename)
(2160, 2560, 21)
>>> data = resampleData(filename, sink = None, resolutionSource = (1,1,1), orientation = (1,2,3), res
>>> print data.shape
(216, 256, 10)
```

fixOrientation (*orientation*)

Convert orientation to standard format number sequence

Parameters **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – orientation sequence

See also:

Orientation

inverseOrientation (*orientation*)

Returns the inverse permuation of the permutation orientation taking axis inversions into account.

Parameters **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – orientation sequence

See also:

Orientation

orientationToPermuation (*orientation*)

Extracts the permuation from an orientation.

Parameters **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – premutation sequence

See also:

Orientation

orientResolution (*resolution*, *orientation*)

Permutes a resolution tuple according to the given orientation.

Parameters

- **resolution** (*tuple*) – resolution specification
- **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – oriented resolution sequence

See also:

Orientation

orientResolutionInverse (*resolution*, *orientation*)

Permutes a resolution tuple according to the inverse of a given orientation.

Parameters

- **resolution** (*tuple*) – resolution specification
- **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – oriented resolution sequence

See also:

Orientation

orientDataSize(*dataSize*, *orientation*)
Permutes a data size tuple according to the given orientation.

Parameters

- **dataSize** (*tuple*) – resolution specification
- **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – oriented dataSize sequence

See also:

Orientation

orientDataSizeInverse(*dataSize*, *orientation*)
Permutes a dataSize tuple according to the inverse of a given orientation.

Parameters

- **dataSize** (*tuple*) – dataSize specification
- **orientation** (*tuple or str*) – orientation specification

Returns *tuple* – oriented dataSize sequence

See also:

Orientation

resampleDataSize(*dataSizeSource*, *dataSizeSink=None*, *resolutionSource=None*, *resolutionSink=None*, *orientation=None*)
Calculate scaling factors and data sizes for resampling.

Parameters

- **dataSizeSource** (*tuple*) – data size of the original image
- **dataSizeSink** (*tuple or None*) – data size of the resmapled image
- **resolutionSource** (*tuple or None*) – resolution of the source image
- **resolutionSink** (*tuple or None*) – resolution of the sink image
- **orientation** (*tuple or str*) – re-orientation specification

Returns *tuple* – data size of the source tuple: data size of the sink tuple: resolution of source tuple: resolution of sink

See also:

Orientation

fixInterpolation(*interpolation*)
Converts interpolation given as string to cv2 interpolation object

Parameters **interpolation** (*str or object*) – interpolation string or cv2 object

Returns *object* – cv2 interpolation type

resampleXY(*source*, *dataSizeSink*, *sink=None*, *interpolation='linear'*, *out=<open file '<stdout>'*, *mode 'w'>*, *verbose=True*)
Resample a 2d image slice

This routine is used for resampling a large stack in parallel in xy or xz direction.

Parameters

- **source** (*str or array*) – 2d image source
- **dataSizeSink** (*tuple*) – size of the resmapled image
- **sink** (*str or None*) – location for the resmapled image
- **interpolation** (*str*) – interpolation method to use: 'linear' or None (nearest pixel)
- **out** (*stdout*) – where to write progress information
- **vebose** (*bool*) – write progress info if true

Returns *array or str* – resampled data or file name

resampleData(*source*, *sink=None*, *orientation=None*, *dataSizeSink=None*, *resolutionSource=(4.0625, 4.0625, 3)*, *resolutionSink=(25, 25, 25)*, *processingDirectory=None*, *processes=1*, *cleanup=True*, *verbose=True*, *interpolation='linear'*, ***args*)

Resample data of source in resolution and orientation

Parameters

- **source** (*str or array*) – image to be resampled
- **sink** (*str or None*) – destination of resampled image
- **orientation** (*tuple*) – orientation specified by permuation and change in sign of (1,2,3)
- **dataSizeSink** (*tuple or None*) – target size of the resampled image
- **resolutionSource** (*tuple*) – resolution of the source image (in length per pixel)
- **resolutionSink** (*tuple*) – resolution of the resampled image (in length per pixel)
- **processingDirectory** (*str or None*) – directory in which to perform resmapling in parallel, None a temporary directry will be created
- **processes** (*int*) – number of processes to use for parallel resampling
- **cleanup** (*bool*) – remove temporary files
- **verbose** (*bool*) – display progress information
- **interpolation** (*str*) – method to use for interpolating to the resmapled image

Returns *(array or str)* – data or file name of resampled image

Notes

•resolutions are assumed to be given for the axes of the intrinsic orientation of the data and reference as when viewed by matplotlib or ImageJ

•orientation: permuation of 1,2,3 with potential sign, indicating which axes map onto the reference axes, a negative sign indicates reversal of that particular axes

•only a minimal set of information to detremine the resampling parameter has to be given, e.g. dataSize-Source and dataSizeSink

resampleDataInverse(*sink*, *source=None*, *dataSizeSource=None*, *orientation=None*, *resolutionSource=(4.0625, 4.0625, 3)*, *resolutionSink=(25, 25, 25)*, *processingDirectory=None*, *processes=1*, *cleanup=True*, *verbose=True*, *interpolation='linear'*, ***args*)

Resample data inversely to `resampleData()` routine

Parameters

- **sink** (*str or None*) – image to be inversly resampled (=sink in `resampleData()`)
- **source** (*str or array*) – destination for inversly resmapled image (=source in `resampleData()`)
- **dataSizeSource** (*tuple or None*) – target size of the resampled image
- **orientation** (*tuple*) – orientation specified by permuation and change in sign of (1,2,3)
- **resolutionSource** (*tuple*) – resolution of the source image (in length per pixel)
- **resolutionSink** (*tuple*) – resolution of the resampled image (in length per pixel)
- **processingDirectory** (*str or None*) – directory in which to perform resmapling in parallel, None a temporary directry will be created
- **processes** (*int*) – number of processes to use for parallel resampling
- **cleanup** (*bool*) – remove temporary files
- **verbose** (*bool*) – display progress information
- **interpolation** (*str*) – method to use for interpolating to the resmapled image

Returns *(array or str)* – data or file name of resampled image

Notes

•resolutions are assumed to be given for the axes of the intrinsic orientation of the data and reference as when viewed by matplotlib or ImageJ

•orientation: permuation of 1,2,3 with potential sign, indicating which axes map onto the reference axes, a negative sign indicates reversal of that particular axes

•only a minimal set of information to detremine the resampling parameter has to be given, e.g. dataSizeSource and dataSizeSink

resamplePoints(*pointSource*, *pointSink=None*, *dataSizeSource=None*, *dataSizeSink=None*, *orientation=None*, *resolutionSource=(4.0625, 4.0625, 3)*, *resolutionSink=(25, 25, 25)*, ***args*)

Resample Points to map from original data to the coordinates of the resampled image

The resampling of points here corresponds to he resampling of an image in `resampleData()`

Parameters

- **pointSource** (*str or array*) – image to be resampled
- **pointSink** (*str or None*) – destination of resampled image
- **orientation** (*tuple*) – orientation specified by permuation and change in sign of (1,2,3)
- **dataSizeSource** (*str, tuple or None*) – size of the data source
- **dataSizeSink** (*str, tuple or None*) – target size of the resampled image
- **resolutionSource** (*tuple*) – resolution of the source image (in length per pixel)

- **resolutionSink** (*tuple*) – resolution of the resampled image (in length per pixel)

Returns *(array or str)* – data or file name of resampled points

Notes

•resolutions are assumed to be given for the axes of the intrinsic orientation of the data and reference as when viewed by matplotlib or ImageJ

•orientation: permuation of 1,2,3 with potential sign, indicating which axes map onto the reference axes, a negative sign indicates reversal of that particular axes

•only a minimal set of information to detremine the resampling parameter has to be given, e.g. dataSize-Source and dataSizeSink

resamplePointsInverse(*pointSource*, *pointSink=None*, *dataSizeSource=None*, *dataSizeSink=None*, *orientation=None*, *resolutionSource=(4.0625*, *4.0625*, *3)*, *resolutionSink=(25*, *25*, *25)*, ***args*)

Resample points from the coordinates of the resampled image to the original data

The resampling of points here corresponds to he resampling of an image in `resampleDataInverse()`

Parameters

- **pointSource** (*str or array*) – image to be resampled
- **pointSink** (*str or None*) – destination of resampled image
- **orientation** (*tuple*) – orientation specified by permuation and change in sign of (1,2,3)
- **dataSizeSource** (*str, tuple or None*) – size of the data source
- **dataSizeSink** (*str, tuple or None*) – target size of the resampled image
- **resolutionSource** (*tuple*) – resolution of the source image (in length per pixel)
- **resolutionSink** (*tuple*) – resolution of the resampled image (in length per pixel)

Returns *(array or str)* – data or file name of inversely resampled points

Notes

•resolutions are assumed to be given for the axes of the intrinsic orientation of the data and reference as when viewed by matplotlib or ImageJ

•orientation: permuation of 1,2,3 with potential sign, indicating which axes map onto the reference axes, a negative sign indicates reversal of that particular axes

•only a minimal set of information to detremine the resampling parameter has to be given, e.g. dataSize-Source and dataSizeSink

sagittalToCoronalData(*source*, *sink=None*)

Change from saggital to coronal orientation

Parameters

- **source** (*str or array*) – source data to be reoriented
- **sink** (*str or None*) – destination for reoriented image

Returns *str or array* – reoriented data

ClearMap.ImageProcessing package

This sub-package provides routines for volumetric image processing in parallel

This part of the *ClearMap* toolbox is designed in a modular way to allow for fast and flexible extension and addition of specific image processing algorithms.

The toolbox part consists of two parts:

- *Volumetric Image Processing*
- *Parallel Image Processing*

Volumetric Image Processing

The image processing routines provided in the standard package are listed below

Module	Descrition
`BackgroundRemoval`	Background estimation and removal via morphological opening
`IlluminationCorrection`	Correction of vignetting and other illumination errors
`GreyReconstruction`	Reconstruction of images
`Filter`	Filtering of images via a large set of filter kernels
`MaximaDetection`	Detection of maxima and h-max transforms
`SpotDetection`	Detection of local peaks / spots / nuclei
`CellDetection`	Detection of cells
`CellSizeDetection`	Detection of cell shapes and volumes via e.g. watershed
`IlastikClassification`	Classification of voxels via interface to Ilastik

While some of these modules provide basic volumetric image processing functionality some routines combine those functions to provide predefined higher level cell detection, cell size and intensity measurements.

The higher level routines are optimized for iDISCO+ cleared mouse brain samples stained for c-Fos expression. Other data sets might require a redesign of these higher level functions.

Parallel Image Processing

For large volumetric image data sets from e.g. light sheet microscopy parallel processing is essential to speed up calculations.

In this toolbox the image processing is parallelized via splitting a volumetric image stack into several sub-stacks, typically in z-direction. Because most of the image processing steps are non-local sub-stacks are created with overlaps and the results rejoined accordingly to minimize boundary effects.

Parallel processing is handled via the `StackProcessing` module.

External Packages

The `ImageProcessing` module makes use of external image processing packages including:

- Open Cv2
- Scipy
- Scikit-Image
- Ilastik

Routines form these packages were freely chosen to optimize for speed and memory consumption

ClearMap.ImageProcessing.StackProcessing module

Process a image stack in parallel or sequentially

In this toolbox image processing is parallized via splitting a volumetric image stack into several sub-stacks, typically in z-direction. As most of the image processig steps are non-local sub-stacks are created with overlaps and the results rejoined accordingly to minimize boundary effects.

Parallel processing is handled via this module.

Sub-Stacks The parallel processing module creates a dictionary with information on the sub-stack as follows:

Key	Description
`stackId`	id of the sub-stack
`nStacks`	total number of sub-stacks
`source`	source file/folder/pattern of the stack
`x`, `y`, `z`	the range of the sub-stack with in the full image
`zCenters`	tuple of the centers of the overlaps
`zCenterIndices`	tuple of the original indices of the centers of the overlaps
`zSubStackCenterIndices`	tuple of the indices of the sub-stack that correspond to the overlap centers

For exmaple the *`writeSubStack()`* routine makes uses of this information to write out only the sub-parts of the image that is will contribute to the final total image.

`printSubStackInfo`(*subStack*, *out=<open file '<stdout>', mode 'w'>*)
Print information about the sub-stack

Parameters

- **subStack** (*dict*) – the sub-stack info
- **out** (*object*) – the object to write the information to

`writeSubStack`(*filename*, *img*, *subStack=None*)
Write the non-redundant part of a sub-stack to disk

The routine is used to write out images when porcessed in parallel. It assumes that the filename is a patterned file name.

Parameters

- **filename** (*str or None*) – file name pattern as described in `FileList`, if None return as array
- **img** (*array*) – image data of sub-stack
- **subStack** (*dict or None*) – sub-stack information, if None write entire image see *Sub-Stacks*

Returns *str or array* – the file name pattern or image

`joinPoints`(*results*, *subStacks=None*, *shiftPoints=True*, ***args*)
Joins a list of points obtained from processing a stack in chunks

Parameters

- **results** (*list*) – list of point results from the individual sub-processes
- **subStacks** (*list or None*) – list of all sub-stack information, see *Sub-Stacks*
- **shiftPoints** (*bool*) – if True shift points to refer to origin of the image stack considered when range specification is given. If False, absolute position in entire image stack.

Returns *tuple* – joined points, joined intensities

calculateChunkSize(*size*, *processes=2*, *chunkSizeMax=100*, *chunkSizeMin=30*, *chunkOverlap=15*, *chunkOptimization=True*, *chunkOptimizationSize=<built-in function all>*, *verbose=True*)

Calculates the chunksize and other info for parallel processing

The sub stack information is described in *Sub-Stacks*

Parameters

- **processes** (*int*) – number of parallel processes
- **chunkSizeMax** (*int*) – maximal size of a sub-stack
- **chunkSizeMin** (*int*) – minial size of a sub-stack
- **chunkOverlap** (*int*) – minimal sub-stack overlap
- **chunkOptimization** (*bool*) – optimize chunck sizes to best fit number of processes
- **chunkOptimizationSize** (*bool or all*) – if True only decrease the chunk size when optimizing
- **verbose** (*bool*) – print information on sub-stack generation

Returns *tuple* – number of chunks, z-ranges of each chunk, z-centers in overlap regions

calculateSubStacks(*source*, *z=<built-in function all>*, *x=<built-in function all>*, *y=<built-in function all>*, ***args*)

Calculates the chunksize and other info for parallel processing and returns a list of sub-stack objects

The sub-stack information is described in *Sub-Stacks*

Parameters

- **source** (*str*) – image source
- **x,y,z** (*tuple or all*) – range specifications
- **processes** (*int*) – number of parallel processes
- **chunkSizeMax** (*int*) – maximal size of a sub-stack
- **chunkSizeMin** (*int*) – minial size of a sub-stack
- **chunkOverlap** (*int*) – minimal sub-stack overlap
- **chunkOptimization** (*bool*) – optimize chunck sizes to best fit number of processes
- **chunkOptimizationSize** (*bool or all*) – if True only decrease the chunk size when optimizing
- **verbose** (*bool*) – print information on sub-stack generation

Returns *list* – list of sub-stack objects

noProcessing(*img*, ***parameter*)

Perform no image processing at all and return original image

Used as the default functon in `parallelProcessStack()` and `sequentiallyProcessStack()`.

Parameters **img** (*array*) – imag

Returns *(array)* – the original image

parallelProcessStack(*source*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *sink=None*, *processes=2*, *chunkSizeMax=100*, *chunkSizeMin=30*, *chunkOverlap=15*, *chunkOptimization=True*, *chunkOptimizationSize=<built-in function all>*, *function=<function noProcessing>*, *join=<function joinPoints>*, *verbose=False*, ***parameter*)

Parallel process a image stack

Main routine that distributes image processing on paralllel processes.

Parameters

- **source** (*str*) – image source
- **x,y,z** (*tuple or all*) – range specifications
- **sink** (*str or None*) – destination for the result
- **processes** (*int*) – number of parallel processes
- **chunkSizeMax** (*int*) – maximal size of a sub-stack
- **chunkSizeMin** (*int*) – minial size of a sub-stack
- **chunkOverlap** (*int*) – minimal sub-stack overlap
- **chunkOptimization** (*bool*) – optimize chunck sizes to best fit number of processes
- **chunkOptimizationSize** (*bool or all*) – if True only decrease the chunk size when optimizing
- **function** (*function*) – the main image processing script
- **join** (*function*) – the fuction to join the results from the image processing script
- **verbose** (*bool*) – print information on sub-stack generation

Returns *str or array* – results of the image processing

sequentiallyProcessStack(*source*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *sink=None*, *chunkSizeMax=100*, *chunkSizeMin=30*, *chunkOverlap=15*, *function=<function noProcessing>*, *join=<function joinPoints>*, *verbose=False*, ***parameter*)

Sequential image processing on a stack

Main routine that sequentially processes a large image on sub-stacks.

Parameters

- **source** (*str*) – image source
- **x,y,z** (*tuple or all*) – range specifications
- **sink** (*str or None*) – destination for the result
- **processes** (*int*) – number of parallel processes
- **chunkSizeMax** (*int*) – maximal size of a sub-stack
- **chunkSizeMin** (*int*) – minial size of a sub-stack
- **chunkOverlap** (*int*) – minimal sub-stack overlap
- **chunkOptimization** (*bool*) – optimize chunck sizes to best fit number of processes
- **chunkOptimizationSize** (*bool or all*) – if True only decrease the chunk size when optimizing
- **function** (*function*) – the main image processing script

- **join** (*function*) – the fuction to join the results from the image processing script
- **verbose** (*bool*) – print information on sub-stack generation

Returns *str or array* – results of the image processing

ClearMap.ImageProcessing.CellDetection module

Cell Detection Module

This is the main routine to run the individual routines to detect cells om volumetric image data.

ClearMap supports two predefined image processing pipelines which will extend in the future:

Method	Description	Reference
"SpotDetection"	uses predefined spot detection pipline	*detectCells()*
"Ilastik"	uses predefined pipline with cell classification via Ilastik	classifyCells()
function	a user defined function	NA

Example

```
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> from ClearMap.ImageProcessing.CellDetection import detectCells;
>>> fn = os.path.join(settings.ClearMapPath, 'Test/Data/Synthetic/test_iDISCO_\d{3}.tif');
>>> parameter = {"filterDoGParameter" : {"size": (5,5,5)}, "findExtendedMaximaParameter" : {"thresho
>>> img = io.readData(fn);
>>> img = img.astype('int16'); # converting data to smaller integer types can be more memory efficie
>>> res = detectCells(img, parameter);
>>> print res[0].shape
```

See also:

ImageProcessing

detectCells (*source*, *sink=None*, *method='SpotDetection'*, *processMethod=<built-in function all>*, *verbose=False*, ***parameter*)

Detect cells in data

This is a main script to start running the cell detection.

Parameters

- **source** (*str or array*) – Image source
- **sink** (*str or None*) – destination for the results
- **method** (*str or function*) –

Method	Description
"SpotDetection"	uses predefined spot detection pipline
"Ilastik"	uses predefined pipline with cell classification via Ilastik
function	a user defined function

- **processMethod** (*str or all*) – 'sequential' or 'parallel'. if all its choosen automatically
- **verbose** (*bool*) – print info
- ****parameter** (*dict*) – parameter for the image procesing sub-routines

Returns:

ClearMap.ImageProcessing.CellSizeDetection module

Cell shape and size detection routines

The cell shape detection is based on a seeded and masked watershed.

detectCellShape(*img*, *peaks*, *detectCellShapeParameter=None*, *threshold=None*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find cell shapes as labeled image

Parameters

- **img** (*array*) – image data
- **peaks** (*array*) – point data of cell centers / seeds
- **detectCellShape** (*dict*) –

Name	Type	Descritption
threshold	(float or None)	threshold to determine mask, pixel below this are background if None no mask is generated
save	(tuple)	size of the box on which to perform the *method*
verbose	(bool or int)	print / plot information about this step

- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – labeled image where each label indicates a cell

findCellSize(*imglabel*, *findCelSizeParameter=None*, *maxLabel=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find cell size given cell shapes as labled image

Parameters

- **imglabel** (*array or str*) – labeled image, where each cell has its own label
- **findCelSizeParameter** (*dict*) –

Name	Type	Descritption
maxLabel	(int or None)	maximal label to include, if None determine automatically
verbose	(bool or int)	print / plot information about this step

- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – measured intensities

findCellIntensity(*img*, *imglabel*, *findCellIntensityParameter=None*, *maxLabel=None*, *method='Sum'*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find integrated cell intensity given cell shapes as labled image

Parameters

- **img** (*array or str*) – image data
- **imglabel** (*array or str*) – labeled image, where each cell has its own label
- **findCellIntensityParameter** (*dict*) –

Name	Type	Descritption
maxLabel	(int or None)	maximal label to include, if None determine automatically
method	(str)	method to use for measurment: 'Sum', 'Mean', 'Max', 'Min'
verbose	(bool or int)	print / plot information about this step

- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – measured intensities

Subpackages

ClearMap.ImageProcessing.Filter package This sub-package provides various volumetric filter kernels and structure elments

A set of linear filters can be applied to the data using `LinearFilter`.

Because its utility for cell detection the difference of Gaussians filter is implemented directly in `DoGFilter`.

The fitler kernels defined in `FilterKernel` can be used in combination with the `Convolution` module.

Structured elements defined in `StructureElements` can be used in combination with various morphological operations, e.g. used in the :mod:~ClearMap.ImageProcessing.BackgroundRemoval' module.

ClearMap.ImageProcessing.Filter.LinearFilter module Linear filter module

filterLinear(*img*, *filterLinearParameter=None*, *ftype=None*, *size=None*, *sigma=None*, *sigma2=None*, *save=None*, *subStack=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Applies a linear filter to the image

Parameters

- **img** (*array*) – image data
- **filterLinearParameter** (*dict*) –

Name	Type	Descritption
ftype	(str or None)	the type of the filter, see *Filter Type* if None do ot perform any fitlering
size	(tuple or None)	size for the filter if None, do not perform filtering
sigma	(tuple or None)	std of outer Guassian, if None autmatically determined from size
sigma2	(tuple or None)	std of inner Guassian, if None autmatically determined from size
save	(str or None)	file name to save result of this operation if None dont save to file
verbose	(bool or int)	print progress information

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – filtered image

Note: Converts image to float32 type if filter is active!

ClearMap.ImageProcessing.Filter.DoGFilter module DoG filter module

filterDoG(*img*, *filterDoGParameter=None*, *size=None*, *sigma=None*, *sigma2=None*, *save=None*, *verbose=None*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)
Difference of Gaussians (DoG) filter step

Parameters

- **img** (*array*) – image data
- **filterDoGParameter** (*dict*) –

Name	Type	Descritption
size	(tuple or None)	size for the DoG filter if None, do not correct for any background
sigma	(tuple or None)	std of outer Guassian, if None autmatically determined from size
sigma2	(tuple or None)	std of inner Guassian, if None autmatically determined from size
save	(str or None)	file name to save result of this operation if None dont save to file
verbose	(bool or int)	print progress information

- **subStack** (*dict or None*) – sub-stack information
- **out** (*object*) – object to write progress info to

Returns *array* – DoG filtered image

ClearMap.ImageProcessing.Filter.Convolution module Convolve volumetric data with a 3d kernel, optimized for memory / float32 use

Based on scipy.signal routines.

Author

Original code from scipy.signal.

Modified by Chirstoph Kirst to optimize memory and sped and integration into ClearMap. The Rockefeller University, New York City, 2015

convolve(*x*, *k*, *mode='same'*)
Convolve array with kernel using float32 / complex64, optimized for memory consumption and speed

Parameters

- **x** (*array*) – data to be convolved
- **k** (*array*) – filter kernel

Returns *array* – convolution

ClearMap.ImageProcessing.Filter.FilterKernel module Implementation of various volumetric filter kernels

Filter Type Filter types defined by the `ftype` key include:

Type	Descrition
`mean`	uniform averaging filter
`gaussian`	Gaussian filter
`log`	Laplacian of Gaussian filter (LoG)
`dog`	Difference of Gaussians filter (DoG)
`sphere`	Sphere filter
`disk`	Disk filter

filterKernel (*ftype='Gaussian'*, *size=(5, 5)*, *sigma=None*, *radius=None*, *sigma2=None*)
Creates a filter kernel of a special type

Parameters

- **ftype** (*str*) – filter type, see *Filter Type*
- **size** (*array or tuple*) – size of the filter kernel
- **sigma** (*tuple or float*) – std for the first gaussian (if present)
- **radius** (*tuple or float*) – radius of the kernel (if applicable)
- **sigma2** (*tuple or float*) – std of a second gaussian (if present)

Returns *array* – structure element

filterKernel2D (*ftype='Gaussian'*, *size=(5, 5)*, *sigma=None*, *sigma2=None*, *radius=None*)
Creates a 2d filter kernel of a special type

Parameters

- **ftype** (*str*) – filter type, see *Filter Type*
- **size** (*array or tuple*) – size of the filter kernel
- **sigma** (*tuple or float*) – std for the first gaussian (if present)
- **radius** (*tuple or float*) – radius of the kernel (if applicable)
- **sigma2** (*tuple or float*) – std of a second gaussian (if present)

Returns *array* – structure element

filterKernel3D (*ftype='Gaussian'*, *size=(5, 5, 5)*, *sigma=None*, *sigma2=None*, *radius=None*)
Creates a 3d filter kernel of a special type

Parameters

- **ftype** (*str*) – filter type, see *Filter Type*
- **size** (*array or tuple*) – size of the filter kernel
- **sigma** (*tuple or float*) – std for the first gaussian (if present)
- **radius** (*tuple or float*) – radius of the kernel (if applicable)
- **sigma2** (*tuple or float*) – std of a second gaussian (if present)

Returns *array* – structure element

test()
Test FilterKernel module

ClearMap.ImageProcessing.Filter.StructureElement module Routines to generate various structure elements

Structured elements defined by the `setype` key include:

Structure Element Types

Type	Descrition
`sphere`	Sphere structure
`disk`	Disk structure

Note: To be extended!

`structureElement`(*setype='Disk'*, *sesize=(3, 3)*)
Creates specific 2d and 3d structuring elements

Parameters

- **setype** (*str*) – structure element type, see *Structure Element Types*
- **sesize** (*array or tuple*) – size of the structure element

Returns *array* – structure element

`structureElementOffsets`(*sesize*)
Calculates offsets for a structural element given its size

Parameters **sesize** (*array or tuple*) – size of the structure element

Returns *array* – offsets to center taking care of even/odd ummber of elements

`structureElement2D`(*setype='Disk'*, *sesize=(3, 3)*)
Creates specific 2d structuring elements

Parameters

- **setype** (*str*) – structure element type, see *Structure Element Types*
- **sesize** (*array or tuple*) – size of the structure element

Returns *array* – structure element

`structureElement3D`(*setype='Disk'*, *sesize=(3, 3, 3)*)
Creates specific 3d structuring elements

Parameters

- **setype** (*str*) – structure element type, see *Structure Element Types*
- **sesize** (*array or tuple*) – size of the structure element

Returns *array* – structure element

ClearMap.ImageProcessing.IlluminationCorrection module

Illumination correction toolbox.

The module provides a function to correct illumination/vignetting systematic variations in intensity.

The intensity image $I(x)$ given a flat field $F(x)$ and a background $B(x)$ the image is corrected to $C(x)$ as:

The module also has functionality to create flat field corections from measured intensity changes in a single direction, useful e.g. for lightsheet images, see e.g. `flatfieldLineFromRegression()`.

References

Fundamentals of Light Microscopy and Electronic Imaging, p. 421

DefaultFlatFieldLineFile = '/home/mtllab/Programs/ClearMap/idisco/ClearMap/Data/lightsheet_flatfield_correction.cs
Default file of points along the illumination changing line for the flat field correction

See also:

correctIllumination()

correctIllumination(*img*, *correctIlluminationParameter=None*, *flatfield=None*, *background=None*, *scaling=None*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)
Correct illumination variations

The intensity image $I(x)$ given a flat field $F(x)$ and a background $B(x)$ the image is corrected to $C(x)$ as:

If the background is not given $B(x) = 0$.

The correction is done slice by slice assuming the data was collected with a light sheet microscope.

The image is finally optionally scaled.

Parameters

- **img** (*array*) – image data
- **findCenterOfMaximaParameter** (*dict*) –

Name	Type	Descritption
flatfield	(str, None or array)	flat field intensities, if None d onot correct image for illumination, if True the
background	(str, None or array)	background image as file name or array if None background is assumed to be zero
scaling	(str or None)	scale the corrected result by this factor if 'max'/'mean' scale to keep max/mean invariant
save	(str or None)	save the corrected image to file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – illumination corrected image

References

Fundamentals of Light Microscopy and Electronic Imaging, p 421

See also:

DefaultFlatFieldLineFile

flatfieldFromLine(*line*, *xsize*)
Creates a 2d flat field image from a 1d line of estimated intensities

Parameters

- **lines** (*array*) – array of intensities along y axis
- **xsize** (*int*) – size of image in x dimension

Returns *array* – full 2d flat field

flatfieldLineFromRegression(*data*, *sink=None*, *method='polynomial'*, *reverse=None*, *verbose=False*)

Create flat field line fit from a list of positions and intensities

The fit is either to be assumed to be a Gaussian:

or follows a order 6 radial polynomial

Parameters

- **data** (*array*) – intensity data as vector of intensities or (n,2) dim array of positions d=0 and intensities measurements d=1:-1
- **sink** (*str or None*) – destination to write the result of the fit
- **method** (*str*) – method to fit intensity data, 'Gaussian' or 'Polynomial'
- **reverse** (*bool*) – reverse the line fit after fitting
- **verbose** (*bool*) – print and plot information for the fit

Returns *array* – fitted intensities on points

ClearMap.ImageProcessing.BackgroundRemoval module

Functions to remove background in images

The main routine subtracts a morphological opening from the original image for background removal.

removeBackground(*img*, *removeBackgroundParameter=None*, *size=None*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Remove background via subtracting a morphological opening from the original image

Background removal is done z-slice by z-slice.

Parameters

- **img** (*array*) – image data
- **removeBackGroundParameter** (*dict*) –

Name	Type	Descritption
size	(tuple or None)	size for the structure element of the morphological opening if None, do not correct for any background
save	(str or None)	file name to save result of this operation if None dont save to file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – background corrected image

ClearMap.ImageProcessing.GreyReconstruction module

Grey reconstruction module

This morphological reconstruction routine was adapted from CellProfiler.

Author

Original author: Lee Kamentsky Copyright (c) 2003-2009 Massachusetts Institute of Technology Copyright (c) 2009-2011 Broad Institute

Modified by Chirstoph Kirst to optimize integration into ClearMap, The Rockefeller University, New York City, 2015

reconstruct(*seed*, *mask*, *method='dilation'*, *selem=None*, *offset=None*)
Performs a morphological reconstruction of an image.

Reconstruction uses a seed image, which specifies the values to dilate and a mask image that gives the maximum allowed dilated value at each pixel.

The algorithm is taken from [1]. Applications for greyscale reconstruction are discussed in [2] and [3].

Parameters

- **seed** (*array*) – seed image to be dilated or eroded.
- **mask** (*array*) – maximum (dilation) / minimum (erosion) allowed
- **method** (*str*) – {'dilation'|'erosion'}
- **selem** (*array*) – structuring element
- **offset** (*array or None*) – offset of the structuring element, None is centered

Returns *array* – result of morphological reconstruction.

Note: Operates on 2d images.

Reference:

greyReconstruction(*img*, *mask*, *greyReconstructionParameter=None*, *method=None*, *size=3*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>'*, *mode 'w'>*, ***parameter*)
Calculates the grey reconstruction of the image

Reconstruction is done z-slice by z-slice.

Parameters

- **img** (*array*) – image data
- **removeBackGroundParameter** (*dict*) –

Name	Type	Descritption
method	(tuple or None)	'dilation' or 'erosion', if None return original image
size	(int or tuple)	size of structuring element
save	(str or None)	file name to save result of this operation if None dont save to file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

[1] Robinson, "Efficient morphological reconstruction: a downhill filter", Pattern Recognition Letters 25 (2004) 1759-1767.
[2] Vincent, L., "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms", IEEE Transactions on Image Processing (1993)
[3] Soille, P., "Morphological Image Analysis: Principles and Applications", Chapter 6, 2nd edition (2003), ISBN 3540429883.

Returns *array* – grey reconstructed image

ClearMap.ImageProcessing.SpotDetection module

Functions to detect spots in images

The main routine `detectCells()` uses a difference of gaussian filter (see *Filter*) followed by a peak detection step.

Example

```
>>> import os
>>> import ClearMap.IO as io
>>> import ClearMap.Settings as settings
>>> import ClearMap.ImageProcessing.SpotDetection as sd
>>> fn = os.path.join(settings.ClearMapPath, 'Test/Data/Synthetic/test_iDISCO_\d{3}.tif');
>>> img = io.readData(fn);
>>> img = img.astype('int16'); # converting data to smaller integer types can be more memory efficie
>>> res = sd.detectSpots(img, dogSize = (5,5,5), flatfield = None, threshold = 5, cellShapeThreshold
>>> print 'Found %d cells !' % res[0].shape[0]
Illumination: flatfield          : None
Illumination: illuminationScaling: True
Illumination: background         : None
Background: backgroundSize: (15, 15)
Background: elapsed time: 0:00:00
DoG: dogSize: (5, 5, 5)
DoG: elapsed time: 0:00:00
Extended Max: threshold   : 5
Extended Max: localMaxSize: 5
Extended Max: hMax        : None
Extended Max: elapsed time: 0:00:00
Cell Centers: elapsed time: 0:00:00
Cell Shape: cellShapeThreshold: 1
Cell Shape:: elapsed time: 0:00:00
Cell Size:: elapsed time: 0:00:00
Cell Intensity: cellIntensityMethod: Max
Cell Intensity:: elapsed time: 0:00:00
Cell Intensity: cellIntensityMethod: Max
Cell Intensity:: elapsed time: 0:00:00
Cell Intensity: cellIntensityMethod: Max
Cell Intensity:: elapsed time: 0:00:00
Found 38 cells !
```

After execution this example inspect the result of the cell detection in the folder 'Test/Data/CellShape/cellshape_d{3}.tif'.

detectSpots(*img*, *detectSpotsParameter=None*, *correctIlluminationParameter=None*, *removeBackgroundParameter=None*, *filterDoGParameter=None*, *findExtendedMaximaParameter=None*, *detectCellShapeParameter=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Detect Cells in 3d grayscale image using DoG filtering and maxima detection

Effectively this function performs the following steps:

- illumination correction via *correctIllumination()*
- background removal via *removeBackground()*

- difference of Gaussians (DoG) filter via `filterDoG()`
- maxima detection via *findExtendedMaxima()*
- cell shape detection via *detectCellShape()*
- cell intensity and size measurements via: *findCellIntensity()*, *findCellSize()*.

detectCells .. note:: Processing steps are done in place to save memory.

Parameters

- **img** (*array*) – image data
- **detectSpotParameter** – image processing parameter as described in the individual sub-routines
- **verbose** (*bool*) – print progress information
- **out** (*object*) – object to print progress information to

Returns *tuple* – tuple of arrays (cell coordinates, raw intensity, fully filtered intensty, illumination and background corrected intensity [, cell size])

test()
Test Spot Detection Module

ClearMap.ImageProcessing.MaximaDetection module

Collection of routines to detect maxima

Used for finding cells or intensity peaks.

hMaxTransform(*img*, *hMax*)
Calculates h-maximum transform of an image

Parameters

- **img** (*array*) – image
- **hMax** (*float or None*) – h parameter of h-max transform

Returns *array* – h-max transformed image if h is not None

localMax(*img*, *size=5*)
Calculates local maxima of an image

Parameters

- **img** (*array*) – image
- **size** (*float or None*) – size of volume to search for maxima

Returns *array* – mask that is True at local maxima

extendedMax(*img*, *hMax=0*)
Calculates extened h maxima of an image

Extended maxima are the local maxima of the h-max transform

Parameters

- **img** (*array*) – image
- **hMax** (*float or None*) – h parameter of h-max transform

Returns *array* – extended maxima of the image

findExtendedMaxima(*img*, *findExtendedMaximaParameter=None*, *hMax=None*, *size=5*, *threshold=None*, *save=None*, *verbose=None*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find extended maxima in an image

Effectively this routine performs a h-max transfrom, followed by a local maxima search and thresholding of the maxima.

Parameters

- **img** (*array*) – image data
- **findExtendedMaximaParameter** (*dict*) –

Name	Type	Descritption
hMax	(float or None)	h parameter for the initial h-Max transform if None, do not perform a h-max transform
size	(tuple)	size for the structure element for the local maxima filter
threshold	(float or None)	include only maxima larger than a threshold if None keep all localmaxima
save	(str or None)	file name to save result of this operation if None do not save result to file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – binary image with True pixel at extended maxima

See also:

hMaxTransform(), *localMax()*

findCenterOfMaxima(*img*, *imgmax*, *findCenterOfMaximaParameter=None*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find center of detected maxima weighted by intensity

Parameters

- **img** (*array*) – image data
- **findCenterOfMaximaParameter** (*dict*) –

Name	Type	Descritption
save	(str or None)	saves result of labeling the differnet maxima if None, do the lableling is not saved
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – coordinates of centers of maxima, shape is (n,d) where n is number of maxima and d the dimension of the image

findPixelCoordinates(*imgmax*, *subStack=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find coordinates of all pixel in an image with positive or True value

Parameters

- **img** (*array*) – image data
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – coordinates of centers of True pixels, shape is (n,d) where n is number of maxima and d the dimension of the image

findIntensity(*img*, *centers*, *findIntensityParameter=None*, *method=None*, *size=(3, 3, 3)*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Find instensity value around centers in the image

Parameters

- **img** (*array*) – image data
- **findIntensityParameter** (*dict*) –

Name	Type	Descritption
method	(str, func, None)	method to use to determine intensity (e.g. "Max" or "Mean") if None take intensities at the given pixels
size	(tuple)	size of the box on which to perform the *method*
verbose	(bool or int)	print / plot information about this step

- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – measured intensities

ClearMap.ImageProcessing.IlastikClassification module

Inteface to Illastik pixel classification

This module allows to integrate ilastik pixel classification into the *ClearMap* pipeline.

To train a classifier ilastik 0.5 should be used following these steps:

- genereate a classifier from illastik 0.5
- press 'Train and classify' button (eventhough the online traning is running) !
- save the classifier to a file
- use the classifiers file name in the ClearMap routine `classifyPixel()`
- try to avoid too many features in the classifier as classification gets very memory intensive otherwise

Note: Note that ilastik classification works in parallel, thus it is advised to process the data sequentially, see `sequentiallyProcessStack()`

Note: Ilastik 0.5 works for images in uint8 format !

References

- Ilastik

- Based on the ilastik interface from cell profiler

initializeIlastik (*path=None*)
Set system path for illastik installation

Initialized = True
bool: True if ilastik interface was sucessfully initialized .

rescaleToIlastik (*img*, *rescale=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)
Rescale image to achieve uint8 format used by ilasstik

The function rescales the image and converts the image to uin8 to fit with the image representation used by ilastik.

Parameters

- **img** (*array*) – image data
- **rescale** (*float or None*) – rescaling factor

Returns *array* – uint8 version of the image

rescaleFactorIlastik (*source*, *processes=12*)
Determines rescale factor given a image file

Parameters **source** – source file / image

Returns *float* – rescale factor

classifyPixel (*img*, *classifyPixelParameter=None*, *subStack=None*, *verbose=False*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)
Detect Cells Using a trained classifier in Ilastik

Parameters

- **img** (*array*) – image data
- **classifyPixelParameter** (*dict*) –

Name	Type	Descritption
classifier	(str or None)	saves result of labeling the differnet maxima if None dont correct image for illumination, if True the
rescale	(float,all, or None)	optional rescaling of the image to fit uint8 format used by ilastik, rescale
save	(str or None)	save the propabilities to belong to the classes to a file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array* – probabilities for each pixel to belong to a class in the classifier, shape is (img.shape, number of classes)

classifyCells (*img*, *classifyCellsParameter=None*, *classifier=None*, *rescale=None*, *save=None*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)
Detect Cells Using a trained classifier in Ilastik

The routine assumes that the first class is identifying the cells.

Parameters

- **img** (*array*) – image data
- **classifyPixelParameter** (*dict*) –

Name	Type	Descritption
classifier	(str or None)	saves result of labeling the differnet maxima if None dont correct image for illumination, if True the
rescale	(float or None)	optional rescaling of the image to fit uint8 format used by ilastik
save	(str or None)	save the detected cell pixel to a file
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *tuple* – centers of the cells, intensity measurments

---

Note: The routine could be poteNtially refined to make use of background detected by ilastik

---

ClearMap.ImageProcessing.ImageStatistics module

Functions to gather iamge statistics in large volumetric images

The main routines extract information from a large volumetric image, such as the maximum or mean.

calculateStatistics(*source*, *sink=None*, *calculateStatisticsParameter=None*, *method='Max'*, *remove=True*, *processMethod=<built-in function all>*, *verbose=False*, ***parameter*)

Calculate statisticsfrom image data

This is a main script to start extracting statistics of volumetric image data.

Parameters

- **source** (*str or array*) – Image source
- **sink** (*str or None*) – destination for the results
- **calculateStatisticsParameter** (*dict*) –

Name	Type	Descritption
method	(str or function)	function to extract statistic, must be trivially distributable if None, do not extract information
remove	(bool)	remove redundant overlap
verbose	(bool or int)	print / plot information about this step

- **method** (*str or function*)
- **processMethod** (*str or all*) – 'sequential' or 'parallel'. if all its choosen automatically
- **verbose** (*bool*) – print info
- ****parameter** (*dict*) – parameter for the image processing sub-routines

Returns list of statistics

calculateStatisticsOnStack(*img*, *calculateStatisticsParameter=None*, *method='Max'*, *remove=True*, *verbose=False*, *subStack=None*, *out=<open file '<stdout>', mode 'w'>*, ***parameter*)

Calculate a statistics from a large volumetric image

The statistics is assumed to be trivially distributable, i.e. max or mean.

Parameters

- **img** (*array*) – image data
- **calculateStatisticsParameter** (*dict*) –

Name	Type	Descritption
method	(str or function)	function to extract statistic, must be trivially distributable if None, do not extract information
remove	(bool)	remove redundant overlap
verbose	(bool or int)	print / plot information about this step

- **subStack** (*dict or None*) – sub-stack information
- **verbose** (*bool*) – print progress info
- **out** (*object*) – object to write progress info to

Returns *array or number* – extracted statistics

Note: One might need to choose zero overlap in the stacks to function properly!

---

joinStatistics(*results*, *calculateStatisticsParameter=None*, *method='Max'*, *subStacks=None*, ***parameter*)

Joins a list of calculated statistics

Parameters

- **results** (*list*) – list of statics results from the individual sub-processes
- **calculateStatisticsParameter** (*dict*) –

Name	Type	Descritption
method	(str or function)	function to extract statistic, must be trivially distributable if None, do not extract information

- **subStacks** (*list or None*) – list of all sub-stack information, see *Sub-Stacks*

Returns *list or object* – joined statistics

ClearMap.Analysis package

ClearMap analysis and statistics toolbox.

This part of ClearMap provides a toolbox for the statistical analysis and visualization of detected cells or structures and region specific analysis of annotated data.

For cleared mouse brains aligned to the Allen brain atlas, a wide range of statistical analysis tools with respect to the annotated brain regions in the atlas is supported.

Key modules are:

Module	Description
Voxelization	Voxelization of cells for visualization and analysis
Statistics	Statistical tools for the analysis of detected cells
Label	Tools to analyse data with respect to annotated references

Subpackages

ClearMap.Analysis.Tools package Analysis and statistics tools not in standard python packages.

ClearMap.Analysis.Tools.Extrapolate module Method to extend interpolation objects to constantly / linearly extrapolate.

extrap1d(*x*, *y*, *interpolation='linear'*, *exterpolation='constant'*)
Interpolate on given values and extrapolate outside the given data

Parameters

- **x** (*numpy.array*) – x values of the data to interpolate
- **y** (*numpy.array*) – y values of the data to interpolate
- **interpolation** (*Optional[str]*) – interpolation method, see kind of scipy.interpolate.interp1d, default: "linear"
- **exterpolation** (*Optional[str]*) – interpolation method, either "linear" or "constant"

Returns *(function)* – inter- and extra-polation function

extrap1dFromInterp1d(*interpolator*, *exterpolation='constant'*)
Extend interpolation function to extrapolate outside the given data

Parameters

- **interpolator** (*function*) – interpolating function, see e.g. scipy.interpolate.interp1d
- **exterpolation** (*Optional[str]*) – interpolation method, either "linear" or "constant"

Returns *(function)* – inter- and extra-polation function

ClearMap.Analysis.Tools.MultipleComparisonCorrection module Correction methods for multiple comparison tests

correctPValues(*pvalues*, *method='BH'*)
Corrects p-values for multiple testing using various methods

Parameters

- **pvalues** (*array*) – list of p values to be corrected
- **method** (*Optional[str]*) – method to use: BH = FDR = Benjamini-Hochberg, B = FWER = Bonferoni

References

- Benjamini Hochberg, 1995
- Bonferoni correction
- R statistics package

Notes

•modified from http://statsmodels.sourceforge.net/ipdirective/generated/scikits.statsmodels.sandbox.stats.multicomp.multiple

estimateQValues(*pvalues*, *m=None*, *pi0=None*, *verbose=False*, *lowMemory=False*)
Estimates q-values from p-values

Parameters

- **pvalues** (*array*) – list of p-values
- **m** (*int or None*) – number of tests. If None, m = pvalues.size
- **pi0** (*float or None*) – estimate of m_0 / m which is the (true null / total tests) ratio, if None estimation via cubic spline.
- **verbose** (*bool*) – print info during execution
- **lowMemory** (*bool*) – if true use low memory version

Notes

•The q-value of a particular feature can be described as the expected proportion of false positives among all features as or more extreme than the observed one

•The estimated q-values are increasing in the same order as the p-values

References

•Storey and Tibshirani, 2003

•modified from https://github.com/nfusi/qvalue

ClearMap.Analysis.Tools.StatisticalTests module Some statistics tests not in standard python packages

testCramerVonMises2Sample(*x*, *y*)
Computes the Cramer von Mises two sample test.

This is a two-sided test for the null hypothesis that 2 independent samples are drawn from the same continuous distribution.

Parameters

- **x, y** (*sequence of 1-D ndarrays*) – two arrays of sample observations
- **assumed to be drawn from a continuous distribution, sample sizes**
- **can be different**

Returns *(float, float)* – T statistic, two-tailed p-value

References

•modified from https://github.com/scipy/scipy/pull/3659

ClearMap.Analysis.Label module

Label and annotation info from Allen Brain Atlas (v2)

Notes

- The annotation file is assumed to be in './Data/Annotation/annotation_25_right.tif' but can be set in the constant *DefaultLabeledImageFile*
- The mapping between labels and brain area information is found in the './Data/ARA2_annotation_info.csv' file. In the './Data/ARA2_annotation_info_collapse.csv' file a cross marks an area to which all sub-areas can be collapsed. The location of this file is set in *DefaultAnnotationFile*.
- For consistency certain labels of the Allen brain atlas without annotation were assigned to their correct parent regions.
- A collapse column in the mapping file was added to allow for a region based collapse of statistics based on the inheritance structure of the annotated regions. These might need to be adjusted to the particular scientific question.

References

- Allen Brain Atlas

DefaultLabeledImageFile = '/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/Data/Annotation/annotation_25_r
str: default volumetric annotated image file

This file is by default the Allen brain annotated mouse atlas with 25um isotropic resolution.

DefaultAnnotationFile = '/home/mtllab/Programs/ClearMap/idisco/ClearMap/Data/ARA2_annotation_info_collapse.csv
str: default list of labels in the annotated image and names of annotated regions

This file is by default the labels for the Allen brain annotated mouse atlas with 25um isotropic resolution.

An extra column for collapse indicates how to automatically collapse data into the different brain regions if the `collapse` option is given.

class LabelRecord(*id*, *name*, *acronym*, *color*, *parent*, *collapse*)
Bases: `tuple`

Structure of a label for a annotated region

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

acronym
Alias for field number 2

collapse
Alias for field number 5

color
Alias for field number 3

id
Alias for field number 0

name
Alias for field number 1

parent
Alias for field number 4

class LabelInfo (*slf*, *annotationFile='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Data/ARA2_annotation_info_collapse.cs*
Bases: object

Class that holds information of the annotated regions

ids = None

names = None

acronyms = None

colors = None

parents = None

levels = None

collapse = None

collapseMap = None

initialize (*slf*, *annotationFile='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Data/ARA2_annotation_info_collapse.*

name (*slf*, *iid*)

acronym (*slf*, *iid*)

color (*slf*, *iid*)

parent (*slf*, *iid*)

level (*slf*, *iid*)

toLabelAtLevel (*slf*, *iid*, *level*)

toLabelAtCollapseMap (*slf*, *iid*)

toLabelAtCollapse (*slf*, *iid*)

Label = <ClearMap.Analysis.Label.LabelInfo object>
Information on the annotated regions

initialize (*annotationFile='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Data/ARA2_annotation_info_collapse.csv'*)

labelAtLevel (*label*, *level*)

labelAtCollapse (*label*)

labelPoints (*points*, *labeledImage='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/Data/Annotation/annotation_25_rig*
level=None, *collapse=None*)

countPointsInRegions (*points*, *labeledImage='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/Data/Annotation/annot*
intensities=None, *intensityRow=0*, *level=None*, *allIds=False*, *sort=True*, *returnIds=True*, *returnCounts=False*, *collapse=None*)

labelToName (*label*)

labelToAcronym (*label*)

labelToColor (*label*)

writePAL (*filename*, *cols*)

writeLUT (*filename*, *cols*)

makeColorPalette (*filename=None*)
Creates a pal file for imaris based on label colors

makeColorAnnotations (*filename*, *labeledImage=None*)

test ()
Test Label module

ClearMap.Analysis.Statistics module

Create some statistics to test significant changes in voxelized and labeled data

TODO: cleanup / make generic

readDataGroup (*filenames*, *combine=True*, ***args*)
Turn a list of filenames for data into a numpy stack

readPointsGroup (*filenames*, ***args*)
Turn a list of filenames for points into a numpy stack

tTestVoxelization (*group1*, *group2*, *signed=False*, *removeNaN=True*, *pcutoff=None*)
t-Test on differences between the individual voxels in group1 and group2, group is a array of voxelizations

cutoffPValues (*pvals*, *pcutoff=0.05*)

colorPValues (*pvals, psign, positive=[1, 0], negative=[0, 1], pcutoff=None, positivetrend=[0, 0, 1, 0], negativetrend=[0, 0, 0, 1], pmax=None*)

mean (*group*, ***args*)

std (*group*, ***args*)

var (*group*, ***args*)

thresholdPoints (*points*, *intensities*, *threshold=0*, *row=0*)
Threshold points by intensities

weightsFromPrecentiles (*intensities, percentiles=[25, 50, 75, 100]*)

countPointsGroupInRegions (*pointGroup*, *labeledImage='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/Data/Anno*
intensityGroup=None, *intensityRow=0*, *returnIds=True*, *returnCounts=False*, *collapse=None*)
Generates a table of counts for the various point datasets in pointGroup

tTestPointsInRegions (*pointCounts1*, *pointCounts2*, *labeledImage='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/D*
signed=False, *removeNaN=True*, *pcutoff=None*, *equal_var=False*)
t-Test on differences in counts of points in labeled regions

testCompletedCumulatives (*data*, *method='AndersonDarling'*, *offset=None*, *plot=False*)
Test if data sets have the same number / intensity distribution by adding max intensity counts to the smaller sized data sets and performing a distribution comparison test

testCompletedInvertedCumulatives (*data*, *method='AndersonDarling'*, *offset=None*, *plot=False*)
Test if data sets have the same number / intensity distribution by adding zero intensity counts to the smaller sized data sets and performing a distribution comparison test on the reversed cumulative distribution

testCompletedCumulativesInSpheres(*points1*, *intensities1*, *points2*, *intensities2*, *dataSize='/home/mtllab/Programs/ClearMap/idisco/ClearMap/Test/Data/Annotation/an*, *radius=100*, *method='AndresonDarling'*)
Performs completed cumulative distribution tests for each pixel using points in a ball centered at that cooridnates, returns 4 arrays p value, statistic value, number in each group

test()
Test the statistics array

ClearMap.Analysis.Voxelization module

Converts point data into voxel image data for visulaization and analysis

voxelize(*points*, *dataSize=None*, *sink=None*, *voxelizeParameter=None*, *method='Spherical'*, *size=(5, 5, 5)*, *weights=None*)
Converts a list of points into an volumetric image array

Parameters

- **points** (*array*) – point data array
- **dataSize** (*tuple*) – size of final image
- **sink** (*str, array or None*) – the location to write or return the resulting voxelization image, if None return array
- **voxelizeParameter** (*dict*) –

Name	Type	Descritption
method	(str or None)	method for voxelization: 'Spherical', 'Rectangular' or 'Pixel'
size	(tuple)	size parameter for the voxelization
weights	(array or None)	weights for each point, None is uniform weights

Returns *(array)* – volumetric data of smeared out points

voxelizePixel(*points*, *dataSize=None*, *weights=None*)
Mark pixels/voxels of each point in an image array

Parameters

- **points** (*array*) – point data array
- **dataSize** (*tuple or None*) – size of the final output data, if None size is determined by maximal point coordinates
- **weights** (*array or None*) – weights for each points, if None weights are all 1s.

Returns *(array)* – volumetric data with with points marked in voxels

test()
Test voxelization module

ClearMap.Visualization package

This sub-package provides tools for the visualization of the alignment and analysis results

ClearMap.Visualization.Plot module

Plotting routines for overlaying labels, tilings, and sectioning of 3d data sets

Supported functionality:

- plot volumetric data as a sequence of tiles via `plotTiling()`
- overlay points on images via `overlayPoints()` and `plotOverlayPoints()`
- overlay labeled images on gray scale images via `overlayLabel()` and `plotOverlayLabel()`

plotTiling(*dataSource*, *tiling='automatic'*, *maxtiles=20*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *inverse=False*)

Plot 3d image as 2d tiles

Parameters

- **dataSouce** (*str or array*) – volumetric image data
- **tiling** (*str or tuple*) – tiling specification
- **maxtiles** – maximalnumber of tiles
- **x, y, z** (*all or tuple*) – sub-range specification
- **inverse** (*bool*) – invert image

Returns *(object)* – figure handle

overlayLabel(*dataSource*, *labelSource*, *sink=None*, *alpha=False*, *labelColorMap='jet'*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*)

Overlay a gray scale image with colored labeled image

Parameters

- **dataSouce** (*str or array*) – volumetric image data
- **labelSource** (*str or array*) – labeled image to be overlayed on the image data
- **sink** (*str or None*) – destination for the overlayed image
- **alpha** (*float or False*) – transparency
- **labelColorMap** (*str or object*) – color map for the labels
- **x, y, z** (*all or tuple*) – sub-range specification

Returns *(array or str)* – figure handle

See also:

`overlayPoints()`

plotOverlayLabel(*dataSource*, *labelSource*, *alpha=False*, *labelColorMap='jet'*, *x=<built-in function all>*, *y=<built-in function all>*, *z=<built-in function all>*, *tiling='automatic'*, *maxtiles=20*)

Plot gray scale image overlayed with labeled image

Parameters

- **dataSouce** (*str or array*) – volumetric image data
- **labelSource** (*str or array*) – labeled image to be overlayed on the image data
- **alpha** (*float or False*) – transparency
- **labelColorMap** (*str or object*) – color map for the labels

- **x, y, z** (*all or tuple*) – sub-range specification
- **tiling** (*str or tuple*) – tiling specification
- **maxtiles** – maximalnumber of tiles

Returns *(object)* – figure handle

See also:

overlayLabel()

overlayPoints(*dataSource, pointSource, sink=None, pointColor=[1, 0, 0], x=<built-in function all>, y=<built-in function all>, z=<built-in function all>*)

Overlay points on 3D data and return as color image

Parameters

- **dataSouce** (*str or array*) – volumetric image data
- **pointSource** (*str or array*) – point data to be overlayed on the image data
- **pointColor** (*array*) – RGB color for the overlayed points
- **x, y, z** (*all or tuple*) – sub-range specification

Returns *(str or array)* – image overlayed with points

See also:

overlayLabel()

plotOverlayPoints(*dataSource, pointSource, pointColor=[1, 0, 0], x=<built-in function all>, y=<built-in function all>, z=<built-in function all>*)

Plot points overlayed on gray scale 3d image as tiles.

Parameters

- **dataSouce** (*str or array*) – volumetric image data
- **pointSource** (*str or array*) – point data to be overlayed on the image data
- **pointColor** (*array*) – RGB color for the overlayed points
- **x, y, z** (*all or tuple*) – sub-range specification

Returns *(object)* – figure handle

See also:

plotTiling()

test()

Test Plot module

ClearMap.Parameter module

ClearMap default parameter module.

This module defines default parameter used by various sub-packages.

See also:

Settings

detectCellParameter = {'findExtendedMaximaParameter': {'threshold': 0, 'save': None, 'verbose': False, 'hMax': 20, 'si
dict: Paramters for cell detection using the spot detection algorithm

See also:

IlastikParameter, StackProcessingParameter

IlastikParameter = {'rescale': None, 'backgroundSize': (15, 15), 'classifier': '/Test/Ilastik/classifier.h5'}
dict: Paramters for cell detection using Ilastik classification

- "classifier": ilastic classifier to use
- "rescale": rescale images before classification
- "backgroundSize": Background correctoin: None or (y,x) which is size of disk for gray scale opening

See also:

SpotDetectionParameter, StackProcessingParameter

processStackParameter = {'chunkOptimizationSize': <built-in function all>, 'processes': 2, 'chunkSizeMin': 30, 'chunkO

dict: Parameter for processing an image stack in parallel

- "processes": max number of parallel processes
- "chunkSizeMax" : maximal chunk size in z
- "chunkSizeMin" : minimal chunk size in z,
- "chunkOverlap" : overlap between two chunks,
- "chunkOptimization": optimize chunk size and number to number of processes
- "chunkOptimizationSize": increase chunk size for optimizaition (True, False or all = automatic)

See also:

SpotDetectionParameter, *IlastikParameter*

AlignmentParameter = {'fixedImageMask': None, 'alignmentDirectory': None, 'movingImage': '/Test/Data/Elastix/150524
dict: Parameter for Elastix alignment

- "alignmentDirectory" : directory to save the alignment result
- "movingImage": image to be aligned
- "fixedImage": reference image
- "affineParameterFile": elastix parameter files for affine alignment
- "bSplineParameterFile" : elastix parameter files for non-linear alignment

See also:

Elastix

ResamplingParameter = {'orientation': None, 'source': None, 'resolutionSink': (25, 25, 25), 'sink': None, 'resolutionSource
dict: Parameter for resampling data

- "source" : data source file
- "sink" : data output file

"resolutionSource": resolution of the raw data (in um / pixel) as (x,y,z)

"resolutionSink" : resolution of the reference / atlas image (in um/ pixel) as (x,y,z)

"orientation" [Orientation of the data set wrt reference as (x=1,y=2,z=3)] (-axis will invert the orientation, for other hemisphere use (-1, 2, 3), to exchnge x,y use (2,1,3) etc)

See also:

Resampling

VoxelizationParameter = {'method': 'Spherical', 'voxelizationSize': (1, 1, 1)}
dict: Parameter to calculate density voxelization

- "method": Method to voxelize: 'Spherical','Rectangular, 'Gaussian'
- "voxelizationSize": max size of the volume to be voxelized

See also:

voxelization

ClearMap.Settings module

Module to set *ClearMap's* internal parameter and paths to external programs.

Notes

Edit the *setup()* routine to point to the ilastik and elastix paths for specific hosts

See also:

- *IlastikPath*
- *ElastixPath*
- *Parameter*

IlastikPath = None
str: Absolute path to the Ilastik 0.5 installation

Notes

Ilastik Webpage

Ilastik 0.5 Download

ElastixPath = None
str: Absolue path to the elastix installation

Notes

Elastix Webpage

setup()
Setup ClearMap for specific hosts

Notes

Edit this routine to include special setttings for specific hosts

See also:

IlastikPath, *ElastixPath*

clearMapPath()
Returns root path to the ClearMap software

Returns *str* – root path to ClearMap

ClearMapPath = '/Users/nicolasrenier/Documents/ClearMap/idisco/ClearMap'
str: Absolute path to the ClearMap root folder

ClearMap.Utils package

This sub-package provides utility functions used throughout the package

ClearMap.Utils.ParameterTools module

ParameterTools

Provides simple formatting tools to handle / print parameter dictionaries organized as key:value pairs.

getParameter(*parameter*, *key*, *default=None*)
Gets a parameter from a dict, returns default value if not defined

Parameters

- **parameter** (*dict*) – parameter dictionary
- **key** (*object*) – key
- **default** (*object*) – deault return value if parameter not defined

Returns *object* – parameter value for key

writeParameter(*head=None*, *out=None*, ***args*)
Writes parameter settings in a formatted way

Parameters

- **head** (*str or None*) – prefix of each line
- **out** (*object or None*) – write to a specific output, if None return string
- ****args** – the parameter values as key=value arguments

Returns *str or None* – a formated string with parameter info

joinParameter(**args*)
Joins dictionaries in a consitent way

For multiple occurences of a key the value is defined by the first key : value pair.

Parameters ***args** – list of parameter dictonaries

Returns *dict* – the joined dictionary

ClearMap.Utils.ProcessWriter module

Provides simple formatting tools to print text with parallel process header

class ProcessWriter(*process=0*)
Bases: `object`

Class to handle writing from parallel processes

process
int

the process number

writeString(*text*)
Generate string with process prefix

Parameters **text** (*str*) – the text input

Returns *str* – text with [process prefix

write(*text*)
Write string with process prefix to sys.stdout

Parameters **text** (*str*) – the text input

ClearMap.Utils.Timer module

Provides tools for timing

class Timer(*verbose=False*)
Bases: `object`

Class to stop time and print results in formatted way

time
float

the time since the timer was started

start()
Start the timer

reset()
Reset the timer

elapsedTime(*head=None*, *asstring=True*)
Calculate elapsed time and return as formated string

Parameters

- **head** (*str or None*) – prefix to the string
- **asstring** (*bool*) – return as string or float

Returns *str or float* – elapsed time

printElapsedTime(*head=None*)
Print elapsed time as formated string

Parameters **head** (*str or None*) – prefix to the string

formatElapsedTime(*t*)
Format time to string

Parameters **t** (*float*) – time in seconds prefix

Returns *str* – time as hours:minutes:seconds