

---

# **clickpoints Documentation**

***Release 1.8.0***

**Richard Gerum, Sebastian Richter**

**Apr 25, 2019**



---

## Contents

---

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Windows Installer . . . . .	3
1.2	Python Packages . . . . .	3
1.3	Developer Version . . . . .	4
1.4	Possible Errors . . . . .	4
<b>2</b>	<b>General</b>	<b>5</b>
2.1	Zooming, Panning, Rotating . . . . .	6
2.2	Jumping frames . . . . .	6
2.3	Interfaces . . . . .	6
<b>3</b>	<b>Tutorial</b>	<b>7</b>
3.1	Opening Files and Folders . . . . .	7
3.2	Using ConfigFiles . . . . .	8
3.3	Manual Tracking . . . . .	9
3.4	Taking Measurements . . . . .	14
<b>4</b>	<b>Modules</b>	<b>15</b>
4.1	Timeline . . . . .	15
4.2	GammaCorrection . . . . .	17
4.3	VideoExporter . . . . .	18
4.4	Annotations . . . . .	20
4.5	Marker . . . . .	21
4.6	Mask . . . . .	25
4.7	Info Hud . . . . .	29
<b>5</b>	<b>Add-ons</b>	<b>31</b>
5.1	Tracking . . . . .	31
5.2	Fluorescence Diffusion . . . . .	33
5.3	Drift Correction . . . . .	35
5.4	Cell Detector . . . . .	35
5.5	Grab Plot Data . . . . .	37
<b>6</b>	<b>Examples</b>	<b>39</b>
6.1	Count Animals . . . . .	39
6.2	Flourescence intensities in plant roots . . . . .	42
6.3	Supervised Tracking of Fiducial Markers in Magnetic Tweezer Measurements . . . . .	45

6.4	Using ClickPoints for Visualizing Simulation Results . . . . .	48
<b>7</b>	<b>Database API</b>	<b>53</b>
7.1	Database Models . . . . .	53
7.2	DataFile . . . . .	59
<b>8</b>	<b>Add-on API</b>	<b>61</b>
8.1	Meta Data File . . . . .	61
8.2	Script File . . . . .	62
8.3	Defining Options . . . . .	62
8.4	Events . . . . .	63
8.5	Commands . . . . .	64
<b>9</b>	<b>Note</b>	<b>65</b>
<b>10</b>	<b>Citing ClickPoints</b>	<b>67</b>
<b>11</b>	<b>Who uses ClickPoints</b>	<b>69</b>
	<b>Bibliography</b>	<b>71</b>



Click Points is a program written in the Python programming language, which serves on the one hand as an image viewer and on the other hand as an data display and annotation tool. Every frame can be annotated by a description, marked points/tracks, or marked areas (paint brush). This helps to view image data, do manual evaluation of data, help to create semi-automatic evaluation or display the results of automatic image evaluation.



ClickPoints can be installed in different ways, you can choose the one which is the most comfortable for you and the operating system you are using.

### 1.1 Windows Installer

If you have no Python installation and just want to get started, our installer is the best option for you. Just download and execute the following installer:

[Download: ClickPoints Installer](#)

This will install the miniconda environment, if it is not already installed and download the clickpoints conda package.

---

**Note:** ClickPoints will be by default installed in a new conda environment called *\_app\_own\_environment\_clickpoints*.

---

### 1.2 Python Packages

If you are already familiar with python and have a python installation, you can choose one of the following ways:

- If you are in a conda env: `conda install -c conda-forge -c rgerum clickpoints (recommended)`
- If you have pip: `pip install clickpoints`
- Or with `python setup.py install`

We recommend the conda installation, as this should always be the newest version of ClickPoints.

If you want to register clickpoints for the typical file extensions, e.g. add it to the right click menu, execute

```
clickpoints register
```

If you want to remove it again, call

```
clickpoints unregister
```

## 1.3 Developer Version

If you want to have ClickPoints installed from the repository and be able to update to the newest changesets, you can follow this guide. First of all you need to have mercurial installed ([Mercurial](#)). Then you can open a command line in the folder where you want to install ClickPoints (e.g. C:Software) and run the following command:

```
hg clone https://bitbucket.org/fabry_biophysics/clickpoints
```

To install the package with all dependencies, go to the folder where ClickPoints has been downloaded (e.g. C:Softwareclickpoints) and execute:

```
python install_requirements_with_conda.py
```

in the downloaded repository directory. Then execute the command

```
clickpoints register
```

which will add clickpoints to the right click menu in the file explorer.

## 1.4 Possible Errors

Here is a short list of possible error messages after installation and how they can be fixed.

```
This application failed to start because it could not find or load
the Qt platform plugin "windows" in "".
```

reinstall pyqt5 with `pip install pyqt5`.



## CHAPTER 2


---


### General

---

Once ClickPoints has been [installed](#) it can be started directly from the Start Menu/Program launcher.

This will open ClickPoints with an empty project.

Images can be added to the project by using .

The project can be saved by clicking on .

ClickPoints can also be used to directly open images, videos or folder by right clicking on them, which will open an unsaved project which already contains some images. This way ClickPoints functions as an image viewing tool.

ClickPoints can be opened with various files as target:

- an **image**, loading all images in the folder of the target image.
- a **video** file, loading only this video.
- a **folder**, loading all image and video files of the folder and its sub folders, which are concatenated to one single image stream.
- a previously saved `.cdb` **ClickPoints Project** file, loading the project as it was saved.

Pressing `ESC` closes ClickPoints.

To easily access marker, masks, track or other information, stored in the `.cdb` ClickPoints Project file, we provide a python based [API](#)

**Attention:** If you plan to evaluate your data set or continue working on the same data set you must save the project - otherwise all changes will be lost upon closing the program. If a project was saved, all changes are saved automatically upon frame change or by pressing `S`

## 2.1 Zooming, Panning, Rotating

ClickPoints opens with a display of the current image fit to the window. The display can be

- zoomed, using the mouse wheel
- panned, holding down the right mouse button
- rotated using R.

To fit the image into the window press F and switch to full screen mode by pressing W.

---

**Note:** Default rotation on startup or and rotation steps with each press of R can be defined in the `ConfigClickPoints.txt` with the entries `rotation =` and `rotation_steps =`.

---

## 2.2 Jumping frames

ClickPoints provides various options to change the current frame.

- The keys `Left` and `Right` go to the previous or next frame.
- The keys `Home` and `End` jump to the first or last frame.
- Click or Drag & Drop the [timeline](#) slider

Key pairs on the numpad allow for jumps of speciefied

- Numpad 2, Numpad 3: `-/+ 1`
- Numpad 5, Numpad 6: `-/+ 10`
- Numpad 8, Numpad 9: `-/+ 100`
- Numpad /, Numpad \*: `-/+ 1000`

Be sure to have the numpad activated, or the keys won't work.

---

**Note:** The step size of the jump keys can be redefined by the `jumps = variable` in the `ConfigClickPoints.txt`

---

For continuous playback of frames see [timeline](#) module.

## 2.3 Interfaces

The interfaces for Marker, Mask and GammaCorretion can be shown/hidden pressing F2.

The recipes section contains some basic usage examples on how to get started using ClickPoints and explains different ways for different tasks.


### 3.1 Opening Files and Folders

ClickPoints was designed with multiple usage keys in mind, and therefore provides multiple ways to open files.

**Attention:** Opening a set of files for the first time can take some time to extract time and meta information from the filename, TIFF or EXIF header. For large collections of files it is recommended to save the collection as a project and use the `.cdb` file for starting ClickPoints. Saving time as no file system search is necessary and all meta information is already stored in the `.cdb`

#### 3.1.1 via Interface

ClickPoints can be started empty by using a desktop link or calling `ClickPoints.bat` from CMD (Windows), or respectively `ClickPoints` from a terminal (Linux).

Images can be added by using .

#### 3.1.2 via Context Menu

A fast and comfortable way to open files and folders with ClickPoints is the context menu.

ClickPoints can be opened with various files as target:

- an **image**, loading all images in the folder of the target image.
- a **video** file, loading only this video.

- a **folder**, loading all image and video files of the folder and its sub folders, which are concatenated to one single image stream.
- a previously saved `.cdb` **ClickPoints Project** file, loading the project as it was saved.

### 3.1.3 via Commandline Parameter

ClickPoints can be run directly from the commandline, e.g. to open the files in the current or a specific folder

```
ClickPoints "C:\Images"
```

or

```
python ClickPoints.py -srcfile="C:\Images"
```

---

**Note:** To use the short version of calling ClickPoints without the path, you have to add ClickPoints base path to the systems or users PATH variable (Windows) or create an alias (Linux).

---

### 3.1.4 via .txt File

Furthermore it is possible to supply a text file where each line contains the path to an image or video file. This is useful e.g. to open a fixed set of files, a list of files extract by another application or a database interface.

```
ClickPoints "sample.txt"
```

Listing 1: sample.txt

```
1  20120919_colonydensity.gif
   ↪      # relativ path (to txt file)
2  C:\Users\Desktop\images\20160601-141408_GE4000.jpg      # absolut path
3  \\192.168.0.99\2014\20140323\03\20140323-030151_31n2.JPG  # network path
```

---

**Note:** It is possible to open files over the network e.g. via samba shares. On Linux systems it is necessary do mount the network drive first!

---

## 3.2 Using ConfigFiles

The config file contains parameters to adjust ClickPoints to the users needs, adjusts default behaviour or configure key bindings. A GUI is available to set parameters during runtime, the ConfigFile is used to set default behaviour.

### 3.2.1 Scope

Config files in ClickPoints are designed to optimize the work flow. Upon opening a file the path structure is searched for the first occurrence of a valid config file. Thereby allowing the user to specify default for files grouped in one location.

If no config file is found, the default config values as set in click points base path are used (green). A config file located in the path “research” (blue) will overwrite these values and is used for files opened in child paths. Allowing the user

to define a preferred setup. In addition we can add a second config file lower in the path tree to specify a specific setup for all files that were stored under “Experiment\_3”. This can contain a set of default marker names, which features to use or which add-ons to include.

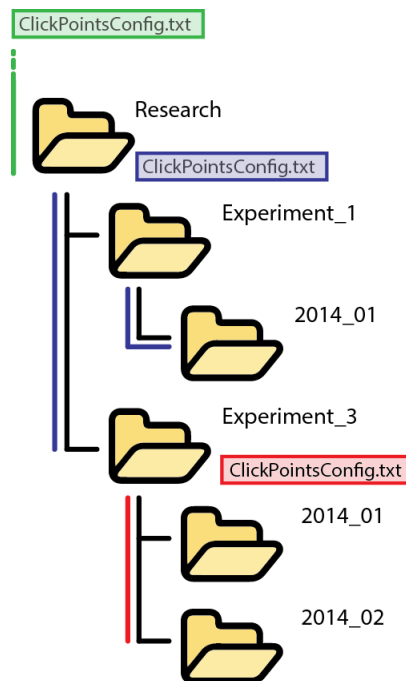


Fig. 1: Scope of ConfigFiles

---

**Note:** A graphical user interface to view and change config values is available too.

---

## 3.3 Manual Tracking


This tutorial gives a short introduction how to get started manually labeling your own tracks, for a quick evaluation or ground truths for the evaluation of automated algorithms.

### 3.3.1 Getting started

#### 1. Open the image sequence or video(s) in ClickPoints

For example: right click on the folder containing your images and select ClickPoints on the context menu

#### 2. Save the project

Marked results and correlated images must be stored somewhere, therefore the project has to be named and saved. Click on the save button  and select a storage location and file name.


---

**Note:** Reference to images and video is stored relative as long as the files reside parallel or below in the path tree. If the files reside above or on a different branch, drive, or network location, the absolute path is stored.

---

### 3. Define Marker types

Before we can get started we have to specify a marker type. Marker types are like classes of objects, e.g. we might use a class for birds and another one for ships. Every marker type can have multiple tracks.

To open the marker menu either press F2 or click on the Marker button  to switch to edit mode (Fig. A). Then right click onto the marker list to open the marker menu (Fig. B). You can reuse the default marker or create a new marker by selecting + add type. Choose a name and color for your new marker type and make sure to set the type to TYPE\_track. Confirm your changes by pressing save. To add more tracking types select + add type and repeat the procedure.

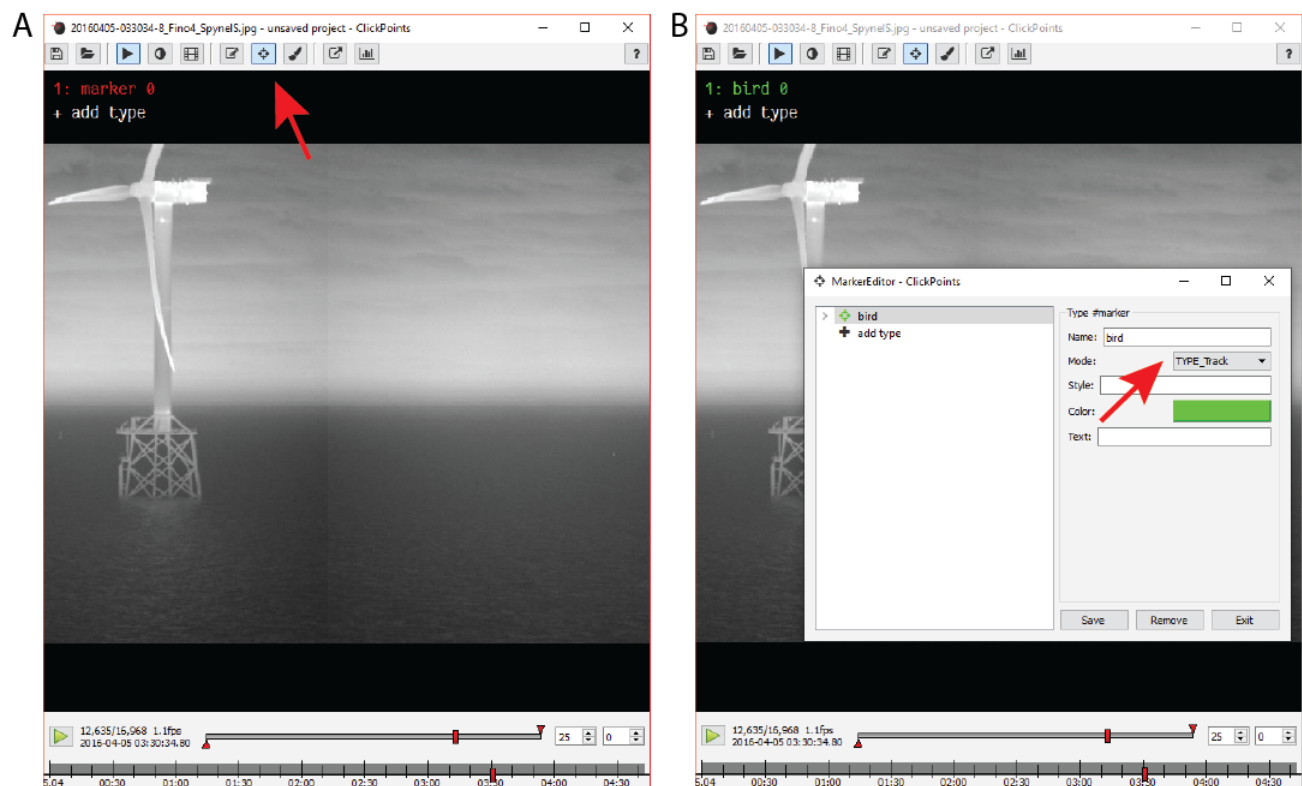


Fig. 2: Figure 1 | Defining a marker for tracking

### 4. Navigating the dataset

- Navigating the current frame:
  - right mouse button (hold) - to pan the image
  - mouse wheel - zoom the image
  - F - fit to view


W - full screen mode

H - hide time line

See [General](#)

- Navigating the dataset:

left & right cursor keys to go one frame forward and backward

- Jump a specified set of frames with the numbad keys. See [Jumping Frames](#)
- Use the frame and time navigation slider to by clicking or dragging the cursor to the desired position.
- Jump to a specific frame by clicking on the frame counter and entering the desired frame number
- Press  to play the dataset with the specified frame rate or as fast as feasible.

---

**Note:** Due to the sequential compression of videos, traversing a video backwards is computational expensive. ClickPoints provides a buffer so that the last N frames are stored and can be retrieved without any further computational cost. The default buffer size can be specified in the config.

---

**Warning:** Be careful not to reserve too much RAM for the frame buffer as it will drastically reduce performance!

## 5. Basic Tracking Procedure

The setup steps are completed, we can begin to mark some tracks.

1. Activate the type of marker you want to use by clicking on the label “bird” or press the associated number key.
2. Set the first marker by clicking on the image.
3. Switch to the next frame using the right cursor key.
4. The track now shows up with reduced opacity, indicating there is no marker for the current frame.
5. Upon dragging the marker (left click & hold) to the current position (release) a line indicates the connection to the last position. The track shows up with full opacity again.
6. If a frame is skipped, the marker can be dragged as usual but no connecting line will appear. Indicating a fragmentation of the track.
7. To create a second track, repeat step 1.
8. Markers are automatically save upon frame change or by pressing the S key.

## 6. “Connect-nearest” Tracking Mode

For low density tracks ClickPoints provides the “connect nearest” mode. Clicking on the image will automatically connect the new marker to the closest Track in the last frame. Speeding up tracking for low track density scenes. The dragging of markers is still support and is usefull for intersecting tracks.

To activate “connect nearest” mode, set the config parameter `tracking_connect_nearest = True`.

See [ConfigFiles](#) for more details.

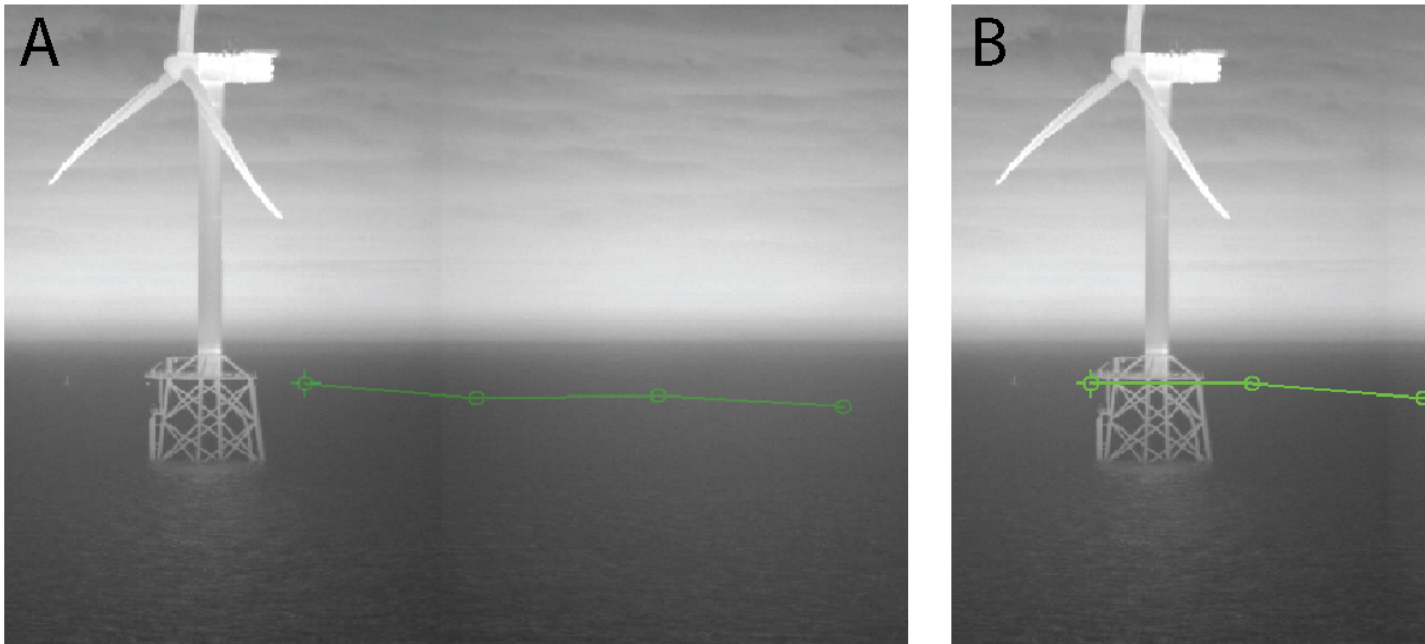


Fig. 3: Figure 2 | Track States

A - Track without update in current frame B - Track with update in current frame C - Track with missing marker

## 7. Important Controls

A list of useful controls for labeling tracks. Connect-nearest mode extends the list of default controls

- **default** left click - create new track (default mode) ctrl + left click - remove marker right click - open marker menu, see XXXXX
- **connect-nearest mode** left click - place marker, autoconnect to nearest track alt + left click - create new track shift + left click - place marker & load next frame

## 8. Advances Options

- Use SmartText to display additional information

See [SmartText](#)

### Example: Display Track IDs

- open the marker menu
- navigate to “bird” marker type
- edit the text field by inserting

```
$track_id
```

All current markers of the type `bird` now display their internal track ID

- Use Styles to modify the display of markers and tracks

See [Marker Styles](#)





Fig. 4: Figure 3 | Tracks with SmartText ID

**Example: Change track point display**

- open the marker menu
- navigate to “bird” marker type
- edit the style field by inserting

```
{"track-line-style": "dash", "track-point-shape": "none"}
```

All tracks of the type `bird` now are displayed with dashed lines and without track points



Fig. 5: Figure 3 | Tracks with modified style

## 3.4 Taking Measurements

How to perform quick measurements with ClickPoints and SmartText Markers

The modules are the different functions of the ClickPoints program. They can be accessed by the icons in the upper panel.




Fig. 1: The icon panel where all modules can be accessed.

## 4.1 Timeline

ClickPoints provides two timelines for navigation, a frame based and a timestamp based timeline. The frame based timeline is used by default, the timestamp timeline can be activated if time information of the displayed files is available. Time information extraction is implemented for the filename, the EXIF or TIFF headers.

### 4.1.1 Frame Timeline

The timeline is an interface at the bottom of the screen which displays range of currently loaded frames and allows for navigation through these frames. It can be displayed by clicking on .

To start/stop playback use the playback button at the left of the timeline or press `Space`. The label next to it displays which frame is currently displayed and how many frames the frame list has in total. The time bar has one slider to denote the currently selected frame and two triangular marker to select start and end frame of the playback. The keys `b` and `n` set the start/end marker to the current frame. The two tick boxes at the right contain the current frame rate and the number of frames to skip during playback between each frame. To go directly to a desired frame simply click on the frame display (left) and enter the frame number.

Each frame which has selected marker or masks is marked with a green tick mark (see [Marker](#) and [Mask](#)) and each frame marked with an annotation (see [Annotations](#)) is marked with a red tick. To jump to the next annotated frame press `Ctrl+Left` or `Ctrl+Right`.



Fig. 2: Frame Timeline example showing tick marks for marker and annotations.

### Config Parameter

- **fps** = (int, value  $\geq 0$ ) if not 0 overwrite the frame rate of the video
- **play\_start** = (float)
  - $> 1$ : at which frame to start playback at what
  - $0 > \text{value} < 1$ : fraction of the video to start playback
- **play\_end** =
  - $> 1$ : at which frame to start playback at what
  - $0 > \text{value} < 1$ : fraction of the video to start playback
- **playing** = (bool) whether to start playback at the program start
- **timeline\_hide** = (bool) whether to hide the timeline at the program start

### Keys

- H - hide control elements
- Space - run/pause
- Ctrl + Left - previous image with marker or annotation
- Ctrl + Right - next image with marker or annotation

### 4.1.2 Date Timeline

The date timeline displays the timestamps of the loaded data set. To navigate to desired time point simply drag the current position marker or click on the point on the date timeline. The timeline can be panned and zoomed by holding the left mouse button (pan) and the mouse wheel (zoom). It aims to make it easier to get an idea of the time distribution of the data set, to find sections of missing data and facilitate navigation by a more meaningful metric than frames.

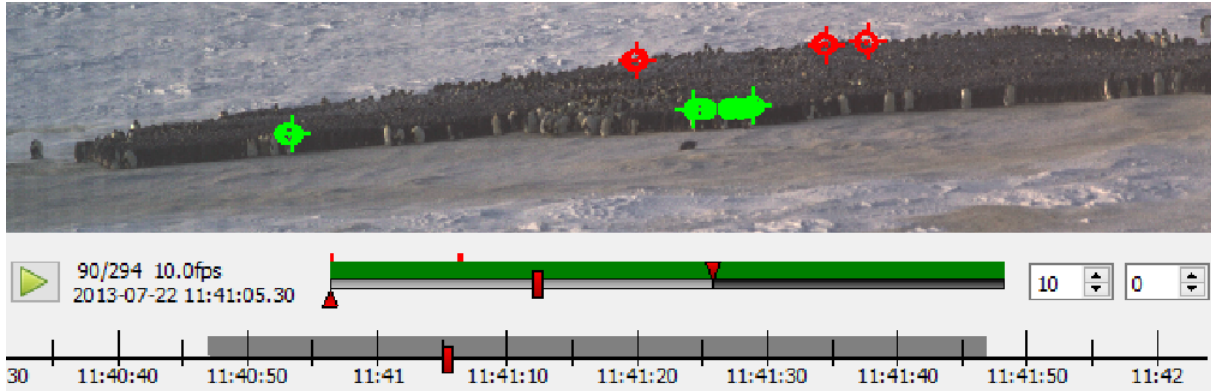


Fig. 3: Date Timeline example

The extraction of timestamps by filename is fast than by EXIF. If you plan to repeatedly open files, without using a `.cdb` to store the time stamps, renaming them once might be beneficial. A list of timestamp search strings can be specified in the config file as shown in the code example below. As the search will be canceled after the first match it is necessary to order the the search strings by decreasing complexity.

Recommended naming template: `%Y%m%d-%H%M%S-%f_Location_Camera.type`


### Config Parameter

- **datetimeline\_show** = (bool) enable or disable the date timeline by setting this value to True or False
- **timestamp\_formats** = (list of strings) list of match strings for images, with decreasing complexity
- **timestamp\_formats2** = (list of strings)

list of match strings for videos with 2 timestamps, with decreasing complexity

```
# default values:
# for image formats with 1 timestamp
timestamp_formats = [r'%Y%m%d-%H%M%S-%f',      # e.g. 20160531-120011-2   with ↵
                    ↵fraction of second
                    r'%Y%m%d-%H%M%S']          # e.g. 20160531-120011
# for video formats with 2 timestamps (start & end)
timestamp_formats2 = [r'%Y%m%d-%H%M%S_%Y%m%d-%H%M%S']
```

## 4.2 GammaCorrection

The gamma correction is a slider box in the right bottom corner which allows to change the brightness and gamma of the currently displayed image. It can be opened by clicking on .

The box in the bottom right corner shows the current gamma and brightness adjustment. Moving a slider changes the display of the currently selected region in the images. The background of the box displays a histogram of brightness values of the current image region and a red line denoting the histogram transform given by the gamma and brightness adjustment. Pressing update the key `G` sets the currently visible region of the image as the active region for the adjustment. Especially for large images it increases performance significantly if only a portion of the image is adjusted. A click on `reset` resets gamma and brightness adjustments.



Fig. 4: An example gamma correction.

### 4.2.1 Gamma

The gamma value changes how bright and dark regions of the images are treated. A low gamma value ( $<1$ ) brightens the dark regions up while leaving the bright regions as they are. A high gamma value ( $>1$ ) darkens the dark regions of the image while leaving the bright regions as they are.

### 4.2.2 Brightness

The brightness can be adjusted by selecting the Max and Min values. Increasing the Min value darkens the image by setting the Min value (and everything below) to zero intensity. Decreasing the Max value brightens the image by setting the Max value (and everything above) to maximum intensity.

### 4.2.3 Keys

- G: update rect

## 4.3 VideoExporter

The video exporter allows for the export of parts of the currently loaded images as a video, image sequence or gif file.


It can be opened using the  or by pressing z. A dialog will open, which allows to select an output filename for a video, an image sequence (which has to contain a %d number placeholder) or a gif file. Frames are exported starting from the start marker in the timeline to the end marker in the timeline. The framerate is also taken from the timeline. Images are cropped according to the current visible image part in the main window.



Fig. 5: The same image for different gamma values or 1, 0.5 and 1.5.



Fig. 6: The same image for different brightness values, where once the lower and once the upper range was adjusted.

### 4.3.1 Keys

- Z: Export Video

## 4.4 Annotations

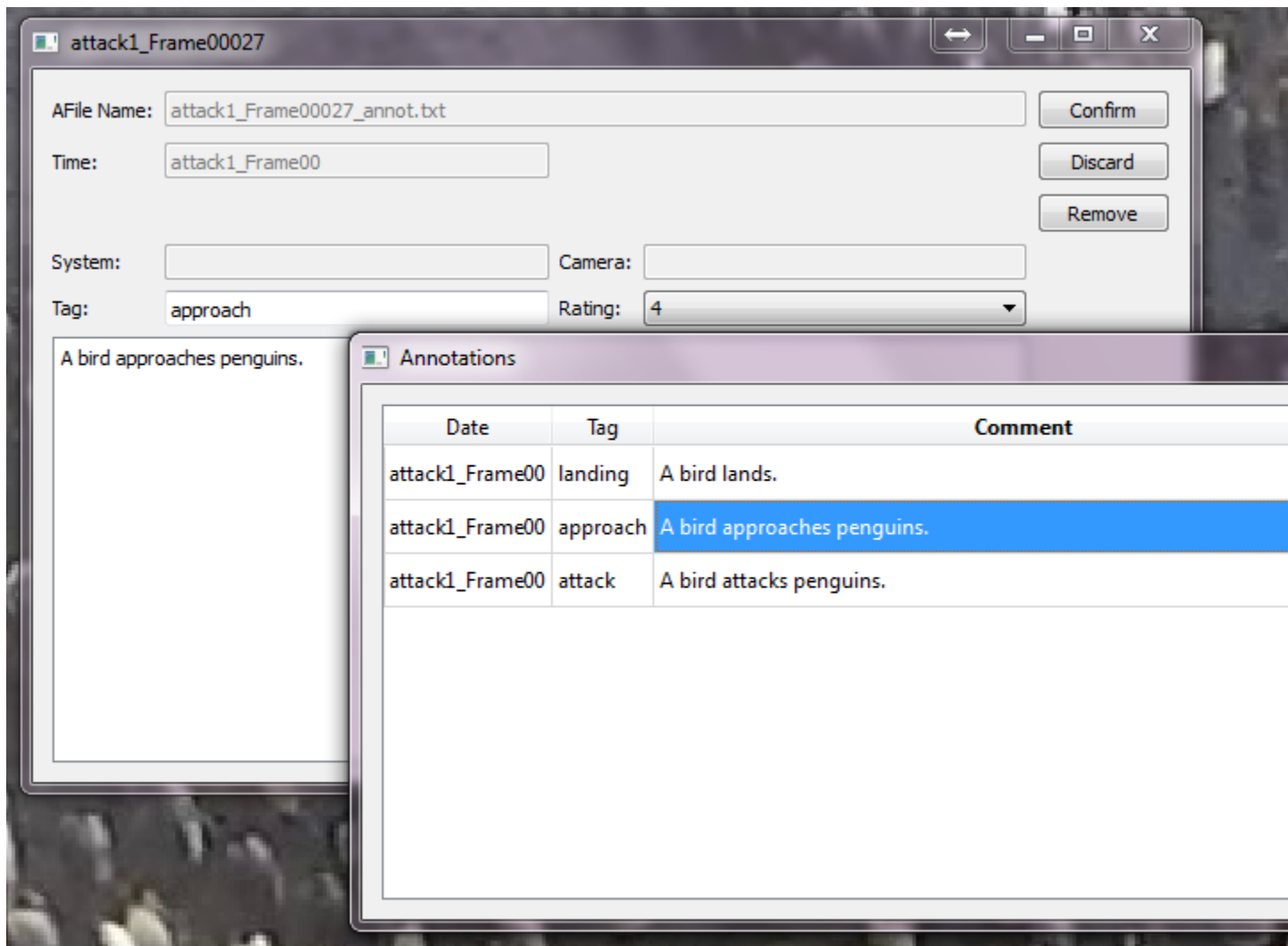



Fig. 7: An example of both the annotation editor and the annotation overview window.

Annotations are text comments which can include a rating and tags, which is attached to a frame. To annotate a frame or edit the annotation of a frame press A or  and fill in the information in the dialog. The frame will be marked with a red tick in the timeline. To get a list of all annotated frames press Y. In this list clicking an annotation results in a jump to the frame of the annotation.

### 4.4.1 Keys

- A: add/edit annotation



- Y: show annotation overview

## 4.5 Marker

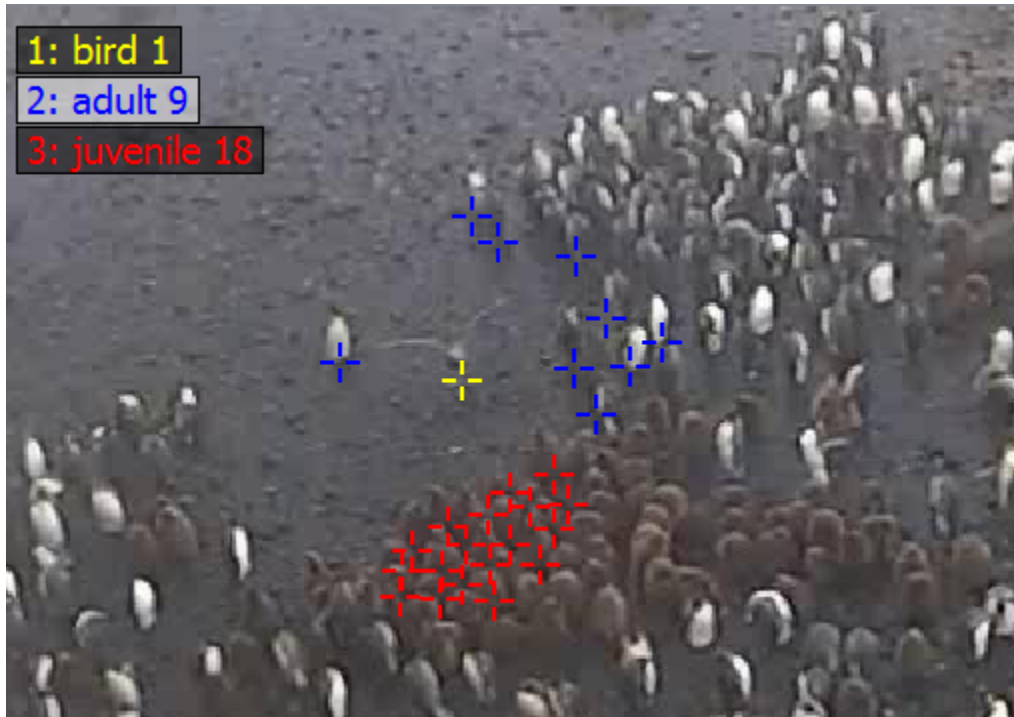



Fig. 8: An example image showing three different marker types and some markers placed on the image.

Markers are added to a frame to refer to pixel positions. Markers can have different types to mark different objects. They can also be used in tracking mode to recognize an object over different frames.

The marker editor can be opened by clicking on .

The list of available markers is displayed at the top left corner. A marker type can be selected either by clicking on its name or by pressing the corresponding number key. A left click in the image places a new marker of the currently selected type. Existing markers can be dragged with the left mouse button and deleted by clicking on them while holding the control key.

To save the markers press S or change to the next image, which automatically saves the current markers.

**MB1** place a marker or track point at the current mouse position

**ctrl + MB1** delete the marker under the mouse pointer

**MB2** open the marker editor

### 4.5.1 Marker types

A right click on any marker or type opens the Marker Editor window. There types can be created, modified or deleted.

Marker types have a name, which is displayed in the HUD, a color and a mode.



Fig. 9: Different marker type modes.

TYPE\_Normal results in single markers. TYPE\_Rect joins every two consecutive markers as a rectangle. TYPE\_Line joins every two consecutive markers as a line. TYPE\_Track specifies that this markers should use tracking mode (see section Tracking Mode).

### 4.5.2 Marker display

Pressing **T** toggles between three different marker displays. If the smallest size is selected, the markers can't be moved. This makes it easier to work with a lot of markers on a small area.



Fig. 10: The same marker in different size configurations.

### 4.5.3 Tracking mode

Often objects which occur in one image also occur in another image (e.g. the images are part of a video). Then it is necessary to make a connection between the object in the first image and the object in the second image. Therefore ClickPoints features a tracking mode, where markers can be associated between images. It can be enabled using the TYPE\_Track for a marker type. The following images displays the difference between normal mode (left) and tracking mode (right):

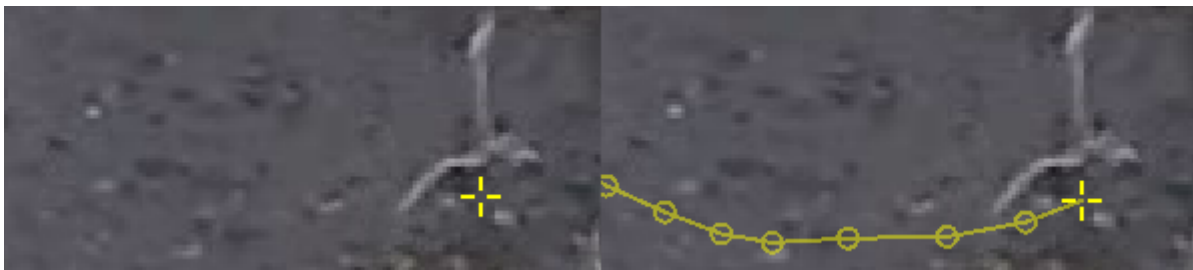


Fig. 11: The same marker in normal mode (left) and in tracking mode (right). The track always displays all previous positions connected with a line, when they are from two consecutive images.

To start a track, mark the object in the first image. Then switch to the next image and the marker from the first image will still be displayed but only half transparent. To add a second point to the track grab the marker and move it to the new position of the object. Continue this process through the images where you want to track the object. If the object didn't move from the last frame or isn't visible, an image can be left out, which results in a gap in the track. To remove a point from the track, click it while holding control.

#### 4.5.4 Marker Editor

The Marker Editor is used to manage marker types. New marker types can be created, existing ones can be modified or deleted.

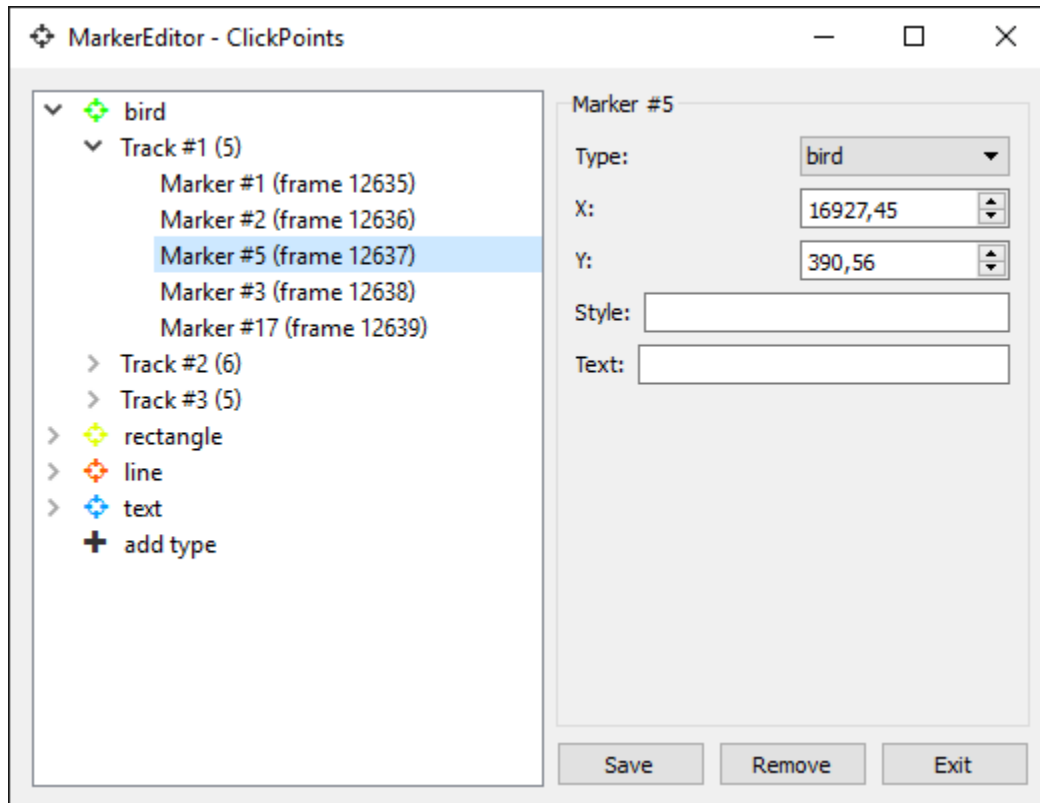



Fig. 12: The Marker Editor used to create and change marker types, navigate to tracks and marks and delete marker, tracks and types

**Creating Marker Types** To create a new marker type open the marker editor via  or right click on the marker display or a marker. Select the +add type field, enter a name, set the marker mode to marker, line, rectangle or track and choose a color. Further modifications can be achieved via the text and style field, for more details see the following sections.

**Editing Marker Types** To edit a marker type, simply select the type from the menu, changes the desired values and save the changes by pressing **Save**

---

**Note:** It is NOT possible to change marker types as long as marker objects of this type exist. E.g. you can't make lines out of regular markers as they don't have a second point.

---

**Navigation** The editor can also be used to navigate. Selecting a marker will bring you to the frame the marker is placed in. By clicking on the arrow in front of the type name the marker or track overview unfolds. Selecting a marker of a track will bring you to the frame it is placed in.

**Deleting Types, Tracks and Markers** Types, tracks and markers can be removed by selecting the object in the tree and pressing the `Remove` button. By removing a marker type all markers and tracks of this type are removed, removing a track will remove all markers of this track.

**Warning:** There is no undo button!

## 4.5.5 Marker Style Definitions

Style definitions can provide additional features to change the appearance of marker. They are inherited from the marker type to the track and from the track to the marker itself. If no track is present the marker inherits its style directly from the type. This allows to define type, track and marker specific styles.

Styles can be set using the Marker Editor (right click on any marker or type).

The styles use the JSON format for data storage. The following fields can be used:

- **Marker Color - "color":** `"#FF0000"` Defines the color of the marker in hex format. Color can also be a `matplotlib` colormap followed optionally by a number (e.g. `jet(30)`), then that many colors (default 100) are extracted from the color map and used for the marker/tracks to color every marker/track differently.
- **Marker Shape - "shape":** `"cross"` Defines the shape of the marker. All shapes can be converted to outlines by appending `"-o"` to the name.  
*values:* `cross` (default), `circle`, `ring`, `rect`
- **Marker Line Width - "line-width":** `1` Defines the line width of the markers symbol (e.g. width of the circle). Ignored if a filled symbol (e.g. the cross) is used.
- **Marker Scale - "scale":** `1` Scaling of the marker.
- **Marker Transform - "transform":** `"screen"` If the marker should have a fixed size with respect to the screen or the image.  
*values:* `screen` (default), `image`
- **Track Line Style - "track-line-style":** `"solid"` The style of the line used to display the track history.  
*values:* `solid` (default), `dash`, `dot`, `dashdot`, `dashdotdot`
- **Track Line Width - "track-line-width":** `2` The line width of the line used to display the track history.
- **Track Gap Line Style - "track-gap-line-style":** `dash` The style of the line used to display gaps in the track history.  
*values:* `solid`, `dash` (default), `dot`, `dashdot`, `dashdotdot`
- **Track Gap Line Width - "track-gap-line-width":** `2` The line width of the line used to display gaps in the track history.
- **Track Marker Shape - "track-point-shape":** `"circle"` The marker shape used to display the track history.  
*values:* `circle`, `ring` (default), `rect`, `cross`, `none`

- **Track Marker Scale - "track-point-scale": 1** The scaling of markers used to display the track history.

#### Style Examples:

```
{ "color": "jet(30)" } # style for providing a marker type with 30 different colors
{ "track-line-style": "dash", "track-point-shape": "none" } # change the track style
```

### 4.5.6 Marker Text & SmartText

The text field allows to attach text to marker, line, rectangle and track objects. Text properties are inherited from the marker type to the track and from the track to the marker itself. If no track is present the marker inherits its text directly from the type. This allows to define type, track and marker specific texts.

Text can be set using the Marker Editor (right click on any marker or type).

ClickPoints provides a SmartText feature, enabling the display of self updating text in to display pre defined values. SmartText keyword always start with a \$ character. The keywords are depending on the type for marker, as explained in the following overview:

#### General

`\n` insert a new line

`$marker_id` inserts the id of the marker, line or rectangle object

`$x_pos` inserts the x position of the marker, first marker of a line or top left marker of a rectangle

`$y_pos` inserts the y position of the marker, first marker of a line or top left marker of a rectangle

#### Line

`$length` inserts the length of the line in pixel with 2 decimals.

#### Rectangle

`$area` inserts the area of the rectangle in pixel with 2 decimals.

#### Track

`$track_id` inserts the track id of the track.

#### Text Examples:

```
# regular Text
Marker: "Hello World!" # shows the text Hello World!

# SmartText
Track: "ID_$track_id" # shows the track ID
Line: "$x_pos | $y_pos \n$length px" # shows the x & y coordinate and
↳length
Rect: "ID_$marker_id\n$x_pos | $y_pos \n$area px²" # shows the object_id, its x & y
↳coordinate and area
```

## 4.6 Mask

A mask can be painted to mark regions in the image with a paint brush. The mask editor can be opened by clicking on



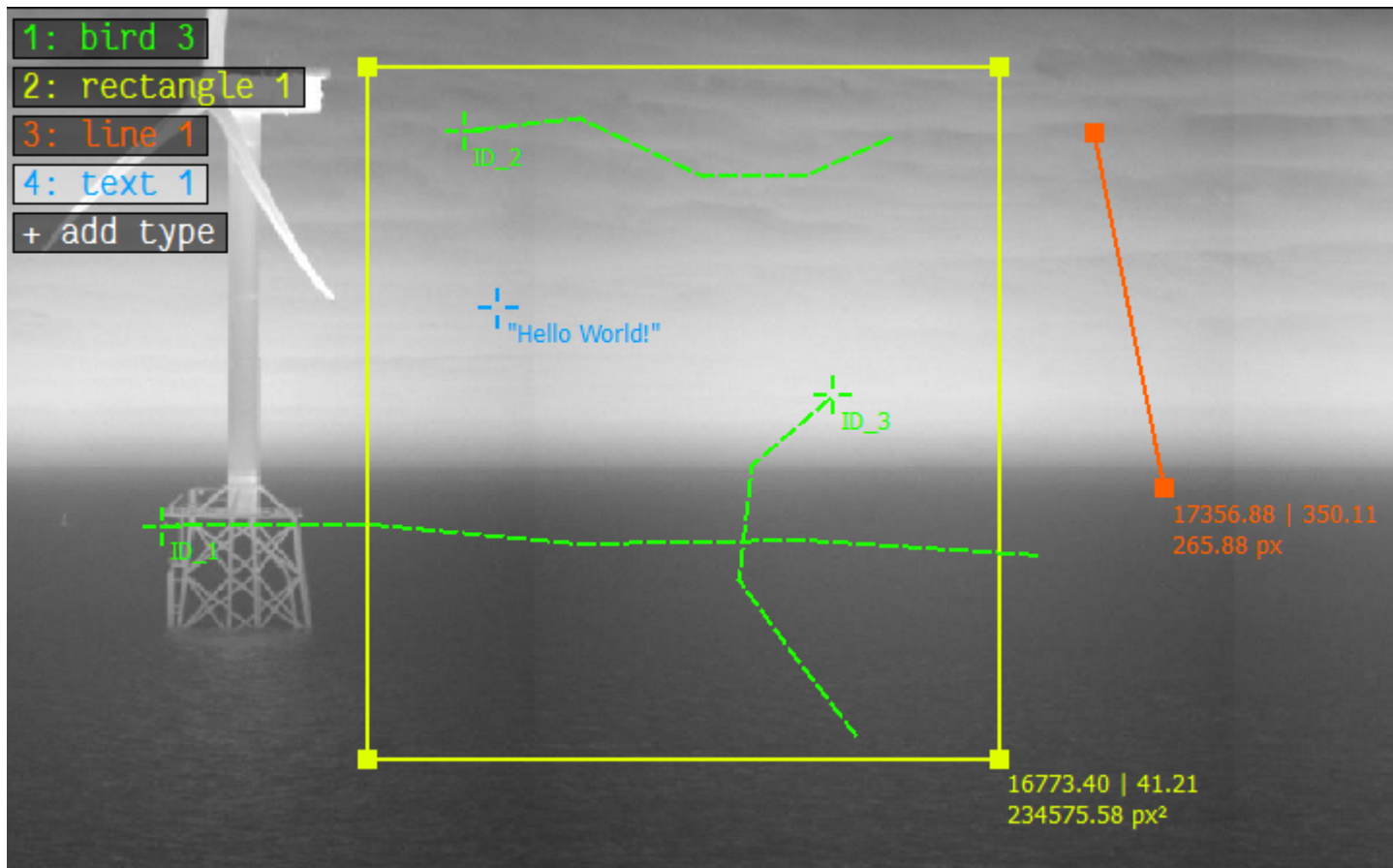


Fig. 13: Using regular text and SmartText features for lines, rectangles and tracks

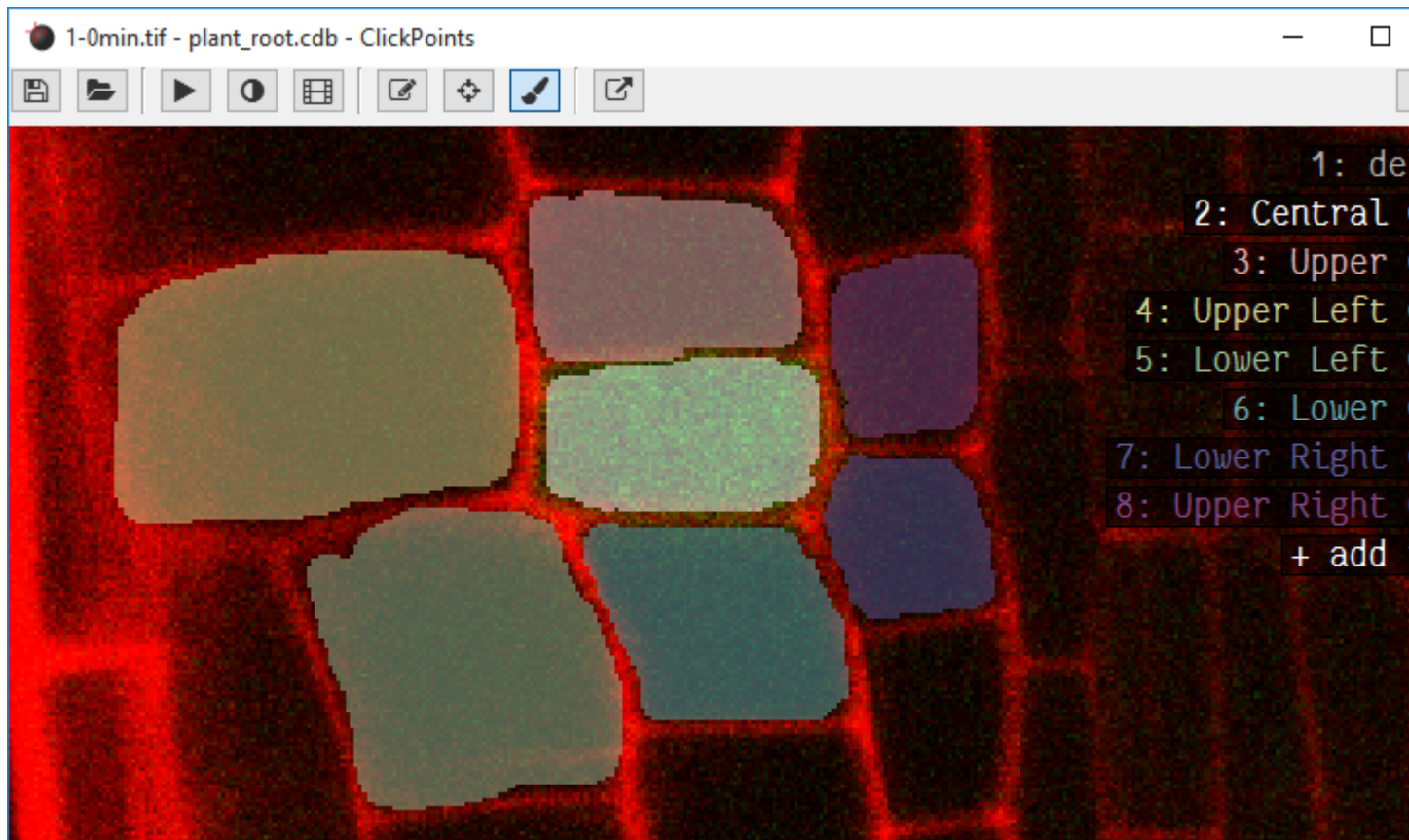


Fig. 14: An image where 7 regions have been marked with different masks.

A list of available mask colors is displayed in the top right corner. Switching to paint mode can be done using the key `P`, pressing it again switches back to marker mode. Colors can be selected by clicking on its name or pressing the corresponding number key. Holding the left mouse button down draws a line in the mask using the selected color. To save the mask press `S` or change to the next image, which automatically saves the current mask. The mask type `delete` acts as an eraser and allows to remove labeled regions.

### 4.6.1 Define colors

A right click on a color name opens the mask editor menu, which allows the creation, modification and deletion of mask types. Every mask type consists of a name and a color.

### 4.6.2 Brush size

The brush radius can be de- and increased using the keys `-` and `+`.

### 4.6.3 Color picker

The color can alternatively to selection via number buttons or a click on the names be selected by using `K` to select the color which is currently below the cursor.

### 4.6.4 Mask transparency

The transparency of the mask can be adjusted with the keys `I` and `O`.

### 4.6.5 Mask update

Updating masks can be slow if the images are very large. To enable fast painting of large masks, ClickPoints can disable the automatic updates of the mask by disabling the option `Auto Mask Update`. If automatic updates are disabled the key `M` redraws the currently displayed mask.

### 4.6.6 Config Parameter

- `auto_mask_update` = whether to update the mask display after each stroke or manually by key press
- `draw_types` = `[[0, [255, 0, 0]]` specifies what categories to use for mask drawing. Every category is an array with two entries: index and color.

### 4.6.7 Keys

- `0-9`: change brush type
- `K`: pick color of brush
- `-`: decrease brush radius
- `+`: increase brush radius
- `O`: increase mask transparency
- `I`: decrease mask transparency
- `M`: redraw the mask



## 4.7 Info Hud

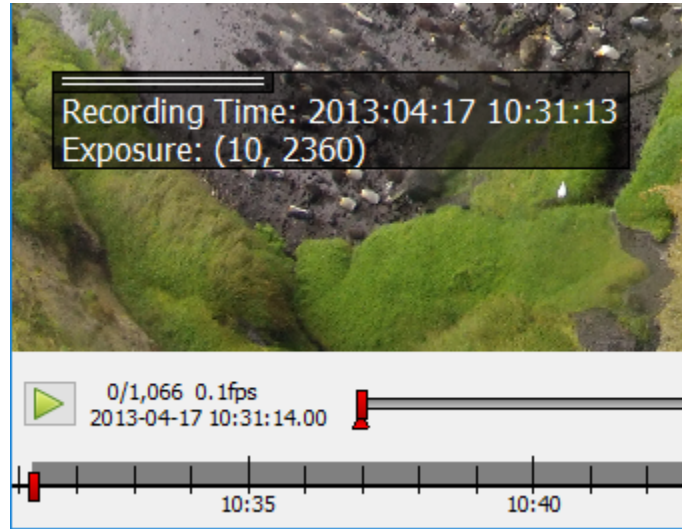


Fig. 15: Example of the info hud displaying time and exposure exif data from a jpg file.

This info hud can display additional information for each image. Information can be obtained from the filename, jpeg exif information or tiff metadata or be provided by an external script.

The text can be set using the options dialog. Placeholders for additional information are written with curly brackets {}. The keyword from the source (regex, exif or meta) is followed by the name of the information in brackets [], e.g. {exif[rating]}. If the text is set to @script the info hud can be filled using an external script. Use \n to start a new line.

To extract data from the filename a regular expression with named fields has to be provided.

### 4.7.1 Examples

#### Data from filename

```
file: "penguins_5min.jpg"

Info Text: "Animal: {regex[animal]} Time: {regex[time]}"
Filename Regex: '(?P<animal>.+?(^_))_(?P<time>.+)'

Output: "Animal: penguin Time: 5"
```

#### Data from exif

```
file: "P1000236.jpg"

Info Text: "Recording Time: {exif[DateTime]} Exposure: {exif[ExposureTime]}"

Output: "Recording Time: 2016:09:13 10:31:13 Exposure: (10, 2360)"
```

The keys can be any field of the jpeg exif header as e.g. shown at <http://www.exiv2.org/tags.html>

## Data from meta


```
file: "20160913_134103.tif"


Info Text: "Magnification: {meta[magnification]} PixelSize: {meta[pixelsize]}"

Output: "Magnification: 10 PixelSize: 6.45"
```

The values presented in the meta field of tiff files varies by the tiff writer. ClickPoints can only access tiff meta data written in the json format in the tiff meta header field, as done by the `tifffile` python package.

Add-ons are helpful scripts which are not part of the main ClickPoints program, but can be loaded on demand to do some evaluation task.

They can be loaded by clicking on  and selecting the add-on from the list. ClickPoints already comes with a couple of add-ons, but it is easy to add your own or extend existing ones.


Each add-on will be assigned to a key from F12 downwards (F12, F11, F10 and so on) and a button will appear for each add-on next to . Hitting the key or pressing the button will start the `run` function of the add-on in a separate thread, or tell the thread to stop if it is already running.

To configure ClickPoints to already have scripts loaded on startup, you can define them in the `ConfigClickPoints.txt` file as `launch_scripts =`.

For writing your own add-ons please refer to the [add-on api](#).

## 5.1 Tracking

This add-on takes markers in one image and tries to find the corresponding image parts in the subsequent images.

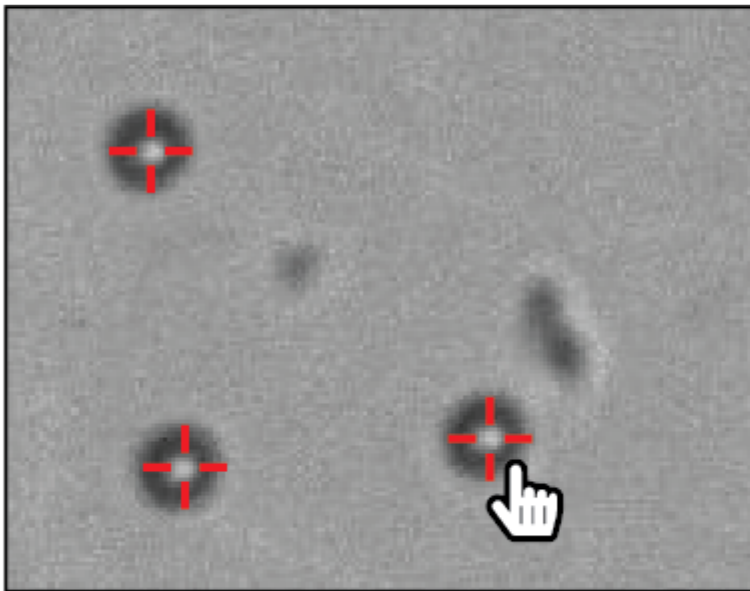
To use it, open a ClickPoints session and add the add-on `Track.py` by clicking on .

Create a marker type with mode `TYPE_Track`. Mark every object which should be tracked with a marker of this type. Then hit F12 (or the button you assigned the `Track.py` to) and watch the objects to be tracked. You can at any point hit the key again to stop the tracking. If the tracker has made errors, you can move the marker by hand and restart the tracking from the new position.

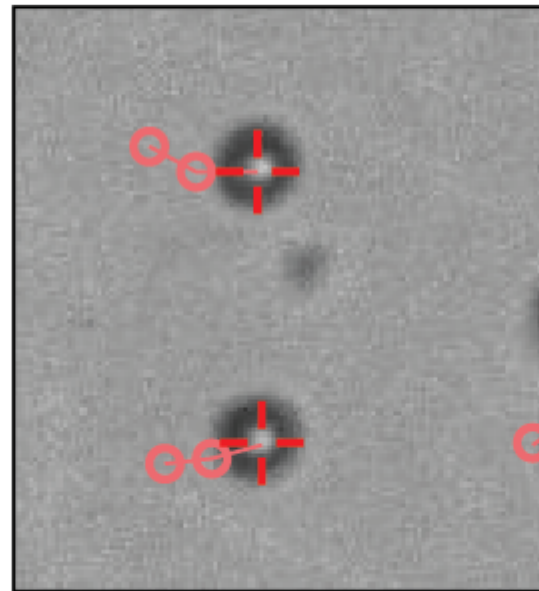
The algorithm uses the position using a sparse iterative Lucas-Kanade optical flow algorithm [bouguet2001pyramidal].

**Attention:** If the markers are not in a `TYPE_Tracking` type, they are not tracked by `Track.py`. Also marker which already have been tracked are only tracked again, if they were moved in ClickPoints.

select starting  
points for tracks



start tracking algo  
(add-on script)



## References

## 5.2 Fluorescence Diffusion

This add-on implements diffusion of fluorescence proteins between adjacent cells.


The add-on can be used as follows:

1. add the add-on “Fluorescence Diffusion”
2. create a mask-type for each cell (see module [Mask](#) on creating mask-types)
3. paint the cell area for each cell in each image
4. add line marker of the type “connect” to mark between which cells diffusion is allowed. (Only necessary for the first image)
5. specify the “Delta T”, the time between two images
6. specify the image channel to use (0: Red, 1: Green, 2: Blue)
7. click on “Calculate Intensities” to obtain the intensity values for the masked regions
8. click on “Calculate Diffusion” to obtain the diffusion values for the links

### 5.2.1 1. Add the add-on

Open a ClickPoints session and add the add-on `Fluorescence Diffusion` by clicking on .

### 5.2.2 2. Create a mask-type for each cell

Click on  and select the button “+ add type” to add a mask type for each cell.

### 5.2.3 3. Paint the cell area for each cell in each image

Click on the paint brush and the name of the mask-type. Then paint the area of the cell in the image. Use the arrow keys to navigate through the images and paint the cell in each image. From these regions the green image channel will be summed as the total fluorescence intensity of the cell at this time.

Repeat this process for all images.

**See also:**

For more information on the usage of masks, see the page on [Masks](#).

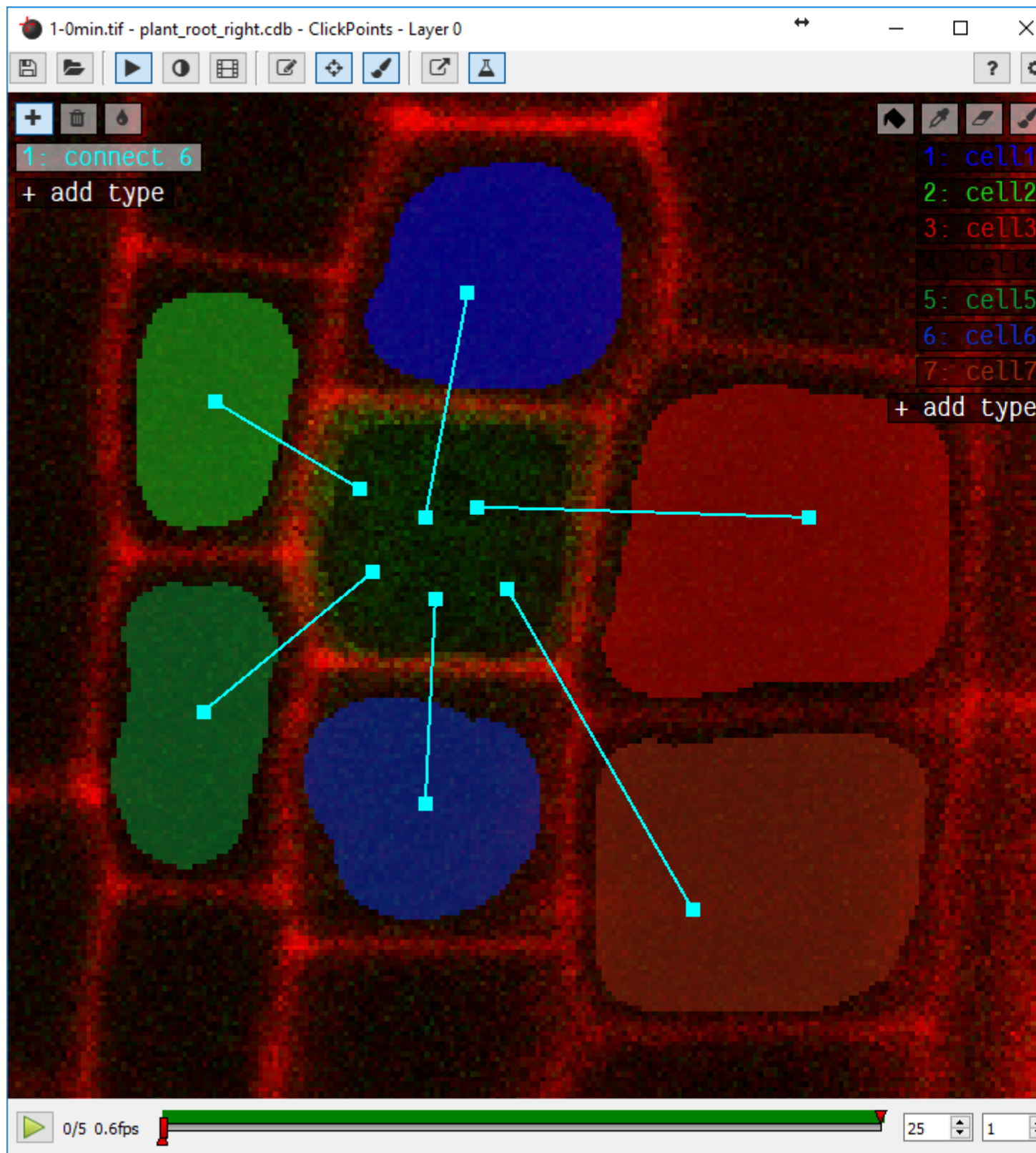
### 5.2.4 4. Specify the links between cells

Click on the button “connect” on the left side of the window. Now click on one cell and drag the mouse to another cell that should be connected to the first cell. Repeat this for all cells you want to link. Links indicate that diffusion between these cells is allowed and a diffusion value will be fitted for this link later.

Links are only needed in one image of the sequence, not for all images.

**See also:**

For more information on the usage of markers, see the page on [Markers](#).



### 5.2.5 5. Specify the time between two images

Open the add-on by clicking on . Add a value for “Delta T”, which specifies the time between two subsequent images in seconds.

### 5.2.6 6. Specify which image channel to use

In the add-on window, specify the value “Color Channel”, which specifies which color channel to use: 0: Red, 1: Green, 2: Blue.

### 5.2.7 7. Calculate Intensities


Click in the button “Calculate Intensities”. The add-on sums all pixel intensity values for the regions covered by the masks. The outputs can be seen in the table directly under the button. The graph right of the table shows the intensities over time.

### 5.2.8 8. Calculate Diffusion

Click on the button “Calculate Diffusion”. The add-on tries to find diffusion constants, so that the diffusion equation describes the change of the fluorescence intensities in the cells. The resulting values are printed in the table under the button and the simulated diffusion are shown in the graph right of the table.

## 5.3 Drift Correction


This add-on takes a region in the image and tries to find it in every image. The offset saved for every image to correct for drift in the video.

To use it, open a ClickPoints session and add the add-on `DriftCorrection.py` by clicking on .

When you first start the script a marker type named `drift_rect` is created. Use this type to select a region in the image which remains stable over the course of the video. Start the drift correction script by using F12 (or the key the script is connected to). The drift correction can be stopped and restarted at any time using the key again.

## 5.4 Cell Detector

This add-on is designed to take a microscope image of fluorescently labeled cell nuclei and find the coordinates of every cell.

To use it, open a ClickPoints session and add the add-on `CellDetector.py` by clicking on .

Start the cell detector script by using F12 (or the key the script is connected to). All found cell nuclei will be labeled with a marker.

**Attention:** The Cell Detector won’t work for cells which are too densely clustered. But ClickPoints allows you to review and adjust the results if some cells were not detected.

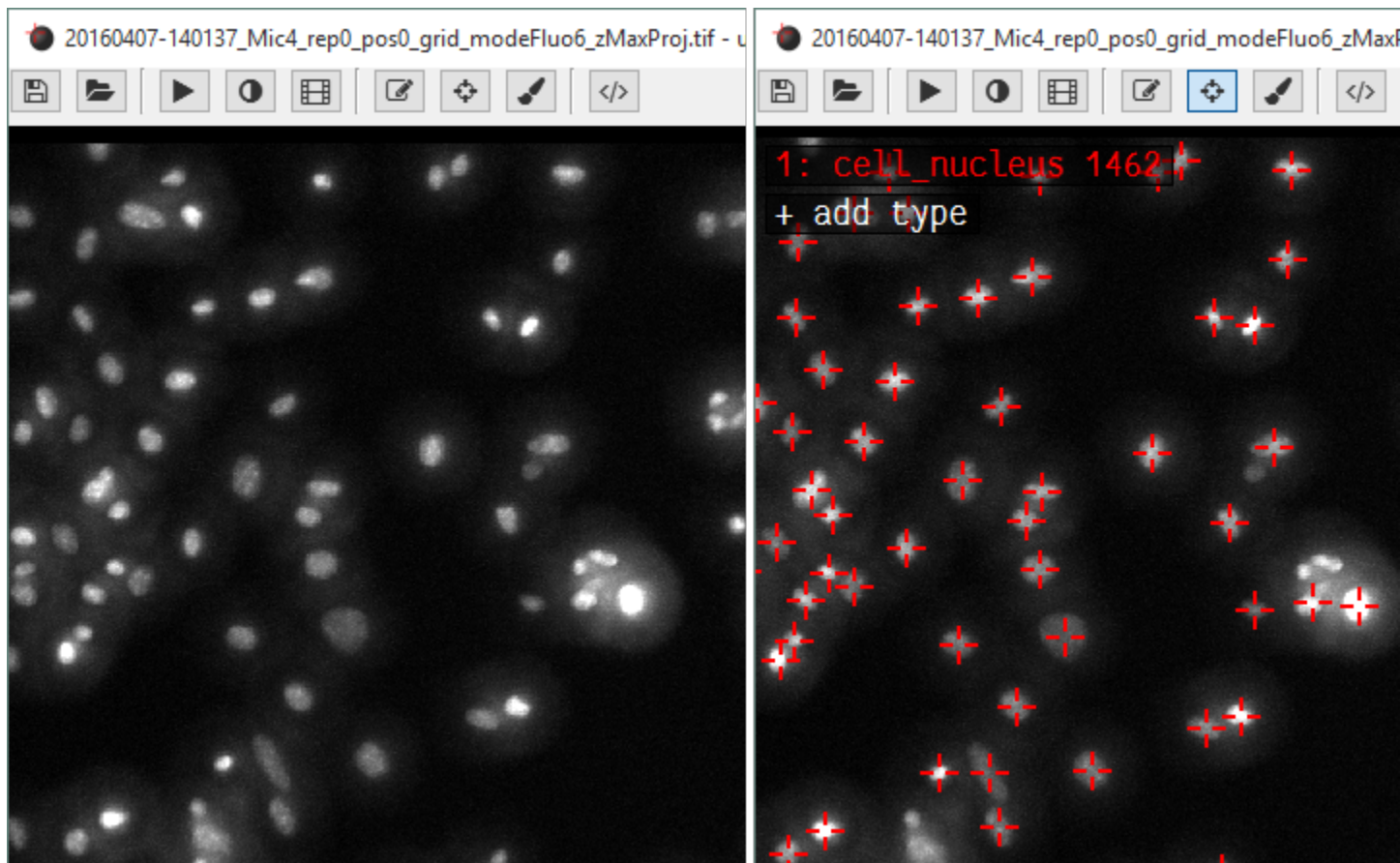



Fig. 1: An image of cell nuclei before and after executing the Cell Detector addon.



## 5.5 Grab Plot Data

This add-on helps to retrieve data from plots.

To use it, open a ClickPoints session and add the add-on `GrabPlotData.py` by clicking on .

Sometimes it is useful to extract data from plotted results found in publications to compare them with own results or simulations. ClickPoints therefore provides the add-on “GrabPlotData”. It uses three marker types. The types “x\_axis” and “y\_axis” should be used to mark the beginning and end of the x and y axis of the plot. Markers should be assigned a text containing the value which is associated with this point on the axis. These axis markers are used to remap the pixel coordinates of the “data” markers to the values provided by the axis. These remapped values are stored in a “.txt” file that has the same name as the image.

**Attention:** This can only be used if the axe is not scaled logarithmically. Only linear axes are supported.

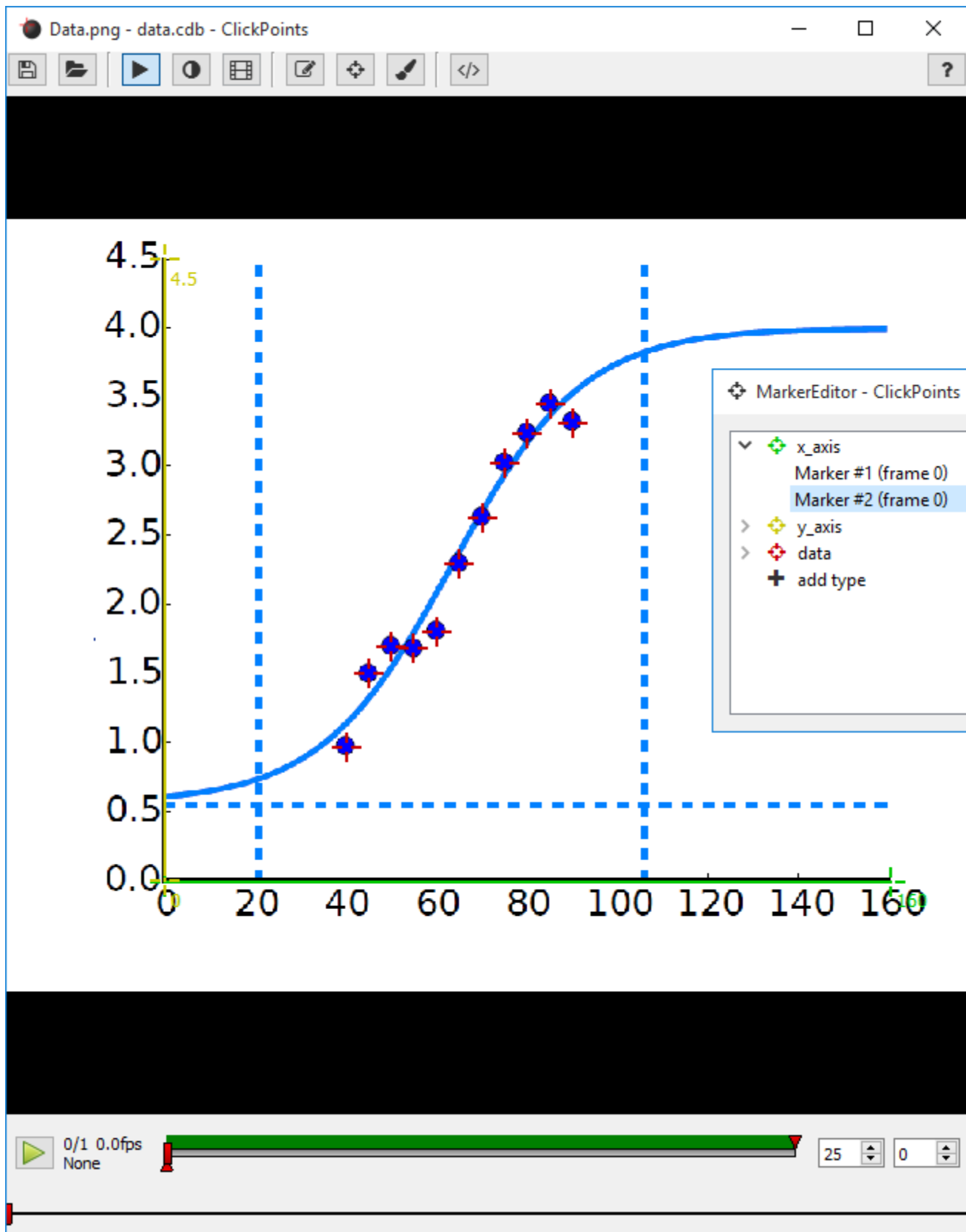


Fig. 2: The two axis are marked with the corresponding markers and the data points with the data markers. The start and end points of the axis are assigned a text containing the corresponding axis value.

The examples provide some usage examples of ClickPoints to demonstrate its various functionalities and how data can be processed with ClickPoints and later easily evaluated with ClickPoints.

To keep the download size of ClickPoints down, the examples are kept in a separate repository. They can be downloaded [here](#).

## 6.1 Count Animals

In the example, we show how the ClickPoints can be used to count animals in an image.

The example contains some images recorded with a GoPro Hero 2 camera, located at the Baie du Marin King penguin colony on Possession Island of the Crozet Archipelago [Bohec137]. Two marker types, “adult” and “juvenile” were added in ClickPoints to count two types of animals.

The counts can be evaluated using a small script.

Open the database where the animals were clicked.

```
[2]: %matplotlib inline
import matplotlib.pyplot as plt
import clickpoints

# open database
db = clickpoints.DataFile("count.cdb")

path count.cdb
Open database with version 18
```

Iterate over the images using `getImages()` to get *Image* objects. Then query the *Marker* objects for each image and for the two marker types (see `getMarkers()`)

```
[3]: # iterate over images
for index, image in enumerate(db.getImages()):
```

(continues on next page)

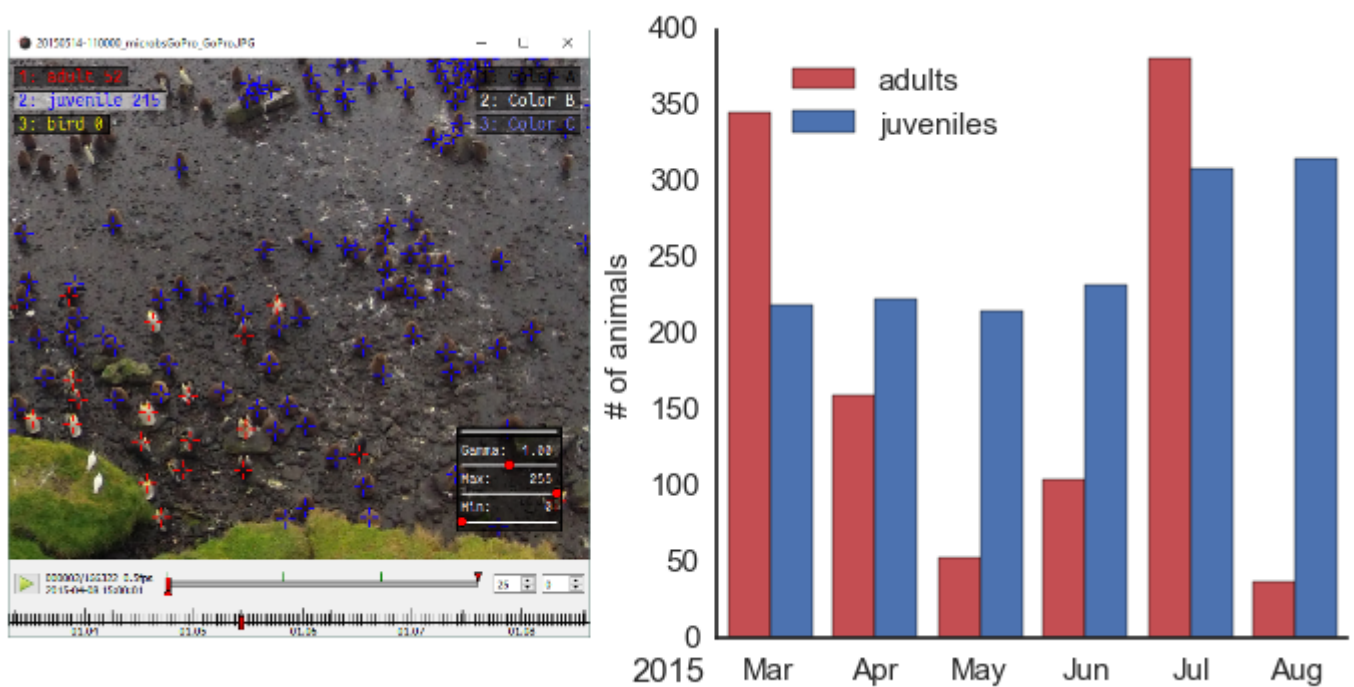


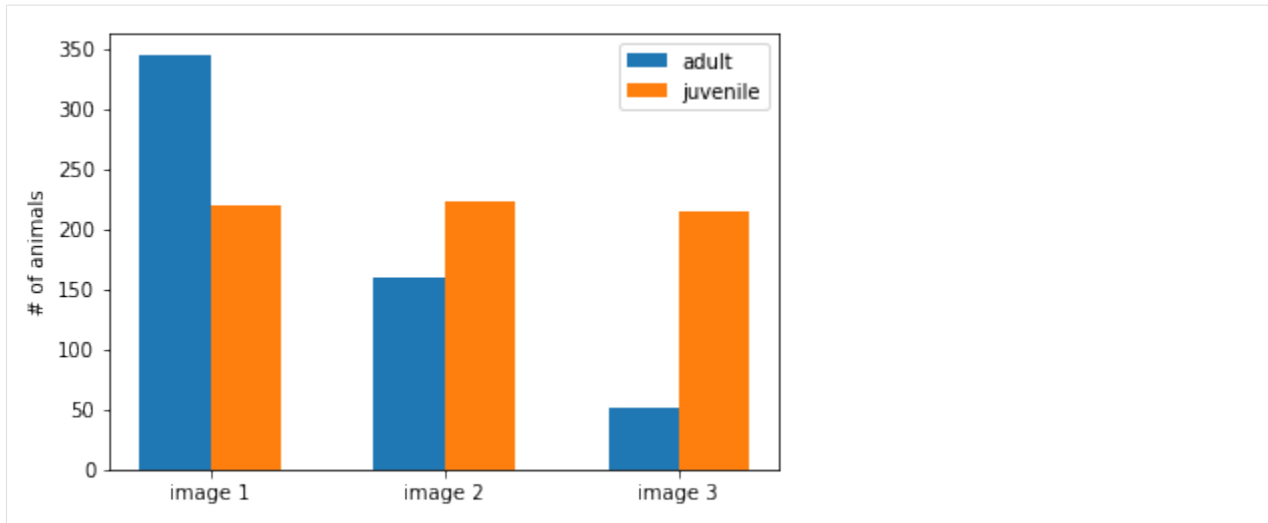
Fig. 1: Left: image of clickpoints to count penguins. Right: number of penguins counted.

(continued from previous page)

```
# get count of adults in current image
marker = db.getMarkers(image=image, type="adult")
bar1 = plt.bar(index-0.15, marker.count(), color='C0', width=0.3)

# get count of juveniles in current image
marker = db.getMarkers(image=image, type="juvenile")
bar2 = plt.bar(index+0.15, marker.count(), color='C1', width=0.3)

# add labels
plt.ylabel("# of animals")
plt.xticks([0, 1, 2], ["image 1", "image 2", "image 3"])
# add a legend
plt.legend([bar1[0], bar2[0]], ("adult", "juvenile"))
# display the plot
plt.show()
```



Now we want to plot an example image with the image and the markers as points in the image. We can now use `getImage()` to get the first image of the sequence and load the data from this file. This can now be displayed with `matplotlib`. Then we use the `image` and `type` keyword of the `getMarkers()` function to filter out only markers from this image and the given type.

```
[4]: # get the first image
im_entry = db.getImage(0)

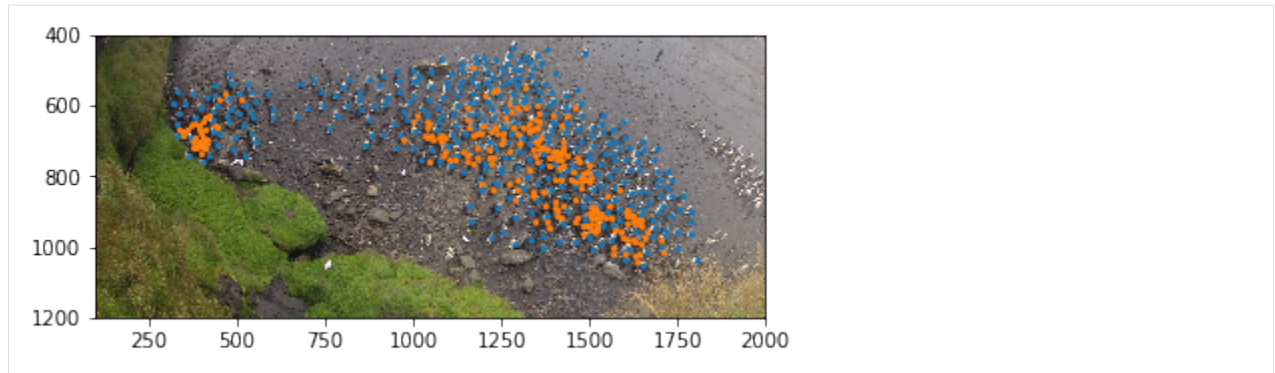
# we load the pixel data from the Image database entry
im_pixel = im_entry.data

# plot the image
plt.imshow(im_pixel, cmap="gray")

# iterate over the adults in the image
for marker in db.getMarkers(image=im_entry, type="adult"):
    # plot the coordinates of the marker
    plt.plot(marker.x, marker.y, 'C0o', ms=2)

# iterate over the juveniles in the image
for marker in db.getMarkers(image=im_entry, type="juvenile"):
    # plot the coordinates of the marker
    plt.plot(marker.x, marker.y, 'C1o', ms=2)

# zoom into the image
plt.xlim(100, 2000)
plt.ylim(1200, 400)
plt.show()
```



## References

### 6.2 Fluorescence intensities in plant roots

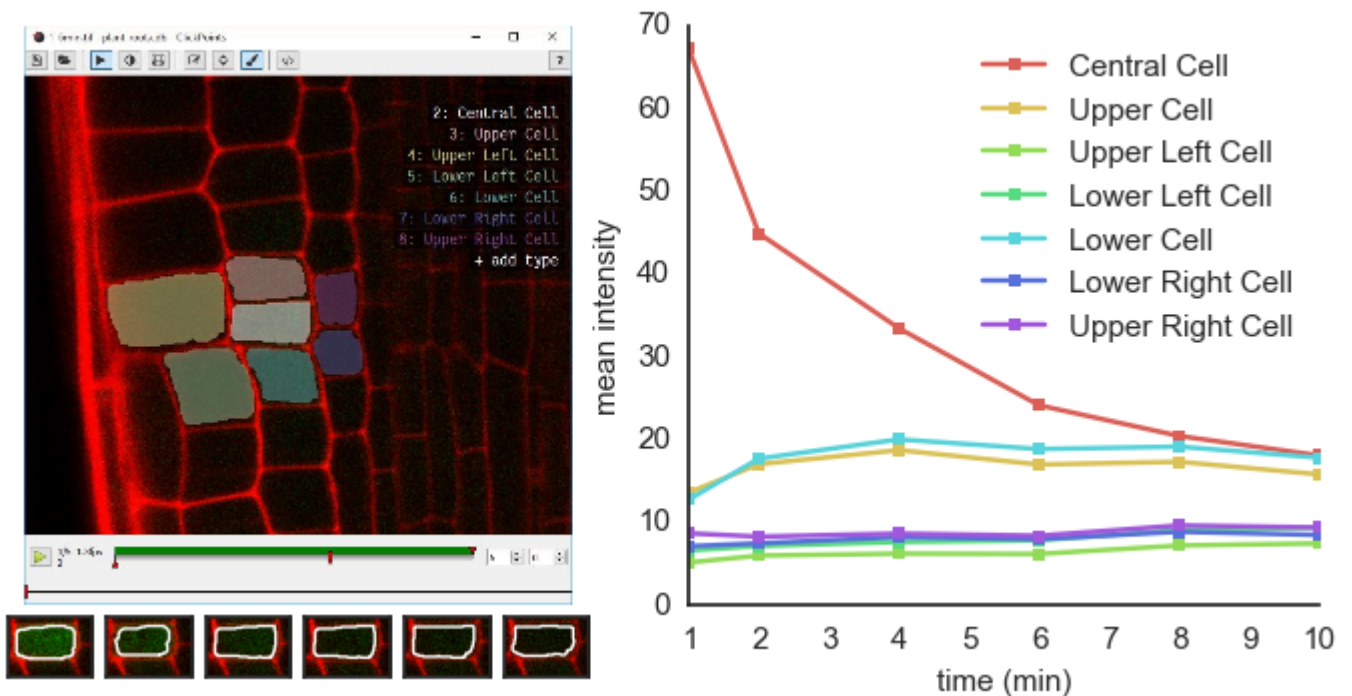


Fig. 2: Left: image of a plant root in ClickPoints. Right: fluorescence intensities of the cells over time.

In the example, we show how the mask painting feature of ClickPoints can be used to evaluate fluorescence intensities in microscope recordings.

Images of an *Arabidopsis thaliana* root tip, obtained using a two-photon confocal microscope [Gerlitz2016], recorded at 1 min time intervals are used. The plant roots expressed a photoactivatable green fluorescent protein, which after activation with a UV pulse diffuses from the activated cells to the neighbouring cells.

For each time step a mask is painted to cover each cell in each time step.

The fluorescence intensities be evaluated using a python script.

Open the the database where the masks are painted:

```
[2]: import re
import numpy as np
from matplotlib import pyplot as plt
import clickpoints

# open the ClickPoints database
db = clickpoints.DataFile("plant_root.cdb")

path plant_root.cdb
Open database with version 18
```

Get a list of *Image* objects (`getImages()`) and a list of all *MaskType* objects (`getMaskTypes()`). Then we iterate over the images, load the green channel of the image (`image.data[:, :, 1]`) and get the mask data for that image (`image.mask.data`). The mask data is a numpy array with the same dimensions as the image having 0s for the background and the value `mask_type.index` for each pixel that belongs to the `mask_type`. Therefore we iterate over all the mask types and filter the pixels of the mask that belong to each type. This mask can then be used to filter the pixels of the green channel that belong to the *MaskType*.

```
[3]: # get images and mask_types
images = db.getImages()
mask_types = db.getMaskTypes()

# regular expression to get time from filename
regex = re.compile(r".*(?P<experiment>\d*)-(?P<time>\d*)min")

# initialize arrays for times and intensities
times = []
intensities = []

# iterate over all images
for image in images:
    print("Image", image.filename)
    # get time from filename
    time = float(regex.match(image.filename).groupdict()["time"])
    times.append(time)

    # get mask and green channel of image
    mask = image.mask.data
    green_channel = image.data[:, :, 1]

    # iterate over the mask types
    intensity = []
    for mask_type in mask_types:
        # filter from the mask the current mask type
        mask_for_this_type = (mask == mask_type.index)
        # calculate the mean intensity of this cell in the green channel
        mean_intensity_in_cell = np.mean(green_channel[mask_for_this_type])
        # and add it to the list
        intensity.append(mean_intensity_in_cell)
    # add all the mean intensities of the cells in this image to a list
    intensities.append(intensity)

Image 1-0min.tif
Image 1-2min.tif
Image 1-4min.tif
Image 1-6min.tif
Image 1-8min.tif
```

(continues on next page)

(continued from previous page)

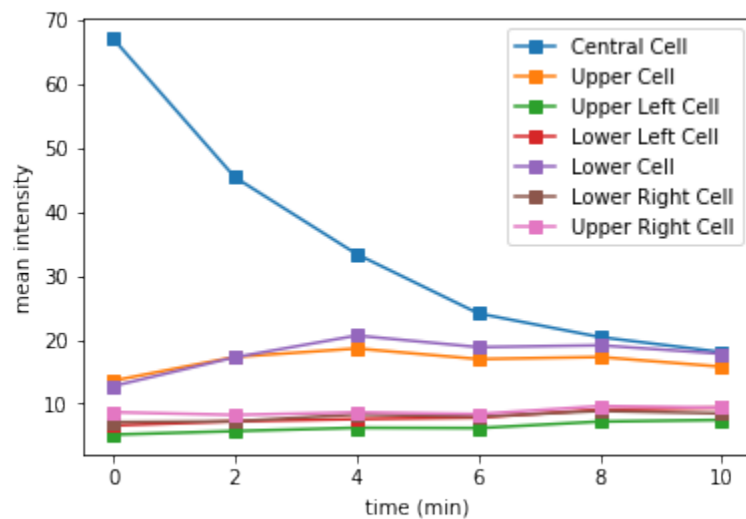
Image 1-10min.tif

We now plot the intensity for each cell over the time. The label of each line is the name of the corresponding *MaskType*.

```
[4]: # convert lists to numpy arrays
intensities = np.array(intensities).T
times = np.array(times)

# iterate over cells
for mask_type, cell_int in zip(mask_types, intensities):
    plt.plot(times, cell_int, "-s", label=mask_type.name)

# add legend and labels
plt.legend()
plt.xlabel("time (min)")
plt.ylabel("mean intensity")
# display the plot
plt.show()
```



How we want to visualize the cells in the image. Therefore we fetch the first image and its mask. Then we iterate over all mask types and draw contours around each masked region. Then we plot the centroid of the mask and the mask index.

```
[5]: from skimage.measure import regionprops, label, find_contours

# get the first image
image = db.getImage(0)
# get the corresponding mask data
mask_data = image.mask.data
# iterate over all mask types
for mask_type in mask_types:
    # get the mask data for that mask type
    mask = (mask_data == mask_type.index)

    # get the contour of the masked region and draw it
    contour = find_contours(mask, 0.5)[0]
```

(continues on next page)



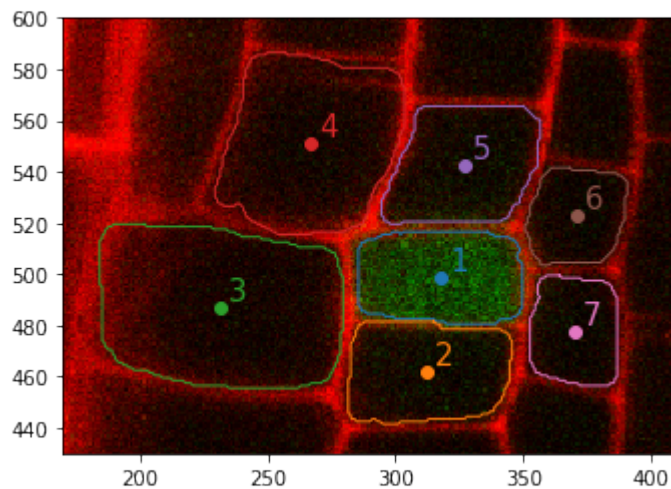
(continued from previous page)

```

line, = plt.plot(contour[:, 1], contour[:, 0], '-', lw=1)

# get the centroid and draw a dot with the same color
prop = regionprops(label(mask))[0]
y, x = prop.centroid
plt.plot(x, y, 'o', color=line.get_color())
# and a text showing the index of the mask type with the same color
plt.text(x+3, y+3, '%d' % mask_type.index, color=line.get_color(), fontsize=15)
# draw the image
plt.imshow(image.data)
# and zoom in
plt.xlim(170, 410)
plt.ylim(430, 600)
plt.show()

```



## References

### 6.3 Supervised Tracking of Fiducial Markers in Magnetic Tweezer Measurements

In the example, we show how the ClickPoints addon `Track` can be used to track objects in an image and how the resulting tracks can later on be used to calculate displacements. [Bonakdar2016]

The data we show in this example are measurements of a magnetic tweezer, which uses a magnetic field to apply forces on cells. The cell is additionally tagged with non magnetic beads, which are used as fiducial markers.

The images can be opened with ClickPoints and every small bead (the fiducial markers) is marked with a marker of type `tracks`. Then the `Track` addon is started to find the position of these beads in the subsequent images.

The resulting tracks can now be accessed and evaluated with Python and the ClickPoints package. Therefore we first open the ClickPoints file:

```

[2]: %matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

```

(continues on next page)

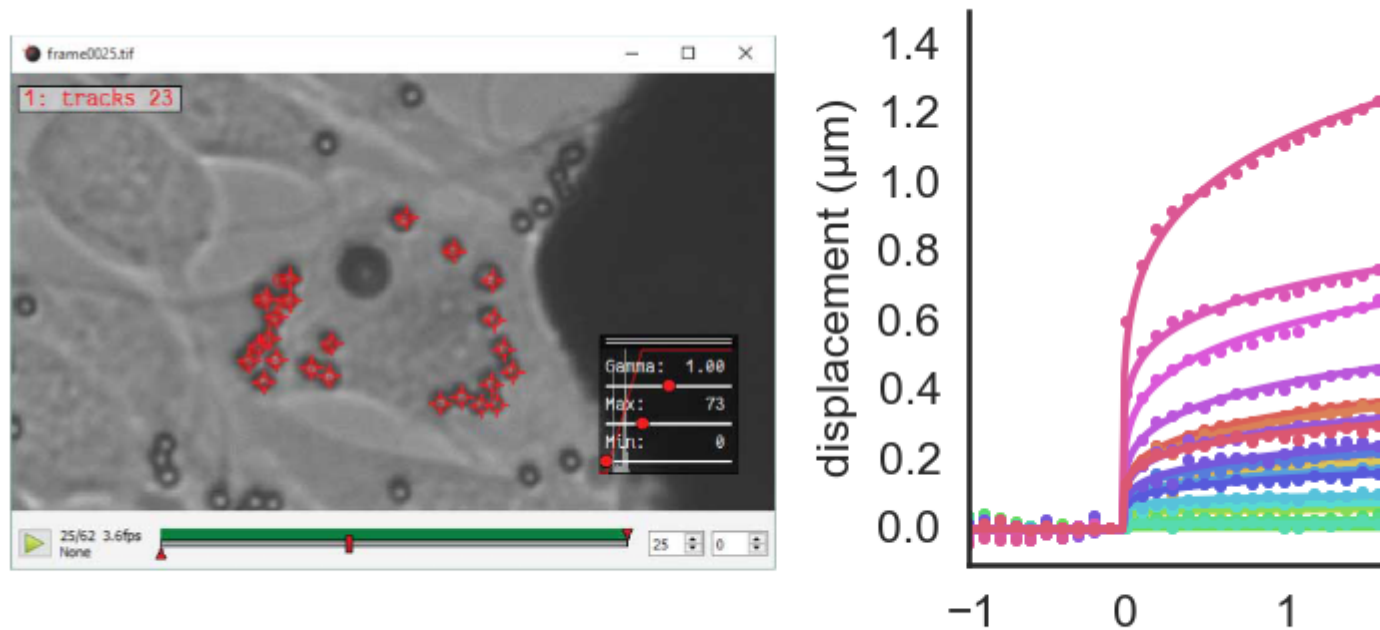


Fig. 3: Left: the image of beads on cells loaded in ClickPoints. Right: displacement of beads.

(continued from previous page)

```
# connect to ClickPoints database
# database filename is supplied as command line argument when started from ClickPoints
import clickpoints
db = clickpoints.DataFile("track.cdb")

path track.cdb
Open database with version 18
```

Then we get all the tracks with `getTracks()`. In this example without any filtering, as we just want all the tracks available. We can iterate over the received object and access the `Track` objects. From them we can get the points and calculate their displacements with respect to their starting points.

```
[3]: # get all tracks
tracks = db.getTracks()

# iterate over all tracks
for track in tracks:
    # get the points
    points = track.points
    # calculate the distance to the first point
    distance = np.linalg.norm(points[:, :] - points[0, :], axis=1)
    # plot the displacement
    plt.plot(track.frames, distance, "-o")

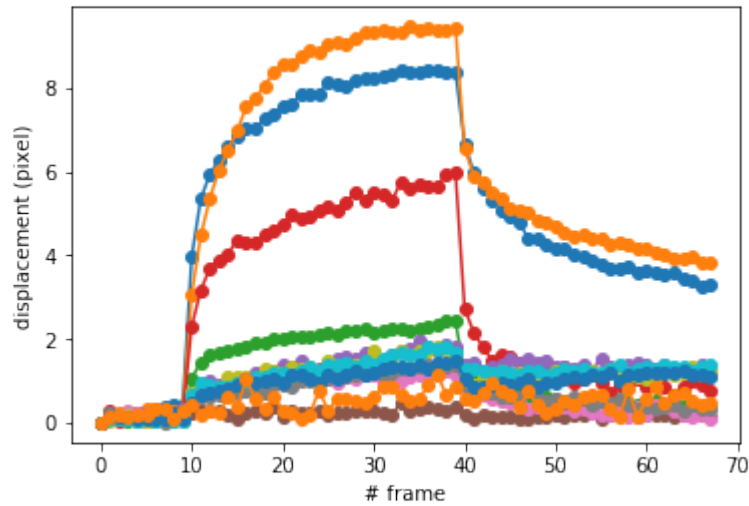
# label the axes
plt.xlabel("# frame")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("displacement (pixel)")
```

```
[3]: <matplotlib.text.Text at 0x2b3f44019e8>
```



We can now use `getImage()` to get the first image of the sequence and load the data from this file. This can now be displayed with matplotlib. To draw the tracks, we iterate over the track list again and plot all the points, as well, as the starting point of the track for a visualisation of the tracks.

```
[4]: # get the first image
im_entry = db.getImage(0)

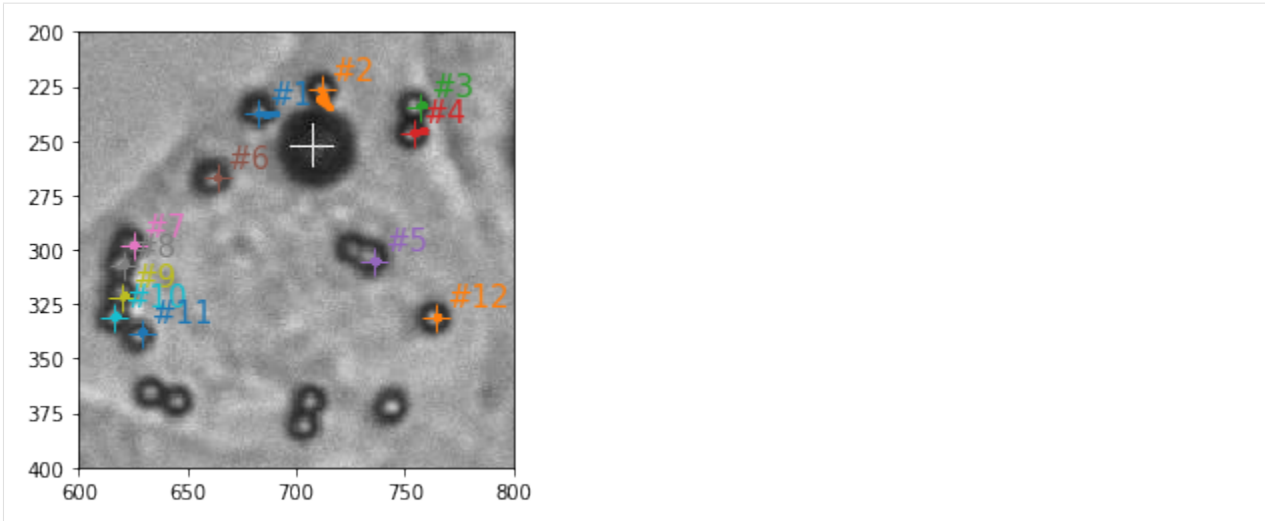
# we load the pixel data from the Image database entry
im_pixel = im_entry.data

# plot the image
plt.imshow(im_pixel, cmap="gray")

# iterate over all tracks
for track in tracks:
    # get the points
    points = track.points
    # plot the beginning of the track
    cross, = plt.plot(points[0, 0], points[0, 1], '+', ms=14, mew=1)
    # plot the track with the same color
    plt.plot(points[:, 0], points[:, 1], lw=3, color=cross.get_color())
    # plot the track id with a little offset and the same color
    plt.text(points[0, 0]+5, points[0, 1]-5, "%d" % track.id, color=cross.get_
->color(), fontsize=15)

# zoom into the image
plt.xlim(600, 800)
plt.ylim(400, 200)
```

```
[4]: (400, 200)
```



## References

### 6.4 Using ClickPoints for Visualizing Simulation Results

Here we show how ClickPoints can be apart from viewing and analyzing images also be used to store simulation results in a ClickPoints Project file. This has the advantages that the simulation can later be viewed in ClickPoints, with all the features of playback, zooming and panning. Also the coordinates of the objects used in the simulation can later be accessed through the ClickPoints database file.

This simple example simulates the movement of 10 object which follow a random walk.

First some imports:

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import clickpoints
```

Then we define the basic parameters for the simulation.

```
[2]: # Simulation parameters
N = 10
size = 100
size = 100
frame_count = 100
```

We create a new database. The attribute “w” stands for write, which ensures that we create a new database.

```
[3]: # create new database
db = clickpoints.DataFile("sim.cdb", "w")

path sim.cdb
```

We define a new marker type with the name “point” the color red (“#FF0000”) and the type TYPE\_Track. Then we create N track instances for it.

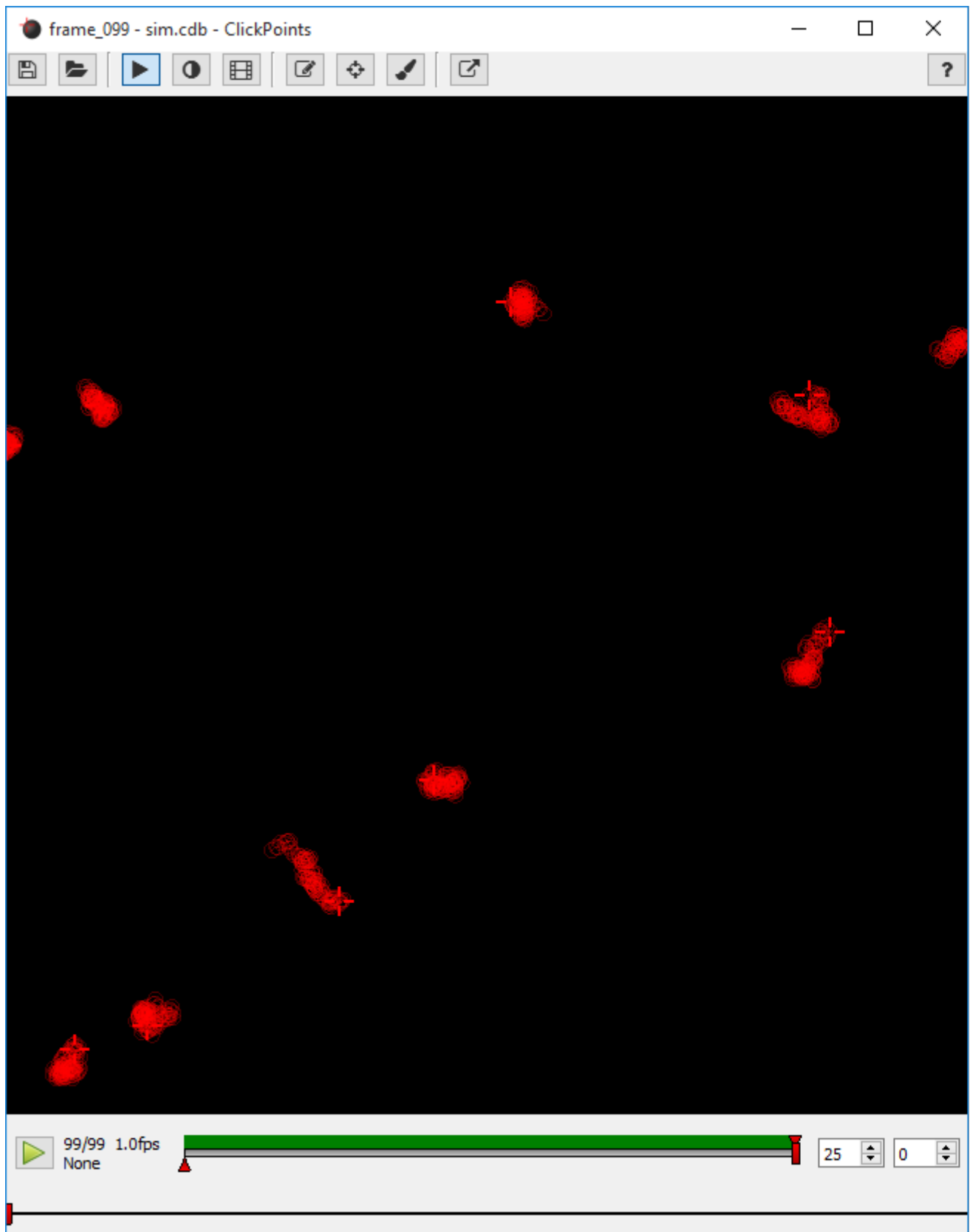


Fig. 4: Left: Tracks of the random walk simulation in ClickPoints. Right: Tracks plotted all starting from (0, 0).

```
[4]: # Create a new marker type
type_point = db.setMarkerType("point", "#FF0000", mode=db.TYPE_Track)

# Create track instances
tracks = [db.setTrack(type_point) for i in range(N)]
```

We create some random initial positions for the tracks. We now iterate over the amount of frames we want to simulate. For each iteration we add a new image to the database using `setImage()`, then we add some random movement to the positions and store the new positions for the tracks using `setMarkers()`. Here we have to provide the current image and the list of *Track* objects.

```
[5]: # Fix the seed
np.random.seed(0)

# Create initial positions
points = np.random.rand(N, 2)*size

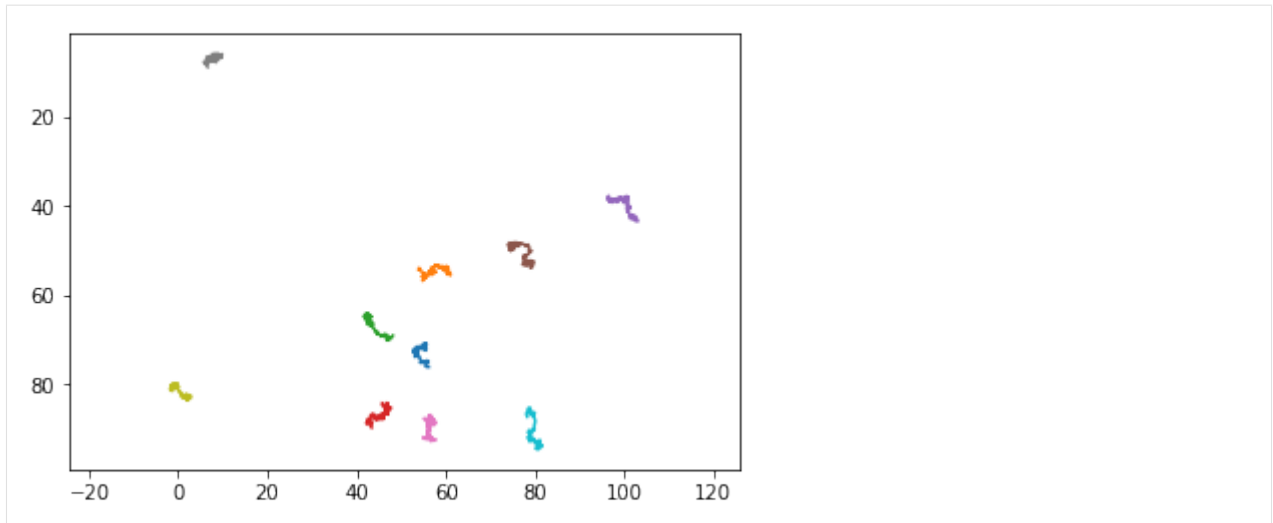
# iterate
for i in range(frame_count):
    # Create a new frame
    image = db.setImage("frame_%03d" % i, width=size, height=size)

    # Move the positions
    points += np.random.rand(N, 2)-0.5

    # Save the new positions
    db.setMarkers(image=image, x=points[:, 0], y=points[:, 1], track=tracks)
```

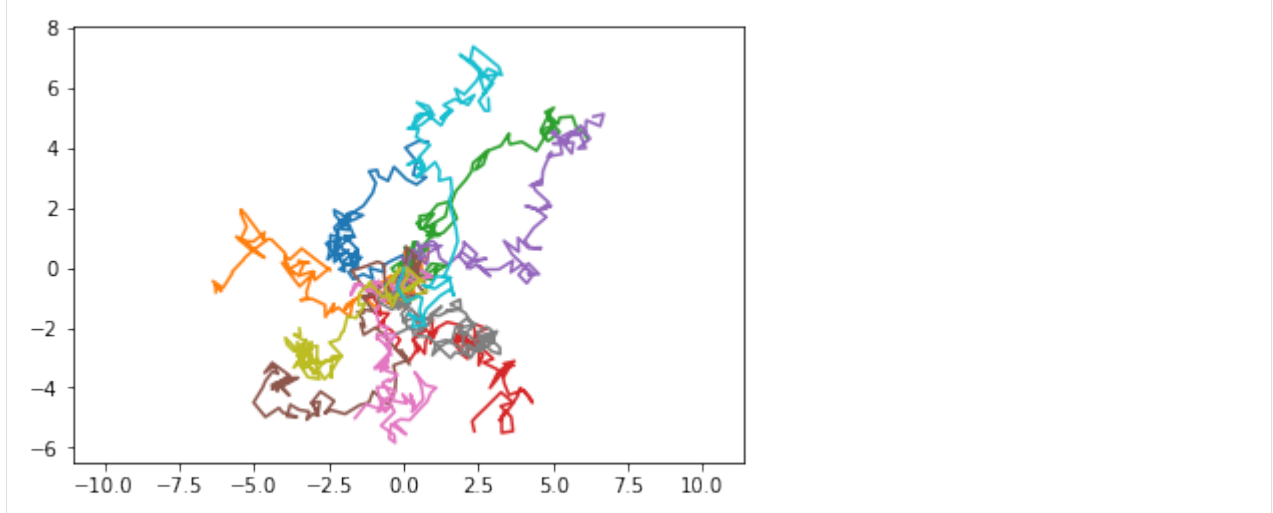
After the simulation, we want to access the tracks again to plot them. From every track we query the points and plot them.

```
[6]: # plot the results
for track in tracks:
    plt.plot(track.points[:, 0], track.points[:, 1], '-')
# adjust the plot ranges
plt.xlim(0, size)
plt.ylim(size, 0)
# and adjust the scaling to equal
plt.axis("equal")
plt.show()
```



We now plot the tracks in a different fashion, as a “rose” plot, where all the tracks start at the same position.

```
[7]: # plot the results
for track in tracks:
    # get the points
    points = track.points
    # subtract the initial point
    points = points-points[0, :]
    # plot the track from the new origin
    plt.plot(points[:, 0], points[:, 1], '-')
# and adjust the scaling to equal
plt.axis("equal")
plt.show()
```



```
[ ]:
```





# CHAPTER 7

---

## Database API

---

ClickPoints comes with a powerful API which enables access from within python to ClickPoints Projects which are stored in a `.cdb` ClickPoints SQLite database.

To get started reading and writing to a database use:

```
1 import clickpoints
2 db = clickpoints.DataFile("project.cdb")
```

This will open an existing project file called `project.cdb`.

---

**Note:** The [Examples](#) section demonstrates the use of the API with various examples and provides a good starting point to write custom evaluations.

---

## 7.1 Database Models

The `.cdb` file consists of multiple SQL tables in which it stores its information. Each table is represented in the API as a peewee model. Users which are not familiar can use the API without any knowledge of peewee, as the API provides all functions necessary to access the data. For each table a `get` (retrieve entries), `set` (add and change entries) and `delete` (remove entries) function is provided. Functions with a plural name always work on multiple entries at once and all arguments can be provided as single values or arrays if multiple entries should be affected.

The tables are: *Meta*, *Path*, *Layer*, *Image*, *Offset*, *Track*, *MarkerType*, *Marker*, *Line*, *Rectangle*, *Mask*, *MaskType*, *Annotation*, *Tag*, *TagAssociation*.

### **class Meta**

Stores key value pairs containing meta information for the ClickPoints project.

#### **Attributes:**

- **key** (*str*, *unique*) - the key
- **value** (*str*) - the value for the key

**class Path**

Stores a path. Referenced by each image entry.

See also: `getPath()`, `getPaths()`, `setPath()`, `deletePaths()`.

**Attributes:**

- **path** (*str, unique*) - the path
- **images** (*list of Image*) - the images with this path.

**class Image**

Stores an image.

See also: `getImage()`, `getImages()`, `getImageIterator()`, `setImage()`, `deleteImages()`.

**Attributes:**

- **filename** (*str, unique*) - the name of the file.
- **ext** (*str*) - the extension of the file.
- **frame** (*int*) - the frame of the file (0 for images,  $\geq 0$  for images from videos).
- **external\_id** (*int*) - the id of the file entry of a corresponding external database. Only used when ClickPoints is started from an external database.
- **timestamp** (*datetime*) - the timestamp associated to the image.
- **sort\_index** (*int, unique*) - the index of the image. The number shown in ClickPoints next to the time line.
- **width** (*int*) - None if it has not be set, otherwise the width of the image.
- **height** (*int*) - None if it has not be set, otherwise the height of the image.
- **path** (*Path*) - the linked path entry containing the path to the image.
- **layer** (*int*) - the id separating different kinds of images for the same sort\_index.
- **offset** (*Offset*) - the linked offset entry containing the offsets stored for this image.
- **annotation** (*Annotation*) - the linked annotation entry for this image.
- **markers** (*list of Marker*) - a list of marker entries for this image.
- **lines** (*list of Line*) - a list of line entries for this image.
- **rectangles** (*list of Rectangle*) - a list of rectangle entries for this image.
- **mask** (*Mask*) - the mask entry associated with the image.
- **data** (*array*) - the image data as a numpy array. Data will be loaded on demand and cached.
- **data8** (*array, uint8*) - the image data converted to unsigned 8 bit integers.
- **getShape()** (*list*) - a list containing height and width of the image. If they are not stored in the database yet, the image data has to be loaded.

**class Offset**

Offsets associated with an image.

See also: `setOffset()`, `deleteOffsets()`.

**Attributes:**

- **image** (*Image*) - the associated image entry.
- **x** (*int*) - the x offset

- **y** (*int*) - the y offset

#### **class Layer**

Stores the name of a layer. Referenced by each image entry.

See also: `getLayer()`, `getLayers()`, `setLayer()`, `deleteLayers()`.

##### **Attributes:**

- **name** (*str, unique*) - the name of the layer
- **images** (*list of Image*) - the images with this layer.

#### **class Track**

A track containing multiple markers.

See also: `getTrack()`, `getTracks()`, `setTrack()`, `deleteTracks()`, `getTracksNanPadded()`.

##### **Attributes:**

- **style** (*str*) - the style for this track.
- **text** (*str*) - an additional text associated with this track. It is displayed next to the markers of this track in ClickPoints.
- **hidden** (*bool*) - whether the track should be displayed in ClickPoints.
- **points** (*array*) - an Nx2 array containing the x and y coordinates of the associated markers.
- **points\_corrected** (*array*) - an Nx2 array containing the x and y coordinates of the associated markers corrected by the offsets of the images.
- **markers** (*list of Marker*) - a list containing all the associated markers.
- **times** (*list of datetime*) - a list containing the timestamps for the images of the associated markers.
- **frames** (*list of int*) - a list containing all the frame numbers for the images of the associated markers.
- **image\_ids** (*list of int*) - a list containing all the ids for the images of the associated markers.

##### **Methods:**

##### **split** (*marker*)

Split the track after the given marker and create a new track which contains all the markers after the given marker.

##### **Parameters:**

- **marker** (*int, Marker*) - the marker id or marker entry at which to split.

##### **Returns:**

- **new\_track** (*Track*) - the new track which contains the markers after the given marker.

##### **removeAfter** (*marker*)

Remove all the markers from the track after the given marker.

##### **Parameters:**

- **marker** (*int, Marker*) - the marker id or marker entry after which to remove markers.

##### **Returns:**

- **count** (*int*) - the amount of deleted markers.

##### **merge** (*track*)

Merge the track with the given track. All markers from the other track are moved to this track. The other track which is then empty will be removed. Only works if the tracks don't have markers in the same images, as this would cause ambiguities.

##### **Parameters:**

- **track** (*int, Track*) - the track id or track entry whose markers should be merged to this track.

**Returns:**

- **count** (*int*) - the amount of new markers for this track.

**changeType** (*new\_type*)

Change the type of the track and its markers to another type.

**Parameters:**

- **new\_type** (*str*, *int* *MarkerType*) - the id, name or entry for the marker type which should be the new type of this track. It has to be of mode TYPE\_Track.

**Returns:**

- **count** (*int*) - the amount of markers that have been changed.

**class MarkerType**

A marker type.

See also: `getMarkerTypes()`, `getMarkerType()`, `setMarkerType()`, `deleteMarkerTypes()`.

**Attributes:**

- **name** (*str*, *unique*) - the name of the marker type.
- **color** (*str*) - the color of the marker in HTML format, e.g. #FF0000 (red).
- **mode** (*int*) - the mode, has to be either: TYPE\_Normal, TYPE\_Rect, TYPE\_Line or TYPE\_Track
- **style** (*str*) - the style of the marker type.
- **text** (*str*) - an additional text associated with the marker type. It is displayed next to the markers of this type in ClickPoints.
- **hidden** (*bool*) - whether the markers of this type should be displayed in ClickPoints.
- **markers** (*list of Marker*) - a list containing all markers of this type. Only for TYPE\_Normal and TYPE\_Track.
- **lines** (*list of Line*) - a list containing all lines of this type. Only for TYPE\_Line.
- **markers** (*list of Rectangle*) - a list containing all rectangles of this type. Only for TYPE\_Rect.

**class Marker**

A marker.

See also: `getMarker()`, `getMarkers()`, `setMarker()`, `setMarkers()`, `deleteMarkers()`.

**Attributes:**

- **image** (*Image*) - the image entry associated with this marker.
- **x** (*float*) - the x coordinate of the marker.
- **y** (*float*) - the y coordinate of the marker.
- **type** (*MarkerType*) - the marker type.
- **processed** (*bool*) - a flag that is set to 0 if the marker is manually moved in ClickPoints, it can be set from an add-on if the add-on has already processed this marker.
- **style** (*str*) - the style definition of the marker.
- **text** (*str*) - an additional text associated with the marker. It is displayed next to the marker in ClickPoints.
- **track** (*Track*) - the track entry the marker belongs to. Only for TYPE\_Track.
- **correctedXY()** (*array*) - the marker position corrected by the offset of the image.
- **pos()** (*array*) - an array containing the coordinates of the marker: [x, y].

**Methods:****changeType** (*new\_type*)

Change the type of the marker.

**Parameters:**

- **new\_type** (*str*, *int* *MarkerType*) - the id, name or entry for the marker type which should be the new type of this marker. It has to be of mode TYPE\_Normal.

**class Line**

A line.

See also: `getLine()`, `getLines()`, `setLine()`, `setLines()`, `deleteLines()`.**Attributes:**

- **image** (*Image*) - the image entry associated with this line.
- **x1** (*float*) - the first x coordinate of the line.
- **y1** (*float*) - the first y coordinate of the line.
- **x2** (*float*) - the second x coordinate of the line.
- **y2** (*float*) - the second y coordinate of the line.
- **type** (*MarkerType*) - the marker type.
- **processed** (*bool*) - a flag that is set to 0 if the line is manually moved in ClickPoints, it can be set from an add-on if the add-on has already processed this line.
- **style** (*str*) - the style definition of the line.
- **text** (*str*) - an additional text associated with the line. It is displayed next to the line in ClickPoints.
- **correctedXY()** (*array*) - the line positions corrected by the offset of the image.
- **pos()** (*array*) - an array containing the coordinates of the line: [x, y].
- **length()** (*float*) - the length of the line in pixel.
- **angle()** (*float*) - the angle of the line to the horizontal in radians.

**Methods:****changeType** (*new\_type*)

Change the type of the line.

**Parameters:**

- **new\_type** (*str*, *int* *MarkerType*) - the id, name or entry for the marker type which should be the new type of this line. It has to be of mode TYPE\_Line.

**class Rectangle**

A rectangle.

See also: `getRectangle()`, `getRectangles()`, `setRectangle()`, `setRectangles()`, `deleteRectangles()`.**Attributes:**

- **image** (*Image*) - the image entry associated with this rectangle.
- **x** (*float*) - the x coordinate of the rectangle.
- **y** (*float*) - the y coordinate of the rectangle.
- **width** (*float*) - the width of the rectangle.
- **height** (*float*) - the height of the rectangle.

- **type** ( *MarkerType* ) - the marker type.
- **processed** ( *bool* ) - a flag that is set to 0 if the rectangle is manually moved in ClickPoints, it can be set from an add-on if the add-on has already processed this line.
- **style** ( *str* ) - the style definition of the rectangle.
- **text** ( *str* ) - an additional text associated with the rectangle. It is displayed next to the rectangle in ClickPoints.
- **correctedXY()** ( *array* ) - the rectangle positions corrected by the offset of the image.
- **pos()** ( *array* ) - an array containing the coordinates of the rectangle: [x, y].
- **slice\_x(border=0)** ( *slice* ) - a slice object to use the rectangle to cut out a region of an image
- **slice\_y(border=0)** ( *slice* ) - a slice object to use the rectangle to cut out a region of an image
- **slice(border=0)** ( *tuple* ) - a tuple of a y-slice and an x-slice, border specifies an additional border to slice around the rectangle
- **area()** ( *float* ) - the area of the rectangle

**Methods:****changeType** ( *new\_type* )

Change the type of the rectangle.

**Parameters:**

- **new\_type** ( *str*, *int* *MarkerType* ) - the id, name or entry for the marker type which should be the new type of this rectangle. It has to be of mode TYPE\_Rect.

**cropImage** ( *image=None*, *with\_offset=True*, *with\_subpixel=True*, *border=0* )

Crop a region of the given image provided by the rectangle.

**Parameters:**

- **image** ( *ndarray*, \* :*py:class: 'Image'* \*) - the image as a database entry or a numpy array.
- **with\_offset** ( *bool* ) - whether to apply the offset of the image. Default True.
- **with\_subpixel** ( *bool* ) - whether to apply a subpixel shift for the region. Default True.
- **border** ( *number* ) - a number of border pixels to add to the region. Default 0.

**class Mask**

A mask entry.

See also: `getMask()`, `getMasks()`, `setMask()`, `deleteMasks()`.**Attributes:**

- **image** ( *Image* ) - the image entry associated with this marker.
- **data** ( *array* ) - the mask image as a numpy array. Mask types are stored by their index value.

**class MaskType**

A mask type.

See also: `getMaskType()`, `getMaskTypes()`, `setMaskType()`, `deleteMaskTypes()`.**Attributes:**

- **name** ( *str* ) - the name of the mask type.
- **color** ( *str* ) - the color of the mask type in HTML format, e.g. #FF0000 (red).
- **index** ( *int* ) - the integer value used to represent this type in the mask.

**class Annotation**

An annotation.

See also: `getAnnotation()`, `getAnnotations()`, `setAnnotation()`, `deleteAnnotations()`.

**Attributes:**

- **image** ( *Image* ) - the image entry associated with this annotation.
- **timestamp** ( *datetime* ) - the timestamp of the image linked to the annotation.
- **comment** ( *str* ) - the text of the comment.
- **rating** ( *int* ) - the value added to the annotation as rating.
- **tags** ( *list of Tag* ) - the tags associated with this annotation.

**class Tag**

A tag for an *Annotation*.

See also: `getTag()`, `getTags()`, `setTag()`, `deleteTags()`.

**Attributes:**

- **name** ( *str* ) - the name of the tag.
- **annotations** ( *list of Annotation* ) - the annotations associated with this tag.

**class TagAssociation**

A link between a *Tag* and an *Annotation*

**Attributes:**

- **annotation** ( *Annotation* ) - the linked annotation.
- **tag** ( *Tag* ) - the linked tag.

## 7.2 DataFile

The DataFile is the interface to the ClickPoints database. This can either be used in external evaluation scripts that take data clicked in ClickPoints for further evaluation or in add-on scripts where it is accessible through the `self.db` class variable.





ClickPoints allows to easily write add-on scripts.

**Note:** The [Addons](#) section demonstrates how the add-ons can be used and may serve as a good starting point to write custom add-ons.

The add-on consists of at least two files. A meta data file with `.txt` ending which contains basic information on the add-on and a script file providing an overloaded class of `clickpoints.Addon` as shown above.

## 8.1 Meta Data File

The file has to start with `[addon]` followed by lines with key and value pairs: A typical meta file looks like this:

```
1 [addon]
2 name=My new Add-on
3 file=Addon.py
4 icon=fa.flask
5 description=This add-on makes cool new things.
6 image=Image.png
7 requirements=xlwt
```

- **name - name=My new Add-on** Defines the name of the add-on. This name is displayed in ClickPoints in the add-on list.
- **file - file=Addon.py** Defines the filename of the python file that contains the add-on class.
- **icon - icon=fa.flask** Defines the icon of the add-on. It can be either a filename or a valid qtawesome icon name (see <https://github.com/spyder-ide/qtawesome>, e.g. it starts with `fa` . followed by the name of a font awesome icon see the [font awesome iconlist](#)).
- **image - image=Image.png** Defines the image of the add-on. The image will be displayed in ClickPoints in the add-on list directly above the description. The image should have a dimension of 300x160 pixel.

- **description - description=This add-on makes cool new things.** Defines a short description for the add-on. If a longer description is desired, a file called `Desc.html` next to the `*.txt` file can be used. This file supports rich text with an html subset defined by [Qt Html Subset](#).
- **requirements - requirements=xlwt, skimage** Define the packages that this add-on needs. Multiple packages have to be separated by a komma.

## 8.2 Script File

The script file has to contain a class called `Addon` which is derived from a prototype Add-on class:

```
1 import clickpoints
2
3 class Addon(clickpoints.Addon):
4     def __init__(self, *args, **kwargs):
5         clickpoints.Addon.__init__(self, *args, **kwargs)
6
7         print("I am initialized with the database", self.db)
8         print("and the ClickPoints interface", self.cp)
9
10    def run(self, *args, **kwargs):
11        print("The user wants to run me")
```

This class will allow you to overload the `init` function where your add-on can set up its configuration, e.g. add some new marker types to ClickPoints.

To process data, you can overload the `run` function. Here the add-on can do its heavy work. Some caution has to be taken when executing interface actions, as `run` is called in a second thread to not block ClickPoints during its execution. For a good example of an add-on that uses the `run` function, refer to the [Track Add-on](#).

But add-ons can also provide passive features that are not executed by a call of the `run` method, but rely on callbacks. Here a good example is the [Measure Tool Add-on](#), which just reacts on the `MarkerMoved` callback.

The add-on class has two main member variables: `self.db` and `self.cp`.

- `self.db` is a [DataFile](#) instance which gives access to the ClickPoints database. For details on the interface see [Database API](#).
- `self.cp` is a [Commands](#) instance which allows for communication with the ClickPoints interface.

## 8.3 Defining Options

Add-ons can define their own options that are saved in the database along the ClickPoints options. They are also included in the ClickPoints options menu and the export of options.

New options should be defined in the `__init__` function of the add-on. Therefore, the add-on class has some methods to add, get and set options:

**addOption** (*key*, *default*, *value\_type="string"*, *values=None*, *display\_name=""*, *hidden=False*, *tooltip=""*,  
*min\_value=None*, *max\_value=None*, *value\_count=1*)

Define a new option value for the add-on.

**Parameters:**

- **key** (*str*) - the name of the option.
- **default** (*str*, *int*, *float*, *list*) - the default value for the option.

- **value\_type** (*str*) - the type of the value, can be *string*, *int*, *float*, *bool*, *choice*
- **values** (*list*) - allowed values if the type is *choice*.
- **display\_name** (*str*) - the name to display in the option menu.
- **hidden** (*bool*) - weather the option should be displayed in the option menu.
- **tooltip** (*str*) - the tooltip of the option in the option menu.
- **min\_value** (*number*) - the minimal value for a *int* or *float* option.
- **max\_value** (*number*) - the maximum value for a *int* or *float* option.
- **decimals** (*number*) - the number of decimals to allow for a *float* option.
- **value\_count** (*int*) - it the option should accept a list of values. Only for *int* values.

**getOption** (*key*)

Return the current value of an option.

**Parameters:**

- **key** (*str*) - the name of the option.

**setOption** (*key*, *value*)

Set the current value of an option.

**Parameters:**

- **key** (*str*) - the name of the option.
- **value** (*str*, *int*, *float*, *list*) - the new value of the option.

**getOptions** ()

Return a list of all options of this add-on. The list contains option objects with the following properties:

**Properties:**

- **key** (*str*) - the name of the option.
- **value** (*str*, *int*, *float*, *list*) - the current value of the option.
- **default** (*str*, *int*, *float*, *list*) - the default value for the option.
- **value\_type** (*str*) - the type of the value, can be *string*, *int*, *float*, *bool*, *choice*
- **values** (*list*) - allowed values if the type is *choice*.
- **display\_name** (*str*) - the name to display in the option menu.
- **hidden** (*bool*) - weather the option should be displayed in the option menu.
- **tooltip** (*str*) - the tooltip of the option in the option menu.
- **min\_value** (*number*) - the minimal value for a *int* or *float* option.
- **max\_value** (*number*) - the maximum value for a *int* or *float* option.
- **value\_count** (*int*) - it the option should accept a list of values. Only for *int* values.

## 8.4 Events

Events are actions that occur in the main ClickPoints program. The add-ons are notified for this events and can react to them.

**markerAddEvent** (*entry*)

A marker (line or rectangle) was added to the current image.

**Parameters:**

- **entry** (*Marker*, *Line*, *Rectangle*) - the new marker.

**markerRemoveEvent** (*entry*)

A marker (line or rectangle) was removed to the current image.

**Parameters:**

- **entry** (*Marker*, *Line*, *Rectangle*) - the removed marker.

**markerMoveEvent** (*entry*)

A marker (line or rectangle) was moved.

**Parameters:**

- **entry** (*Marker*, *Line*, *Rectangle*) - the moved marker.

**buttonPressedEvent** ()

The button for this add-on was pressed. If not overloaded it will just call *self.run\_threaded()* to executed the add-on's *self.run* method in a new thread.

A typical overloading for gui based add-ons would be to call the *self.show* method:

```
def buttonPressedEvent(self):  
    self.show()
```

**zoomEvent** (*scale*, *pos*)

The zoom of the ClickPoints window has changed.

**Parameters:**

- **scale** (*number*) - the new scale factor of the displayed image.
- **pos** (*QPoint*) - the origin point of the zoom. Typically the mouse cursor position.

**frameChangedEvent** ()

The current frame in ClickPoints changed. Called when the image data is loaded, before it is displayed.

**imageLoadedEvent** (*filename*, *framenummer*)

The displayed image in ClickPoints changed. Called when the new image is loaded and displayed.

**Parameters:**

- **filename** (*string*) - the filename of the new image.
- **framenummer** (*int*) - the frame number of the new image.

**keyPressEvent** (*event*)

A key has been pressed in the ClickPoints window.

**Parameters:**

- **event** (*QKeyEvent*) - the key press event. With *event.key()* they key can be queried and compared to the key constants (see *Qt::Key*).

## 8.5 Commands

Add-ons have some basic functions to communicate with the main ClickPoints window. This interface is accessible through the *self.cp* class variable in the add-on class.

## CHAPTER 9

---

### Note

---

If you encounter any bugs or unexpected behaviour, you are encouraged to report a bug in our Bitbucket [bugtracker](#).



# CHAPTER 10

---

## Citing ClickPoints

---

If you use ClickPoints for academic research, you are highly encouraged (though not required) to cite the following paper:

- Gerum, R., Richter, S., Fabry, B. and Zitterbart, D.P. (2016), “[ClickPoints: an expandable toolbox for scientific image annotation and analysis](#)”. *Methods Ecol Evol.* doi:10.1111/2041-210X.12702

ClickPoints is developed primarily by academics, and so citations matter a lot to us. Citing ClickPoints also increases it's exposure and potential user (and developer) base, which is to the benefit of all users of ClickPoints. Thanks in advance!





## Who uses ClickPoints

Here is a list of publications that used ClickPoints for their evaluation.

- Bonakdar, N., Gerum, R. C., Kuhn, M., Spörrer, M., Lippert, A., Schneider, W., ... Fabry, B. (2016). *Mechanical plasticity of cells*. Nature Materials, 15(10), 1090–1094. <https://doi.org/10.1038/nmat4689>
- Braniš, J., Pataki, C., Spörrer, M., Gerum, R. C., Mainka, A., Cermak, V., ... Rosel, D. (2017). *The role of focal adhesion anchoring domains of CAS in mechanotransduction*. Scientific Reports, 7. <https://doi.org/10.1038/srep46233>
- Gerum, R. C., Richter, S., Winterl, A., Fabry, B., & Zitterbart, D. P. (2017). *CameraTransform: a Scientific Python Package for Perspective Camera Corrections*. ArXiv <http://arxiv.org/abs/1712.07438>
- Richter, S., Gerum, R., Schneider, W., Fabry, B., Le Bohec, C., & Zitterbart, D. P. (2018). *A remote-controlled observatory for behavioural and ecological research: A case study on emperor penguins*. Methods in Ecology and Evolution. <https://doi.org/10.1111/2041-210X.12971>
- Gerum, R., Richter, S., Fabry, B., Le Bohec, C., Bonadonna, F., Nesterova, A., Zitterbart, D. (2018). *Structural organisation and dynamics in kin penguin colonies*, Journal of Physics D: Applied Physics, 51(16), 164004. <https://doi.org/10.1088/1361-6463/AAB46B>
- Richter, S., Gerum, R., Winterl, A., Houstin, A., Seifert, M., Peschel, J., Fabry, B., Le Bohec, C., Zitterbart, D.P., (2018). *Phase transitions in huddling emperor penguin colonies*, Journal of Physics D, 51(21), 214002. <https://doi.org/10.1088/1361-6463/aabb8e>
- Gerlitz, N., Gerum, R., Sauer, N., Stadler, R. (2018). *Photoinducible DRONPA-s: a new tool to investigate cell-cell connectivity*, The Plant Journal, <https://doi.org/10.1111/tpj.13918>
- Pârvulescu, L. (2019). *Introducing a new Austropotamobius crayfish species (Crustacea, Decapoda, Astacidae): A Miocene endemism of the Apuseni Mountains, Romania*, Zoologischer Anzeiger, 279, 94–102. <https://doi.org/10.1016/j.jcz.2019.01.006>
- Córdor, M., Mark, C., Gerum, R. C., Grummel, N. C., Bauer, A., García-Aznar, J. M., & Fabry, B. (2019). *Breast cancer cells adapt contractile forces to overcome steric hindrance*, Biophysical Journal. <https://doi.org/10.1016/j.bpj.2019.02.029>



---

## Bibliography

---

- [Bohec137] Celine Le Bohec. Programme 137 ‘ecophy-antavia’ of the french polar institute paul-emile victor (ipev).
- [Gerlitz2016] Nadja Gerlitz. Dronpa. 2016.
- [Bonakdar2016] Navid Bonakdar, Richard Gerum, Michael Kuhn, Marina Spörrer, Anna Lippert, Werner Schneider, Katerina E Aifantis, and Ben Fabry. Mechanical plasticity of cells. Nature Materials, 2016.



## A

`addOption()` (*built-in function*), 62  
`Annotation` (*built-in class*), 58

## B

`buttonPressedEvent()` (*built-in function*), 64

## F

`frameChangedEvent()` (*built-in function*), 64

## G

`getOption()` (*built-in function*), 63  
`getOptions()` (*built-in function*), 63

## I

`Image` (*built-in class*), 54  
`imageLoadedEvent()` (*built-in function*), 64

## K

`keyPressEvent()` (*built-in function*), 64

## L

`Layer` (*built-in class*), 55  
`Line` (*built-in class*), 57  
`Line.changeType()` (*built-in function*), 57

## M

`Marker` (*built-in class*), 56  
`Marker.changeType()` (*built-in function*), 57  
`markerAddEvent()` (*built-in function*), 63  
`markerMoveEvent()` (*built-in function*), 64  
`markerRemoveEvent()` (*built-in function*), 64  
`MarkerType` (*built-in class*), 56  
`Mask` (*built-in class*), 58  
`MaskType` (*built-in class*), 58  
`Meta` (*built-in class*), 53

## O

`Offset` (*built-in class*), 54

## P

`Path` (*built-in class*), 53

## R

`Rectangle` (*built-in class*), 57  
`Rectangle.changeType()` (*built-in function*), 58  
`Rectangle.cropImage()` (*built-in function*), 58

## S

`setOption()` (*built-in function*), 63

## T

`Tag` (*built-in class*), 59  
`TagAssociation` (*built-in class*), 59  
`Track` (*built-in class*), 55  
`Track.changeType()` (*built-in function*), 56  
`Track.merge()` (*built-in function*), 55  
`Track.removeAfter()` (*built-in function*), 55  
`Track.split()` (*built-in function*), 55

## Z

`zoomEvent()` (*built-in function*), 64