
Mastodon

Jean-Yves Tinevez

Apr 24, 2024

A. USING MASTODON

1	Table of content.	3
1.1	Getting started with Mastodon. Automated tracking.	3
1.2	Manually editing tracks in Mastodon. TrackScheme.	15
1.3	Inspecting large datasets.	28
1.4	Numerical features and tags. The table view.	34
1.5	Semi-automated tracking.	51
1.6	The selection creator.	57
1.7	Scripting Mastodon in Fiji.	67
1.8	Moving around in the BDV views.	78
1.9	Editing spots and links in the BDV views.	79
1.10	Moving around in the TrackScheme views.	80
1.11	Editing spots and links in the TrackScheme views.	80
1.12	Shortcuts for the table views.	81
1.13	Navigation through lineages in BDV and TrackScheme views.	81
1.14	Setting the selection.	82
1.15	Existing plugins	82
1.16	Mastodon Deep Lineage - a collection of plugins to analyse lineages of tracked objects in Mastodon	83
1.17	Mastodon data structures.	92
1.18	Creating custom plugins in Mastodon.	95
1.19	Custom simple numerical features in Mastodon.	105
1.20	Mastodon numerical features.	117
1.21	The graph data structure of Mastodon.	122
1.22	Containment in Convex Polytopes using k -D trees.	127
1.23	Scripting functions	128
2	Documentation tools.	145
	Index	147



Mastodon is a large-scale tracking and track-editing framework for large, multi-view images, such as the ones that are typically generated in the domain Development Biology or Stem-Cell Biology or Cell Biology.

Why using Mastodon?

Modern microscopy technologies such as light sheet microscopy allows live sample *in toto* 3D imaging with high spatial and temporal resolution. Such images will be 3D over time, possibly multi-channels and multi-view. Computational analysis of these images promises new insights in cellular, developmental and stem cells biology. However, a single image can amount to several terabytes, and in turn, the automated or semi-automated analysis of these large images can generate a vast amount of annotations. The challenges of big data are then met twice: first by dealing with a very large image, and second with generating large annotations from this image. They will make interacting and analyzing the data especially difficult.

Mastodon is our effort to provide a tool that can harness these challenges.

This pages centralize the user and developer documentation for Mastodon. It is divided in four parts, that group sections by interest.

A. The first part, *Using mastodon* contains tutorials, aimed at end-users and focused on cell tracking. They are meant to guide users with the Mastodon software and cover three applications of cell tracking in Mastodon:

- automated cell or particle tracking;
- manual curation and correction of tracking results;
- manual and semi-automatic tracking.

This also where we introduce all the user-oriented features, such as numerical features, tags, navigation facilities, data export *etc.* If you are new to Mastodon, it is best starting with this part, and reading the tutorial in order.

B. The second part contains tables that summarize the main keyboard shortcuts of Mastodon. They are put in a separate section to facilitate browsing to them quickly.

C. The third path documents the various plugins, extensions and miscellaneous functionality of Mastodon. Some of the features documented there are already shipped with Mastodon, others are distributed via optional update sites. They are all based on the extension mechanisms described in the next part.

D. The fourth part is aimed at developers, that want to extend Mastodon. Mastodon, like [TrackMate](#), is a software platform meant to be extended by you so that new features and algorithms can be added to it in a relatively simpler manner. This part detail the plugin interface of Mastodon and the discovery mechanism.

E. The last part is made of technical information that serve as a reference for the specificities of algorithms and features currently implemented in Mastodon.

TABLE OF CONTENT.

1.1 Getting started with Mastodon. Automated tracking.

This tutorial is the starting point for new Mastodon users. It will walk you through basic operations in Mastodon, opening a dataset and creating a Mastodon project, automatically detect cells and link them, and show you how to use the main views of Mastodon. We don't go into details, and will revisit the features we survey here later.

1.1.1 The image data.

Exporting your image to file format.

Mastodon uses (BDV) files as input images. You need to prepare your images so that they can be opened in the [BigDataViewer](#).

BDV files are used more and more by several software projects in the Fiji ecosystem and beyond. This tutorial focuses on Mastodon not on BDV, however we will take a very small detour to explain what makes it fit and how to turn your images into this format. If you know already, you can skip this part, because we simply recapitulate what is being explained in the [original BDV publication](#).

For this tutorial we will use a ready-made dataset, in the adequate format, but it is a good idea to know how to export or create an image in such a format. We lazily rely on the excellent documentation and point directly to the instructions to prepare your images, depending on whether

- they are opened as an [ImageJ stack](#), or
- they come from a [SPIM processing pipeline](#), or
- they come from the [BigStitcher](#) plugin, *etc.*

Once you have prepared your images for opening in the , you should have a .xml file and a possibly very large .h5 file on your computer. The .xml file must be the output of the data preparation. It should start with the following lines:

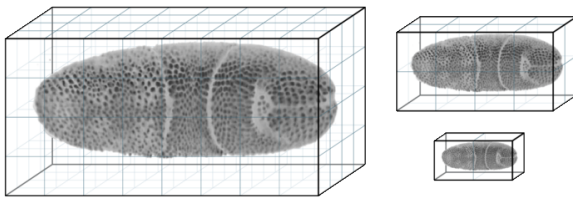
```
<?xml version="1.0" encoding="UTF-8"?>
<SpimData version="0.2">
  <BasePath type="relative">./</BasePath>
  <SequenceDescription>
    <ImageLoader format="bdv.hdf5">
      <hdf5 type="relative">dataset.hdf5</hdf5>
    ...
```

Key advantages of the file format.

The BDV file format solves mainly two challenges in image visualization and analysis, that arise with modern microscopy, namely:

- Modern microscopes can generate images that are very large in size. Much larger than what can be fitted in RAM, even with the increase in computer power. It is now common to find single movies acquired on SPIM microscopes that are several TBs in size. Computers with several TBs of RAM are not so common.
- Multiple views of the same sample can be acquired, and they need to be visualized in the same viewer. The first use case is also the multi-view images generated by SPIM microscopes, but we can also think of correlative light-electron microscopy.

If we focus on the first challenge, you see that we need to stream the image data directly from the disk, instead of fully loading it into RAM. But at the same time, we need a tool that allows for interactive browsing of the data. The view must be responsive to the user input, and not block when it has to load the data from the file. The BDV file format offers a clever file format design that does this, coupled to a specialized viewer. The image data are stored in small chunks corresponding to a neighborhood. As the viewer shows a slice through the image, the required chunks are loaded on demand and cached. All the chunks are organized in a HDF5 file, which is like a [file-system in a file](#), and accessing single chunks is fast with current computer hardware. On top of this, the image is also stored as a [multi-scale pyramid](#), to speed-up zooming and unzooming. The BDV display component exploits this file format in a clever way, and ensures that the view still answers to user interactions (mouse pan, zoom, clicks) even if the chunks are not fully loaded.



There are several implementations of this strategy, for instance in [Imaris](#) and with the new file format [N5](#) proposed by the Saalfeld lab. Some of them are inter-compatible. We will pick the BDV file format all along this document. The has proved its value and impact on our field. For instance our previous work on cell lineaging in large images, [MaMuT](#), is based on BDV.

The tutorial dataset.

Mastodon was created specially because we needed to harness very big, multi-view images. We wanted to generate comprehensive lineages and follow a large number of cells over a very long time. This accumulation of inflated words is tied to the very large - in objective disk space occupation - images we deal with using modern microscopy tools. Such datasets might not be optimal for a first contact with Mastodon. So just for this tutorial we will use a smaller dataset. It is a small region cut into a movie following the development of a drosophila embryo, acquired by William Lemon in Patrick Keller lab (HHMI, Janelia Farm). This was created from the example dataset released with the [TGMM](#) software. You can find it on Zenodo here: [DOI 10.5281/zenodo.3336346](https://doi.org/10.5281/zenodo.3336346)

It is a zip file that contains 3 files:

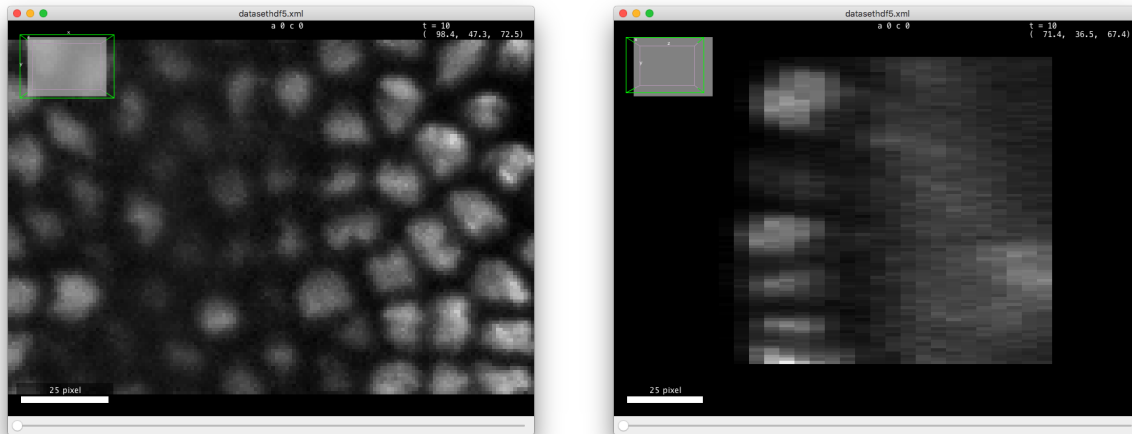
```
14M  datasethdf5.h5
2.7K  datasethdf5.settings.xml
8.7K  datasethdf5.xml
```

The `.h5` file is the HDF5 file mentioned above, that contains the image data itself. The `datasethdf5.xml` is a text file following the XML convention, specific to the BDV file format, that contains information about the image data and metadata. When we want to open a BDV file, we point the reader to this file. The `datasethdf5.settings.xml`

is an optional file that stores user display parameters, such as channel colors, min and max display value, as well as bookmarks in the data. We refer you to the [BDV documentation](#) about this file. Mastodon uses this settings file to store that same information.

If you open this data in the (in Fiji in the menu), you should see something like in the figure below. There is about 70 cells in each of the 30 time-points, arranged in a layer at the top of the sample. The deeper part of the sample (low Z coordinates) has some hazy, diffuse signal from which we cannot individualize cells. As time progresses, the cells move towards the middle part and bottom (high Y coordinates) part of the image, and some of them move deeper in Z, initiating gastrulation.

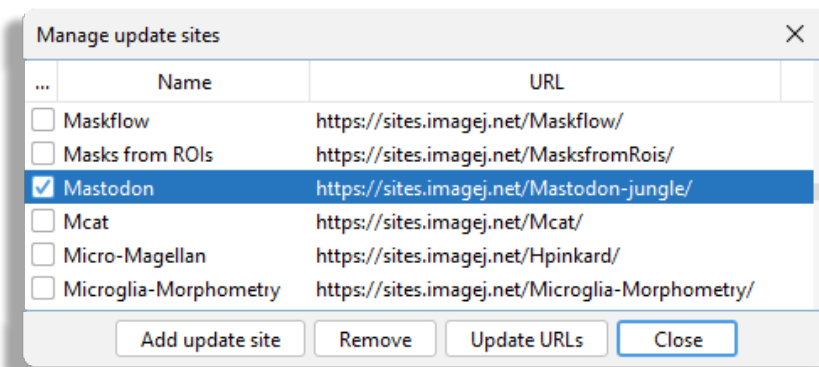
The goal of this short tutorial is to track all these cells in Mastodon.



1.1.2 Getting Mastodon.

As of today, Mastodon is available as a beta. We are still working on adding and validating features. Nonetheless the beta has everything we need to track these cells. Also, Mastodon is independent of ImageJ or Fiji, it can operate as a standalone software. However we currently distribute it via Fiji, because the updater and the dependency management are so convenient. So the first thing to do is to grab [Fiji](#), if you do not have it already.

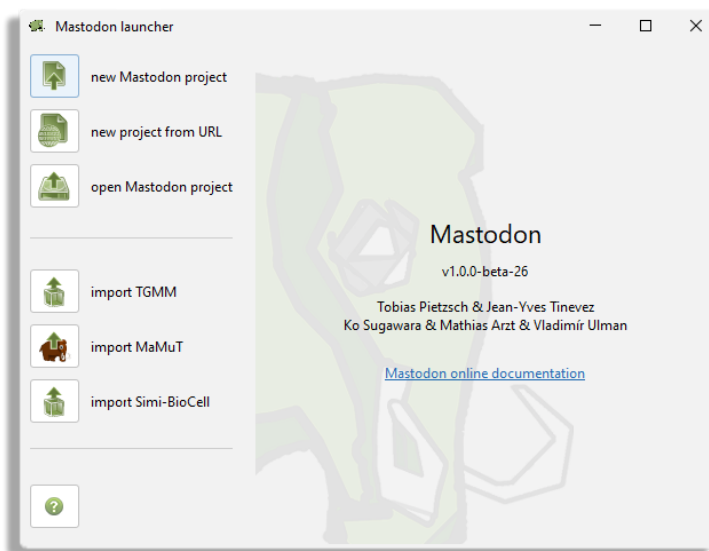
Then launch the [Fiji updater](#) and once your Fiji is up to date, click on the [Manage update site](#) button. We will add the [Mastodon update site](#). You should find the Mastodon site in the list.



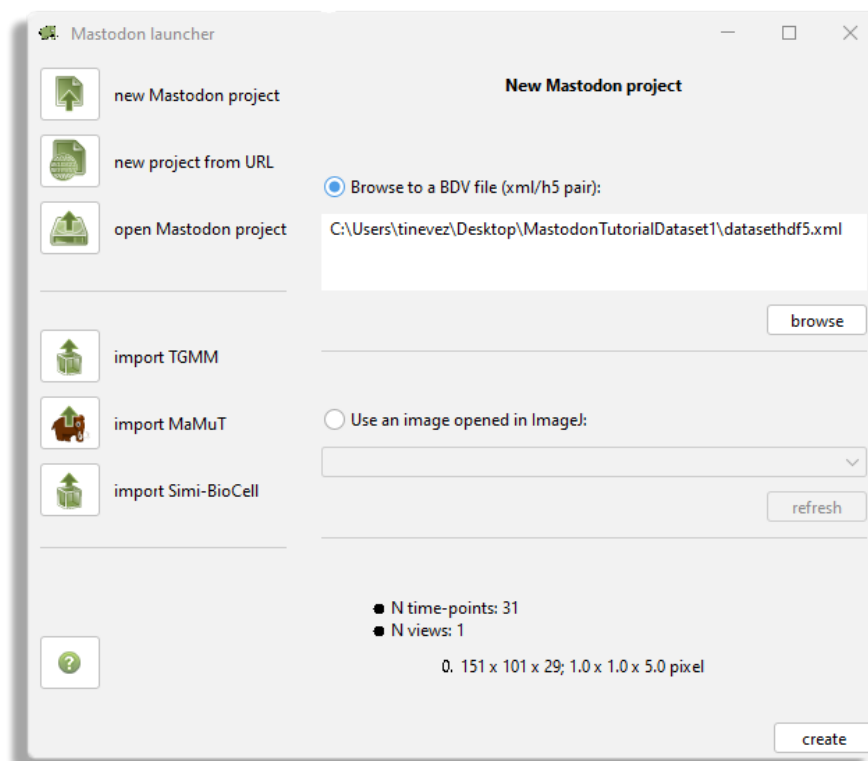
Select it, update Fiji and restart it. After restarting, you should find the command *Plugins > Mastodon* at the bottom of the *Plugins* menu.

1.1.3 Creating a new Mastodon project.

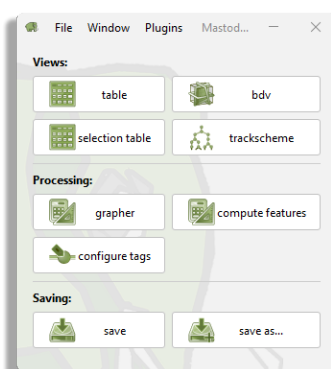
After launching the command, this Mastodon launcher appears.



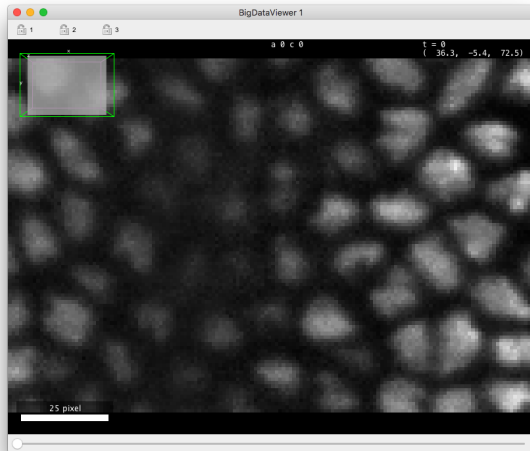
In our case, we want to create a new project from an existing BDV file. Click on the **new Mastodon project** and make sure the **Browse to a BDV file pair (xml/h5 pair)** option is selected. Use the **Browse** button to browse to the XML file location.



Then click the create button. You should now see the Mastodon main window, that is used to control the project.



Click on the bdv button. If a BDV window appears, everything is right.



It is almost a regular BDV window and if you already know how to use it and the key bindings you should find your marks quickly. The BDV view displays a *slice* of the image through arbitrary orientation. Below we give the commands and key-bindings for navigation in a Mastodon-BDV window. They are indeed close to what is found in the standard but some changes. **Please note:** You can reconfigure almost everything in Mastodon, as we will see later, including key-bindings. In this tutorial and the next ones, the key-bindings we present are for the `Default` configuration. In the [BDV navigation shortcuts](#) table you will find the key bindings to navigate through the image data.

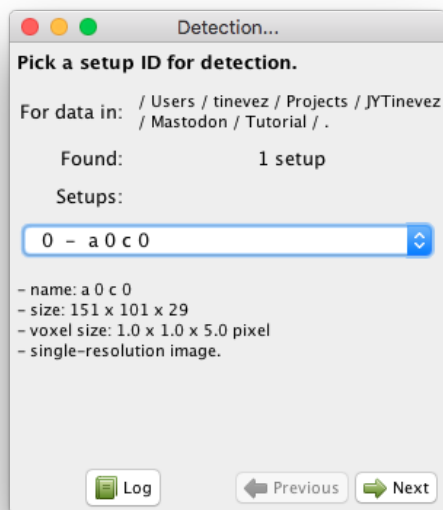
Now you want to save the project. Go back to the main window, and click on the `save as...` button. This will create a single file, called for instance `drosophila_crop.mastodon` file. This file is actually a zip file that contains the tracks and *links* to the image data. The image data is kept separate from the Mastodon file, which allows for using it with another software, independently. So if you want to transfer or move a full Mastodon project, you need to take the `.mastodon` file and all the `.xml` and `.h5` files from the dataset.

Next time you want to open this project, just click on the button and point the file browser to the `.mastodon` file. The image data will be loaded along with the lineages.

1.1.4 Detecting cells.

We want to track automatically all the cells in this dataset, and the first step is therefore to detect them. Mastodon ships a wizard to perform cell detection. It is very much inspired by the [TrackMate](#) GUI, and if you know this software you will find your marks here. Also, the basic algorithms are very close to what was in TrackMate, but they have been heavily optimized for Mastodon.

Mastodon windows have their own menus. The detection wizard can be launched from the `Plugins > Tracking > Detection...` menu item. You should have a window like the one depicted below:

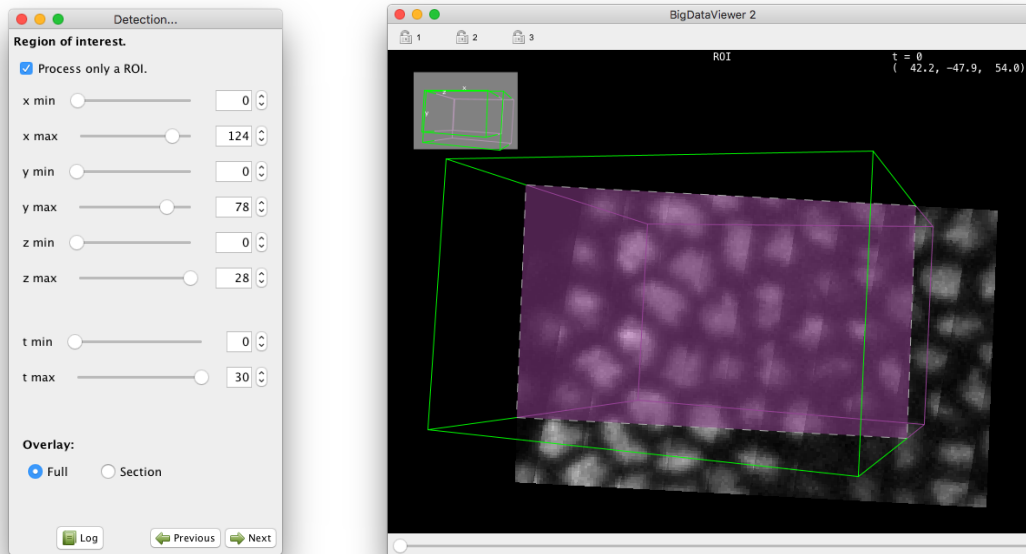


Like for TrackMate, the automated tracking user interface uses *wizards* to enter parameters, select algorithms, *etc.* You can navigate back and forth with the *Previous* and *Next* buttons. The *Next* button will bring an independent panel where all the activity in the wizard are logged as text.

This first panel allows for selecting the target *source* on which the detection will be run. Since we use the *for images*, a channel or a view is stored and displayed as a source. A source can have multiple resolutions stored, as explain above, but for the data used in this tutorial this is not the case. The sources are nicknamed ‘setups’ in this panel. They are numbered from 0 and can be selected from the drop-down list. Below the list we try to display the metadata we could retrieve from the file. Just pick the first and only channel, and click *Next*.

You can now choose to operate only on a rectangular ROI in the image. If you check the *Process only a ROI* button, new controls appear in the panel, and a ROI is drawn into an open BDV view (a new one is created if one is not opened). The ROI is painted as a wire-frame box, green for vertices that point towards the camera from the displayed slice, and purple for vertices that points away from the camera, below the displayed slice. The intersection of the ROI box with the displayed slice is painted with a purple semi-transparent overlay, with a white dotted line as borders. You can control the ROI bounds with the controls in the panel, or by directly dragging the ROI corners in the BDV view. Time bounds can also be set this way. In our case we want to segment the full image over all time-points, so leave the *Process only a ROI* button unchecked.

You cannot have non-rectangular ROIs in Mastodon. Nonetheless they are super useful as is. You can for instance combine several detection steps using different parameters in different region of your image. Or different time interval.



The next panel lets you choose the detector you want to use. In Mastodon, three detectors are available. Right now, we will use the default one, the DoG detector, which should be good enough for most cases. DoG means ‘difference-of-Gaussians’. It is an efficient approximation of the LoG (‘Laplacian of Gaussian’) filter, and there is also a detector in Mastodon based on the latter.

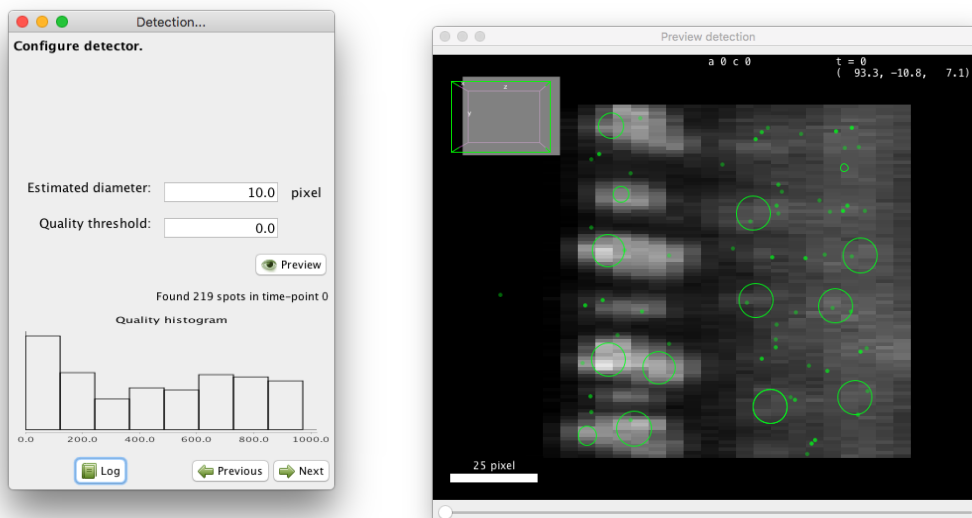
These detectors excel at finding roundish structures in the image that are bright over a dark background. The structures must have a shape somewhat close to a sphere, but they can accommodate a lot of variability. As a rule of thumb, if you can define rough estimate of the radius of these structures, they are eligible to be picked up by our detectors. This also implies that in Mastodon, we cannot segment complex shapes, or object labeled by their contour (*e.g.* cell membranes), or even exploit these shapes to have an accurate measurements of the volume. This is an important limitation of Mastodon.

For now, select the DoG detector and click Next.

Here is briefly how it works. The LoG detector, and its approximation the DoG detector, is the best detector for blob-like particles in the presence of noise (See [Sage et al, 2005](#)). It is based on applying a Laplacian of Gaussian (LoG) filter on the image and looking for local maxima. The result is obtained by summing the second order spatial derivatives of the gaussian- filtered image, and normalizing for scale. Local maxima in the filtered image yields spot detections. Each detected spot is assigned a **quality** value, that is obtained by taking the intensity value in the LoG filtered image at the location of the spot. So by properties of the LoG filter, this quality value is larger for :

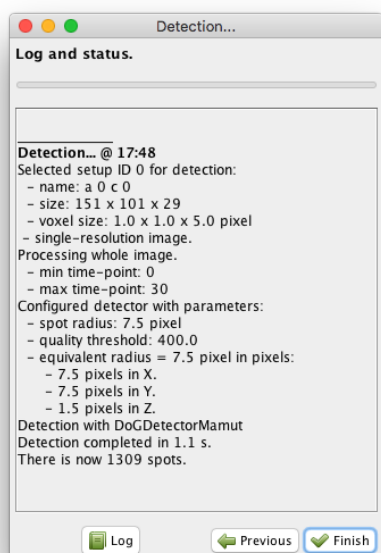
- bright spots;
- spots which diameter is close to the specified diameter.

The DoG detector requires only two parameters: the estimated diameter of the object we want to detect, and a threshold on the quality value, that will help separating spurious detections from real ones. The panel you are presented let you specify these parameters, and preview the resulting detection. Try with 10 pixels for Estimated diameter and 0 for the Quality threshold. The click on the Preview button. A preview panel should open shortly, showing detection results on the current frame:



These values are close to be good but not quite. You can see that the diameter value is too small to properly grasp the elongated shape of the cells along Z (the BDV view on the left panel above is rotated to show a YZ plane). Also the threshold value is too low, and some spurious detections are found below the epithelium. These spots have a low quality, that manifests as a peak at low value in the quality histogram displayed on the configuration panel. From the shape of the histogram, we can infer that a threshold value around 100 should work. However we also need to change the diameter parameter, which will change the range of quality values. After trial and errors, values around 15 pixels for the diameter and 400 for the threshold seem to work.

Note that you can run the preview on any frame. You just have to move the time slider on the preview window. Once you are happy with the parameters, click on the button. All the frames specified in the ROI (if any) will be processed. In our case detection should conclude quickly and the following panel should appear:



We now have more than 1000 cells detected and this concludes the detection step. Click on the **Finish** button, and the wizard will disappear.

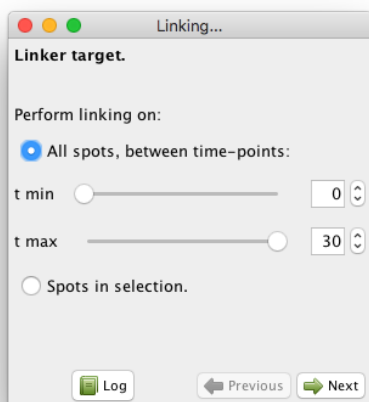
If you have complex images mixing several size of objects, or detection parameters that work for one part of the movie but not for another one, you could restart a new detection now, selecting for instance other parts of the movie with the ROI. You can do this and more, but for this kind of approach, the **Advanced DoG detector** offers more configuration capabilities, that will review later.

1.1.5 Linking cells.

What we just did is the detection step. It yields one Mastodon spot per cell, but the notion of cell identity propagated over time is missing yet. The particle linking step just does that. A particle linking or tracking algorithm accepts a collection of spots, ordered by frames (time-points), and tries to link each spot to the next spot(s) in the next frame or so. All the spots you can reach by starting from one spot and navigating across links build a *track*, and in our case it represents one cell (or any other object) followed over time. In most cases there is one spot per frame for a track, meaning that that a spot has at most one incoming link (spot from previous frame) and one outgoing link (spot in next frame). But some algorithms can accommodate *e.g.* dividing cells (2 outgoing links for the mother cell going to the two daughter cells) and merging events. There is a vast literature behind tracking algorithms, and it is an active domain of Research. A [paper](#) compares implementation and list some pros and pitfalls of many of them.

Selecting target spots for linking.

Like for the detection step, linking in Mastodon happens in a wizard. And also like for detection, the linking algorithms currently available in Mastodon are adapted from TrackMate. Launch the wizard from the GUI, with the *Plugins > Tracking > Linking...* menu item. The first panel you are shown lets you select what spots to include in linking:



There are two modes:

- Either you take all the spots between a start and an end frame. By default, all frames are selected.
- Either you specify you want to link only the spots that are in the selection. This mode offers a lot of flexibility when facing complicated cases. It is best used along with the [selection creator](#), that we will describe later in this manual.

For now, just leave the parameters as they are, which will include all spots in the linking process, and click next. You can now choose between several linking algorithms.

Available linking algorithms in Mastodon.

In Mastodon, they fall mainly in two categories.

The first two LAP trackers are based on the **Linear Assignment Problem (LAP) framework**, first developed by [Jaqaman *et al.*](#), with important differences from the original paper described [elsewhere](#). We focused on this method for it gives us a lot of flexibility and it can be configured easily to handle many cases. You can tune it to allow splitting events, where a track splits in two, for instance following a cell that encounters mitosis. Merging events are handled too in the same way. More importantly are gap-closing events, where a spot disappear for one frame (because it moves out of focus, because detection failed, ...) but the tracker can rescue missing detections and connect with reappearing spots later.

In Mastodon the LAP algorithms exists in two flavors: a simple one and a not simple one. There are again the same, but the simple ones propose fewer configuration options and a thus more concise configuration panel. In short:

- The simple one only allows to deal with gap-closing events, and prevent splitting and merging events to be detected. Also, the costs to link two spots are computed solely based on their respective distance.
- The not simple one allows to detect any kind of event, so if you need to build tracks that are splitting or merging, you must go for this one. If you want to forbid the detection of gap-closing events, you want to use it as well. Also, you can alter the cost calculation to disfavor the linking of spots that have very different feature values.

The third tracker is called **Linear motion Kalman linker**. It can deal specifically with linear motion, or particles moving with a roughly constant velocity. This velocity does not need to be the same for all particles. It relies on the [Kalman filter](#) to predict the most probable position of a particle undergoing (quasi) constant velocity movement.

How to pick the right linking algorithm?

The right choice of a particle linking algorithm is conditioned by the expected motion of the object you track. As a rule of thumb, you can make a decision following these simple rules:

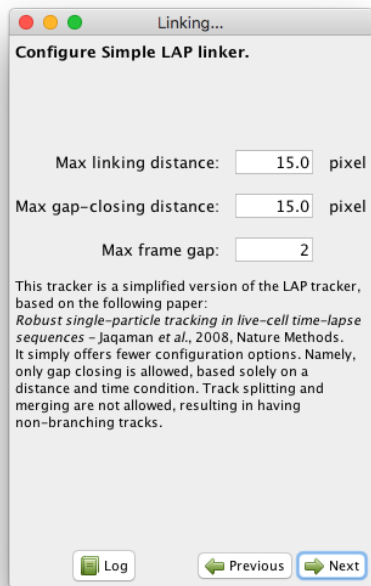
- If the objects you track are transported by an active process and have a motion for which the velocity vector changes slowly, then pick the **Linear motion Kalman linker**.
- If the object motion is random (like in Brownian motion) or unknown, pick on the LAP linker. If the objects you track do not divide, nor merge, pick the **Simple LAP linker**.
- If the objects divide or merge, or if you want to specify linking costs based on numerical features (like spot mean intensity), then pick the **LAP linker**.

In our case, we need the **Simple LAP linker**. Select it and click Next.

Running the Simple LAP linker.

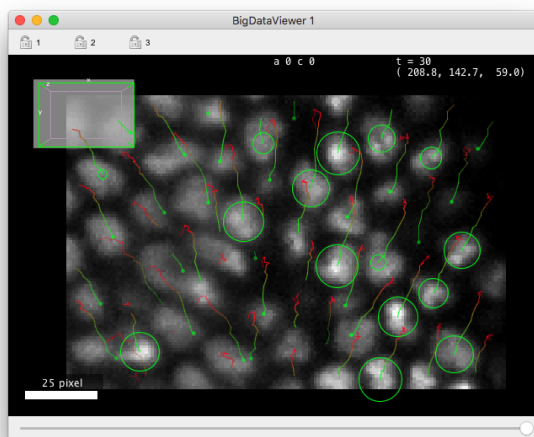
This linker only requires the specification of three parameters.

The first one is the **Max linking distance** during frame-to-frame linking. This is the distance beyond which linking a spot to another one in the next frame will be forbidden. For instance, if you know that your objects move by at most 5 μm from one frame to the next, pick a value slightly larger, for instance 6 μm . Distances are expressed in whatever physical units the BDV dataset specified. In our case it is pixels.



In practical cases, it can happen that the detection step might miss an object in some frames, then detect again later. These gaps will result in generating several small tracks for a single objects, which is one of the main course of spurious results when analyzing tracks. The simple LAP linker can bridge over missed detections. It does so by inspecting small tracks that results from frame-to-frame linking, and tries to connect the end of one with the beginning of another one. The last two parameters of the linker specifies how they are bridged. The `Max gap-closing distance` specifies how far can we look for candidates when we try to bridge the end of a track with the beginning of another one. The `Max frame gap` specifies how far they can be in time. For instance a `Max-frame-gap` of 2 means that we can bridge the end of a track at frame t with the beginning of a track at frame $t+2$. Which results of bridging over detections missed by no more than 1 frame.

In our case, the default parameters turn to work fine. Click `Next` and the linking will proceed. Click on the `Finish` button to end the tracking process. If you have a BDV window opened, it should be updated with the tracking results, like in image below.



By default the tracks are represented by colored lines, extending backward in time. Points in tracks that are close to the

current time-point are green and fade to ref for points that are far back in time. When you change the Z focus, the spots are painted as circle of radius corresponding the the intersection of the sphere with the current Z-plane. When the spot sphere does not interest with current Z-plane, it is painted as a small dots. The points of the track away in time that are not close to the current Z-slice are faded away. We will see later how to customize the display of tracks.

1.1.6 Wrapping up.

This concludes our first tutorial on automated tracking with Mastodon. To continue with the next one, save the project with the tracks you just generated.

As you can see, after creating a project from a BDV file, the process consists mainly in running in succession the two wizards, one for detection, one for particle linking. Even if they provide fully automated tracking, Mastodon is made for interacting with the data as you generate it. We will see in a next section how to manually edit a spot, a link or a track even at the finest granularity. But keep in mind that the tools we quickly surveyed can be used interactively too. First the wizards let you go back to change a tracking parameter and check how the results are improved or not. Second, because you can specify a region-of-interest (ROI) in the detection step, and select the spots you want to track in the linking step, several runs of these wizard can be combined on different parts of the same image, to accommodate *e.g.* for changing image quality over time, or cell shape over time. Mastodon aims at being the workbench for tracking that will get you to results, accommodating a wide range of use cases.

1.2 Manually editing tracks in Mastodon. TrackScheme.

One of the key feature of Mastodon, the implementation of which made our lives extremely hard and rich, is the ability to edit manually at any time any spot or link within an annotation that can contains billions of them, while retaining a good and pleasant response time from the software.

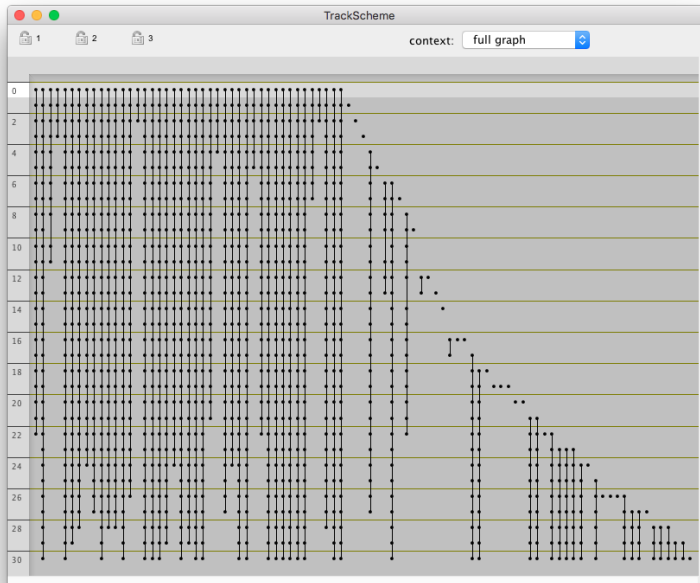
In this tutorial, we will show how to do just that, from existing tracking data, the one we generated in the previous section. We will use it as an opportunity to present **TrackScheme**, the track visualizer of Mastodon, as well as introduce several useful editing features such as the undo/redo mechanism. We will also introduce the various visual hints that help the user knowing what spot or link is currently edited or inspected, such as the **focus**, the **highlight** and the **selection**.

Manual editing is often **not desirable**, first because it is not objective which might be detrimental in situations where the performance of an algorithm is evaluated, or when an unknown motion characteristics is investigated. Second, because it does not look like a realistic solution when *e.g.* thousands of cells are to be followed over thousands of time-points. Regarding the latter however, several courageous individuals sacrificed their time, energy and sometimes sanity in doing so, for the sake of finally a scientific answer, or curating the results of an automated analysis, when there was no other working solutions. See for instance [Wolff, Tinevez, Pietzsch et al, 2018](#) or [McDole et al, 2018](#). If you are going this way, Mastodon aims at providing tools to do so, in the hopefully fastest and least painful way possible. Also, you can combine the fully automated approach of the previous chapter with manual editing to correct hopefully a few numbers of mistakes. Finally, there is also a semi-automated tracking tool, that we will introduce later.

This tutorial will start by presenting you the manual editing tools of Mastodon, as well as and the focus, highlight and selection tools. Because this catalogue of actions can be a bit dry and long to swallow we will in the last paragraph use the techniques presented here to remove and rebuild some tracks from the results of the last tutorial.

1.2.1 TrackScheme, the lineage view and editor.

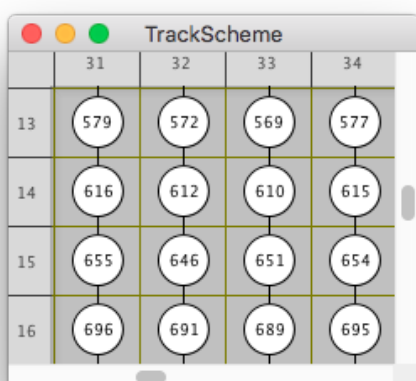
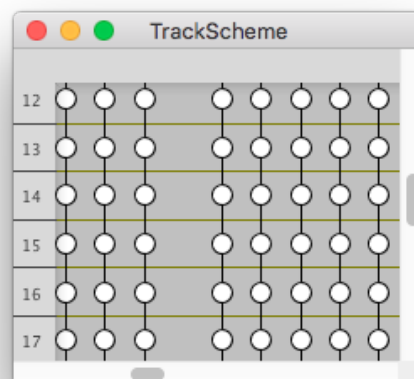
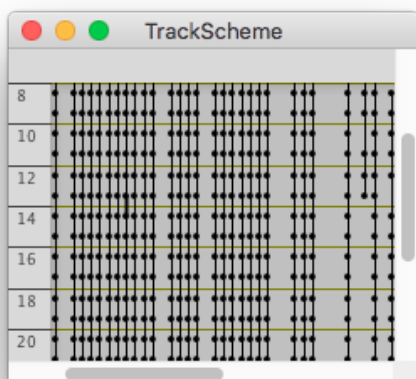
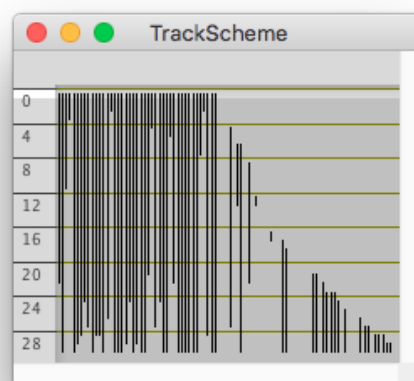
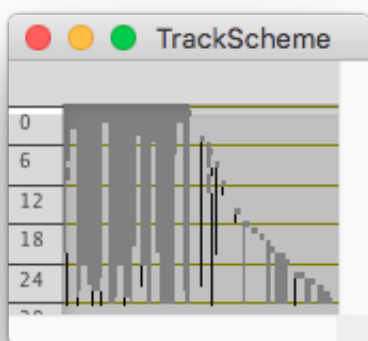
First, load or retrieve the tracking data of the *previous tutorial*. You should have a few hundreds of tracks. To open a TrackScheme view window, press the `trackscheme` button on the main window.



The TrackScheme view offers a special way of displaying tracks. If you are familiar with TrackMate, you will see that we brought the same kind of features here, but scaled to large data. You can think of TrackScheme as a workbench for tracks, where you will edit, cut, stitch and rename them. It displays a kind of “track map”, where a track is laid on a panel, arranged vertically over time, as a Parisian subway train map. Tracks are displayed hierarchically, discarding the spatial location of each spot. Each track is laid out going through time from top to bottom. One horizontal line corresponds to a single time-point in the movie. One vertical column corresponds to a single track, that is all the spots that are connected by links over time, including divisions and merges. It is a great tool in particular when studying and editing cell lineages.

When opened, the view in TrackScheme is scaled so that the full data is shown, all tracks over all time-points. Even on our small dataset, most of the tracks and spots appear as single lines or dense boxes. To see the details of each track, you need to zoom in and navigate around the data. The navigation actions and their mappings are listed in the *TrackScheme keyboard shortcuts table*.

Try to zoom in until you can see the label of a few spots. TrackScheme implements an adaptive level of details depending on the zoom level, so that we can accommodate plotting a large amount of data without compromising the responsiveness of Mastodon. On the finer level of details, spots are plotted as circles, with the spot label shown. As you zoom out, they become just empty circles, then points, then they disappear to only show the track as a line. When the zoom level is so low that several tracks coalesce, they are drawn as a gray box.



1.2.2 The focus and the spot labels.

You probably noticed that some spots are painted differently from their neighbors. Mastodon manages three special collections of spots and links to facilitate making sense of the data across views:

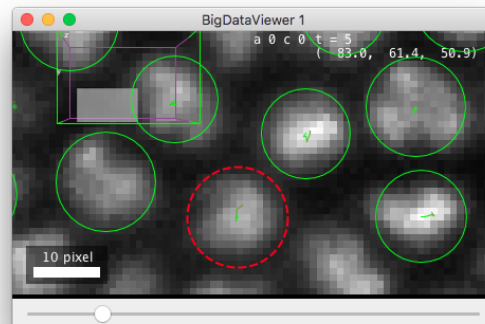
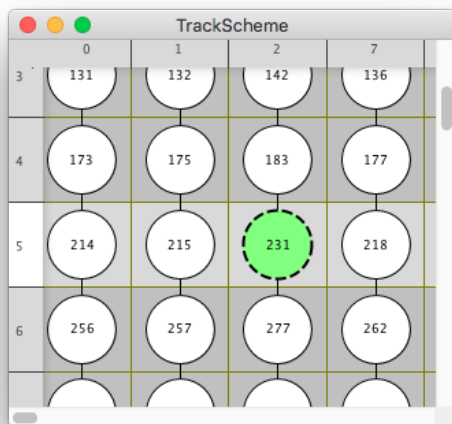
- the **selection**, painted in green, that manages a classical selection of spots and links;
- the **highlight**, used to highlight the spot or link currently under the mouse;
- the **focus**, particularly used in TrackScheme, to indicate what spot is currently focused on by the keyboard interaction.

We will first present the focus.

Moving the focus.

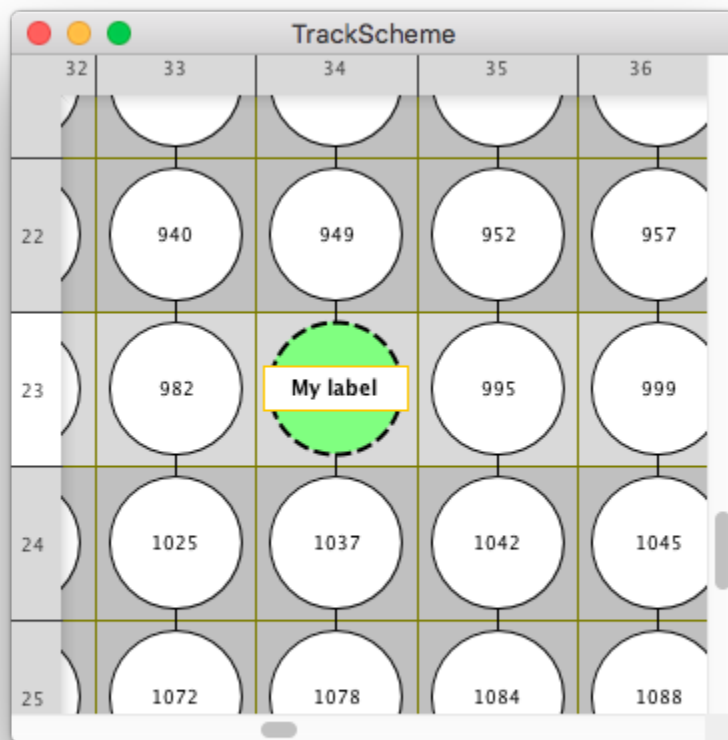
The spot that has the focus is painted with a dashed contour (see figure below). Only one spot can have the focus and it is set by mouse or keyboard interaction. Click on one spot in TrackScheme when fully zoomed, and move across track and time with the arrow cursor keys. If the focus reaches the border of the window, the view will be moved to follow it. You can think of the focus as the caret in a text editor. It is meant to facilitate keyboard interaction. See the [focus navigation keyboard shortcuts](#).

You can also jumps across branches. A branch in a track is a linear section of the track between divisions, fusions or the track end or start. `Alt ↑` and `Alt ↓` will move the focus to the branch start and end. In the case where you could navigate to several branches, for instance you are currently at a cell division and you can go the end of either daughter cells branches, you can navigate to one or the other with `Alt ↓` or `Control Alt ↓`.



Editing the spot labels.

By the way, the focus is used to rename individual spots. Zoom in so that we can see the label of the spots and move the focus to a spot. Press **Enter**. A small editing box appears inside the spot and lets you change its label.



By default the spot label display the spot ID. If you edit the label then it shows the new label you entered. Editing the label of a spot does not affect its ID or any other properties. The spot label is just a convenient text field that you can use to annotate cells and search them later. Only the spots have a label. The links don't.

The order of tracks in TrackScheme.

The spot labels also control how the tracks are ordered in TrackScheme. We said before that in TrackScheme all the spatial information is discarded. However the tracks are laid out in a deterministic order. This order is set by the label of a track.

There is no special structure to follow individual tracks in Mastodon. For this, we simply use the first spot of a track. So when we speak of the label of a track, we simply means the label of the first spot (in time) in the track. The tracks are arranged from left to right following the alphanumerical order of the track labels. So you can change the tracks arrangement by editing theirs first spot's label. For instance a track named A will be laid out to the left of a track named B, and a track named D9 will be put to the left of a track named D10.

	A	B	D9	D10
0	A	B	D9	
1	46	57	56	D10
2	92	100	89	90
3	134	143	133	135

If you change the track labels now, the tracks will not move immediately in TrackScheme. For the new arrangement to happen, you need to either open another TrackScheme window, or to edit the data, which we will see soon.

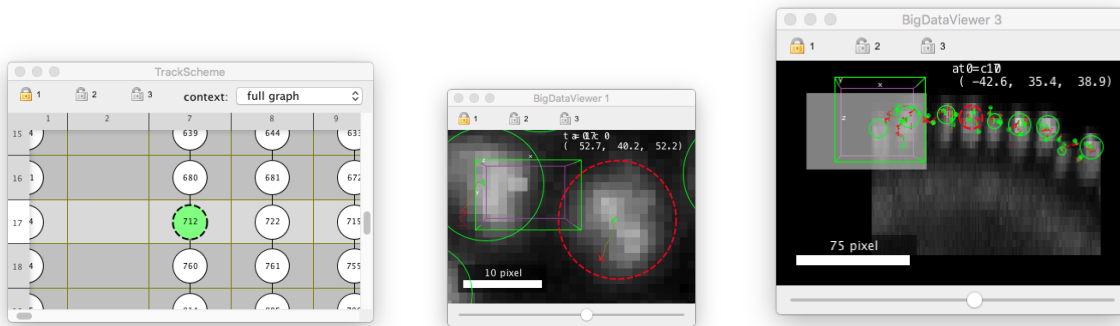
The focus in BDV views.

The focus also works in the BDV views. In these views, the spot that has the focus is also painted with a thick dashed circle. Parenthetically, notice that the focus is shared across all opened views. If you have a TrackScheme view and a BDV view opened, and that they both contain a display of the same spot, setting the focus in one view will update this very spot display in the other views.

When you are in a BDV view and have set the focus, you can also navigate with the arrow keys. After clicking inside a cell, you can navigate in time and follow a cell lineage through the movie with \uparrow and \downarrow . If you reach the border of the view or if the cell moves in Z, the view will be translated to keep the cell in focus. However, you cannot edit the spot labels in a BDV view nor jump from one track to another with the arrow keys. Navigating across branch with $\text{Alt} \leftarrow$ and $\text{Alt} \rightarrow$ however works.

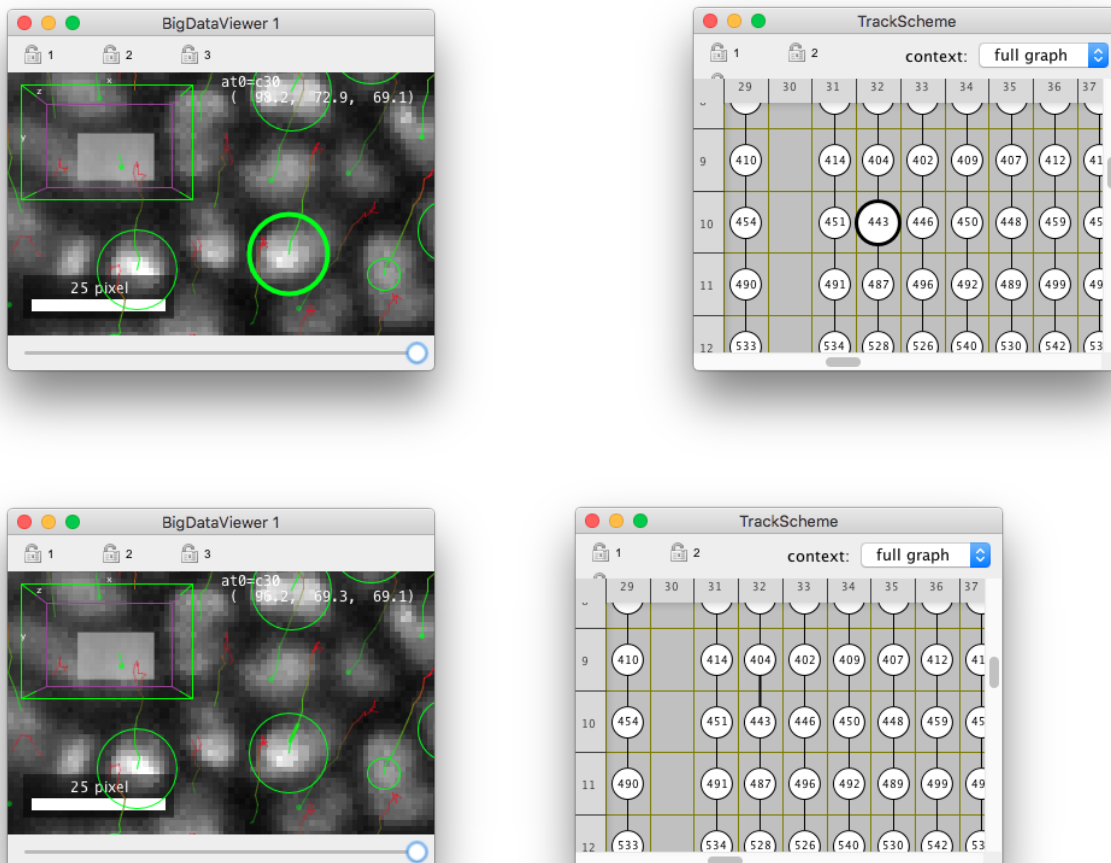
1.2.3 Synchronizing several views together.

You probably noticed that both BDV and views have a toolbar on the of the view itself. This toolbar can be hidden and shown by a press of the T key. In both view types, the toolbar contains 3 gray locks, that are used to link several views together for navigation. For instance, if you click on the lock 1 on a BDV view and the same lock on a view, they will be synchronized. If you move with the focus in the view, the BDV view will be translated to show the focused spot. If you Double click on a spot in any view, all views in sync will translate to show the spot. This is very handy to navigate around in TrackScheme, for instance following a cell over time while the BDV view displays where it is in the sample. You can even combine several BDV views in sync at different magnification to have both an overview of the cell position in the sample and a close view of the cell itself.



1.2.4 The highlight.

The highlight is the second visual hint we will introduce. You probably already noticed it and used it: the display of spots and links that are just below the mouse cursor changes. Highlighted spots and links are painted with a thick continuous line. To highlight a spot or link you just have to lay the mouse over it. As for the focus, the highlight is common to all views, and if you highlight a spot or a link in a view, its display is changed in all the views that show it.

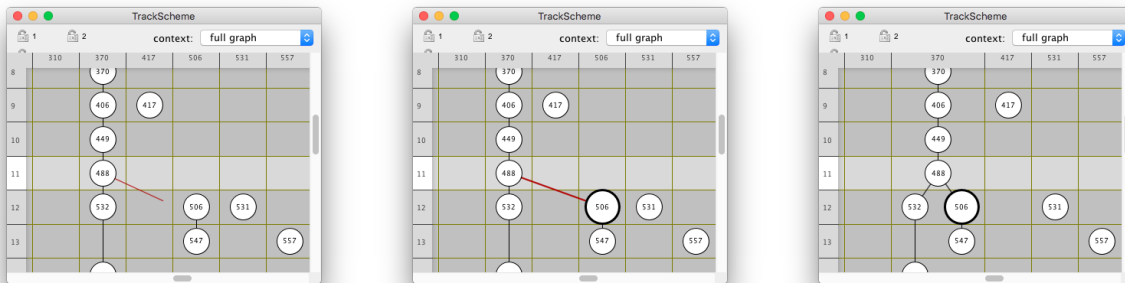


1.2.5 Deleting individual spots and links.

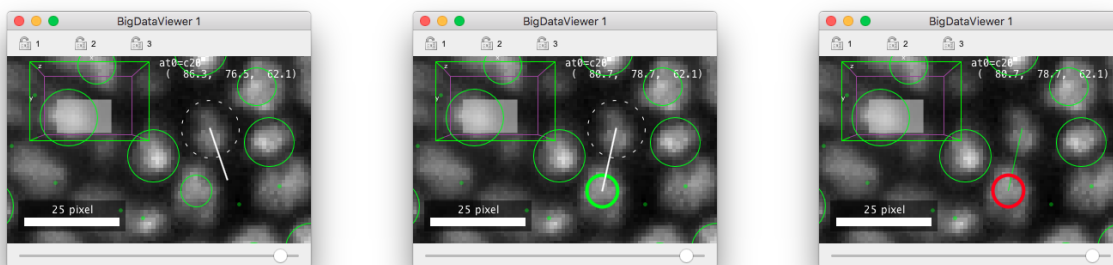
The highlight is especially important for track editing. The commands that delete a single object are actually applied to the highlighted spot or link. For instance, to delete a spot or a link in , simply move the mouse cursor over it so that it is highlighted and press D. You should see the tracks being rearranged following the deletion. In a BDV view, the highlight and deletion mechanism is the same.

1.2.6 Linking spots.

Creating links between existing spots happens in a similar way, except that you have to point to a source spot and another target spot for a single link. In TrackScheme , move inside a spot until it is highlighted (no mouse-click). You must be at a zoom level large enough so that a spot is at least painted as a dot or a circle. Then press and hold the L key. Without releasing the key, move the mouse out of the spot. You should see a red line representing the link to create. Move the mouse to the desired target spot, until it becomes highlighted. The red line should snap to it and become thicker. If you now release the key now, a new link between the source and target spot is created and in TrackScheme it should cause a rearrangement of the tracks. Notice that you cannot add a link between spots in the same frame. Also notice that you can toggle links this way. If you draw a link between two spots that are already connected, their link will be removed.



In BDV views, it is again very similar. Move the mouse over a spot until it is highlighted then press and hold the L key. The view will automatically move to the next frame. The link to create will be painted as a white line, and the ghost shape of the source spot is painted as a dashed white ellipse. Move the mouse to the target spot in this frame until it is highlighted, then release the key. The link is created. If you press **Shift L** in a spot, the BDV view will move to the **previous** frame, and create a backward link.



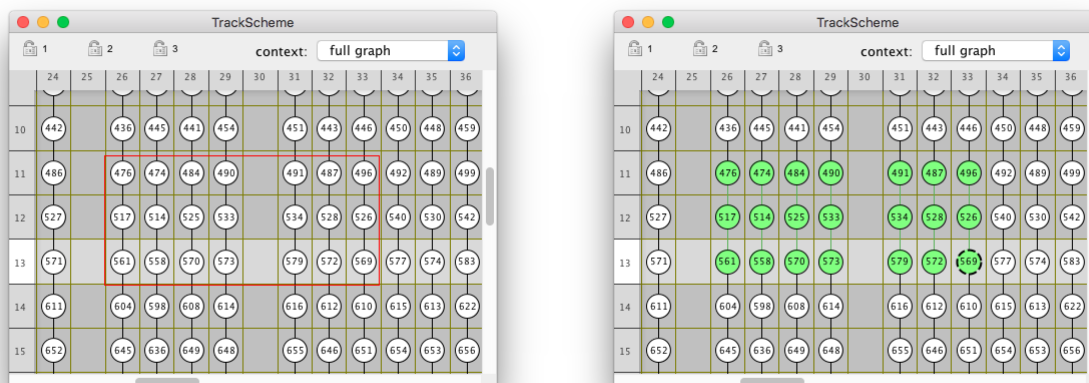
You cannot move in Z while creating a link this way. You must orient the BDV view so that the source and target spot are roughly in the same displayed slice. However you can move in time. While still holding the key, press the or and or key to navigate backward and forward in time. This way you can create links between spots separated by more than one time-point.

1.2.7 The selection.

The selection is the third visual cue of Mastodon and probably the most important one. It behaves and has the role of a classical selection tool you would find in another software that manages a collection of data items. It is meant to select a subset of spots and links and apply operations to this subset. Contrary to the focus and the highlight, any number of spots and links can be put in the selection.

The selection in TrackScheme is built in a classical way. Click and drag from an empty part of the view. A red box appears to let you select an area of the view. Release it when it contains the part of the tracks you want to select, and selected items should now be painted in green. All spots and tracks in the selection box are put in the selection, even if the zoom level is too small for them to be painted. `Control A` selects the whole data. There are variants to select only all the spots (`Control Alt A`) or only all the links (`Control Shift A`).

`Shift` click on a spot or a link to toggle it in the selection. To clear the selection, click on an empty part of the TrackScheme view. In you can also add spots - but not links - to the selection using the focus. For instance, if you combine navigating with the focus while holding the `Shift` key, spots that you navigate to will be added to the selection. `Shift Alt ↓` will add all the spots from the focused spot to the end of the branch it belongs to. Finally there are commands to add a full track from the currently focused spot. `Shift space` will select the whole track of the focused spot. `Shift` selects all the spots and links in the track that are forward in time relative to the focused spot, and `Shift` does the converse backward in time.



The selection is shown in BDV views as red or magenta spots and links by default. In BDV views you can only edit the selection via `Shift` click. The interaction with the focus also work as described above. These default key-bindings are recapitulated in the [table of shortcuts for the selection actions](#).

1.2.8 Editing spots and links with the selection.

The selection actions are all applied in bulk. You can delete all the spots and the links currently in the selection with `Shift` . If your selection includes spots that are not linked, pressing will `Shift K` link them sequentially. If the selection contains more than one spot per frame, only one of them, picked randomly, will be linked.

1.2.9 Manually adding spots and linking them.

Adding new spots requires specifying where to add them. Since does not include spatial information you cannot use it to add spots. You can only add spots in the BDV views.

To add a new spot, simply hover the mouse in a BDV view where you want the spot to be created, and press `A`. A new spot will appear, with a radius given by the last radius value you used. You can respectively decrease or increase the radius of a spot by pressing `Q` and `E` . Modifiers affect the amount of radius changes. `Shift Q` and `Shift E` change the radius by a larger amount and `Control Q` and `Control E` by a finer amount.

Spots created this way are not linked. It might not be the quicker way to follow manually a cell over several frames. To do so, you can use a little trick. When you want to add a spot, and link it to an existing one, arrange the BDV view so that this source spot is visible in the displayed time-point. Then place the mouse cursor *inside* the source spot, and press and hold `A`. Like for creating links, the BDV moves to the next time-point, paint the source spot and link with white, dashed lines, and let you position the new target spot. While you keep pressed, you can move the target spot around. When it is placed in the desired position, release `A`. The new spot is created and linked to the source spot. This way, you can quickly follow a cell or an object and manually creates a track for it by just positioning the mouse with a few presses of `A`. If you press `Shift A` while in the source spot, the new spot will be added in the **previous** frame.

Another equivalent possibility is to use the **auto-linking mode**. To toggle this mode on or off, press `Control L` (the viewer will display a message stating whether it is on or off). When the auto-linking mode is on, everytime a spot is added (`A`), it is *automatically linked to the previously selected spot*, **if** there is exactly one spot in the selection. The newly created spot then becomes selected so that all spots added in succession while in this mode are linked to together. The links are created whether new spots are forward of backward in time, which can be difficult to visualize sometimes. It is wise having a TrackScheme window opened linked to the BDV in which you are doing the editing.

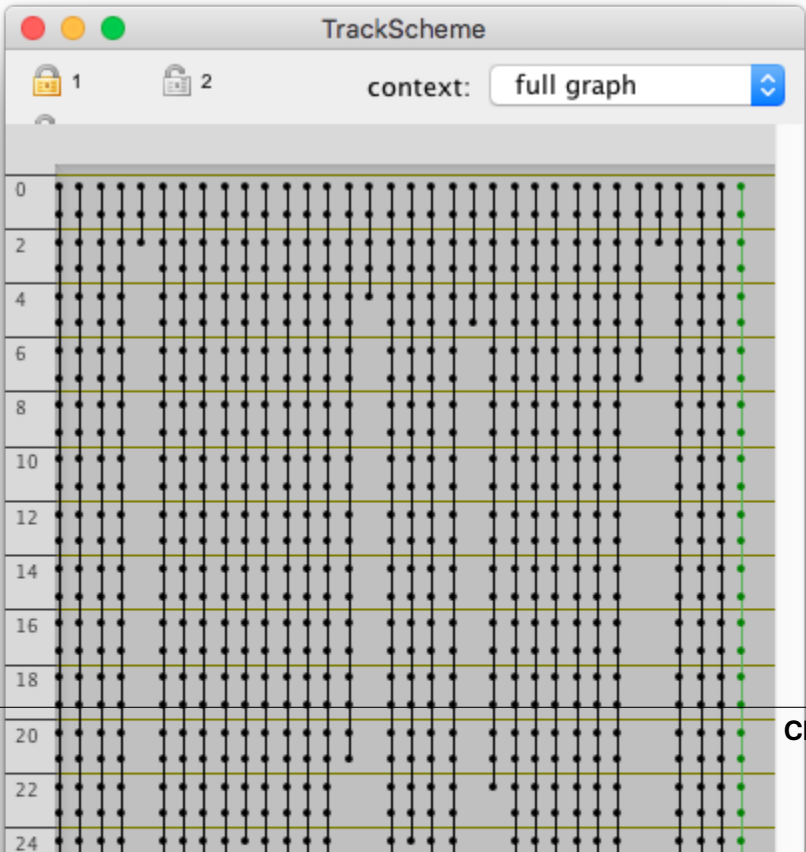
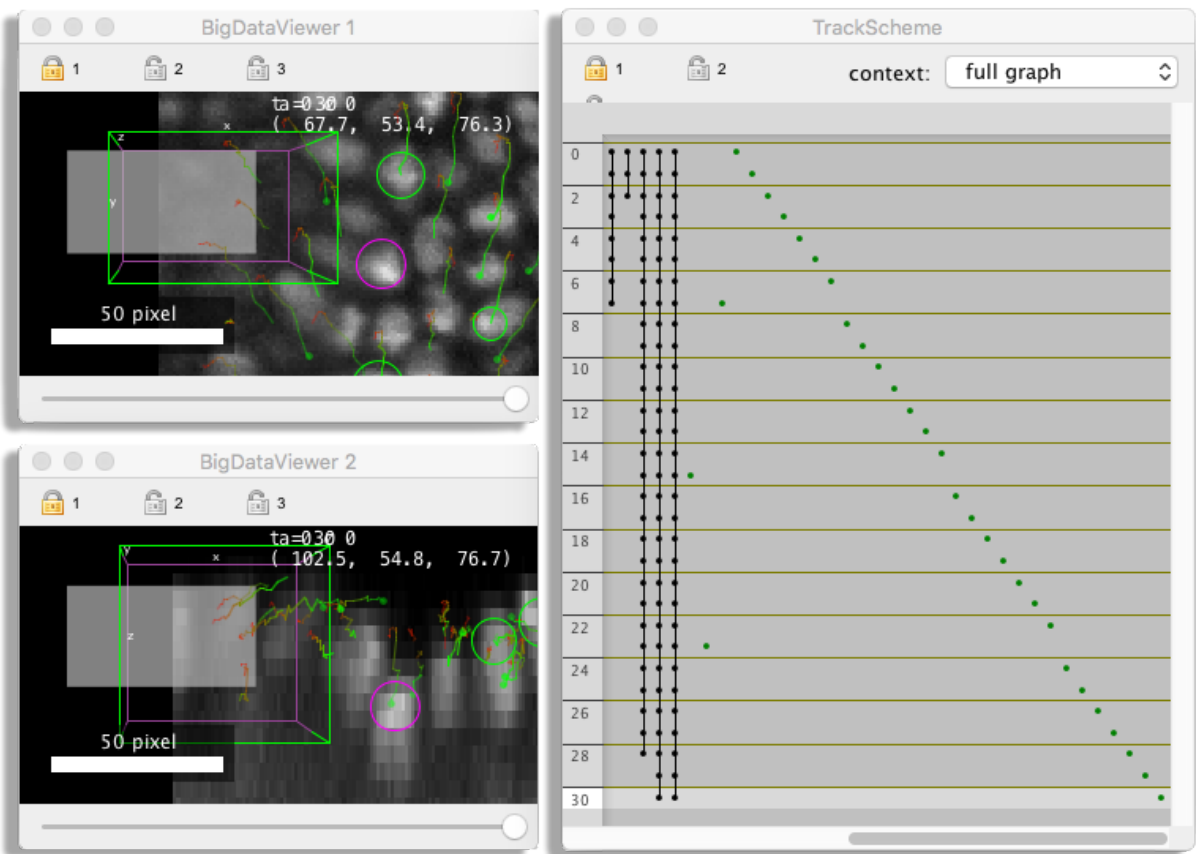
1.2.10 Moving spots around.

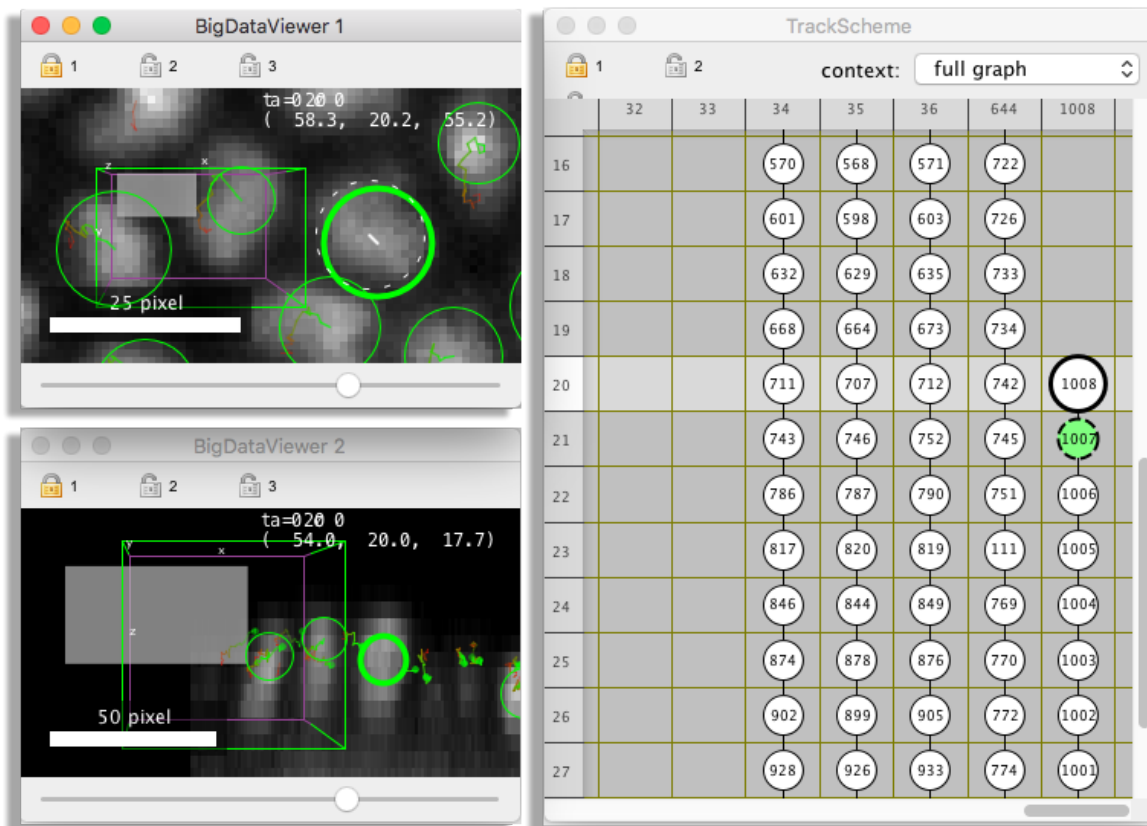
To move an existing spot we use a similar interaction with the spots. In a BDV view, place the mouse cursor inside the spot you want to move, then press and hold `space`. While you hold `space` pressed the spot will move where you move the mouse. Release the `space` key at the desired location.

1.2.11 The undo/redo mechanism.

One of the most useful actions is undoubtedly undo and redo. To undo just press `Control Z` and to redo use `Control Shift Z` . There is no limit to the number of undos you can do. And everything can be undone in Mastodon, even tag definitions. In our humble opinion, this is one of the nice reasons for Mastodon to exists: a tracking and lineaging software that aims at combining automated and manual approaches, in a very efficient way..

1.2.12 Putting things in practice.





Make sure you have one BDV view and one view open, and that both are linked using the same lock. In TrackScheme, select one of the spot or link that belongs to one of the long tracks; the ones that extend from the first time-point to the last. Press **Shift space**. You selected the whole track to which the spot or link you selected belong to. Press **Shift .** The track has been removed.

Let's remove a couple of other tracks another way. Click and drag in to draw a selection box around some long tracks. Then press **Shift .** Our model should now lack several good tracks we will try to put back manually. For clarity, also remove all the tracks on the right part of the view, that do not start at the first time-point.

In the BDV view, go to the first time-point, and look for a cell whose spot has been removed. Pan the view (right-click and drag), adjust the zoom (**Control Shift Mouse-wheel**) and the Z position (**Mouse-wheel** with or without **Shift**) so that the cell is well in view. Then put the mouse cursor over it and press **A**. A spot has been added on the cell. The fine adjustment of the spot position in 3D can be made easier with a second BDV view. Open another BDV view, zoom and rotate it so that it is aligned with the XZ plane by pressing **Shift Y**. Add it to the lock group by toggling the right lock in the toolbar. Go back to the first BDV view, and align it with the XY plane by pressing **Shift Z**. You must now find the spot we just added. If you cannot find it in any of the two BDV view, look for it in TrackScheme. It will be in the rightmost column at the first line, since we added the spot to the first time-point. Double-clicking on it in TrackScheme will center the two BDV views on it. Now that the two BDV views are centered on the spot, you can adjust its XY and Z positions over the cell using the two views. In any of the two BDV views, put the mouse cursor, then press and hold **space**. Move the spot the desired location and release **space**. The spot might not have the right diameter to encompass the cell. Change its radius by pressing **Shift Q** and **Shift E** (without **Shift** for finer adjustments) until you are happy with it.

You must now do it for the same cell in the subsequent time-points. Press in a BDV view to move to the next time-point. Again, put the mouse cursor roughly at the center of the cell and press **.** Adjust its position with and move to the next time-point.

We have to repeat this for all the time-points of the movie. This might be tedious but does not have to be too long. When I need to do this I adopt a posture similar to what I have when I play PC video-games: the right hand on the mouse, the left hand over the left side of the keyboard, over the A, D, Q and E. The **Shift** key can be accessed with the little finger, the space key with the thumb. The **N** and **M** keys to navigate over time are accessible with the left thumb also.

We created a bunch of spots, but they are not linked. Move to the **TrackScheme** view. You will find the spots you last created there always at rightmost part of the view. Since they are not linked, they should appear each in their separate column, in a stairway manner. Select them all by dragging a selection box around them and press **Shift K**. They should now link together into a new track. With one of the spot of this newly created track selected, press to jump to the first spot in the track. Then press to edit its name into something like *My first track*.

We can create a track directly. Let's do this by backtracking a cell from the last time-point to the first. In a **BDV** view, move to the last time-point, and place the mouse cursor over an annotated cell. Create a spot over it by pressing and center and resize it until satisfaction. Now keep the mouse cursor inside this spot, and press and hold **Shift A**. The **BDV** view moves to the previous time-point and creates a new spot there, already linked to the other one. By repeating this you can create a track that will backtrack the cell until it appears in the movie.

We have now restored two of the tracks we removed at the beginning of this paragraph. Hopefully now you feel enough at ease to do these manipulations quickly, efficiently and without too much hassle. But one of the concluding beauty of this tutorial is that you can restore the data as we had it at the beginning by roughly 100 presses of the **Control Z** keys.

1.3 Inspecting large datasets.

In the previous chapter we have seen how to edit single spots and links in Mastodon, what can be called point-wise editing. As we said before, the goal of Mastodon is to let you harness very large images, for which the number of annotations can be very large too. It can be very easy to get lost within such large images and loose track of where we are within the sample image and what cell we follow. So we have added several features that are made especially to get your bearings in large datasets. More than anything, these features are about giving visual cues that ease orientation, and exploit events and signal that would help a human brain get a sense of orientation. We took some inspiration from video-games, that are very good at communicating condensed and synthetic information to the player (but only for a limited part; there is no screenshake when you delete a spot).

1.3.1 Bookmarks in the BDV views.

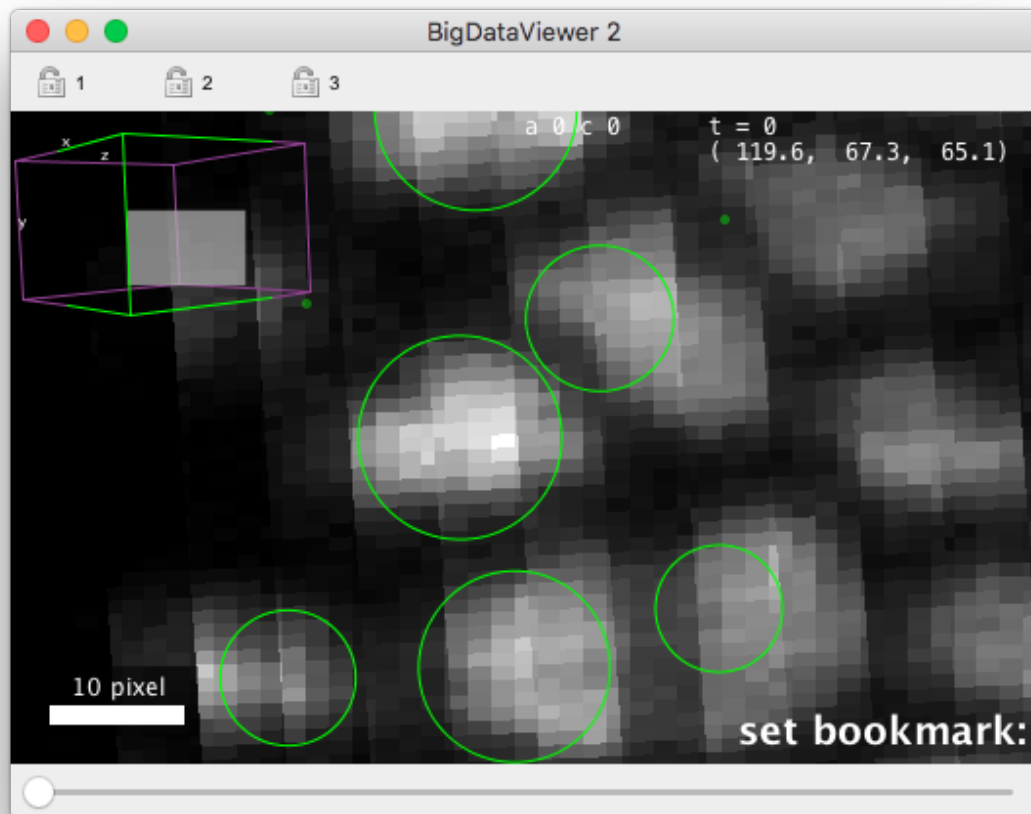
The **BigDataViewer** (BDV) is the image component of Mastodon and is meant to deal with very large images. It offers interactive and responsive user interaction, and to achieve this without any hardware acceleration, it resorts to only displaying a 2D slice through the data. This 2D slice can be arbitrarily positioned and oriented. When it comes to annotating a 3D image, using a 2D slice is a good approach. A full 3D view might actually hinders proper and efficient annotation of the data with a flat 2D screen and a mouse. The 3D view leads to ambiguities about the depth positioning of your cursor, and the image data that stands between the camera eye and the plan of interest may hide it. Parenthetically, these issues with interacting with 3D data are best solved with virtual reality devices, but Mastodon is not a tool that exploit them. The 2D view offer clarity but conversely does not offer a great feeling of the context.

However to facilitate orienting yourself, or retrieving a key point in the data, you can register bookmarks in the **BDV** views. The bookmarks were already implemented in the tool itself before its use in Mastodon. They let you store a position and orientation in space as bookmarks. You can later call them again and retrieve said position.

- First move to the position and orientation you want to store in a bookmark.
- Then press **B**. You should see a message prompting you to press another key (see the image below).
- Pick one and press it. This key will be used as a tag for this bookmark.

- To later retrieve the position and orientation of this bookmark, press B then the bookmark's key. The view should animate and restore the stored position and time-point.
- O does the same things, but only restore the bookmark orientation, not its position.

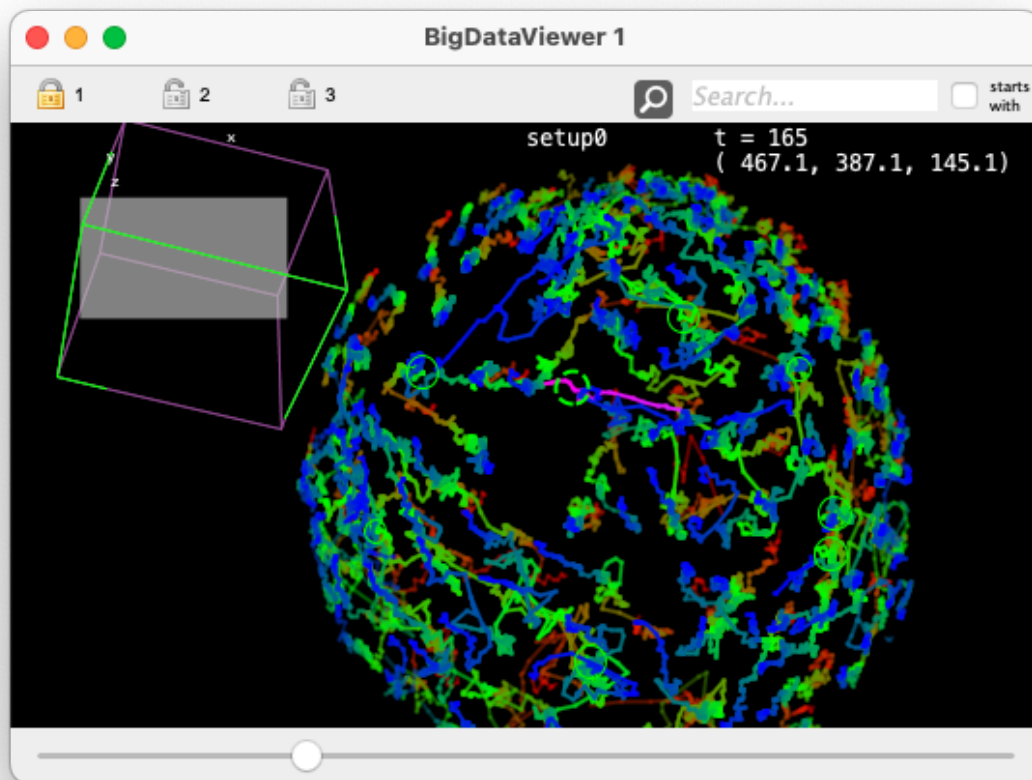
You can have many bookmarks, all identified by the key you press after the bookmark command. Bookmarks are saved with the BDV settings file that also saves the channel color and display range. You can save such a file with the or the key. The settings file is loaded when a new BDV window is displayed.



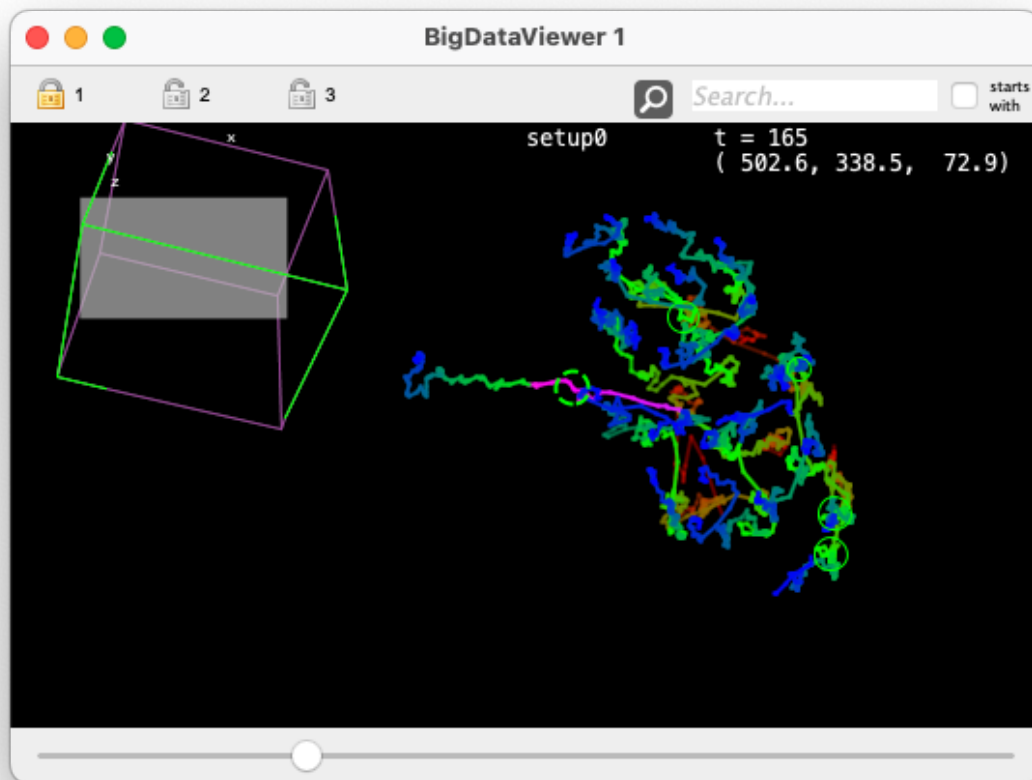
1.3.2 View modes in BDV.

The BDV views has also four different visibility modes that help making sense of tracks of interest. You can cycle between them by pressing the V key with a BDV active. These four views are:

All. In this mode (the default), all the tracks are visible. It follows the display configuration you set with the render settings however.

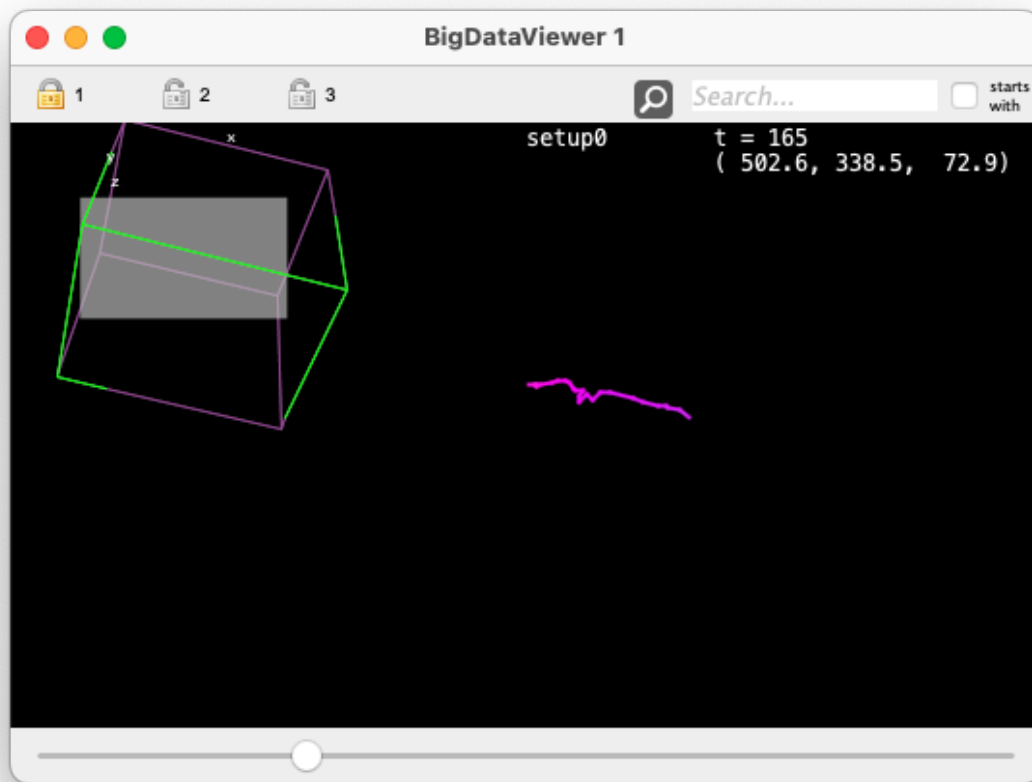


Track of focused vertex. In this mode, only the track (*e.g.* the whole lineage of a cell) of the currently focused spot is displayed.



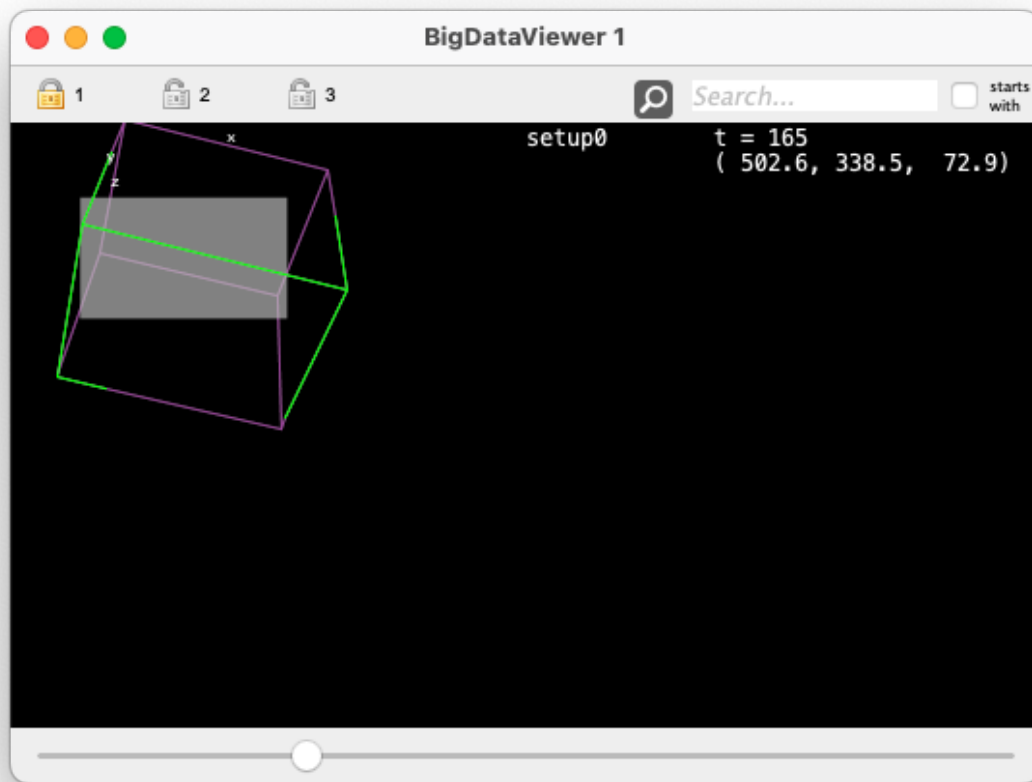
This visibility mode is particularly useful in conjunction with a TrackScheme window, where you would move the focus across different lineages with the keyboard. Comparing the image above with the image of the previous mode (movie and annotation by Mette Handberg-Thorsager and colleagues in the Tomancak lab), you can see with this visibility mode that the lineage of a progenitor tends to colonize only one side of the embryo.

Selection only. In this mode, only the content of the current selection is displayed.



This visibility mode is best used along with the *Selection creator*, the we describe later, and that lets you set the selection based on various criteria.

None. In this mode, the tracks are not displayed.



1.3.3 Linking several views together.

You can generate as many views as you want in Mastodon, and you can link several of them via the lock system. This is a good way to improve the perception of context, by linking several views that display for instance a close-up view of the data and another view displaying a bird-eye view of it. We already presented this feature in the [previous tutorial](#). The figure of this section shows an example of a view configuration with three views in sync, showing each a different level of desired information. We direct you there for details on how to use the lock system.

1.3.4 In TrackScheme everything is animated.

If you went through the previous tutorial, you probably noticed that editing events are associated with animations in TrackScheme. For instance, if you delete a link in the middle of a track, the ‘bottom’ part of the track will move to the left side of TrackScheme, in a quick animation. If you undo the deletion, the branch will move back to its original place the same way. Deleting a spot makes it fade rather than disappear. These animations are more than a toy. Something we learned that hard way with [MaMuT](#) is that point-wise editing the data can completely confuse and disorient the user. A single link deletion will generate a big rearrangement in the track hierarchy, and therefore will change the view a lot. Without any subtle cues to the user, these changes will disorient them quickly. Animating the editing events is a great way to hint them about what happens to the data modify in a user-friendly way.

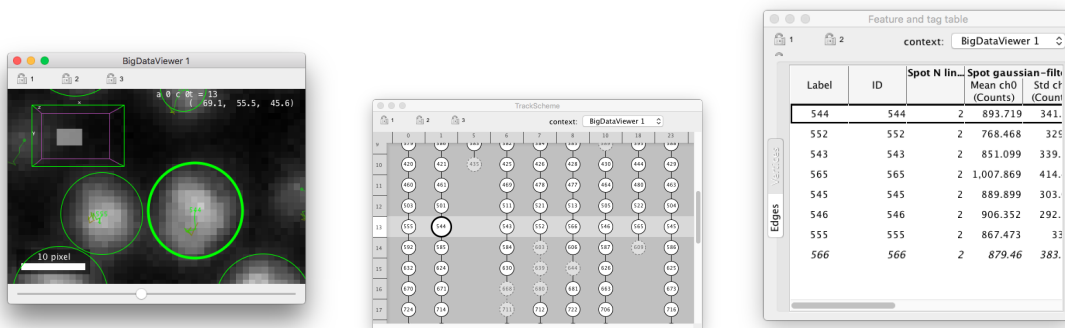
We also added some fluidity and inertia in navigation. In the [TrackMate](#) and [MaMuT](#) version of TrackScheme, panning and zooming were done in discrete discontinuous steps, that would also lead to confusion. In Mastodon, moving and

zooming are done continuously. There is even some inertia again to emulate interacting with a tangible panel.

1.3.5 Spatial context in TrackScheme.

There are situations where the density of cells displayed in is so high that even at high zoom level this view is barely useful. The hierarchical arrangement of tracks in is handy to grasp how tracks behave over time. But since it eliminates spatial information, the neighbors of a track do not bring information to it.

The spatial context in reconciles spatial information with the hierarchical layout. In views, the toolbar has a `context` list box, from which you can select between `full` `graph` and the names of all the BDV views currently opened (typically `BigDataViewer 1`, *etc*). When `full` `graph` is selected, the full lineage data is shown in TrackScheme. This is the classic view. If you pick an item corresponding to an opened BDV view, then this view will only display the lineages of the cells currently displayed in the target BDV view. And the view will be updated (and animated) as you pan, move in Z, zoom or unzoom the BDV view.



Try it now with the data from the previous tutorial. Open a BDV view and a view. In the view, select the `BigDataViewer 1` item (it might not be 1 in your case). Then in the BDV view, zoom so that almost only one cell is displayed. The view should display considerably fewer tracks. As you unzoom, some lineages will appear in TrackScheme. You will probably see that some lineages appear in gray, with dashed lines. They are called “ghost” lineages: These are the lineages of cells that are not within the BDV view at the time-point currently displayed, but that will enter this view later or earlier in time. The context feature is immensely useful when studying tissue development or coordinated cells movement.

1.4 Numerical features and tags. The table view.

Mastodon is a tracking and lineaging tool. Its output is a collection of tracks, and the analysis of these tracks to yield statistics on velocity, displacement is carried out in another software package such as MATLAB or Python. Nonetheless you will find in Mastodon tools to compute *numerical features* on data item. Numerical features are numbers that can be calculated on spots, links and tracks of the data. For instance there are feature for the number of links that touch a spot, or the displacement of a link or the number of spots in a track. You can find them within Mastodon because it is convenient, but also because they are very useful for the interactive exploration of your data. Coupled with feature-based coloring, the display and sorting of values in the table view and the selection creator tool, they can considerably accelerate and facilitate making sense of the data.

Numerical features are numbers that classically relate to a physical quantity. When we need to *categorize* items, we rely on *tags*. We describe them just below. This chapter will also show you how to compute numerical features and create a coloring view from the feature values and tags. Doing so, we will introduce the third kind of data view in Mastodon: the data tables.

1.4.1 Tags and tag-sets.

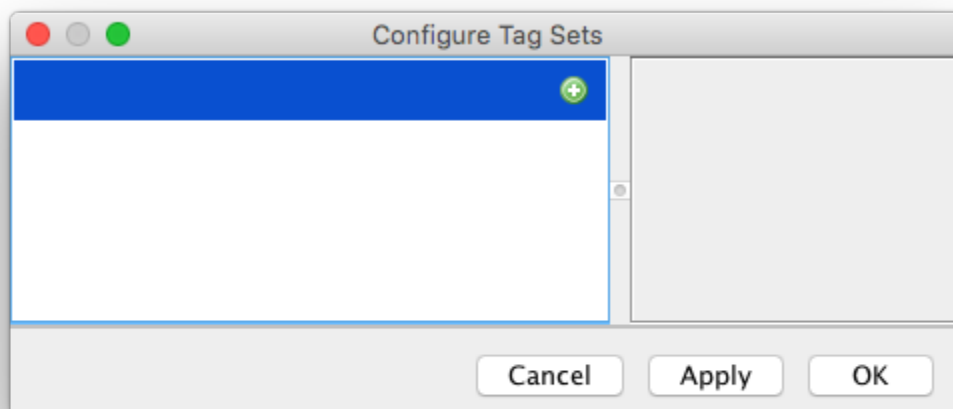
As we said above, every-time you need to categorize certain data items, or need to visualize categories, you should rely on tags. Let's suppose that you are investigating the trajectories of cells in a developing embryo from an early stage to a stage where the embryo is polarized. Some cells will migrate to the anterior part, some others to the posterior part, *etc.* You might want to tag cell tracks with the **Anterior** or **Posterior** tag, to investigate where do these cell come from in the early embryo. Or let's say that you are curating the results of the automated tracking on a large images. The tracking results might have some inaccuracies, and you want to correct them for important tracks. Because there is a lot of tracks, you share the workload with some colleagues. You work asynchronously with them, editing the Mastodon file one after another. Doing so, you can use tags in Mastodon to mark some tracks as reviewed by you. Your colleagues will use a tag for themselves, to ensure that no two scientists are reviewing the same track twice. All the cells that are not tagged in this categorisation are still waiting to be reviewed.

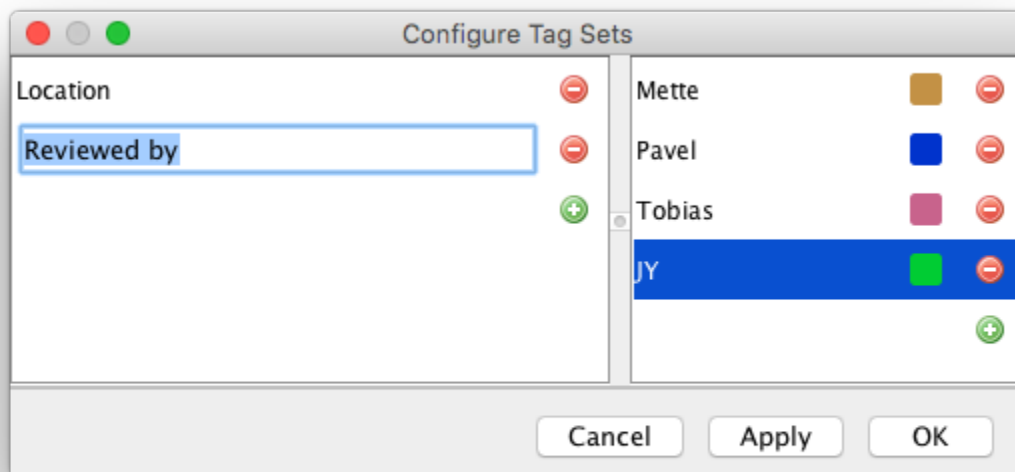
In Mastodon, a categorization corresponds to a **tag-set**. A tag-set defines a property that can have a reasonable number of discrete values, or **tags**. In the first of the two examples above, **Location** would be a tag-set to specify the location of cells. **Anterior** and **Posterior** would be two tags belonging to the **Location** tag-set. In the second example, **Reviewed by** would be a tag-set, and **Mette**, **Pavel**, **Tobias** and **Jean-Yves** would be 4 tags of this tag-set.

You can assign tags to spots and links. To assign a tag to a whole track, you have to assign this tag to all the spots and links of this track. One data item (a spot or a link) can have 1 or 0 tags per existing tag-set. But they can be categorized by as many tag-sets as there is. For instance, a spot can have the tag **Anterior** in the **Location** tag-set, and the tag **Pavel** in the **Reviewed by** tag-set. Or it can be not tagged in the **Reviewed by** tag-set. But it cannot have both the tag **Mette** and the tag **Tobias** because they belong to the same tag-set. Each tag-set works independently, and clearing a tag-set does not affect the others even for one data item. Now that we set things straight, let's see how to create tag-sets. We will base the demonstration in this chapter on the data we generated in the [first tutorial](#).

Creating tag-sets.

On the main Mastodon window, there is a **configure tags** button . Pressing it opens the tag-set dialog. Right now, it appears as an empty table made of two columns. This is where you enter tag-sets and tags.



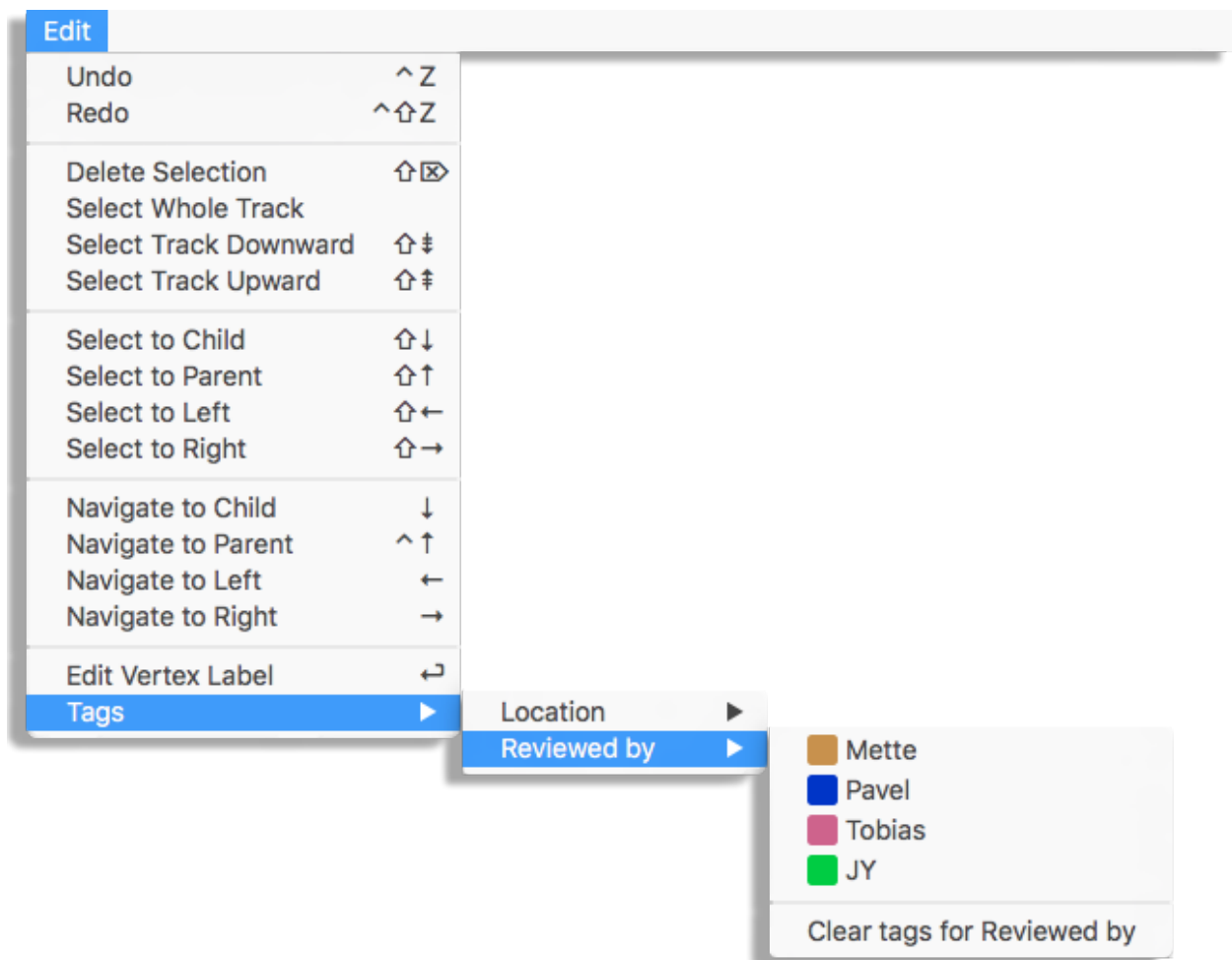


Press the green + button on the left column to create a new tag-set. A default name is shown for the tag-set, that you can edit. You can also directly press the Enter key to immediately start editing the new tag-set. Let's say we want to create the location tag we talked about before. Type `Location` in the text field. An empty line followed by a green + button should appear on the right column. This is where you will enter the tags of this tag-set. Click on this button, or press the `key` followed by the Enter key to create a new tag. For instance, the `Anterior` tag. Note that a tag is only made of a label (the text) and a color. The color will be used in Mastodon views. Create a second tag for the same tag-set called `Posterior`. Pick the color as you like. Now try to create another tag-set called `Reviewed by` and create some tags in this tag-set. The tag-set dialog is normally fully navigable with the cursor keys, so that you can enter tags quickly if you have a lot of them. You can create new tag-set or new tags with the Enter key, and delete them either with the `key` or by pressing the red - button.

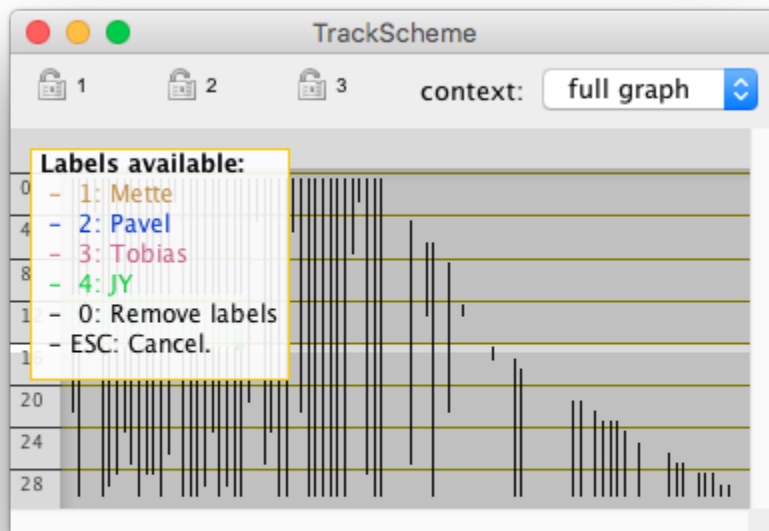
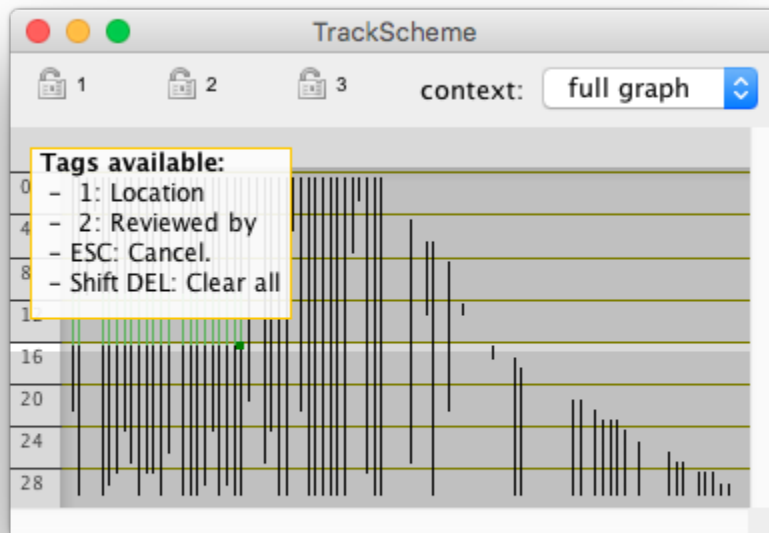
When you are finished, press the OK button.

Assigning tags to data items.

Tags are set via the selection tool, presented in the [second tutorial](#). Once you have some spots and links in the selection, you can assign a tag to it via the menu. The menu content will be updated with the tag-sets and tags you defined in the tag-set dialog, described above. This will work in any Mastodon views, BDV or TrackScheme.

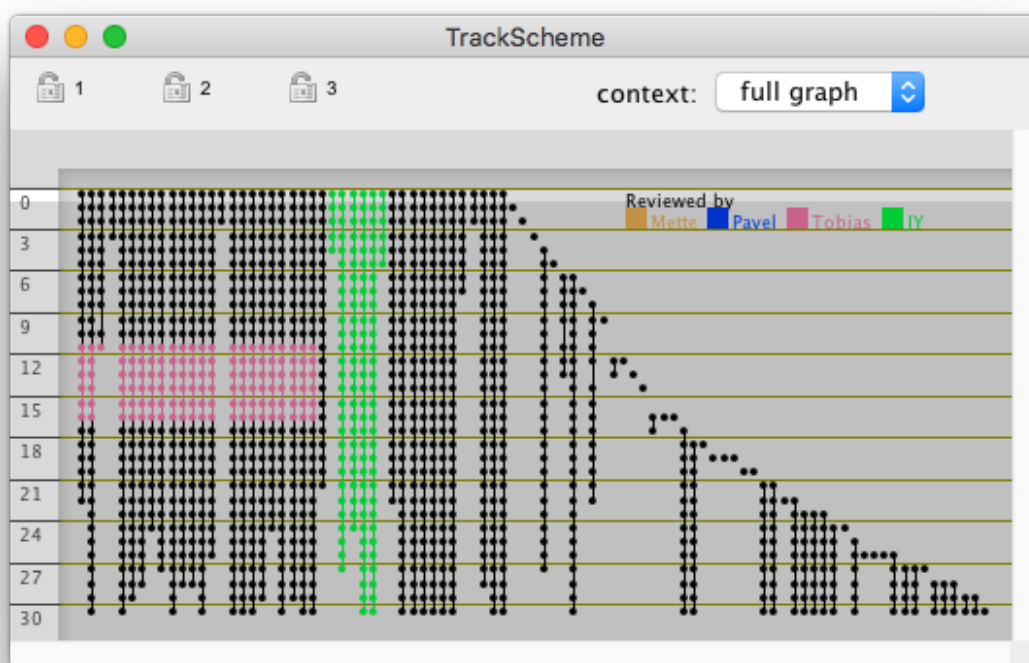
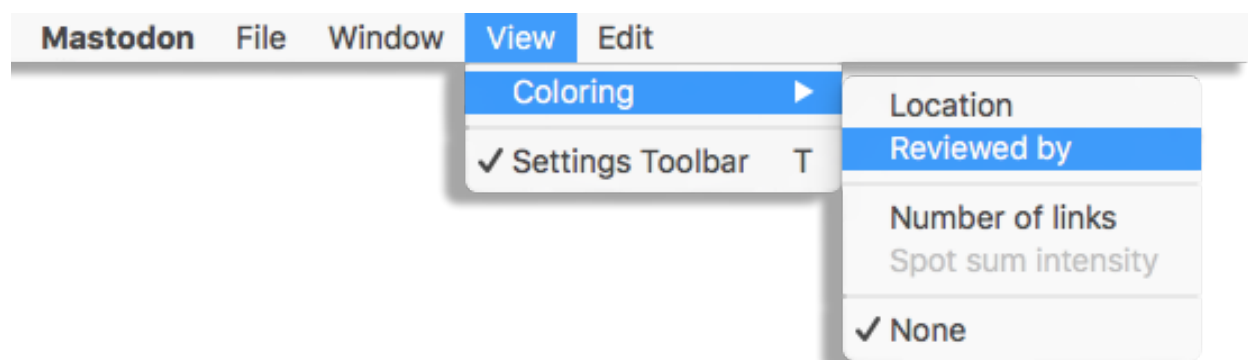


TrackScheme ships a second way to set tags quickly from the keyboard. After selecting the spots and links of interest, press the key. A floating menu should appear on the left part of the view panel. There is a bit of naming clash in the floating menu. Here, tag-sets are called *tags* and tags are called *labels*. Select the desired tag-set with the 1, 2, .. keys. The menu now shows the tags defined within this tag-set, that you can select the same way. Note that there is a way to remove all the tags over all the tag-sets on the selection by pressing on the first menu, or just the tags of the selected tag-set by pressing on the second menu.



Coloring views by tag-sets.

The tags we just defined and assigned can be used in with the views, to highlight the items that are tagged. In the *View > Coloring* menu of any view in Mastodon, you will find a sub-menu updated with the tag-sets you created among other choices. By default, newly created views are colored with the coloring mode, which simply colors all the spots and links the same way, taking colors from display settings. If you select a mode corresponding to a tag-set, tagged spots and links will appear painted with the color you chose for the tags of this tag-set. This is very handy to mark some locations in the image or highlight interesting tracks in the data. Later we will see that tags can be used to retrieve specific items for further processing. Finally, in there is an option to show a legend of the current coloring mode. You can toggle it on or off and set the location of this legend in the *View > Colorbar* menu.



1.4.2 Numerical features.

Numerical features are values that are calculated from the data. For instance the mean intensity within a spot, or the displacement along a link. They are very generic: the main restriction is that there must be a data item (a spot or a link) per feature value. But the feature itself can be scalar, non-scalar, real, integer, a string, a vector, *etc.* They are *labile*. Because they are defined for a data item, they will become invalid as soon as the data item changes. Think of what happens to the spot mean intensity if the spot is moved over the image for instance. Because we want to accommodate extensibility and large data, we have to use a special system that we describe below.

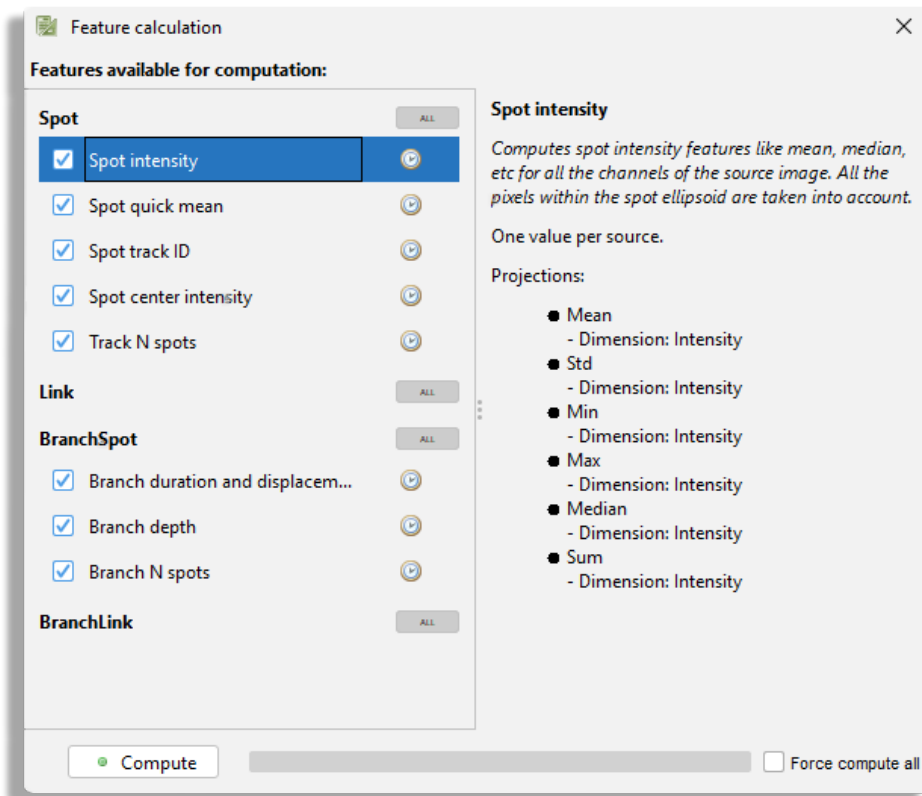
Feature computation.

Some numerical feature values are calculated by **feature computers**. Feature computers are actually specialized Mastodon plugins, made so that it is easy for a 3rd party (you) to implement their own features in Mastodon. We explain you to write your own feature computer in the second part of this documentation, dedicated to technical information.

Because some feature computation can take very long on large images, you have to trigger it manually. On Mastodon main window, you can find a button **compute features**. Pressing it will show the feature computation dialog. The feature computers are listed on the left panel.

You see in the panel **only** the feature computers that **takes a significant time to compute**. The other ones are computed on the fly and are always visible in the data tables (see below). For instance, the **Link displacement** and **Link velocity** are computed on the fly and do not appear in this panel.

Clicking on the computer name displays some information about the feature they compute in the right panel. Note that they are named 'features' on this panel, but they are in reality the feature computers. For instance if you click on the **Spot intensity**, you will see in the information panel that this computer generates a feature for the mean intensity, median intensity, *etc.* Note also that they can have dependencies. For instance, the **Track N spots** feature computer depends on the **Spot track ID** feature to be present at the time of computation.



The check-box on the left of each feature computer name triggers whether they will be part of the next feature computation. Press the Compute button to trigger computation of features.

Once the computation of all the features is complete, all the small clock icons that were shown right to the feature computer names now turned to a green dot. This is how we keep track of the validity of the feature values. Since the feature computation is triggered manually, and that a feature value might invalidated if the data changes (new spots added, removed, moved, changed the radius, added or removed some links), this icon serves as a signal for feature value de-synchronization. If the icon is shows as a clock , it means that the data changed since the last feature computation, and that the feature values are out of sync. If is shows a green dot, then the data did not change since last computation, and the feature values are sure to be valid. This is very important for proper interpretation of the data, and you will have to show the computation dialog often just to check the feature values validity. By the way, you can check now how the validity flag works. While keeping the feature computation dialog open, move a spot in a BDV view. You should see that all the green dot icons now turn to the clock icon. Also, if you now deselect some feature computers before launching a new computation, the validity flag will not turn to the green dot icon for those feature computers.

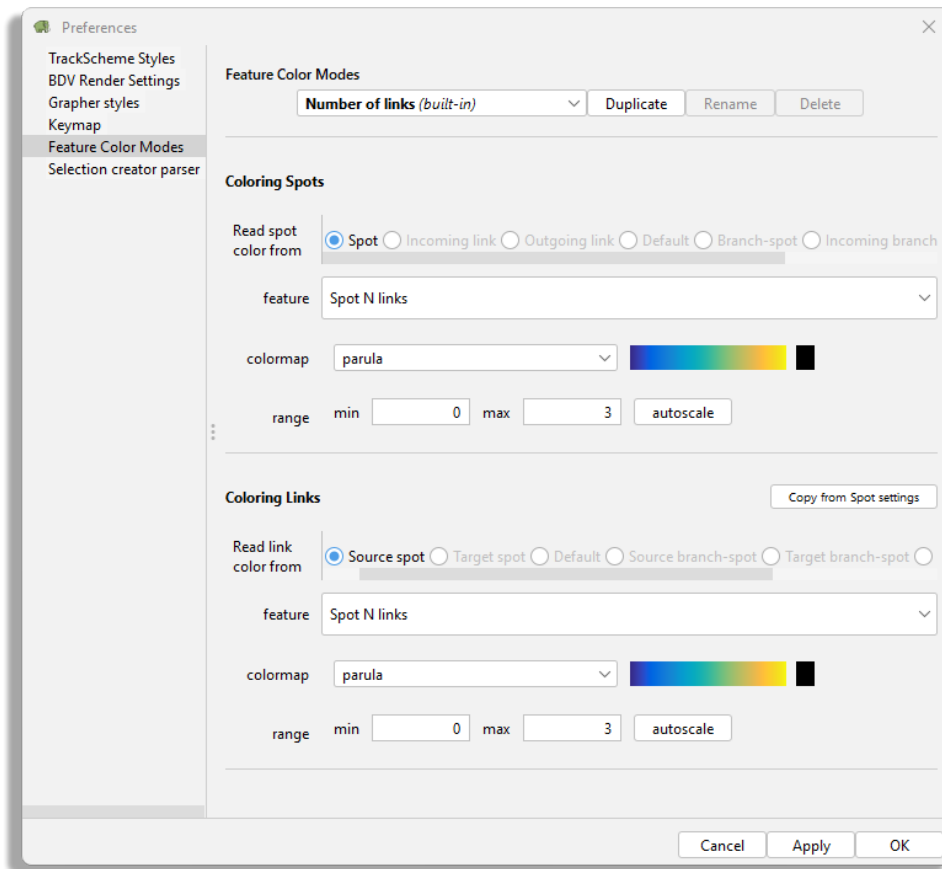
You probably have noticed, thanks to the progress bar, that the intensity-related features are the ones that take the most time to compute. Indeed, they require loading all of the data blocks on which there are spots. This could rapidly become cumbersome for large datasets, if you have to recompute all of the features every-time a spot is added or edited. Fortunately we implemented an update mechanism for feature computation. After the first computation, which possible takes very long, only the spots that have been added or modified since the last computation are considered for computation.

Now that we have feature values computed, we would like to inspect them and export them for further analysis. This is the role of the table view, but before getting to it, we will make a little detour to showing how to use features to generate coloring, accelerating updates of computation and saving them to disk.

Coloring views by numerical features.

We have seen above that tag-sets could be used to generate coloring of the data items shown in a view. Indeed, in the *View > Coloring* menu of each view, that tag-sets are listed and when selected, are used to assign a color to each data item. We can do something similar with feature values, except that feature based coloring requires more input from us.

Feature color modes need to be created first, and this is done in a dedicated user interface. Select the *File > Preferences* menu item in the main window or any view. This shows the preferences dialog. It is organized with a side-bar on the left that contains the various items that can be configured in Mastodon. Parenthetically, you can see that you can configure the display style of the and BDV views, and the keymaps. But we will see this later. In the sidebar select **Feature Color Modes**. The panel on the right now display the feature color mode configuration panel:



Its top line has a drop-down list that contains all the color modes already defined. Right now, there is only one, called **Number of links**. As the *built-in* suffix indicates, it is a built-in color modes, and it cannot be edited. To create a new one you must duplicate it and rename the new one. Do so by clicking on the **Duplicate** button, then on **Rename**. Let's create a color mode that color spots and links based on the mean intensity inside the spots, that we would call **Mean intensity**.

The rest of the configuration panel is made of two parts. The top part configures the spot coloring, or how we color spots. The bottom part configures the link coloring, or how we color links. In Mastodon the data is organized in a *mathematical graph*, in which the vertices are the spots, and the edges are the links that connect spots from one frame to another, so you will sometimes find in Mastodon and in this manual the vocables vertex and edge to design a spot and a link respectively. The vertex color mode specifies where do we take colors from. You can choose between:

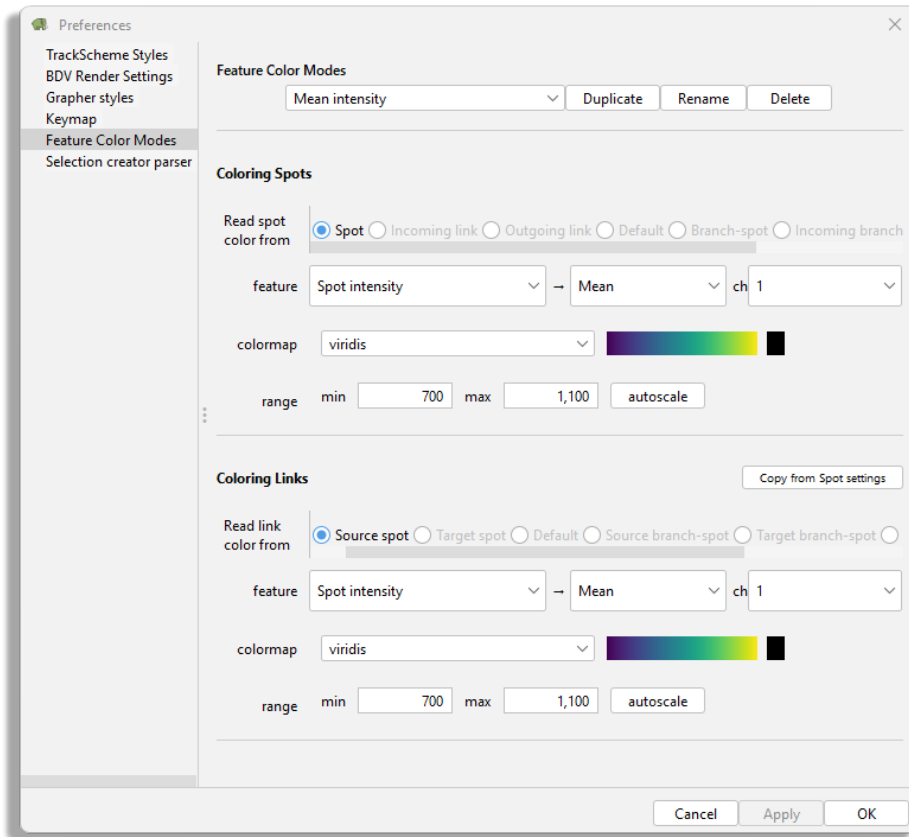
- **Spot**, which means a spot will take its color from a feature value it owns.

- **Incoming link**, which means a spot will take its color from a feature value owned by the single incoming link that targets this spot. This is the link backward in time. If there are no such links or more than one, then the default color is used.
- **Outgoing link** is the same thing, but for the link forward in time.
- **Default mode** does not rely on feature values but simply uses the default color in the view.

There are also modes that are related to the branch graph (Branch-spot, etc.) but we will discuss them in its dedicated tutorial.

Depending on the mode you chose, the content of the drop-down list below, called **Spot feature**, will change to reflect either the list of spot features or the link features. Select **Spot** as a color mode and **Spot intensity** as feature. Two new drop-down list appear on the right of the feature list. One contains the list of projections in the feature and the second one contains the list of channels in the dataset.

We need to explain a bit what are **feature projections**. We said above that a feature could be roughly anything numerical, and was not necessarily a scalar. It could be a vector, a tensor, a complex number, *etc.* However to be usable and useful in Mastodon, features are required to expose a sensible list of projections that compose them. Feature projections are scalar and real values that can decompose or project a feature on a real axis. How they are defined is up to the person that created the feature computer, but we can rely on the *hope* that they choose wisely. For instance, a feature that gives the velocity vector of a link will reasonably expose 3 projections, one for each of the X, Y and Z component of the vector. Or maybe the polar angle, azimuthal angle and norm of this vector. Or maybe the 6 projections since they can be calculated on the fly. A complex feature value will reasonably expose 2 projections, one for the real part, one of the imaginary part. *Etc.* The **Spot intensity** feature has six projections: mean, max, min, median, sum and standard deviation of the intensity inside a spot. Since all can be computed on any of the channel present in the dataset, their number is multiplied by the number of channels. The configuration panel changes according to the number of projections in a feature and its multiplicity. For features that are made of one real value with no multiplicity, the projection list is superfluous and not shown. In our case, we simply want the mean of the only channel in the dataset.



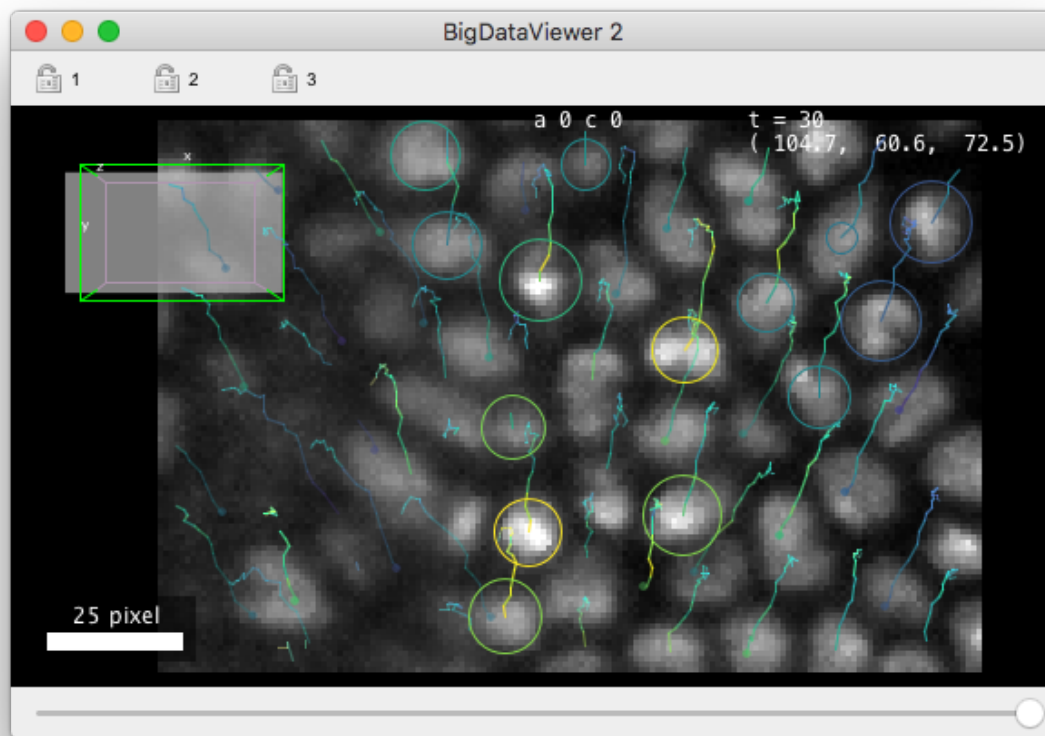
Coming back to the color mode configuration, next we need to pick a color-map. A color-map acts as the LUT for an image, and maps a color to a certain value. Mastodon ships about 20 of them, many taken from the [Matplotlib project](#). Finally, you have to specify a min value and a max value that will act as the brightness and contrast values for an image. Values below the min you defined will all be displayed with the first color of the color-map and values larger than the max with last. The black square you see next to the graded representation of the color-map is the color used for data items for which the feature value is not present or undefined (division by zero, *etc*). The `autoscale` button computes the min and max automatically from the feature currently selected with the values taken from the last feature computation.

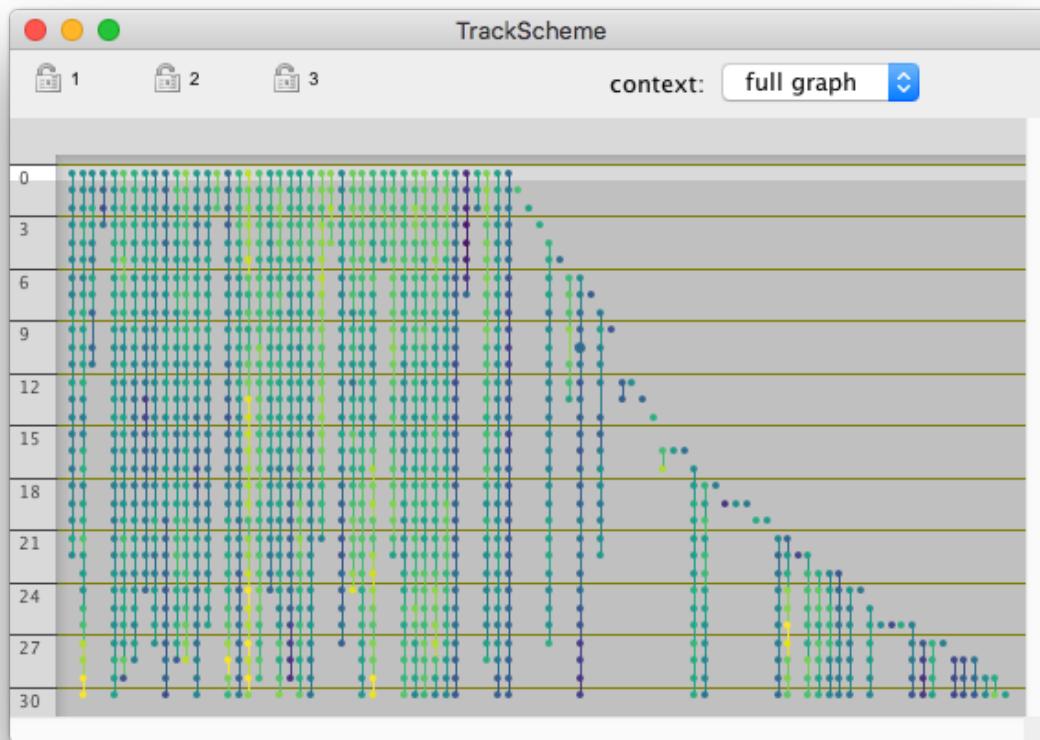
The link coloring works exactly the same, except for the link color mode.

- **Link** means that a link will be colored by a feature value it owns.
- **Source spot** will take a feature value from the source spot of this link, that is, the first in time.
- **Target spot** does the same but for the last spot in time of this link.
- **Default** mode does not rely on feature values but simply uses the default color in the view.

To build our example feature color mode, choose **Source spot** as color mode, and use for instance the **viridis** color-map along with 700 and 1100 for min and max values. You can also click on the **Copy from Spot settings** that will automatically configure the link color mode so that it matches what is set for spots.

Like for tag-sets, the menu is now updated with items corresponding to the color modes we created. If they are grayed-out, it means that the feature values they depend on is not yet computed. This kind of view immediately reveals important aspect of the data, even at a very high level. For instance with our custom color mode, we can quickly find cells that are the brightest, and visually inspect how the intensity in cells change over time.





1.4.3 The data table views.

The main table view.

The BDV and views are not suitable to display all the feature values we computed. The coloring we have been using with them is good only for visualization purpose. There is a nice view to properly inspect and exploit feature values in subsequent steps in your analysis: the table view. In practice, the table view is simply a tabular representation of the data items in Mastodon. Spots and links are displayed in a list where a single row corresponds to a data item, and columns to feature values and tags. You can create a new table view by using the menu . If you did compute all the features, it should look like the table below:

The left screenshot shows a table with columns: Label, ID, Detection, Spot N links, Spot center, and Spot intensity. The right screenshot shows a table with columns: Label, ID, Link cost, Link displacement, and Link target ID.

Spot	Label	ID	Detection	Spot N links	Spot center	Spot intensity
Link	0	0	602	1	963.886	
BranchSpot	1	1	612.571	1	974.561	
BranchLink	2	2	517.502	1	951.851	
	3	3	570.774	1	941.652	
	4	4	602.595	1	983.99	
	5	5	509.332	1	913.227	
	6	6	622.155	1	1,026.911	
	7	7	548.118	1	1,034.784	
	8	8	730.865	1	1,083.628	
	9	9	704.193	1	1,145.156	

Spot	Label	ID	Link cost	Link displacement	Link target ID
Link	1227 → 1264	0	15.413	3.926	1,227
BranchSpot	1226 → 1265	1	11.837	3.44	1,226
BranchLink	1225 → 1262	2	12.325	3.511	1,225
	1224 → 1257	3	9.046	3.008	1,224
	1223 → 1260	4	12.008	3.465	1,223
	1222 → 1248	5	11.136	3.337	1,222
	1221 → 1258	6	11.159	3.341	1,221
	1220 → 1255	7	12.64	3.555	1,220
	1219 → 1254	8	15.729	3.966	1,219
	1216 → 1253	9	2.474	1.573	1,216

The view is made of several tables, the first one for spots and the second one for links. We will discuss the others, made for the branch-graph, later. Right now it is pretty empty. The spot and link tables only show the label and ID of the spots and a few feature values that are built-in. But they do not show the spot intensity features we configured above. Navigating in this table is done classically: the arrow keys \uparrow and \downarrow jump from one row to the next, and \leftarrow and \rightarrow from one column to the next. and $\text{Control} + \text{Page Up}$ and $\text{Control} + \text{Page Down}$ moves across tables..

After computing some features and defining some tag-sets, the table shows new columns. Note that the column headers represent the feature with their projection and physical units on several rows. For instance, the Spot intensity feature name is displayed on the first row of the column header. The header is split in two columns on the second row, one for each projection included in the feature. And in the third and last row, the units of each projection is displayed in brackets (Counts in this case). The header of the tag-set columns are similar. The first row shows the name of the tag-set, and the second row shows each of the tag the set contains, with the tag chosen color as background.

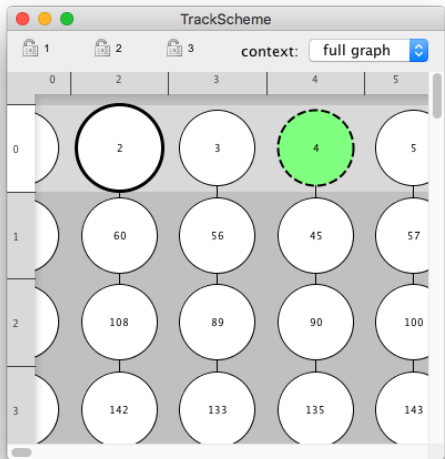
The table displays various features for spots and links, including position (X, Y, Z), radius, track ID, and track N. It also includes a 'Reviewed by' section with checkboxes for different users.

Spot	Label	ID	X (pixel)	Y (pixel)	Z (pixel)	Spot quick ... (Counts)	Spot radius (pixel)	Spot track ID	Track N	Track N spo...	Reviewed by
Link	0	0	51.492	57.266	46.937	794.693	7.5	0	23		
BranchSpot	3	3	67.559	57.686	45.465	814.747	7.5	88	31		
BranchLink	1	1	51.503	87.233	46.383	794.362	7.5	87	11		
	5	5	85.04	56.398	50.529	770.996	7.5	86	28		
	3	3	63.028	39.367	50.465	823.59	7.5	85	25		
	2	2	25.27	59.91	50.968	794.592	7.5	3	31		
	0	0	45.072	36.123	48.885	847.454	7.5	83	29		
	9	9	77.028	72.846	49.934	905.511	7.5	82	29		
	3	3	79.216	35.028	52.464	876.077	7.5	81	30		
	2	2	61.569	76.153	51.259	923.728	7.5	80	31		

The table view can be used to edit in part the data. For instance you can edit the spot label directly in the table. Just navigate to the row of spot you want to change the name of and the Label column, then press F2. The label field becomes editable. When you are done editing, press Enter. The tags are displayed as check-boxes in the table, that you can set directly by clicking on them. Or you can navigate the desired row and column and set them with the space key.

Spot	Link	Label	ID	Detection ...	Spot N links	Spot frame	Spot position			Spot radius	Location						Reviewed by
							X (pixel)	Y (pixel)	Z (pixel)		Anterior	Posterior	Mette	Pavel	Tobias	JY	
0			0	602	1	0	51.492	57.266	46.937	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
1	BranchSpot		1	612.571	1	0	67.559	57.686	45.465	7.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
	BranchLink	EDITED	2	517.502	1	0	51.503	87.233	46.383	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
3			3	570.774	1	0	85.04	56.398	50.529	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
4			4	602.595	1	0	63.028	39.367	50.465	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
5			5	509.332	1	0	25.27	59.91	50.968	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
6			6	622.155	1	0	45.072	36.123	48.885	7.5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
7			7	548.118	1	0	77.028	72.846	49.934	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
8			8	730.865	1	0	79.216	35.028	52.464	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
9			9	704.193	1	0	61.569	76.153	51.259	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
10			10	701.513	1	0	103.268	82.986	51.293	7.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

The highlight and selection are also shared with table views. When the table view is not active, selected items are shown with a gray background. The highlight spot or link is shown in the table with a thick black border.



Spot	Link	Label	ID	Detection ...	Spot N links	Spot frame
0			0	602	1	0
1	BranchSpot		1	612.571	1	0
	BranchLink		2	517.502	1	0
3			3	570.774	1	0
		4	4	602.595	1	0
		EDITED	5	509.332	1	0
6			6	622.155	1	0
7			7	548.118	1	0
8			8	730.865	1	0
9			9	704.193	1	0
10			10	701.513	1	0

To add rows to the selection, the default key-bindings are again standard. Press **Shift Left-click** to add a range of rows to the selection from a table view, or use **Shift ↑** or **Shift ↓** or **Shift** and **Shift .** By pressing **Control Left-click** or **Command Left-click** you can toggle single rows in and out of the selection. Of course, all of the commands related to the selection we have seen before also apply to the table views. The shortcuts to navigate in the table views are summarized in the [table of table shortcuts](#).

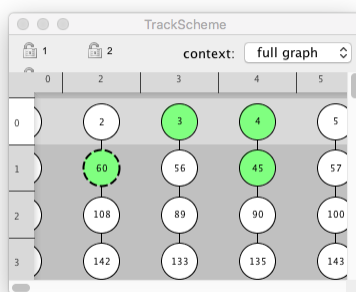
Another feature of data tables is that they can be made slave of a spatial context, like for TrackScheme. When another BDV view is active, you can select its name in the drop-down list on the top-right part of the table. Then the table only shows the data items that are currently displayed in the master BDV view. The notion of spatial context is explained in the [previous tutorial](#).

Sorting rows.

The table can be sorted by clicking on the header of the column you want to use for sorting. It works for labels, IDs, feature values and tags

The selection table.

There exists a variation of the table view, but that display only what is currently in the selection. To display such a table, go to **Window > New selection table** in the menu. The selection table that appears only shows what is in the selection, and is constantly updated to reflect changes in the selection. You cannot use it to edit the selection like in the main table. However the row you pick in this table will set the focus and highlight in other views. Everything else applies to the selection table.



File

Window

View

Edit

Settings

Plugins

Selection table

1

2

3

context: full graph

Search...

start with

Spot	Label ^	ID	Detection ...	Spot N links	Spot frame	X (pixel)	Spot position Y (pixel)
Link	3	3	570.774	1	0	85.04	56.398
BranchSpot	4	4	602.595	1	0	63.028	39.367
BranchLink	45	45	566.273	2	1	55.48	22.907
	60	60	666.943	2	1	79.737	35.85

Feature-based coloring in table views.

Of course, feature based coloring works with the table views. And it can give a pleasant display when combined with sorting rows by a feature column.

Feature and tag table

context: full graph

Label	ID ▲	Spot N lin...	Spot gaussian-filter...		Spot trac...	Locatio	
			Mean ch0 (Counts)	Std ch0 (Counts)		Anterior	P
0	0	1	897.326	382.799	0	<input type="checkbox"/>	
EDITED	1	1	904.884	356.5	90	<input type="checkbox"/>	
2	2	1	887.126	387.602	89	<input type="checkbox"/>	
3	3	1	889.358	360.016	88	<input type="checkbox"/>	
4	4	1	922.528	401.369	87	<input type="checkbox"/>	
5	5	1	1,000.187	460.709	86	<input type="checkbox"/>	
6	6	1	954.934	385.171	85	<input type="checkbox"/>	
7	7	1	913.906	357.682	84	<input type="checkbox"/>	
8	8	1	872.884	382.024	83	<input type="checkbox"/>	
9	9	1	853.587	301.853	82	<input type="checkbox"/>	
10	10	1	962.587	353.652	81	<input type="checkbox"/>	
11	11	1	1,029.641	504.933	80	<input type="checkbox"/>	
12	12	1	915.922	338.421	79	<input type="checkbox"/>	
13	13	1	914.296	319.505	78	<input type="checkbox"/>	
14	14	1	1,020.443	519.421	77	<input type="checkbox"/>	
15	15	1	838.055	308.914	9	<input type="checkbox"/>	
16	16	1	955.984	366.516	75	<input type="checkbox"/>	
17	17	1	1,061.557	493.844	74	<input type="checkbox"/>	

Exporting table data.

The data currently displayed in a table view can be exported to CSV. When a table window is active, select the menu item **File > Export to CSV**. You will have to specify a saving location and a name. Only the data displayed in the currently visible table are saved, and ordered as in the view. This means that if you call the command from a selection table, only the current selection will be saved.

1.5 Semi-automated tracking.

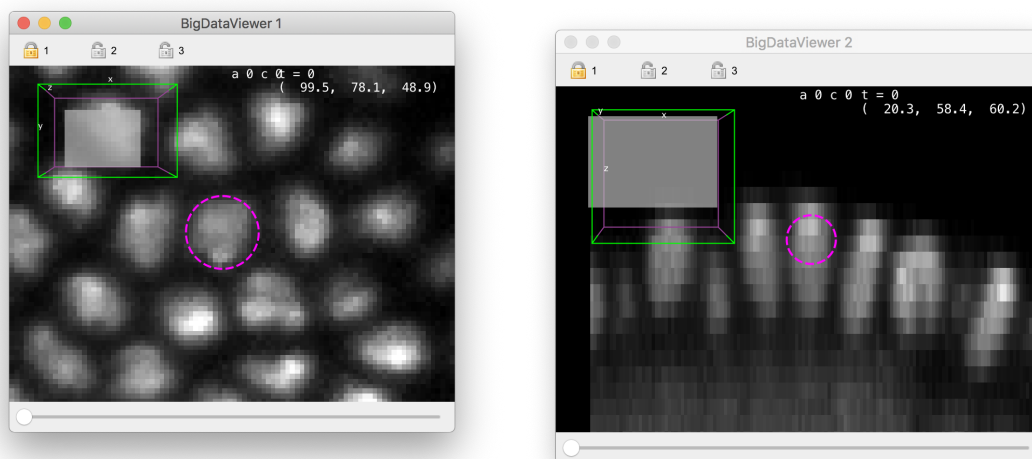
Let us suppose we are dealing with a difficult image in which we must track only a subset of cells. The cells are difficult to track automatically, for instance because there are many spurious structures labelled. Or because there are other cells of different sizes in the tissue we are studying. Or because the image quality varies in time and space. The data is such that that fully automated algorithms introduced in the first tutorial won't give us fully accurate results. We can use the manual editing tools introduced in the second tutorial and curate the results of automated tracking, manually removing spurious detections and links, and fixing incorrect ones. Another approach would be to start from a blank annotation and track manually only the cells we are interested in. Both approaches might be long and tedious. We introduce in this tutorial tools for semi-automated tracking, that should alleviate the work of the second approach.

Semi-automated tracking is simply a way of following a specific cell that you picked-up manually. The tracker will follow the cell over time, and create spots and links for a certain amount of time-points. It searches for the best spot candidate in the next time-point around the location of the spot. It then creates a new spot there and links it to the previous one. This procedure is repeated this for a certain number of time-points you can set, creating or augmenting a track starting from the spot you selected.

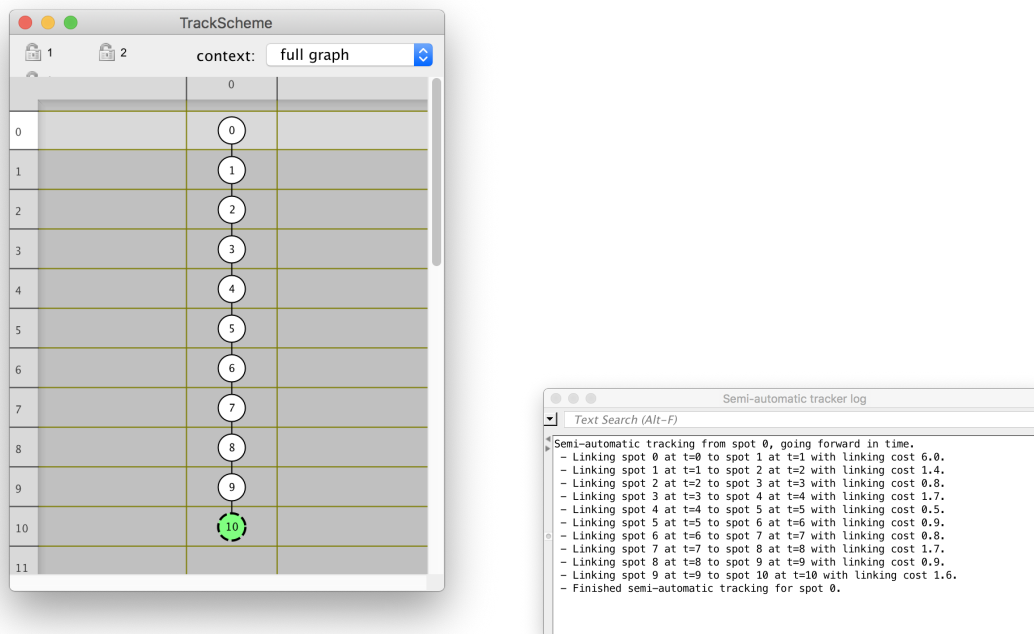
The semi-auto tracker can be configured to work backward in time (backtracking), to have a certain search radius, or a certain sensitivity to spot quality. The way it interacts with existing annotation can be configured too. You can make it stop when it meets an existing spot, linking to it or not. You can make it connect to small tracks and resume tracking when it meets the track end. You can force it to only create links on already existing spots. These configuration options give rise to several use-cases we will also survey in this chapter. But the important message is that semi-automated tracking is a convenient means for dealing with difficult cases, when a fully automated approach fails, and when the data to track is large that doing it manually is inconvenient. Or when you only care for a subset of cells in a dense tissue.

1.5.1 Simple semi-automated tracking.

We will introduce semi-automated tracking on a movie with empty annotation. Open the dataset we have been using so far, and clear all annotations (for instance select all **Control A** then delete selection **Shift**). Then pick a cell in the top layer, and create a spot, centered on its brightest part, as is done below. Adjust its radius and select it. Open a TrackScheme window to visualize the tracking progress.



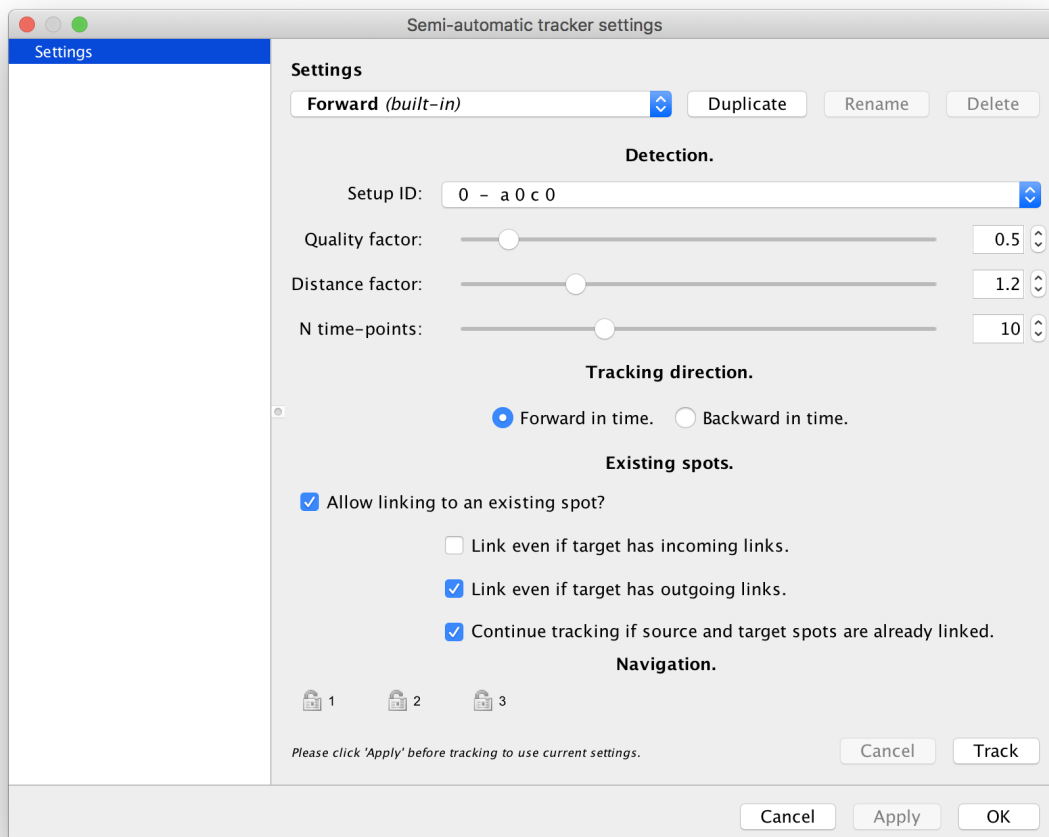
To start semi-automated tracking, press **Control T**. A log window should open, and tracking should proceed. If the log does not complain about candidates being too far, you should end up with something resembling the images below.



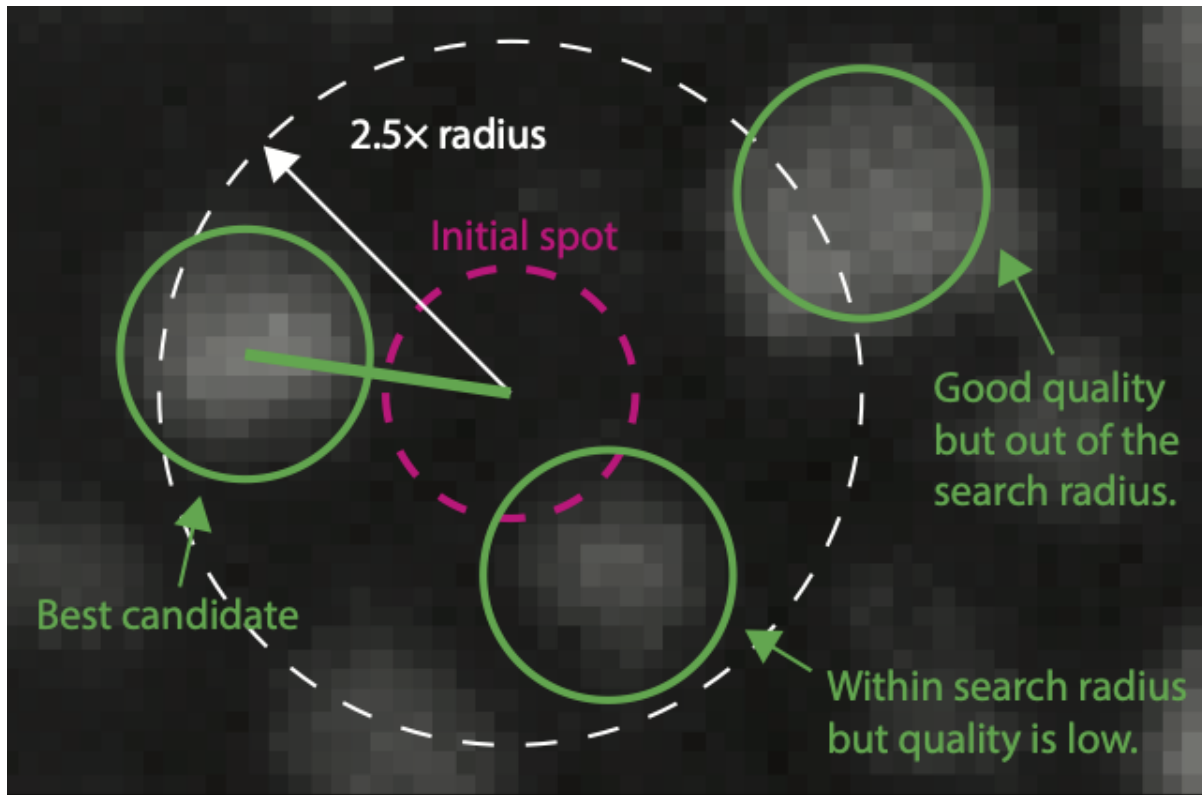
The tracking stopped after 10 frames, and the last spot added is now in the selection. Each spot is centered on the cell we started with, and it has the same radius that of the first spot we created. You can resume semi-automated tracking from the last spot created by just pressing **Control T** again. If you do it one more time you should reach the end of the movie, with a new track following a single cell over the full movie duration. To get it, you just had to press **Control T** 3 times. If you have more than one cell in the selection, they will be tracked one after another.

1.5.2 Configuring the semi-automated tracker.

The tracking does not always succeed. Depending on where you position the initial spot, it might fail, for instance stating in the log that it found a suitable spot, but outside the tolerance radius. (In this example movie, this happens often because the cells are very elongated along the Z direction, and the tracker tends to find candidates near the top, brightest part of the cell.) The parameters that control for instance the tracker search radius can be set in the semi-automated tracking configuration dialog, in the **Plugins > Tracking > Configure semi-automatic tracker**. The following window should appear.



This dialog is very similar to the one used to configure feature color modes, that we have seen the tutorial just before. It also works the same way: the top elements are made to manage several tracking configuration, that will be stored on disk and retrieved in your next Mastodon session. There are two default tracking configuration: the **Forward** one is the default that we just used. The **Backtracking** configuration tracks backward in time. The other parameters controls the tracker behavior for the configuration currently selected in the top drop-down list. To explain what they do we need first to describe how the semi-automated tracker works:



The semi-automated tracker works by processing only a small neighborhood around the initial spot, (called the *source spot* later). This neighborhood is centered on the spot center (in magenta above), but taken in the next time-point (or previous one if you choose to go backward in time). It applies the DoG detector (described in the first tutorial) on this neighborhood, which yields several detections (green circles). The detections that are found outside of a search radius (white, dashed circle) are not considered. Detections inside the search radius but with a low quality are discarded as well. The tracker therefore select the detection with a sufficiently large quality inside the search radius. If there is more than one suitable detection, it selects the one with the highest quality. A new spot is created at this location, with the same radius that of the source spot. If the source spot is not a sphere but an ellipsoid, the smallest radius of the ellipsoid is taken. The newly created spot is then linked to the source spot. This is then repeated for the next time-point (or previous one), using the new spot as initial spot in the same process. If no suitable detections are found within the search radius, the tracker stops, and the reason is printed in the tracker log window.

The configuration panel controls the parameters of this process.

- **Setup ID** specifies what channel (or setup in case you have a multi-view dataset) that will be used for the detection.
- **Quality factor** specifies the threshold on quality below which we reject detections. This threshold is expressed in fraction of the source spot quality. For instance if the source spot quality is 60 and the **Quality factor** is 0.5, detections with a quality lower than 30 will be rejected. If the source spot has no quality value (it was added manually), this parameter is ignored and all quality values are accepted.
- **Distance factor** specifies the search radius, in units of the source spot radius. For instance, for a value of 2.5, only detections that are within 2.5 times the radius of the source spot will be considered. Again, if the source spot is not a sphere but an ellipsoid, the smallest radius of the ellipsoid is taken.
- **N time-points** specifies the number of time-points after which to stop.
- **Tracking direction** lets you specify whether you want tracking to happen forward or backward in time.

If you see that the tracker often stops with a message stating that it could not find a suitable candidate within search radius, try to either decrease the value of the **Quality factor** or increase the value of the **Distance factor**.

1.5.3 Tracker behavior with existing annotations.

The next parameters below the **Existing spots** category configure how the tracker deals with existing annotations. They change the behavior described in the previous section. Indeed, before running the DoG detector on the neighborhood, the tracker first searches for an existing spot within the search radius. If it finds one, it does not run the DoG detection, but links to the existing spot (called *target* spot later) or not, depending on the following parameters.

If the **Allow linking to an existing spot** checkbox is deselected, the tracker stops. If it is selected, the tracker will link to the target spot, provided it has no links already. However, the next 3 parameters allow to add exceptions:

- Link even if target has incoming links
- Link even if target has outgoing links
- Continue tracking if source and target spots are already linked

Their selection have important consequences when you are tracking along existing spots. They are best exemplified by several different use-cases.

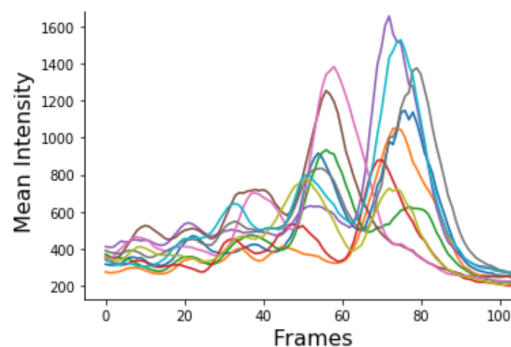
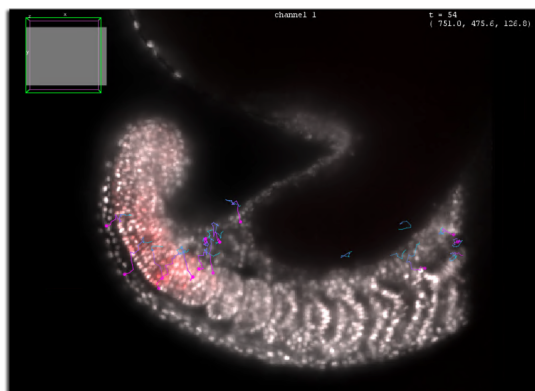
1.5.4 Main use-cases for semi-automated tracking.

Tracking a subset of cells.

This is what we have been doing in the first section of this chapter. This does not require any special configuration and the default tracking configuration called **Forward** will do. Here is an example of when this use-case can be useful.

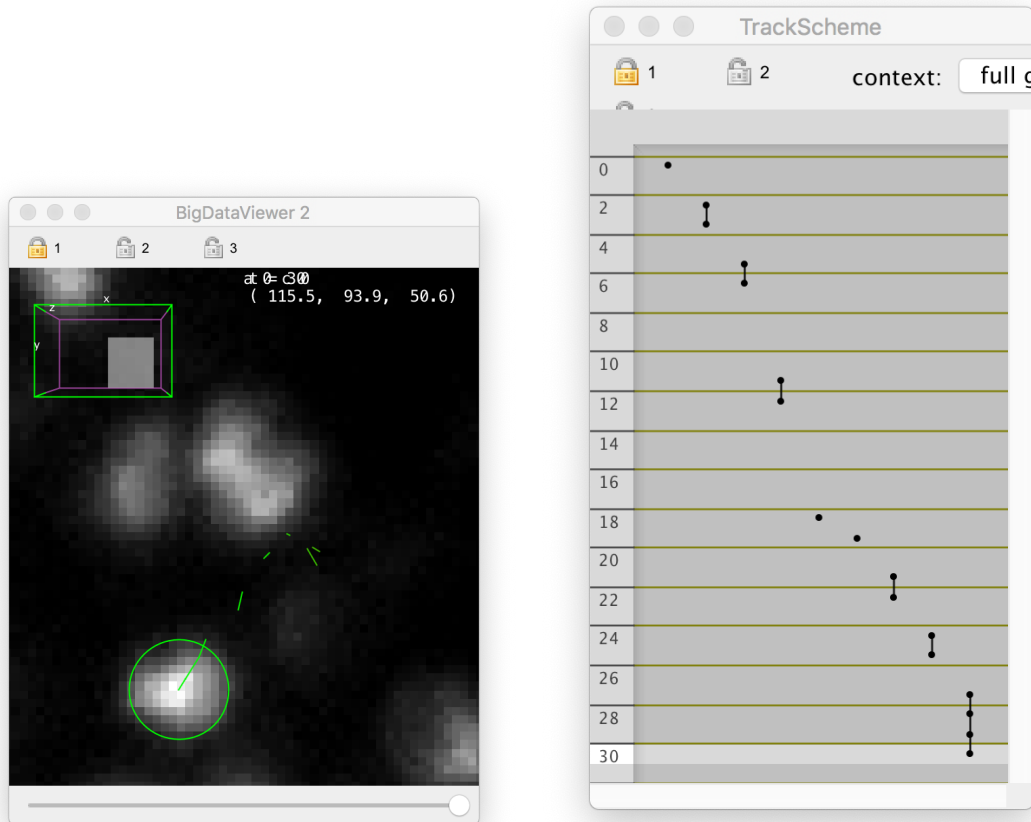
Arianne Bercowski Rama and Laurel Ann Rohde (Segmentation Timing and Dynamics Laboratory, EPFL) are studying the somatogenesis dynamics in the zebrafish embryo. They acquire long-term time-lapse movies of the development of an embryo, with cellular resolution. For a project they wanted to follow a few cells of interest (a few dozens per movie) and investigate the expression of a gene reporter as the cells moved along the zebrafish embryo. The nuclei are all stained with a nuclear marker, so this fluorescence channel could be used for automated detection, but there are thousands of cells. Instead, they relied on semi-automated tracking, selecting a the cells of interest in the first frames of the movie, and tracking them to the end thanks to the semi-automated tracker.

Once the tracking was done, they used feature analysis to extract fluorescence intensity over time for each cell in the gene reporter channel.

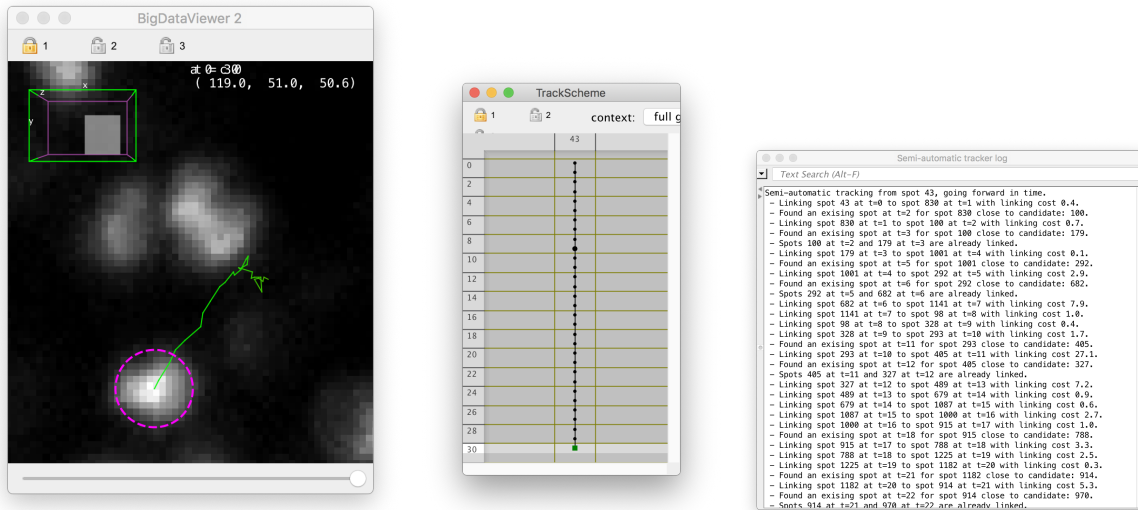


Stitching small track segments.

The semi-automated tracker can be used with the same configuration to stitch small track segments, which follow the same cell but are separated and spread in time. This might be caused linking with missing detections, and the situation could resemble what is pictured below.



We can use the semi-automated tracker to connect them, and add the missing detections. We want the tracker to create spots for the cell in time-points when they are missing, and to connect to existing ones it will find by following the cell. To allow this, we must enable first the 'Link even if target has outgoing links' setting, so that the tracker does not stop when it meets a spot that is at the beginning of a track segment. Second, we must enable the 'Continue tracking if source and target spots are already linked' setting. If we do not, the tracker will stop after connecting the first track segment it meets. Finally, we have to increase the value of the 'N time-points' parameter, so that the tracker iterate through the full movie. After doing this and tracking for the first cell, we get a single track:



Backtracking, branching on cell divisions.

Backtracking is particularly useful when creating lineages of cells that divide often in the movie. Another application is in mapping differentiated cells at a late time-point, to the position of their progenitors at the beginning of the movie. For this we typically start from an empty annotation, select a cell of interest in a late time-point, and backtrack it to its position in the beginning of the movie. The built-in configuration called **Backtracking** is made for this. If the cell we backtrack divided several times during the movie, it might be creating tracks for sibling cells. We want these tracks to branch properly in case we meet the mother cell when it divides.

This is why we need to enable the ‘Link even if target has outgoing links’ setting, but no other. Suppose we already have one track for a cell that divides, following the mother cell then one of the daughter cells. When we will backtrack from the other daughter cell, late in the movie, we will meet the division point of the mother cell. There is an existing spot there, just before the cell divides. It has already an outgoing link (to the first daughter we already tracked), and we want to connect to it. Hence we enable the ‘Link even if target has outgoing links’ setting. Because we want to stop tracking there (the rest of the track is good already), we do not enable any of the two other settings.

Sparse linking over dense spots.

1.6 The selection creator.

The automated detection process we use often generates a lot of spurious detections. In **TrackMate** we complemented it by adding *feature filters* just after the detection step. In **TrackMate** UI it takes the shape of filter windows, where the user can specify a feature and a threshold above or below which spots are rejected. The filters can be stacked to generate a more stringent filtering. This approach is like fishing with a small-hole net, then throwing back unwanted fishes to sea.

In **Mastodon** we take a somewhat different approach. We don’t have a filter interface, but instead work with a dedicated tool called the **selection creator** that we describe it here. It works differently from the interactive selection we have in **TrackMate**. Instead of manually clicking on spots or links, or drawing a selection rectangle in **TrackScheme**, you will enter an *expression* that will be parsed to generate a selection. The expression uses a basic language that allows translating criteria like *select all spots that have an intensity larger than X and their links*.

We thought this approach would be more convenient and powerful, and the selection is especially important in **Mastodon**. Indeed:

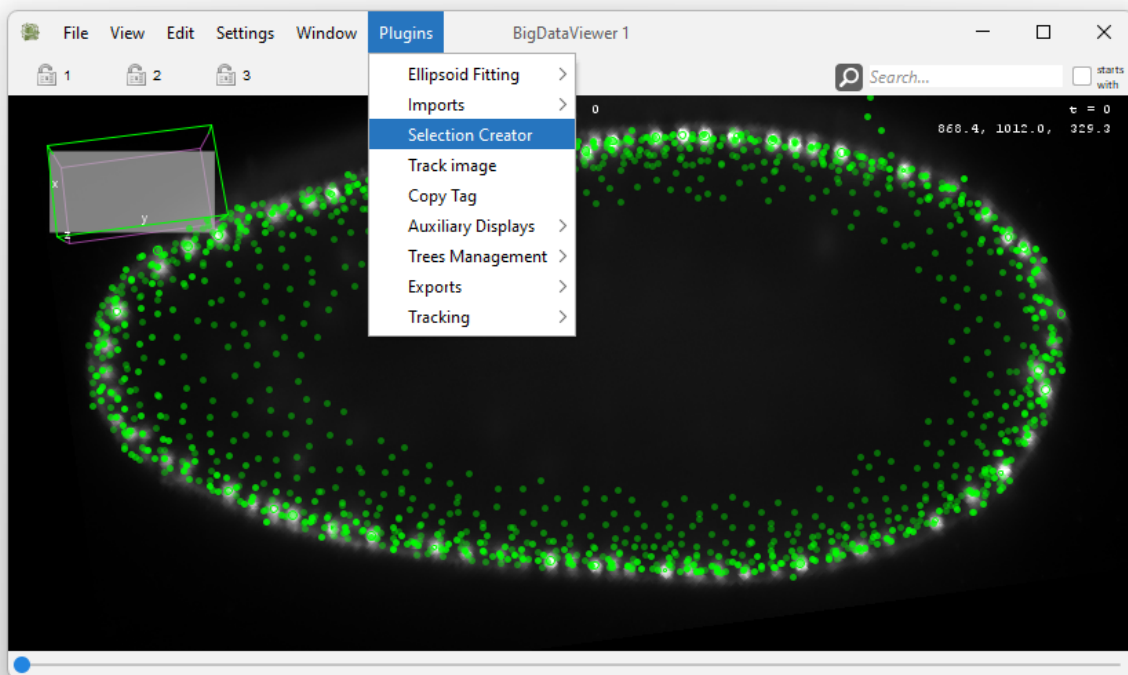
- You can create a selection to tag several data items that follow a certain criterion.
- The selection can be used as an input for the linking algorithms (see the [getting started tutorial](#)).
- The BDV views can be used to only show the selection you just created.
- The selection content can be inspected in the selection table and inspected with the grapher views.

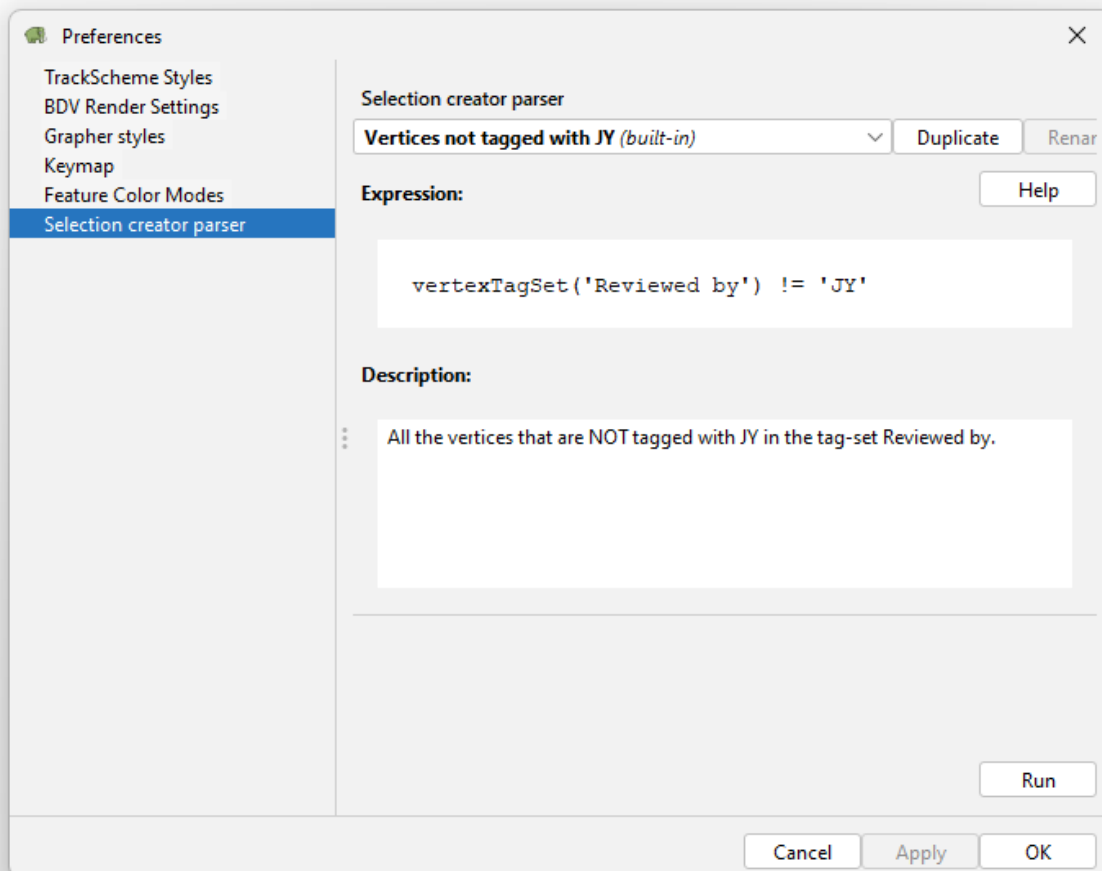
1.6.1 The selection creator window.

We will be using the TRIF dataset from the cell tracking challenge for this tutorial. You can download it from Zenodo:

Download the `.mastodon` and the `.xml` files, and put them in the same folder. The image data is stored on the Pasteur BDV-server and Mastodon will stream the data from there.

The selection creator tool can be called from the *Plugins > Selection Creator* menu, but is also a tab in the Preferences window:





The window for the tool is very simple. It contains a text field for the expression, and another one that contains an editable description. The Run button evaluates the current expression, which results in an error or in the selection being modified. Several built-in examples are included, and there is a Help button that opens a document recapitulating the expression syntax.

Expression are stored and saved with the same menu that for the settings (TrackScheme styles, BDV render settings, etc). When you press the apply button, they are saved to a file in the `.mastodon` folder in your home folder, called `selectioncreatorexpressions.yaml`. For instance:

```
>> pwd
/Users/tinevez/.mastodon
>> cat selectioncreatorexpressions.yaml
```

```
Vertices with a tag
--- !selectioncreatorsettings {name: Spots with 1 link in frame 25, expression:
  vertexFeature('Spot
    N links') == 1 & vertexFeature('Spot frame') == 25, description: Spots with 1
    link in frame 25.}
--- !selectioncreatorsettings {name: Vertices with 1 link minus those in frame 25,
  expression: (vertexFeature('Spot N links') == 1) - (vertexFeature('Spot frame'))
```

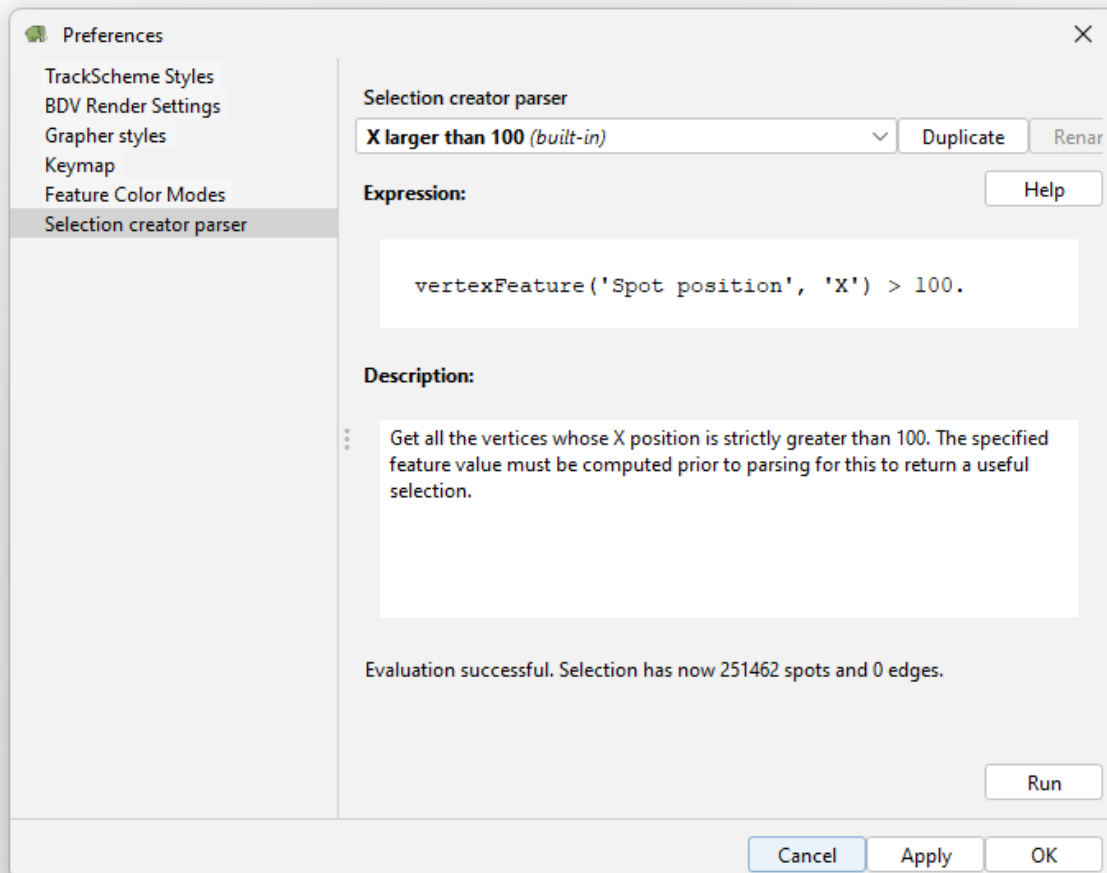
(continues on next page)

(continued from previous page)

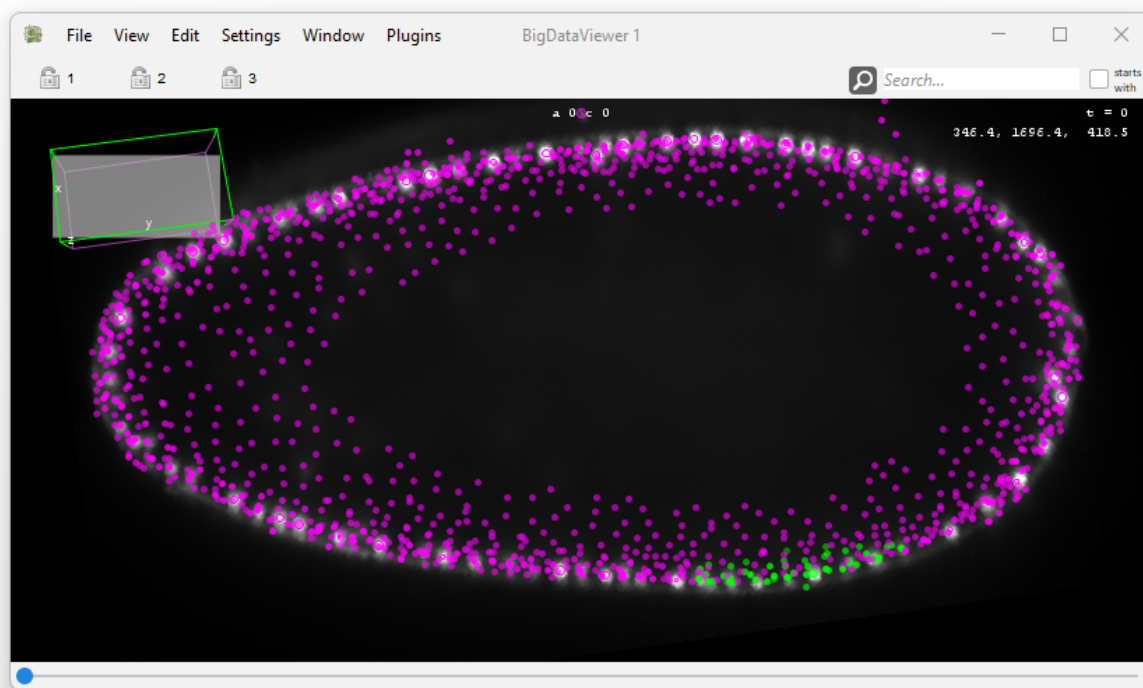
```
== 25), description: Vertices with 1 link minus those in frame 25.}
```

1.6.2 Selection based on spot feature values.

Let's see how we can select spots based on numerical features. We will be mainly playing with the expression field. First, in the top drop-down list, select the builtin example called `X larger than 100`, then click the Run button. This should be displayed:



And in a BDV window, you should see that most of the spots have been selected, saved for a few.



Let's have a look at the expression we used. In the selection creator, you enter expressions that are evaluated for all the data items. The expression must evaluate to a boolean answer: a `true` or `false` answer for each of the data item currently in mastodon. In our example we have:

```
vertexFeature('Spot position', 'X') > 100.
```

The keyword `vertexFeature` is a function that will fetch numerical values from a spot feature, for all spots. The arguments of this function are first, the feature name (`Spot position`) and second, the projection name `X`.

Indeed, many features aggregates several *projections*. A projection is a component of a feature that is always a real scalar. For instance, the `Spot position` feature is composed of 3 projects: `X`, `Y` and `Z`. See the part on numerical feature computation in the [tutorial on the table views](#) for more details. Anyway, we need to remember that the `vertexFeature` function will return a scalar value for all spots, so it needs a feature *projection* to be specified.

In the last part of the expression, we have a boolean comparison resulting in a boolean result: `> 100.` (the dot only stresses that 100 is floating point number). So with this expression, all the spots that have a `X` position larger than 100 will evaluate to `true` with this expression, and therefore will be added to the selection.

We use the **strictly great than** operator `>`, but any comparison operator would have worked:

- `>` strictly greater than
- `<` strictly smaller than
- `>=` greater than
- `<=` smaller than
- `==` equal to
- `!=` different from

This is the basis for the whole selection creator framework. You can enter any kind of mathematical expression based on the keyword supported, and it will be turned in a selection if it evaluates to a boolean results.

1.6.3 Let's make errors.

Because the selection creator accepts expressions that can be anything, it is likely that we will have error messages at some point. Let's spend some time generating errors on purpose.

First let's try removing the comparison:

```
vertexFeature('Spot position', 'X')
```

If we click the Run button we get the following error message:

```
Evaluation failed. Got unexpected result:  
VertexFeature( Spot position → X, 256609 )
```

This is expected. The selection creator expects expressions to evaluate to booleans. When we removed the comparison operator (`< 100.`) we returned the X position for the 256609 spots in the model, and the selection creator does not know how to turn this into a selection. So we need to remember that we always need to use expressions that evaluates to booleans.

Let's test an expression without specifying the projection name in the `vertexFeature` function:

```
vertexFeature('Spot position') > 100.
```

Now we get:

```
Evaluation failed. Incorrect syntax: Calling vertexFeature:  
The projection key 'Spot position' is unknown to the feature  
'Spot position'.
```

Here the selection creator complains because we did not specify the projection in the call. It then tried to find a projection with name identical to the feature name (`Spot position`) and could not find one. So it complained. We would get the same error message if we tried calling the function with the name of a projection that does not exist. For instance `vertexFeature('Spot position', 'U') > 100.`

A similar error would be triggered when calling a feature that does not exist. For instance:

```
vertexFeature('Tralala', 'X') > 100.
```

returns the error message:

```
Evaluation failed. Incorrect syntax: Calling vertexFeature: The  
feature 'Tralala' is unknown to the feature model.
```

Importantly, the same error message is triggered when you call the function **with a feature that has not been computed yet**. For instance, in the Mastodon file you just downloaded, all feature are computed except one: `Track N spots`. If we try to select short tracks with the valid expression that follows:

```
vertexFeature('Track N spots') < 20
```

we would get the same error message (feature unknown). To fix this, you need to go to the `Feature computation` tool, make sure the `Track N spots` feature computer is selected and click `Compute`. After this, the expression will evaluate to a selection correctly:

```
Evaluation successful. Selection has now 33316 spots and 0 edges.
```

1.6.4 Selection with link features.

We can of course run the same kind of expression with link features. For this we use another function: `edgeFeature`. But the principles are the same. For instance, to select all the links that correspond to a cell displacement larger than 3 μm , we write the following:

```
edgeFeature('Link displacement') > 3.
```

which results in:

```
Evaluation successful. Selection has now 0 spots and 16855 edges.
```

Note that in this case we called the function with only one function, the feature name. This worked because this feature has exactly one projection. In that case the function understands that it needs to return the values of this sole projection.

1.6.5 Why are the functions called `vertexFeature` and `edgeFeature`?

In the Mastodon application we deal with data items called spots and links. The data model of Mastodon is a mathematical graph in which a cell in one frame is a vertex. The same cell in the next frame is another vertex, and the two vertices are connected by an edge. Tracks are built by all the vertices connected by edges. Spots and links are specialized versions of the vertices and edges of the graph used in the Mastodon application. But the selection creator is built for any kind of graph that uses the Mastodon API, not only the one in the application we document here. For this reason, the syntax of the functions adopted the most general terms: `vertex` and `edge`, but for us, they points to features of spots and links.

1.6.6 Building selection based on tags.

We can also use tags in our expressions. The main function to do so is `tagSet`. It returns the tags of the specified tag-set for all the spots and links. To create a selection from them, you simply need to check what ones are equal to a certain tag with the equality `==` operator. For instance the expression:

```
tagSet('Reviewed by') == 'JY'
```

will retrieve all the tags in the tag-set called `Reviewed by`, and data items (spots and links) that are tagged with `JY` in this tag-set will be selected. But with our current Mastodon project it returns an error:

```
Evaluation failed. Incorrect syntax: The tag-set 'Reviewed by' is
unknown to the tag-set model.
```

Indeed, the tag-set `Reviewed-by` does not exist.

The function `tagSet` returns spots and links altogether. If you want to retrieve only the spots or only the edges separately, there are two functions: `vertexTagSet` and `edgeTagSet`. Otherwise they work like the `tagSet` function and can be substituted with the same syntax.

There are also special switches to return data items that have no tag in a tag-set or that are tagged with anything in a tag-set.

- `~tagSet('Reviewed by')` will return all the data items that are tagged with any tag in the `Reviewed by` tag-set. It matters not what tag, but there must be one.

- `!tagSet('Reviewed by')` is the converse: it will return all the data items that are not tagged in the Reviewed by tag-set.

And of course, these two switches can be used with the `vertexTagSet` and `edgeTagSet` functions.

1.6.7 Combining expressions.

Since we use boolean expressions we can combine them using logic operator. The first ones are the *and* and *or* operators `&` and `|`. For instance, to select all spots that have 1 link **and** that are in frame 25, we write the following:

```
vertexFeature('Spot N links') == 1 & vertexFeature('Spot frame') == 25
```

To select all spots that have 1 link **or** that are in frame 25, we use the `|` operator instead:

```
vertexFeature('Spot N links') == 1 | vertexFeature('Spot frame') == 25
```

There are other operators which functions have been extended to have a proper meaning with the selection creator. For instance, the above expression is identical to select all spots that have 1 link, and **adding** the spots that are in frame 25. So when it comes to selection, this works as in an addition, and we could write similarly:

```
vertexFeature('Spot N links') == 1 + vertexFeature('Spot frame') == 25
```

But we get an error:

```
Evaluation failed. Incorrect syntax: Improper use of the 'add'
operator, not defined for Integer and VertexFeatureVariable. Use
brackets to clarify operator priority.
```

Here the expression parser is confused due to the classical operation priority of the `+` and `-` operators. We need to use brackets to specify we operate on the results of the feature filtering:

```
(vertexFeature('Spot N links') == 1) + (vertexFeature('Spot frame') == 25)
```

```
Evaluation successful. Selection has now 21814 spots and 0 edges.
```

The `+` operator gives the same results that the `|` operator, they just operate with different priorities.

The **subtract** operator `-` is used to remove data items from a selection. As for the `+` operator, it needs brackets to operate properly. For instance, to remove the spots that belong to the frame 25 from the selection of spots that have 1 link, we write:

```
(vertexFeature('Spot N links') == 1) - (vertexFeature('Spot frame') == 25)
```

```
Evaluation successful. Selection has now 19397 spots and 0 edges.
```

And of course, you can combine expressions using features or tags:

```
(vertexFeature('Spot N links') == 1) - (tagSet('Reviewed by') == 'JY')
```


1.6.8 Editing the selection.

Let's say you already have a selection, and want to alter it with an expression. There are 3 functions that return the content of the *current* selection and let you modify it. As such, they are not exactly functions, but can be considered as variables in an expression.

The `selection` variable simply returns the list of data items that are currently selected. It can then be combined with another expression. For instance, to remove from the selection all the spots that have 2 links, use this expression:

```
selection - ( vertexFeature('Spot N links') == 2 )
```

Of course, because running it will modify the selection, such expressions might have different outcome if you run it several times in a row.

The `vertexSelection` and `edgeSelection` variables work like their selection counterpart, but are limited to return only the spots, respectively the links, of the current selection.

1.6.9 Morphing a selection.

All the expressions we have seen above works like a boolean operation on an array. Except that in our case we have spots and links, and that each have their own separated set of features. This separation is problematic if we want to create selection of spots based on the features of their links, or if we want to include in the selection the links of the spots we selected. To workaroudn this, the selection creator ships the `morph` function.

The `morph` function is a bit special as it is able to change the type of data items that are selected, based on their relations. It is with this function that you will select the links of a spot. For instance:

```
morph(vertexSelection, 'incomingEdges')
```

will select the incoming links of the spots currently in the selection. The vertices themselves will not be included in the final selection.

The `morph` function need two inputs:

- a selection (it can result from `tagSet('TS') == 'A', vertexFeature('F', 'FP') > 3, selection, ...`);
- a list of tokens that specify how to morph the selection.

In the above example, the token was `incomingEdges` which takes all the incoming links of the spots currently selected, and nothing else (not even the spots initially selected). The supported morph tokens, or **morphers** are the following:

- `toVertex` includes the spots currently selected in the `morph` result. When this morpher is not present, the selected spots are removed from the target selection.
- `incomingEdges` includes the incoming links (backward in time) of the selected spots.
- `outgoingEdges` includes the outgoing links (forward in time) of the selected spots.
- `toEdge` includes the links of the source selection in the target selection. When this morpher is not present, the selected links are removed from the target selection.
- `sourceVertex` includes the source spots of the selected links. The source spot of a link is the one backward in time.
- `targetVertex` includes the target spots of the selected links. The target spot of a link is the one forward in time.
- `wholeTrack` includes the whole track of the selected spots and links.

You can combine several morphers, if you put them as a list between brackets. For instance:

```
morph( vertexFeature('Spot N links') == 3, ('toVertex', 'outgoingEdges') )
```

will select the spots that have 3 links, and return them plus their outgoing links.

1.6.10 Summary.

Functions and variables.

Function	Usage
vertexFeature('Spot feature name', 'Projection name')	Returns the values of the specified spot feature and projection. To be used with a comparison operator on a numerical value like >, <, >=, <=, == or !=.
edgeFeature('Link feature name', 'Projection name')	The same, but for link features.
tagSet('Tag set name')	Returns the tag of all data items for the specified tag-set. To be used with the equality operator ==, comparing to a tag belonging to the specified tag set (tagSet('TS') == 'T').
vertexTagSet('Tag set name')	The same, but only returns the spots in the comparison.
edgeTagSet('Tag set name')	The same, but only returns the links in the comparison.
selection	Returns the content of the current selection.
vertexSelection	Returns only the spots in the current selection.
edgeSelection	Returns only the links in the current selection.
morph	Change the type of data items that are selected, based on their relations. To use with the morphers described below.

Morphers.

Morpher name	Effect
toVertex	Include the spots currently selected in the morph result. When this morpher is not present, the selected spots are removed from the target selection.
incomingEc	Include the incoming links (backward in time) of the selected spots.
outgoingEc	Include the outgoing links (forward in time) of the selected spots.
toEdge	Include the links of the source selection in the target selection. When this morpher is not present, the selected links are removed from the target selection.
sourceVert	Include the source spots of the selected links. The source spot of a link is the one backward in time.
targetVert	Include the target spots of the selected links. The target spot of a link is the one forward in time.
wholeTrack	Include the whole track of the selected spots and links.

1.7 Scripting Mastodon in Fiji.

With its integration in Fiji, Mastodon can be scripted in with the [script editor](#) using any of the scripting language there. In the following, we will be using Jython, which is an implementation of Python 2 language.

The example script we will be using is [this one](#) and we will be detailing its section to document the supported functions of Mastodon.

You can find a more comprehensive information on supported scripting functions in the [scripting API reference](#), but it is best starting with this tutorial.

1.7.1 The required import.

The header of a Mastodon script needs to contain the following:

```
#@ Context context

from org.mastodon.mamut import Mamut
import os
```

The scripting gateway of Mastodon is called Mamut and is located in the `org.mastodon.mamut` package. We chose this name to underly that this application is based on the Mastodon libraries, and is similar to the [MaMuT software](#) but with improved functionalities.

The gateway needs to have a `Context` object, ideally coming from the Fiji instance you are running. Thanks to the ImageJ2 scripting API, this is done by adding a special “shabang” at the first line of the script: `#@ Context context`. With this, the `context` variable will contain the context we need in Mastodon.

This is all we need. There is also an import for the `os` package so that we can manipulate file paths.

1.7.2 Creating a new project from a BDV file.

For this tutorial, we can use the small BDV file from the drosophila embryo we used in the first tutorial. You can find it here: [DOI 10.5281/zenodo.3336346](https://doi.org/10.5281/zenodo.3336346)

Download the 3 files, and put them for instance on your Desktop folder. Creating a new Mastodon project from this image file is simply done via the `newProject` function of the gateway:

```
bdvFile = os.path.join( os.path.expanduser('~'), 'Desktop', 'dataset hdf5.xml' )
mamut = Mamut.newProject( bdvFile, context )

logger = mamut.getLogger()
logger.info( 'File opened: %s\n' % bdvFile )
logger.info( 'Mamut instance: %s\n' % mamut )
```

with `bdvFile` being the path to a BDV XML file. If the file cannot be found, you will be shown an error window stating that Mastodon will create the project with an empty image.

Otherwise, the `mamut` object does now contain everything we need to perform tracking and analysis on the image.

Mastodon has a `logger` instance that is used to send messages. By default it sends all messages in the Fiji console:

```
File opened: /Users/tinevez/Desktop/dataset hdf5.xml
Mamut instance: org.mastodon.mamut.Mamut@10dcf50e
```

1.7.3 Basic cell tracking.

The mamut object we have has some functions that readily performs basic tracking. By basic, we mean that the default algorithms are used with a minimal set of sensible parameters:

```
#-----
# Run default detection and linking algorithms.
#-----

logger.info( "\n\n-----" )
logger.info( "\n  Basic detection and linking" )
logger.info( "\n-----\n" )

# Detect with the DoG detector, a radius of 6 and a threshold on quality of 200.
radius = 6.
threshold = 200.
mamut.detect( radius, threshold )
# Link spots with the simple LAP tracker, with a max linking distance of 10, and
↳forbidding gap-closing
max_linking_distance = 10.
gap_closing_n_frames = 0
mamut.link( max_linking_distance, gap_closing_n_frames )
```

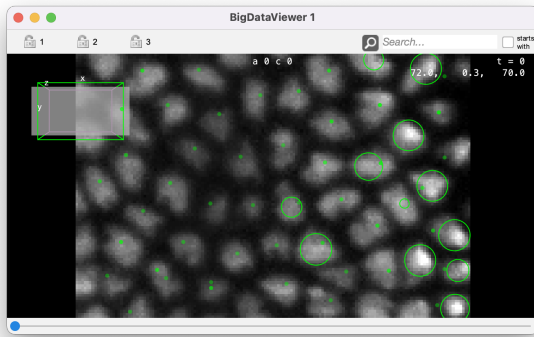
You should get the following output:

```
-----
Basic detection and linking
-----
Detection with DoGDetectorMamut
Detection completed in 6.3 s.
There is now 1622 spots.
Particle-linking with SimpleSparseLAPLinkerMamut
Particle-linking completed in 0.2 s.
There is now 173 tracks.
```

Now we would like to see visually what are the results like. Because we are scripting in Fiji we can generate views like in the GUI. This is done via the WindowManager gateway. For instance, to create a BDV view with the current project, we write:

```
# A new BDV window.
mamut.getWindowManager().createBigDataViewer()
```

You should see this window appearing:



1.7.4 Configure tracking.

The `detect()` and `link()` methods use basic algorithms. You can configure tracking in depth using a dedicated gateway, that called TrackMate (as you can see we did not go very far for names). We called it TrackMate, but it does not have shared code with the TrackMate software. An instance of this gateway is created for a tracking session. It offers methods to configure tracking and run the tracking. But first, let's remove the results of the preceeding tracking:

```
# Reset tracking data.
mamut.clear()
```

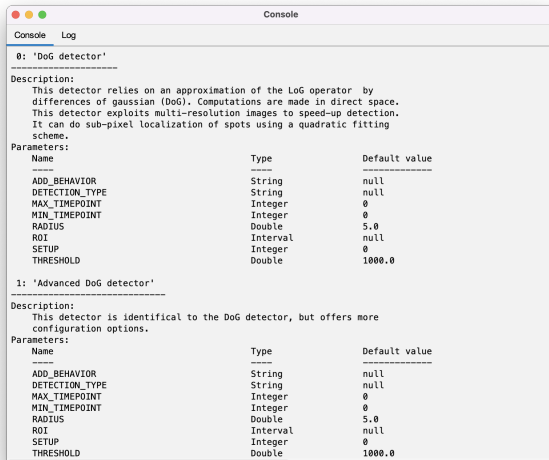
Creating the TrackMate gateway is done as follow:

```
trackmate = mamut.createTrackMate()
```

It can be used to print information on what detectors and linkers are present at runtime. This is important, as Mastodon can be extended with 3rd party algorithms:

```
# Print info on available detectors and linkers.
trackmate.infoDetectors();
trackmate.infoLinkers();
```

A large quantity of text should be printed in the log window. It gives information on the algorithms themselves, and on the parameters they require.



Let's pick and configure some algorithms. We will use the **Advanced DoG detector**, using a special feature it has. This detector allows configuring what happens when it detects a cell in a location where there is already one detection. It can replace the existing one, or let it in place. Parenthetically, this is a very interesting feature to harness complex tissues with for instance different cell sizes depending on the location, or time. You could run a first detection with a first radius, in a specific ROI. Then use this detector a second time in another ROI with another radius. Even if there are some overlap, this detector will be able to avoid creating conflicts or duplicate detections. In this case here it is not super interesting, as we just cleared the previous results.

Selecting and configuring this detector is done as follow:

```

# Configure detection.
trackmate.useDetector( "Advanced DoG detector" );
trackmate.setDetectorSetting( "RADIUS", 8. );
trackmate.setDetectorSetting( "THRESHOLD", 200. );
trackmate.setDetectorSetting( "ADD_BEHAVIOR", "DONTADD" );

```

We just configured the detector. The linker is still the default one. To print what is currently configured, you can call the `info()` method on the TrackMate gateway:

```

# Show info on the config we have.
trackmate.info();

```

We see this:

```

TrackMate settings:
org.mastodon.tracking.mamut.trackmate.Settings@50387161:
- sources: [bdv.viewer.SourceAndConverter@2521e039]
- detector: class org.mastodon.tracking.mamut.detection.AdvancedDoGDetectorMamut
- detector settings: @407185223
  - DETECTION_TYPE = null
  - THRESHOLD = 200.0
  - MIN_TIMEPOINT = 0
  - RADIUS = 8.0
  - ADD_BEHAVIOR = DONTADD
  - ROI = null
  - MAX_TIMEPOINT = 31
  - SETUP = 0

```

(continues on next page)

(continued from previous page)

```
- linker: class org.mastodon.tracking.mamut.linking.SimpleSparseLAPLinkerMamut
- linker settings: @742562126
  - MAX_FRAME_GAP = 2
  - MIN_TIMEPOINT = 0
etc.
```

The linker is configured in the same way. You need to call the

```
useLinker('linkerName')
```

function, with `linkerName` being the name of the linker you want, that can be obtained with the `infoLinkers()` function.

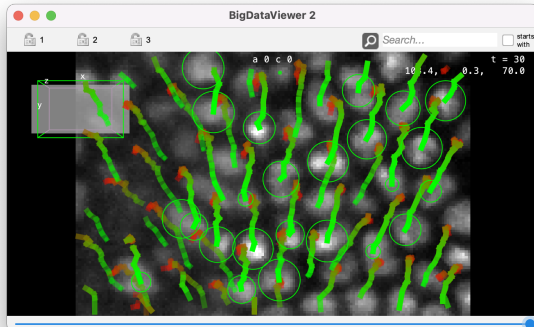
The function

```
setLinkerSetting('key', 'value')
```

is used to set parameters, again using the list of parameters that can be found in the `infoLinkers()` output. In our case we are satisfied with the default linker and parameters, so we can just run the tracking. This is simply done with:

```
# Run the full tracking process.
trackmate.run()
# Show results.
mamut.getWindowManager().createBigDataViewer()
```

We get results that are slightly different from the previous run, as we changed the detection radius.



1.7.5 Computing features.

Numerical features can be also computed via the script. As noted in a [previous tutorial](#), only the features that take a long time to compute have a ‘manual’ computer. You can get the list of computer as follow:

```
mamut.infoFeatures()
```

We get this:

```
Features that can be computed:
- 'Spot center intensity' - Computes the intensity at the center of spots by
  ↳ taking the mean of pixel intensity weighttd by a gaussian. The gaussian weights are
```

(continues on next page)

(continued from previous page)

- centered int the spot, and have a sigma value equal to the minimal radius of the ellipsoid divided by 2.0.
- 'Spot intensity' - Computes spot intensity features like mean, median, etc for all the channels of the source image. All the pixels within the spot ellipsoid are taken into account.
- 'Spot quick mean' - Computes the mean intensity of spots using the highest resolution level to speedup calculation. It is recommended to use the 'Spot intensity' feature when the best accuracy is required.
- 'Spot track ID' - Returns the ID of the track each spot belongs to.
- 'Track N spots' - Returns the number of spots in a track.
- 'Branch depth' - The depth of this branch in the track tree.
- 'Branch duration and displacement' - The displacement and duration of a branch.
- 'Branch N spots' - Returns the number of spots in a branch.

Again, this list is important. Mastodon can also be extended with 3rd party feature computers, so this list above (which is what you see with the vanilla beta-26 Mastodon) will change depending on future developments and future contributions.

To compute them we simply use the `computeFeatures()` function, with a list of names taken from the list above. For instance

```
mamut.computeFeatures( 'Spot intensity', 'Spot center intensity')
```

We get the following simple messages:

```
Feature computation started.
Feature computation finished.
```

Now the features values are available. You get them with a table view, or directly in a text table using the `echo()` method.

To display 2 tables with the first 10 spots and links:

```
mamut.echo( 10 )
```

Console Log

Detection completed in 5.9 s.
There is now 1421 spots.
Particle-linking with SimpleSparseAPLinkerMamut
Particle-linking completed in 0.1 s.
There is now 182 tracks.
Feature computation started.
Feature computation finished.
Model: org.mastodon.mamut.model.Model@22b636b4

Spots:

Id	Label	Frame	X (pixel)	Y (pixel)	Z (pixel)	Detection quality	Spot N links	Spot frame	Mean ch1 (Counts)	Std ch1 (Counts)	Min ch1 (Counts)	Max ch1 (Counts)	Median ch1 (Counts)
0	0	0	51.5	57.3	47.0	555.5	1	0	756.6	338.3	211.0	1674.0	
49	49	1	51.1	57.6	47.3	539.8	2	1	745.7	329.1	201.0	1577.0	
97	97	2	50.7	57.7	46.2	541.2	2	2	725.1	357.6	181.0	1682.0	
143	143	3	50.5	56.9	45.7	555.7	2	3	744.9	357.3	189.0	1813.0	
187	187	4	50.5	57.2	45.7	546.7	2	4	747.0	343.4	195.0	1653.0	
229	229	5	51.0	56.6	44.7	541.7	2	5	715.4	348.1	202.0	1569.0	
275	275	6	50.9	56.2	45.5	534.2	2	6	731.5	348.4	171.0	1755.0	
320	320	7	51.2	57.2	46.5	516.6	2	7	735.5	338.1	193.0	1629.0	
363	363	8	51.3	56.3	45.2	592.3	2	8	741.9	377.1	143.0	1784.0	
407	407	9	51.9	57.2	46.0	544.9	2	9	751.0	358.1	189.0	1711.0	
450	450	10	52.2	57.0	46.1	548.0	2	10	755.6	335.3	208.0	1676.0	

Links:

Id	Source Id	Target Id	Link cost	Link displacement (pixel)	Source spot id	Target spot id	Link velocity (pixel/frame)
0	538	577	4.0	2.0	538.0	577.0	2.0
1	537	584	1.3	1.1	537.0	584.0	1.1
2	536	578	1.6	1.3	536.0	578.0	1.3
3	535	579	2.6	1.6	535.0	579.0	1.6
4	534	571	1.6	1.3	534.0	571.0	1.3
5	532	582	2.7	1.6	532.0	582.0	1.6
6	531	581	0.1	0.3	531.0	581.0	0.3
7	530	583	2.5	1.6	530.0	583.0	1.6

1.7.6 Selecting and editing the model.

We can now use these feature values to select data items, and then edit the model. The syntax for selection based on feature values is the same that in the *selection creator*. Of course we must make sure the features we target are computed to use them for selection. Also, as we are in a script, and because the selection creator uses strings to specify the feature names, we must be cautious with quotes and double-quotes. For instance, to select the short tracks (fewer than 10 spots), we write:

```
mamut.computeFeatures('Track N spots')
mamut.select( "vertexFeature( 'Track N spots' ) < 10" )
```

and we get:

```
Evaluation successful. Selection has now 173 spots and 0 edges.
```

To remove this spots, we now call:

```
mamut.deleteSelection()
```

```
Removed 173 spots and 117 links.
```

1.7.7 Tagging data items.

The selection can also be used to tag objects, exactly in the same way. But as now we do not have tag-sets defined in the current mamut instance:

```
mamut.infoTags()
```

```
No tags are currently defined.
```

Creating tag-sets is done with the `createTag()` function. It accepts a list of strings. The first one must be the name of the tag-sets, and the following list of strings are the tags within the tag-set. For instance:

```
# We create a 'Fruits' tag-set, with 'Apple', 'Banana', 'Kiwi' as tags.
mamut.createTag('Fruits', 'Apple', 'Banana', 'Kiwi')
# The same with a 'Persons' tag-set.
mamut.createTag('Persons', 'Tobias', 'Jean-Yves')
```

You can also set the colors of each tag, by specifying the R,G and B value of the color:

```
mamut.setTagColor('Fruits', 'Apple', 200, 0, 0 )
mamut.setTagColor('Fruits', 'Banana', 200, 200, 0 )
mamut.setTagColor('Fruits', 'Kiwi', 0, 200, 0 )
```

Again, you need to specify first the tag-set name, then the tag, then RGB triplet. And now we get:

```
mamut.infoTags()
```

```
Tags currently defined:
- Fruits:
  - Apple      [ 200,  0,  0 ]
  - Banana     [ 200, 200,  0 ]
```

(continues on next page)

(continued from previous page)

```
- Kiwi                                [  0, 200,  0 ]
- Persons:
  - Tobias                            [ 178, 245, 116 ]
  - Jean-Yves                         [ 159,  98, 229 ]
```

Let's use these tags in our instance. To tag data items, you need to first select them, then calling

```
tagSelectionWith( 'Tag-set name', 'Tag name' )
```

For instance:

```
mamut.select("vertexFeature('Spot position' , 'X' ) > 100.")
mamut.tagSelectionWith('Fruits', 'Kiwi')

mamut.select("vertexFeature('Spot frame' ) == 25")
mamut.tagSelectionWith('Fruits', 'Banana')

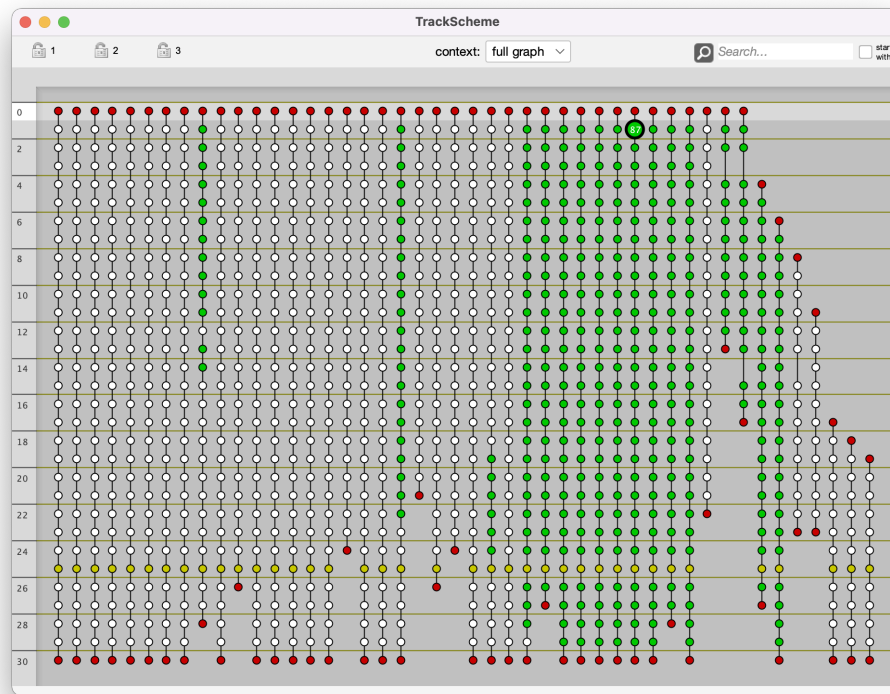
mamut.select("vertexFeature('Spot N links' ) == 1")
mamut.tagSelectionWith('Fruits', 'Apple')

# Clear selection.
mamut.resetSelection()
```

1.7.8 Showing and configuring views.

Let's visualize the tags we just assigned in a TrackScheme window. We will again use the WindowManager gateway, using a python dictionary to configure settings. We set the coloring mode to be a certain tag-set like this:

```
displaySettings = {'TagSet' : 'Fruits'}
mamut.getWindowManager().createTrackScheme( displaySettings )
```



1.7.9 Undo and redo.

The undo and redo operations are also supported with scripting. But you have to set the undo points manually. The undo points are 'states' of the data where you navigate back and forth with undo / redo. In GUI, they are set automatically, but in the script, you need to specify them yourself. To illustrate it, we will need to remove a few spots, then undo this operation.

```
# Print some info on the model before we remove spots.
logger.info( "Before:\n" )
mamut.info()

# Let's delete 500 spots.
it = mamut.getModel().getGraph().vertices().iterator()
for i in range( 500 ):
    if it.hasNext():
        spot = it.next()
        mamut.getModel().getGraph().remove ( spot )

# We mark this point as an undo point. Calling undo() / redo()
# navigates in the stack of these undo points.
mamut.getModel().setUndoPoint()
```

This yields:

```
Before:
Data model #4
- mastodon project file: Not defined.
```

(continues on next page)

(continued from previous page)

```
- dataset: /Users/tinevez/Desktop/datasethdf5.xml
- objects: 1248 spots, 1202 links and 46 tracks.
- units: pixel and frame
```

After deleting some spots:

Data model #4

```
- mastodon project file: Not defined.
- dataset: /Users/tinevez/Desktop/datasethdf5.xml
- objects: 748 spots, 702 links and 46 tracks.
- units: pixel and frame
```

Now undoing this is simply a matter of calling `undo()`

```
mamut.undo()
logger.info( "After undo:\n" )
mamut.info()
```

After undo:

Data model #5

```
- mastodon project file: Not defined.
- dataset: /Users/tinevez/Desktop/datasethdf5.xml
- objects: 1248 spots, 1202 links and 46 tracks.
- units: pixel and frame
```

And redoing it:

```
mamut.redo()
logger.info( "After redo:\n" )
mamut.info()
```

After redo:

Data model #5

```
- mastodon project file: Not defined.
- dataset: /Users/tinevez/Desktop/datasethdf5.xml
- objects: 748 spots, 702 links and 46 tracks.
- units: pixel and frame
```

1.7.10 Saving a Mastodon project.

Saving to an existing Mastodon file is done with the `save()` method. But the project we created since the beginning of this tutorial has never been saved:

```
mamut.save()
```

Indeed, we get an error message:

```
Mastodon file not set. Please use #saveAs() first.
```

We need to provide a path for a new Mastodon file. This is done with the `saveAs()` method:

```
newMastodonFile = os.path.join( os.path.expanduser('~'), 'Desktop', 'test_scripting.
↳mastodon' )
mamut.saveAs( newMastodonFile )
```

From now on, any call to `save()` will save to this file.

Saving to `/Users/tinevez/Desktop/test_scripting.mastodon`

1.7.11 Reloading a Mastodon project.

Again, nothing complicated. We need to use the Mamut gateway again:

```
existingMastodonFile = os.path.join( os.path.expanduser('~'), 'Desktop', 'test_scripting.  
↪mastodon' )  
mamut2 = Mamut.open( existingMastodonFile, context )  
mamut2.info()
```

Data model #7

- mastodon project file: `/Users/tinevez/Desktop/test_scripting.mastodon`
- dataset: `/Users/tinevez/Desktop/datasethdf5.xml`
- objects: 748 spots, 702 links and 46 tracks.
- units: pixel and frame

1.8 Moving around in the BDV views.

Action	Key
View.	
Move in X & Y.	Right-click and Drag.
Move in Z.	Mouse-wheel. Press and hold Shift to move faster, Control to move slower.
Align with XY plane	Shift Z
Align with YZ plane	Shift X
Align with XZ plane	Shift C or Shift A. For these 3 shortcuts, the view will rotate around the mouse position on the BDV.
Zoom / Un-zoom.	Control + Shift + Mouse-wheel or Command + Mouse-wheel. The view will zoom and unzoom around the mouse location.
Time-points.	
Next time-point.] or M
Previous time-point.	[or N
Bookmarks.	
Store a bookmark.	Shift B then press any key to store a bookmark with this key as label. A bookmark stores the position, zoom and orientation in the view but not the time-point. Bookmarks are saved in display settings file.
Recall a bookmark.	Press B then the key of the bookmark.
Recall a bookmark orientation.	Press O then the key of the bookmark. Only the orientation of the bookmark will be restored.
Image display.	
Select source 1, 2 ...	Press 1 / 2 ...
Toggle fused mode.	Press F. In fused mode, several sources are overlaid. Press Shift + 1 / Shift + 2 ... to add / remove the source to the view. In single-source mode, only one source is shown.
Save / load display settings.	F11 / F12. This will create a XYZ_settings.xml file in which the display settings and bookmarks will be saved.

Before mid-2023, the brightness and source visibility dialogs were configured in dialogs which visibility were toggled with the F and F6 shortcuts respectively. With the new version of BDV, they are configured with a side-pane in a BDV window that can be toggled by placing the mouse ofther right part of the images:

1.9 Editing spots and links in the BDV views.

See also the ‘*Linking spots*’ and ‘*Manually adding spots and linking them*’ sections in [this tutorial](#).

Action	Key
<i>Adding, deleting and modifying spots.</i>	
Add a new spot.	Put the mouse at the desired position and press the A key.
Move a spot.	Put the mouse inside a spot, press and hold space, and move the mouse to the desired position.
Delete a spot.	Put the mouse inside a spot and press D.
Change the radius of a spot.	Put the mouse inside a spot and press Q (make it smaller) or E (bigger). Hold Shift to make larger changes or Control for finer changes.
<i>Adding and linking spots.</i>	
Link one spot to an existing one in the <i>next</i> frame.	Put the mouse inside a source spot, and press and hold L. The viewer moves to the next time-point and shows a preview of the link. Release the L key in the desired target spot. A link is created from the source spot to the target spot.
Link one spot to an existing one in the <i>previous</i> frame.	Same procedure, but press Shift L.
Add and link to a spot in the <i>next</i> frame.	Put the mouse inside a source spot, and press and hold A. The viewer moves to the next time-point, creates a spot there and links it to the source spot. While holding A, move the new spot to the desired location, and release A.
Add and link to a spot in the <i>previous</i> frame.	Same procedure, but press Shift A.
Toggle the auto-linking mode.	Control L
<i>Removing links.</i>	
Delete a link.	Put the mouse over the link to delete and press D.
<i>Selection editing.</i>	
Add a spot / link to the selection.	Shift click on a spot or a link to add / remove it to / from the selection.
Clearing the selection.	Click on an empty place of the image.
Remove selection content.	Shift delete
<i>Undo / Redo.</i>	
Undo / Redo	Control Z / Control Shift Z

1.10 Moving around in the TrackScheme views.

Action	Key
<i>View.</i>	
Move around.	Right click and Drag or with the trackpad.
Zoom / unzoom in X.	Shift Mouse-wheel
Zoom / unzoom in Y.	Control Mouse-wheel
Zoom / unzoom in X & Y.	Control Shift Mouse-wheel
Full zoom, full unzoom.	Press Z .The view zoom at max level to the mouse location.Press Z again to unzoom fully.
Zoom in a box.	Press and hold Z then drag a box.The view will zoom to the box.

1.11 Editing spots and links in the TrackScheme views.

Action	Key
<i>Editing spots.</i>	
Remove a spot.	Press D with the mouse inside the spot to remove.
Edit the label of a spot.	Press Enter when a spot is focused, then enter its label and press Enter to validate.
<i>Creating links between spots.</i>	
Create a link between two spots.	Press and hold L with the mouse inside the source spot. Release L when inside the target spot.
Remove a link.	Press D with the mouse on the link to remove.
<i>Selection editing.</i>	
Add a spot / link to the selection.	Shift-click on a spot or a link to add / remove it to / from the selection.
Select all in a box.	Click and drag a box.Shift-click and drag to add the content of the box to the current selection.
Clearing the selection.	Click on an empty place of the image.
Remove selection content.	Shift-delete.
<i>Undo / redo.</i>	
Undo.	Control-Z.
Redo.	Control-shift-Z.

1.12 Shortcuts for the table views.

Action	Key
<i>Navigation.</i>	
Move from row to row.	↓ and ↑
Move column to column	← and →
Jump to next / previous table.	Control / Control
<i>Editing.</i>	
Edit spot label	F2 when the focus is in the spot label column.
Toggle tag	space when the focus is in the desired tag column
<i>Selecting.</i>	
Add next / previous row to selection	Shift ↓ / Shift ↑
Add range to selection	Shift and Shift
Toggle row into selection	Control Left-click or Command Left-click

1.13 Navigation through lineages in BDV and TrackScheme views.

In BDV and TrackScheme views, the focused spot is the last one selected.

Action	Key
<i>Navigation with the Focus in BDV views.</i>	
Follow a spot across time within a track with the focus.	↓ and ↑
Jump to the beginning of another branch.	Control Alt ↑
Jump to the end of another branch.	Control Alt ↓
<i>Navigation with the Focus in TrackScheme.</i>	
Follow a spot across time within a track with the focus.	↓ and ↑
Move the focus around from one track or one track branch to another (i.e. navigate to sibling).	← and →
Jump to the beginning of a branch.	Alt ↑
Jump to the end of a branch.	Alt ↓
Edit the label of the focused spot.	Enter

1.14 Setting the selection.

Action	Key
<i>Setting selection.</i>	
Add a spot or link to the selection.	Shift + Left-click. Do it again to remove a selected spot or link from the selection.
Select all spots and links.	Control A
Select just all the spots.	Control Alt A
Select just all the links.	Control Shift A
Select all the spots and links in the track of the currently focused spot.	Shift space
Select all the spots and links <i>upward</i> (backward in time) in the track of the currently focused spot.	Shift
Select all the spots and links <i>downward</i> (forward in time) in the track of the currently focused spot.	Shift
Add the previous / next spot in time to the selection.	Shift ↑ / Shift ↓
Add all spots to the selection from current spot to the beginning of a branch.	Shift Alt ↑
Add all spots to the selection from current spot to the end of a branch.	Shift Alt ↓
Add all spots to the selection from current spot to the beginning of another branch.	Shift Control Alt ↑
Add all spots to the selection from current spot to the end of another branch.	Shift Control Alt ↓
<i>Selection in TrackScheme.</i>	
Add the spot to the left / right of the focus to the selection.	Shift ← / Shift →
Draw a selection box	Left-click Drag

1.15 Existing plugins

1.15.1 Mastodon Tracking

Automated tracking algorithms for Mastodon.

1.15.2 Mastodon Pasteur

Small collection of plugins developed for the research led in the Institut Pasteur, Paris, but of general utility.

1.15.3 Mastodon Selection Creator

Mastodon plugin to create selections from mathematical expressions.

1.15.4 Mastodon Tomancak

Various Mastodon plugins for use in Tomancak lab projects

1.15.5 Ellipsoid Fitting

Mastodon plugin for fitting ellipsoids around spots

1.15.6 Mastodon Deep Lineage

A collection of plugins to analyse lineages of tracked objects in Mastodon, e.g. Lineage Tree Classification, Export of ellipsoids as image, some more numerical features for Spots and BranchSpots. See the [Mastodon Deep Lineage](#).

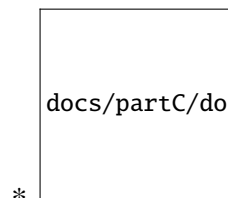
1.16 Mastodon Deep Lineage - a collection of plugins to analyse lineages of tracked objects in Mastodon

1.16.1 Installation instructions

- Add the listed Mastodon update site in Fiji:

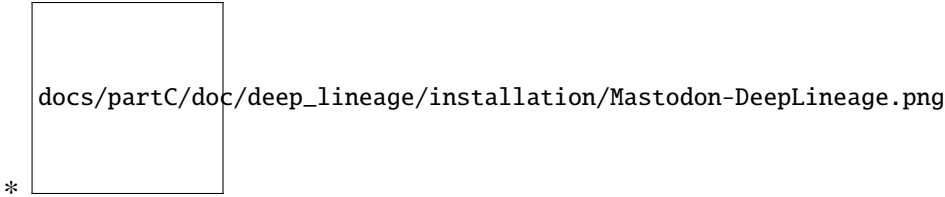
- Help > Update > Manage update sites

- * Name: Mastodon



- *

- Add the unlisted Mastodon Deep Lineage update site in Fiji:
 - Help > Update > Manage update sites > Add Unlisted Site
 - * Name: Mastodon-DeepLineage
 - * URL: <https://sites.imagej.net/Mastodon-DeepLineage/>





1.16.2 Numerical Features added to Mastodon

Spot Features

Feature name	Projections	Description	Formula/Visualisation
Spot Ellipsoid	Short semi axes, Middle semi axis, Long semi axis	The ellipsoid semi axes in ascending order of length.	The semi axes are computed applying the square root to the eigenvalues of the so-called covariance matrix of the spots
	Volume	The volume of the ellipsoid.	$V=\frac{4}{3} \pi a b c$
Spot Ellipsoid Aspect Ratios	Aspect ratio short to middle	The ratio between the short axis and middle axis.	$\frac{\text{short axis}}{\text{middle axis}}$
	Aspect ratio short to long	The ratio between the short axis and long axis.	$\frac{\text{short axis}}{\text{long axis}}$
	Aspect ratio middle to long	The ratio between the middle axis and long axis.	$\frac{\text{middle axis}}{\text{long axis}}$
Spot Branch ID	<i>idem</i>	The ID of the branch spot each spot belongs to.	

Branch-spot features

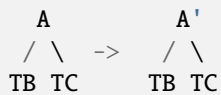
Feature name	Projections	Description	Formula/Visualisation
Branch N Leaves	<i>idem</i>	The total number of leaves of a branch spot in the whole track subtree of this branch spot.	
Branch N Successor	<i>idem</i>	Total number of successors of a branch spot in the whole track subtree of this branch spot.	
Branch Sinuosity	<i>idem</i>	Computes the sinuosity of a spot during its life cycle (cf. Sinuosity), i.e. how much the track represented by the branch is curved. Values close to 1: almost straight movement. Values significantly higher than 1: winding or meandering movement.	i.e.:

1.16.3 Lineage Tree Classification

- This plugin is capable of grouping similar lineage trees together.
- The similarity of a pair of lineage trees is computed based on the Zhang edit distance for unordered trees (Zhang, K. *Algorithmica* 15, 205–222, 1996). This method captures the cost of the transformation of one tree into the other.
- The cost function applied for the tree edit distance uses the attribute branch spot duration, which is computed as a difference of time points between to subsequent divisions reflecting the start and the end of a spot's lifetime.
- Thus, the lineage classification operates on Mastodon's branch graph.
- The Zhang unordered edit distance allows the following edit operations:

Note: The prefix T may represent a node **or** a complete subtree. Nodes without this prefix are just nodes.

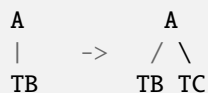
1. Change label



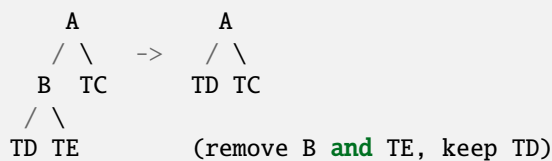
2a: Remove subtree (opposite of 2b)



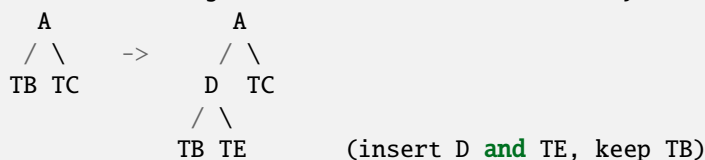
2b: Add new subtree (opposite of 2a)



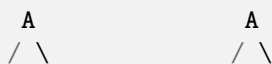
3a: Remove subtree but keep one child (opposite of 3b)



3b: Convert existing subtree into child of a newly inserted subtree (opposite of 3a)

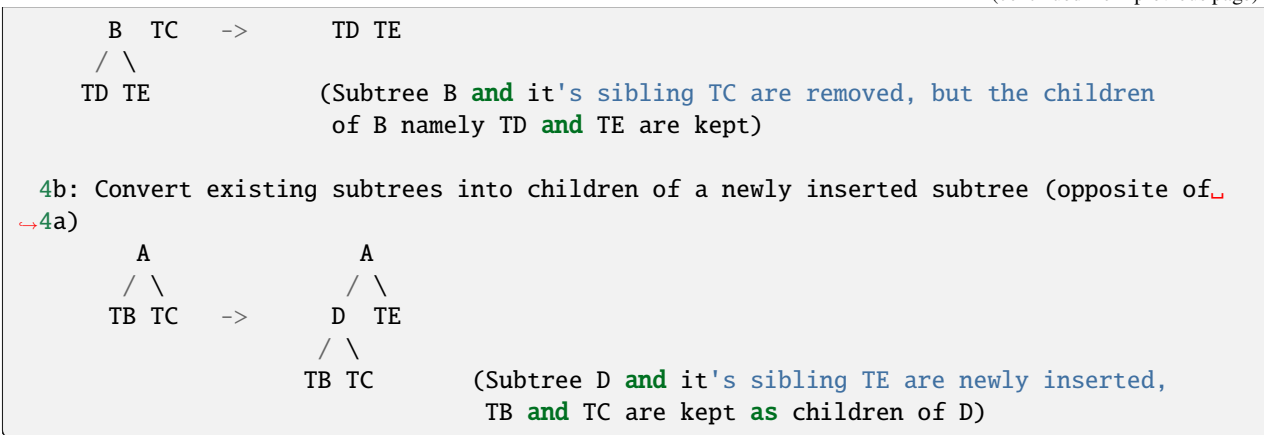


4a: Remove subtree (and siblings) but keep all children (opposite of 4b)



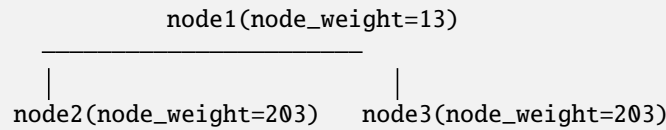
(continues on next page)

(continued from previous page)

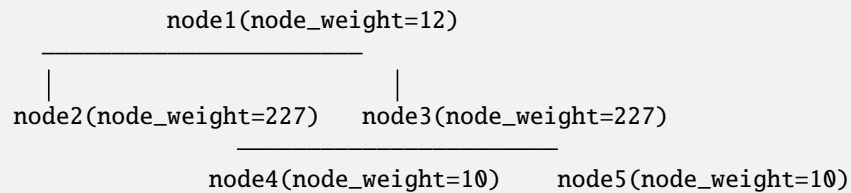


- A basic example of the tree edit distance:

Tree1



Tree2



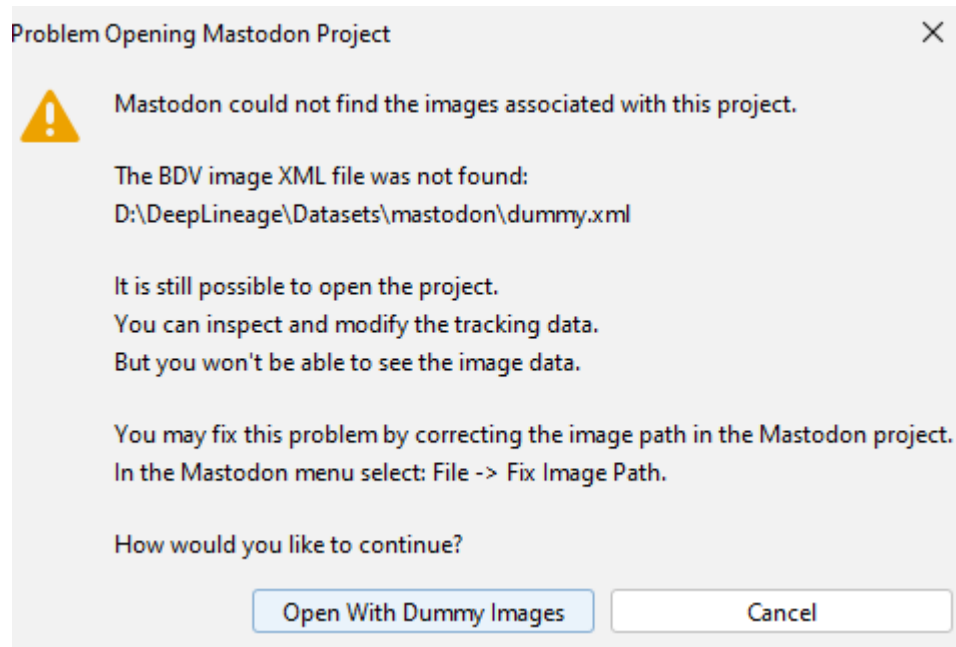
- Edit distance of 69, because:
 - one node has a difference of 1
 - two nodes have a difference of 24 each
 - two extra nodes are added with a weight of 10 each
 - $\text{Zhang Tree Edit Distance}_{\{Tree1, Tree2\}} = 1 + 2 \times 24 + 2 \times 10 = 69$
- The tree edit distances are computed between all possible combinations of lineage trees leading to a two-dimensional matrix. The values in this matrix are considered to reflect similarities of lineage trees. Low tree edit distances represent a high similarity between a discrete pair of lineage trees.
- This similarity matrix is then used to perform an **agglomerative hierarchical clustering** into a specifiable number of classes.

Parameters:

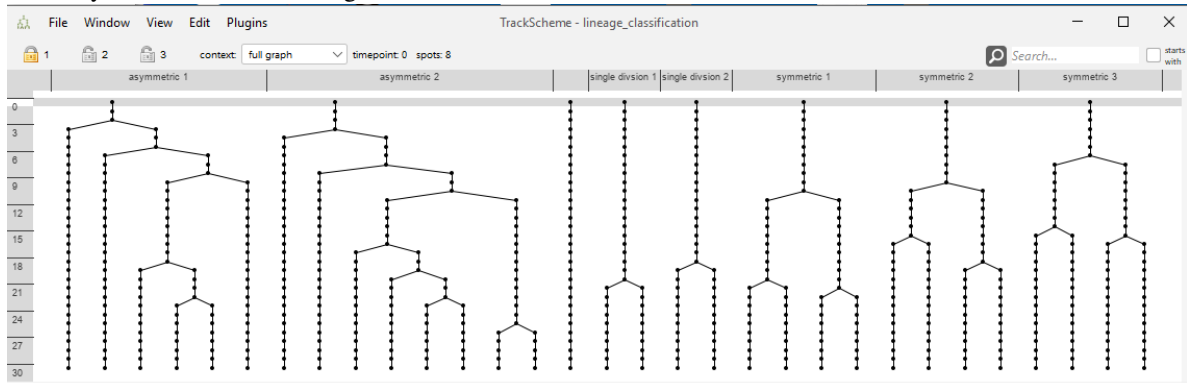
- Crop criterion:
 - Time point (default)
 - Number of spots
- Crop start
- Crop end
- Number of classes (minimum 2)
- Minimum number of divisions
- Similarity measure:
 - Normalized Zhang Tree Edit Distance (default). $\frac{\text{distance_}\{\text{treeA}, \text{treeB}\}}{\text{distance_}\{\text{treeA}, \text{emptyTree}\} + \text{distance_}\{\text{treeB}, \text{emptyTree}\}}$
 - Per Branch Spot Zhang Tree Edit Distance. $\frac{\text{distance_}\{\text{treeA}, \text{treeB}\}}{\text{numBranchSpotsA} + \text{numBranchSpotsB}}$
 - Zhang Tree Edit Distance as described in (Zhang).
- Linkage strategy for hierarchical clustering, cf. [linkage methods](#)
 - Average (default)
 - Single
 - Complete
- Feature:
 - Branch duration (default and currently only selectable feature)
- Show dendrogram of clustering

Example:

- Demo data: Example data set
 - The demo data does not contain any image data.
 - The spatial positions of the spots are randomly generated.
 - When opening the dataset, you should confirm that you open the project with dummy images.



- The track scheme of the demo data containing 8 lineage tree in total. You may see that the “symmetric”, the “asymmetric” and the “single division” trees look similar to each other, but dissimilar to the other trees.



Classification of Lineage Trees

Classification of Lineage Trees

This plugin is capable of grouping similar lineage trees together. This is done by creating a tag set and assigning subtrees that are similar to each other with the same tag.

The similarity between two subtrees is computed based on the Zhang edit distance for unordered trees ([Zhang, K. Algorithmica 15, 205–222, 1996](#)). The similarity measure uses the attribute the cell lifetime, which is computed as a difference of timepoints between two subsequent divisions. It is possible to apply the *absolute difference*, *average difference* or the *normalized difference* of cell lifetimes.

The similarity is computed between all possible combinations of subtrees leading to a two-dimensional similarity matrix. This matrix is then used to perform a [agglomerative hierarchical clustering](#) into a specifiable number of classes. For the clustering three different [linkage methods](#) can be chosen.

Crop criterion	Timepoint	▼
Crop start	0	◊
Crop end	30	◊
Number of classes	4	◊
Minimum number of cell divisions	0	◊
Similarity measure	Normalized Zhang Tree Distance	▼
Linkage strategy for hierarchical clustering	Average linkage	▼
Feature	Branch duration	▼
Show dendrogram of clustering	<input checked="" type="checkbox"/>	

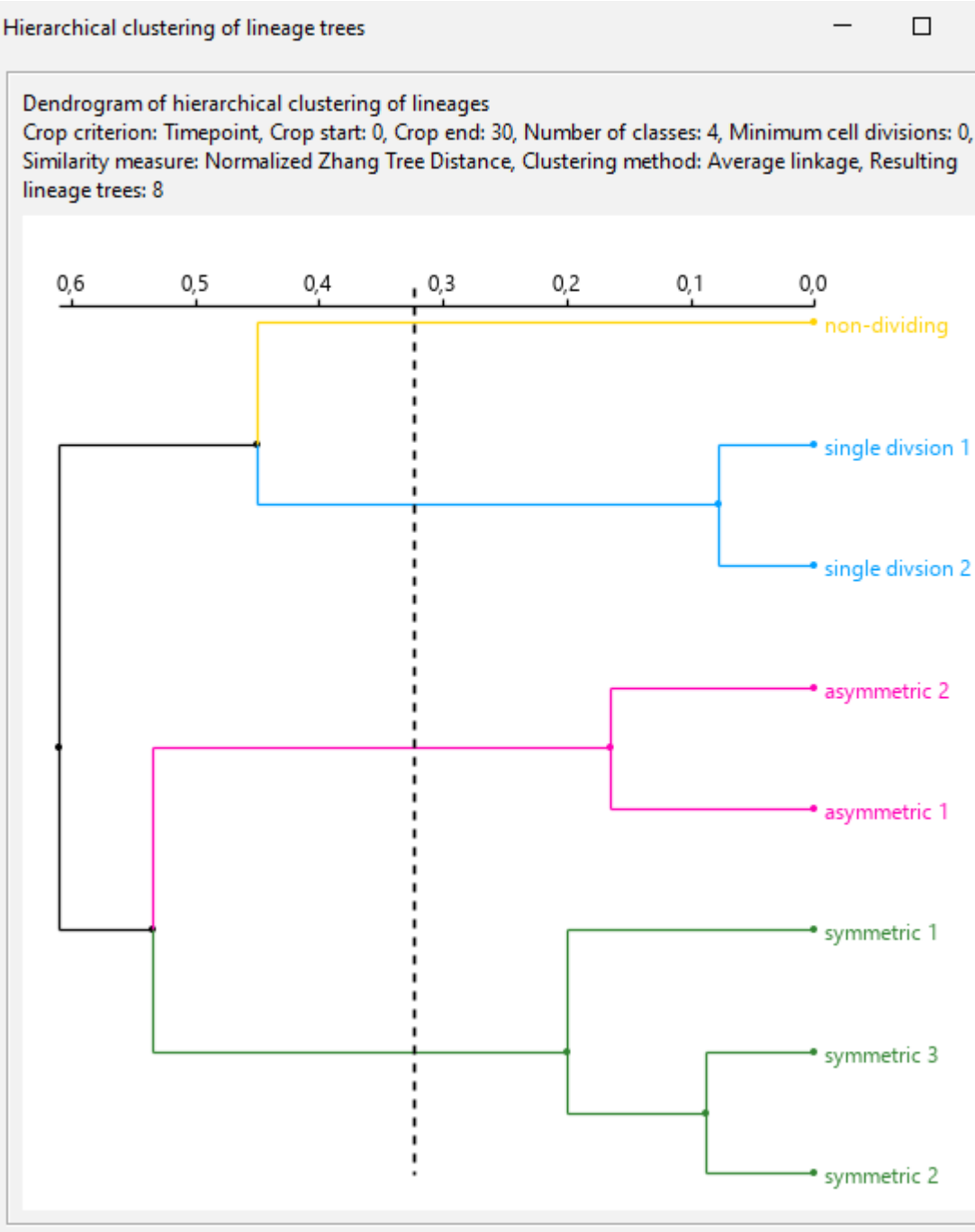
Parameters are valid.

Classified lineage trees.

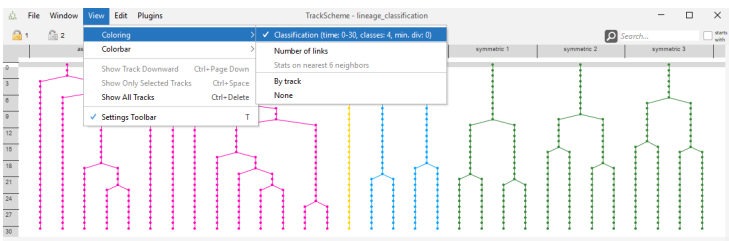
Tag set created.

Classify lineage trees

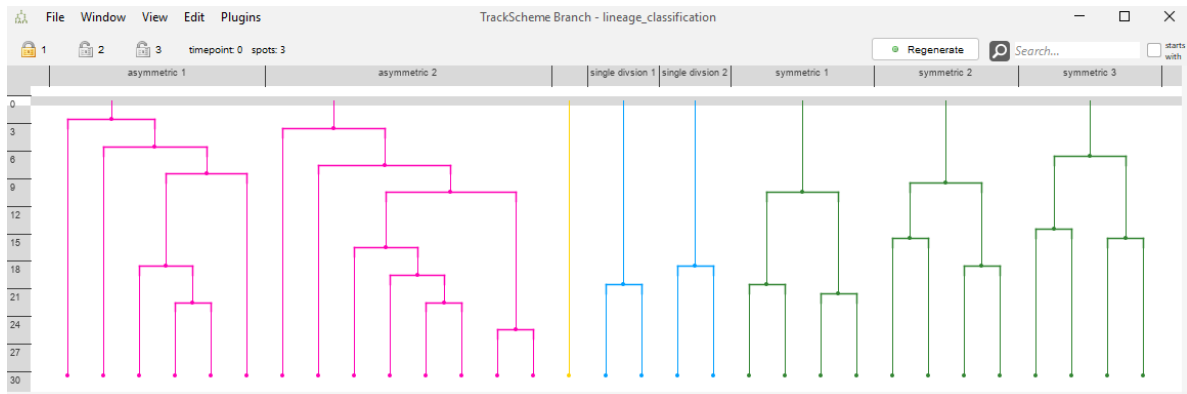
- The lineage classification dialog.



- The resulting dendrogram.

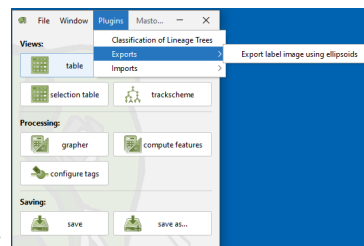


- The resulting tag set used for coloring the track scheme.
- The resulting tag set used for coloring the track scheme branch view.

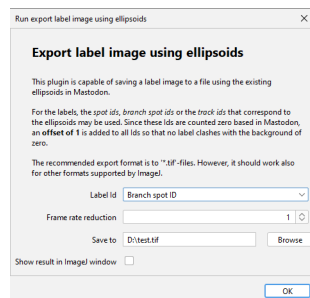


1.16.4 Label image exporter

- The Label image exporter is capable of saving a label image to a file using the existing ellipsoids in Mastodon.
- For the labels, the *spot ids*, *branch spot ids* or the *track ids* that correspond to the spots / ellipsoids may be used. Since these Ids are counted zero based in Mastodon, an **offset of 1** is added to all Ids so that no label clashes with the background of zero.
- The recommended export format is “*.tif”-files. However, it should work also for other formats supported by ImageJ.
- The export uses an image with signed integer value space, thus the maximum allowed id is 2.147.483.646.



- Exporter can be found here:



- The dialog:
 - Label Id: The id that is used for the labels. The default is the Spot track Id.
 - * The ids correspond to the highlighted columns in the feature table:

Spot	Link	Label	ID	Branch spot ID	N incoming	N outgoing	Spot N links	Spot frame	X (pixel)	Y (pixel)	Z (pixel)	Spot radius	Spot track ID
0		0	0	0	0	1	1	0	61,362	76,349	55,142	6,671	0
1	BranchSpot	1	1	1	0	1	1	0	113,153	3,233	67,34	6,188	1
2	BranchLink	2	2	2	0	1	1	0	35,741	20,075	57,84	6,768	2
3		3	3	3	0	1	1	0	79,251	35,248	56,869	6,936	3
4		4	4	4	0	1	1	0	143,312	95,934	77,957	6,414	4
5		5	5	5	0	1	1	0	114,927	97,665	65,717	6,124	5
6		6	6	6	0	1	1	0	129,541	98,146	70,754	5,784	6
7		7	7	7	0	1	1	0	36,41	50,149	57,431	6,771	7
8		8	8	8	0	1	1	0	97,896	26,362	61,752	6,455	8

- Frame rate reduction: Only export every n-th frame. 1 means no reduction. Value must be ≥ 1 .
 - * The frame number corresponds to the *Spot frame* column in the feature table.
- Save to: Path to the file to save the label image to.

Example:

- Demo data: [Example data set](#)
- The timelapse with the ellipsoids in BigDataViewer:
- The exported tif imported into [Napari](#) 3D view:

1.17 Mastodon data structures.

Before we discuss how to extend Mastodon with your own plugins, we need to learn a little bit about how Mastodon stores and deal with the data it can visualize and edit. This page will walk you through the data structure in Mastodon from a Java programmer perspective.

1.17.1 The data model.

All the data is aggregated in the `ProjectModel` class. This is the class you will access for instance when you write a plugin, and it contains many things that relate to the image, the tracking data and the user-interface. We document below its most important components.

Image data.

The image data is stored in a special class `SharedBigDataViewerData`. It aggregates several objects and information from the BDV ecosystem. You can access the list of sources and the `SpimData` objects from it.

```
SharedBigDataViewerData imageData = projectModel.getSharedBdvData();
AbstractSpimData< ? > spimData = imageData.getSpimData();
List< SourceAndConverter< ? > > imageData.getSources();
```

Tracking data, the `Model` class.

The tracking data, including the graph, the feature values, tags, etc. are stored in the `Model` class. This is the main class to focus on to interrogate and manipulate tracking data.

```
Model model = mamutAppModel.getModel();
```

The `model` contains everything about the tracking data, from the individual objects, tracks, tags and numerical feature values. It also controls the undo / redo mechanisms.

The tracking data. The `ModelGraph` class.

The tracks, that is the objects we follow, their shape and position, and how they evolve over time, are stored as a [mathematical graph](#). Precisely, the graph we use in Mastodon is a simple, directed graph: the edges have a direction, and there is at most one edge between two vertices. The vertices of the graph are the objects we track: cells or organelles, and they are represented by spots, described below. One spot, of the `Spot` class represents one cell or one object at one time-point. The edges of the graph link objects over time. They are represented by the `Link` class, also described below. Two spots `s1` and `s2` that represent the same cell at time-points `t1` and `t2=t1+1` are linked together by an edge that goes from `s1` to `s2` labeled `l1=s1→s2`. The direction of the edge is important. In Mastodon, the edges are always oriented forward in time: the source spot of the edge is always in a time-point that is strictly lower than the target spot of the edge. So there cannot be an edge oriented backward in time, and there cannot be an edge between two spots that are in the same time-point.

The class of the graph we use in Mastodon is `ModelGraph`. It is based on a special data structure to manage large graphs that we developed specifically for Mastodon, and described [elsewhere in this documentation](#). The graph instance can be obtained as follow:

```
ModelGraph graph = model.getGraph();
```

This graph class is often required by Mastodon algorithms and plugins, but is not the first entry point to browse and navigate the data from a user point-of view. You can access the links of a spot directly with the `spot` class, and the collections of spots in a time-point via the index.

The data objects. The `Spot` class.

The data objects are stored as spots with the `Spot` class. A spot is a 3D ellipsoid with a label and there are methods in the `Spot` class to access all these properties:

The `Spot` class implements `ImgLib2 RealPositionable` and `RealLocalizable` interfaces, so you will find:

- all the methods to access the position: `localize(double[] pos)`, `getDoublePosition(int d)`. This refers to the center of the ellipsoid represented by the spot.
- conversely, all the methods to set the position: `setPosition()`.
- The `setLabel(String label)` and `getLabel()` are used for the spot label.
- `setCovariance(double[][] cov)` / `getCovariance()` to set or get the spot shape. As spots are ellipsoids, their shape is controlled by a covariance matrix that controls the ellipsoid size and orientation.
- `getTimepoint()` to return the timepoint this spot belongs to. The time-point of a spot is set at construction (see below) and cannot be changed.

Also, a spot is a vertex in the track graph (see below) and the `Spot` class has therefore methods to access the edges it is connected to in this graph:

- `edges()` return all the edges attached to this spot, regardless of their direction.

- `incomingEdges()` returns the collection of edges that are incoming to the spot (for which this spot is the *target* vertex).
- `outgoingEdges()` is the converse: it returns the collection of edges that are outgoing, that is, for which this spot is the *source* vertex). The collection returned by `edges()` is the union of the collections returned by `incomingEdges()` and `outgoingEdges()`.

The Link class.

The index. The `SpatioTemporalIndex` and `SpatialIndex` classes.

The model maintains an index of the spots that are presents in each time-point, in the `SpatioTemporalIndex` class and its implementation. You can get the index from the model with

```
SpatioTemporalIndex< Spot > index = model.getSpatioTemporalIndex();
```

The index class has a generic parameter: `<Spot>`. This just indicates that for the data model we deal with in the Mastodon application, the objects we deal with are of the `Spot` class.

The main use of the index is to retrieve all the spots at one specific time-point:

```
// Retrieve all the spots in the 3rd time-point.
int tp = 2;
SpatialIndex< Spots > spatialIndex = index.getSpatialIndex( tp );
```

We get a `SpatialIndex` instance, which can be iterated over **all** the spots it contains:

```
for (Spot spot : spatialIndex)
{
    // Do something with the spot.
}
```

The methods `isEmpty()` and `size()` methods can tell whether the collection is empty or its size. But the main interest of this `SpatialIndex` class is that it offers efficient methods to query the spots that are in specified volume, or the closest spots around a position. The method:

```
NearestNeighborSearch< Spot > nn = spatialIndex.getNearestNeighborSearch();
```

returns a class that can perform efficient nearest-neighbor search. Here is an example of its use:

```
import net.imglib2.RealPoint;

// ...

// Get the NN instance for this time-point.
SpatioTemporalIndex< Spot > index = model.getSpatioTemporalIndex();
int tp = 2;
SpatialIndex< Spots > spatialIndex = index.getSpatialIndex( tp );
NearestNeighborSearch< Spot > nn = spatialIndex.getNearestNeighborSearch();

// Search the closest spot to a position.
// (We use a RealPoint. It could have been a Spot since a Spot is a RealLocalizable.)
RealLocalizable pos = new RealPoint(1.2, 5.6, 7.8);
```

(continues on next page)

(continued from previous page)

```
// Perform the search
nn.search( pos );

// Get search results.
Spot target = nn.getSampler().get();
double distance = nn.getDistance();
```

The `SpatialIndex` class can also return a `IncrementalNearestNeighborSearch`, which also perform nearest-neighbor search, but can be iterated to return the N closest spots to a position.

1.18 Creating custom plugins in Mastodon.

Here we introduce how to write your own plugins in Mastodon.

A Mastodon plugin is a piece of code that implements a functionality that you can define and distribute, without depending on editing the core of Mastodon code, or on any action from the core developers. We built the plugin system and documented it so that extending Mastodon can be done without the core developers intervention, or even without us knowing about it. To achieve this, we rely on [SciJava](#). This core library offers a very simple and efficient plugin discovery mechanism. With it, you can write some code in Java, compile it in a jar file, drop the jar file in Fiji, and Mastodon will automatically pick it up, and integrate the functionality it ships. As an example, all [TrackMate components](#) use the same mechanism, making it fast and easy to add new functionality to this software.

As opposed to other ways of extending Mastodon reviewed next, a Mastodon plugin is very generic. It can have any goal that you envision, and does not have limitations in what you have access to. You can use a Mastodon plugin to implement and import / export filter, a specific analysis tool, or a feature that edits the model.

1.18.1 The code template.

We will use and document a template file that is included in an example repository. You can clone it from:

<https://github.com/mastodon-sc/mastodon-plugin-example>

The simplest approach to start developing your plugin would be to simply edit this repository, changing its name, editing the `.java` files and modifying the content of the `pom.xml` file. Maven will generate a `.jar` file that you can drop in the `Fiji.app/jars` folder. Provided that this Fiji has Mastodon installed in it, the extensions you will develop within this code will be discovered and integrated into Mastodon.

In the template you will see that there are several examples, each focusing on one of the multiple specific extensions of Mastodon. Here we will just focus on the general plugin, for which an example is in the class:

`src/main/java/org/mastodon/mamut/example/plugin/MastodonPluginExample.java`

(Or [online](#).)

This example creates a Mastodon plugin that, when called, simply displays a dialog showing the date and the number of spots in the model. This class is fully annotated and commented, and can be read from top to bottom. We repeat and precise here the information that you can find in the class and directly edit.

1.18.2 The plugin class hierarchy.

The class file starts with:

```
package org.mastodon.mamut.example.plugin;

...

@Plugin( type = MastodonPluginExample.class )
public class MastodonPluginExample implements MamutPlugin
{
    ...
}
```

The interface to implement.

The class implements the interface `org.mastodon.mamut.plugin.MamutPlugin`, which name and package can be surprising at first. Indeed, in the code base, we tried to separate the common, general code that implements the Mastodon technology, from the user application also termed ‘Mastodon’, that is currently shipped in Fiji. The Mastodon application is a Fiji end-user tool that focuses on tracking cells and building lineages in large 3D+T images. But the Mastodon API could be used to build other scientific applications, not necessarily focused on tracking, or focused on tracking, but using another data model.

The code that provides the general, underlying functionality of Mastodon is placed in the `org.mastodon` package. The code that is specific to the Mastodon application is placed in the `org.mastodon.mamut` package. This part is specific to a data model class `org.mastodon.mamut.model.Model`, based on `Spot` and `Link`. This model class is at the core of Mastodon data structure, and was surveyed in the [previous page](#). The classes specific to this application often have a Mamut in their name, as in the plugin interface we extend below.

Suggested package.

In this example we deal with a plugin specific to the Mastodon application, so we recommend using the prefix `org.mastodon.mamut` for its package. Then add as sub-package the part specific to the feature you implement.

The @Plugin annotation.

The `@Plugin` annotation you see in the line just above the class declaration is used by Mastodon to automatically discover your plugin at runtime. This is the medium of the SciJava discovery mechanism. If you do not add it, Mastodon will not find the plugin and it won’t be usable in the application. In the `type = XXX.class` you simply need to put the name of the current class.

1.18.3 The plugin interface methods.

The interface only defines 4 methods, with only the first one mandatory. But of course it is sensible to implement all of them to present a nice integration to the user.

1. Passing the data to the plugin:

```
public void setAppPluginModel( final ProjectModel projectModel )
```

2. Declaring to Mastodon what actions are contained in the plugin class:


```
public void installGlobalActions( final Actions actions )
```

3. Inserting the actions in the Mastodon menu:

```
public List< MenuItem > getMenuItems()
```

4. Fine-tuning the action names displayed in the menu.

```
public Map< String, String > getMenuTexts()
```

5. And finally, you can document for the user what each action does. This is not done through a method, but with a specific inner class:

```
@Plugin( type = Descriptions.class )
public static class Descriptions extends CommandDescriptionProvider
{
    ...
}
```

We will now detail each.

1.18.4 Passing the data to the plugin.

This is done with the method `setAppPluginModel()`. It is only used by Mastodon to pass the project model (all the important data) to the plugin. It is called only once, automatically after the image data and tracking data is loaded, and before the user interface is displayed.

The project model has all the important project data, window manager, and tools to execute or show something. The role of this method can be first to store it or pass it to the class that will actually perform the work. In this example, the class that will *do* something is a named action, defined below in the code. It is a named action that we can instantiate now, passing the project model in its constructor.

So the code for the first part of the class is the following:

```
package org.mastodon.mamut.example.plugin;

// All the imports.
...

@Plugin( type = MastodonPluginExample.class )
public class MastodonPluginExample implements MamutPlugin
{
    /**
     * The custom action. In our case, this is the piece of code that will run
     * when the user will click on the plugin menu item or use the plugin
     * shortcut.
     */
    private MyMastodonAction action;

    @Override
    public void setAppPluginModel( final ProjectModel projectModel )
    {
        this.action = new MyMastodonAction( projectModel );
    }
}
```

(continues on next page)

(continued from previous page)

```

}

...

```

1.18.5 The actual plugin logic.

The code for the action class `MyMastodonAction` can be defined in the same file. In this example we chose to implement `AbstractNamedAction`. It is a simple refinement of the core class `AbstractAction`, that stores a name for the action. We will see that the action name is very important, and it needs to be unique for each action.

This `MyMastodonAction` class is where you would put the logic of your action. Depending on whether it is about showing a dialog, starting an intensive computation or connecting to a server, the boilerplate code might change. Be mindful for instance about the fact that as is, this code will run on the `Event Dispatch Thread`.

In our case we don't mind, because the code we run is trivial. We want to count the number of spots currently in the model. For this we need to access some of the project model component.

```

private static class MyMastodonAction extends AbstractNamedAction
{

    private static final long serialVersionUID = 1L;

    private final ProjectModel projectModel;

    private MyMastodonAction( final ProjectModel projectModel )
    {
        /*
         * This is where we specify the action name. We would
         * normally recommend to put it in a String constant, at the
         * beginning of the plugin class.
         *
         * By convention, action names are all lower case, with words
         * separated by space. This name is what the user will see in the
         * Keymap table of the Preferences dialog.
         */
        super( "my super mastodon plugin example action" );

        /*
         * We simply store the project model instance.
         */
        this.projectModel = projectModel;
    }

    /*
     * The method below is called as many times as the user triggers the
     * action. The `ActionEvent` parameter is not really useful to our case.
     */
    @Override
    public void actionPerformed( final ActionEvent e )
    {
        /*
         * The data model is one component of the project model.

```

(continues on next page)

(continued from previous page)

```

        */
        final Model model = projectModel.getModel();

        /*
         * It contains the mathematical graph of all tracks.
         */
        final ModelGraph graph = model.getGraph();

        /*
         * The graph has vertices, which are the spots, or cells, in our
         * data.
         */
        final int nSpots = graph.vertices().size();

        /*
         * Let's show this to the user.
         */
        final Date now = new Date();
        final String dateTxt = new SimpleDateFormat( "YYYY-MM-dd HH:MM"
↪).format( now );
        final String message = "On " + dateTxt + ", there were " +
↪nSpots + " spots.";
        JOptionPane.showMessageDialog( null,
                                    message,
                                    "example Mastodon plugin",
                                    JOptionPane.INFORMATION_MESSAGE,
                                    MastodonIcons.MASTODON_ICON_MEDIUM );
    }
}

```

1.18.6 Declaring the actions to Mastodon.

This is well and fine, but how are we going to let the user *launch* anything? We want to have something done when the user triggers the plugin. How can we do that? This is done using the *actions* mechanism, used in our case with the following method:

```
public void installGlobalActions( final Actions actions )
```

When a plugin is registered, it receives via the method above an `Actions` instance, reserved for plugins. We will see how to use it, but these actions are simple mechanism to bind a shortcut or a menu item to something that runs.

But first a word on keyboard shortcut. Mastodon has a nice system to map actions to keyboard shortcuts and the let the user modifies them. If you want to trigger your plugin via keyboard, you need to provide a default shortcut in the method we detail below.

The syntax to specify a keyboard shortcut is simple, but has some gotchas about the accelerators and letters. It is a good idea to check the supported syntax here:

<https://github.com/scijava/ui-behaviour/wiki/InputTrigger-syntax>

If you do not want your action to be callable directly from the keyboard, enter `not mapped` instead. If you want your action to be callable by *several* shortcuts, enter them as an array of strings, as in:

```
String[] keyboardShortcuts = new String[] { "ctrl shift P", "ctrl shift Q" };
```

As coding style extremists we would recommend declaring the shortcut in a `String` constant declared in the beginning of the class, so that it can be easily modified later.

We can declare our custom action we instantiated above in the following way:

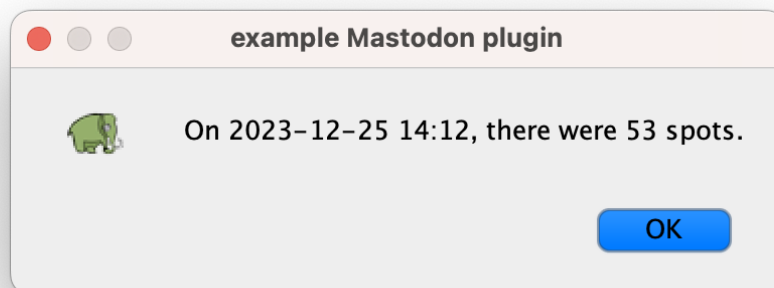
```
actions.namedAction( theActionInstance, "the keyboard shortcut" );
```

Note that this method only required the named action instance and the default keyboard shortcut (could be "not mapped", but not null, nor something empty). The action name will be taken directly from the `AbstractNamedAction` implementation, as seen in the previous paragraph.

In the end, the content of the method looks like this:

```
@Override
public void installGlobalActions( final Actions actions )
{
    final String keyboardShortcut = "ctrl shift P";
    actions.namedAction( action, keyboardShortcut );
}
```

Alright this is done! Now when the user will press the `ctrl shift P` keys, the `actionPerformed()` code of the `MyMastodonAction` above will run.



Before we carry on: did you recognize you could declare *several* action within this plugin class? In this example, we just made one, called `MyMastodonAction`, that is defined and instantiated above. Nothing prevents your from having several implementations of `AbstractNamedAction`, each corresponding to a different action. You can repeat the call to

```
actions.namedAction( action, keyboardShortcut );
```

with each of them, and have several actions declared in the same plugin class. **A Mastodon plugin mainly contains the logic for the declaration of one or several actions to be run, rather than the action itself.**

1.18.7 The action descriptions.

We could stop here.

We have a plugin class, that instantiate an action, declares it into the Actions bundle, and binds it to a specific shortcut. We have some code and a way for the user to trigger it.

But 1/ it does not have a description and 2/ what if you want to put this action into the Mastodon menu? Let's address the first point.

Mastodon lets the user redefine keyboard shortcuts as they please. This is done in the Preferences dialog, in the Keymap section. This page shows a description for each action and this description is specified with the mechanism below. This is not mandatory, but Mastodon will complain if it cannot find a description for the new actions you create, so ...

To inform Mastodon of descriptions, we basically extends a specific class, and annotate it so that it can be discovered at runtime. This is exactly the same discovery mechanism used in the plugin above, except that it is a different type, hence, usage. Because the descriptions live in a specific class, they could be in a separate file. But it is more convenient this way. So we can add the following piece of code to our MastodonPluginExample:

```
@Plugin( type = Descriptions.class )
public static class Descriptions extends CommandDescriptionProvider
{
    public Descriptions()
    {
        /*
         * A quick word on scopes and contexts for keyboard shortcuts. If
         * you do not know nor care, simply use the same syntax as below.
         *
         * It is possible that there are several applications in the same
         * process that use the same plugin mechanism. For instance the
         * BigDataViewer and Mastodon, both in Fiji. To sort descriptions
         * and shortcuts according to the right scope and context, we
        ↪need
         * to declare our scope and context in the constructor.
         *
        ↪use
         * The scope relates to the application itself. In our case we
         * `KeyConfigScopes.MAMUT` and it will be the same for all your
         * Mastodon plugins.
         *
        ↪shortcut
         * The context relates to what part of the application the
         * is used in. For instance you could define an action that only
         * runs within the TrackScheme view. In that case you will use
        ↪the
         * `KeyConfigContexts.TRACKSCHEME` context.
         *
        ↪use
         * To signal that the action can be run everywhere in Mastodon,
         * the global `KeyConfigContexts.MASTODON` context.
         */
        super( KeyConfigScopes.MAMUT, KeyConfigContexts.MASTODON );
    }

    @Override
```

(continues on next page)

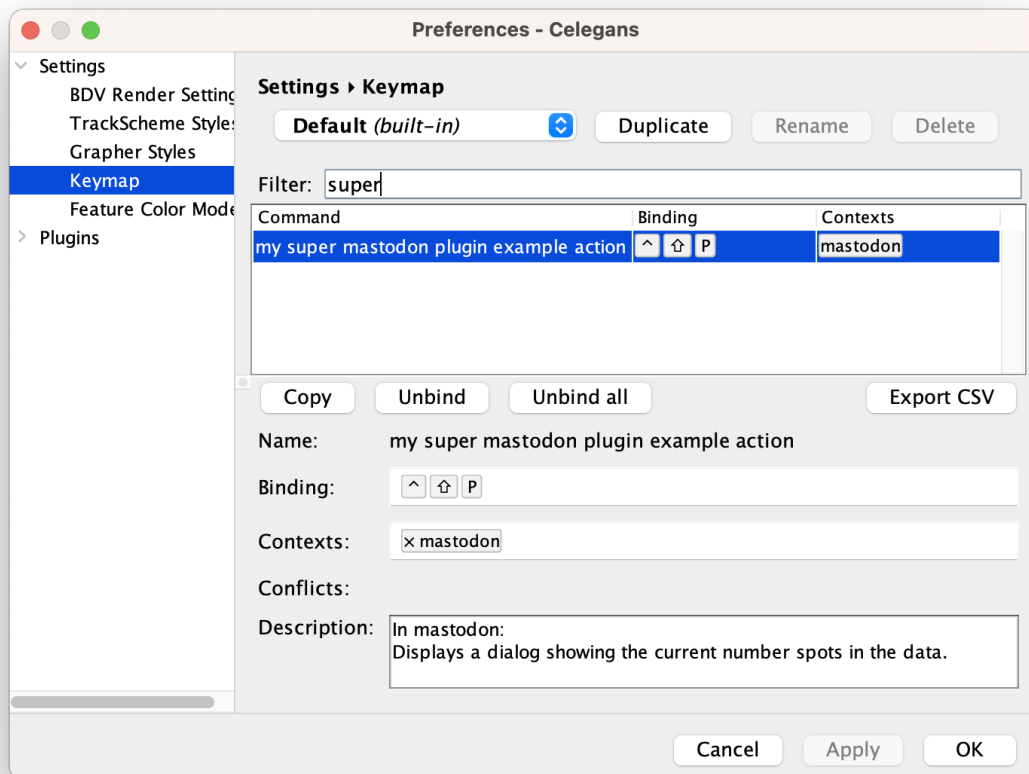
```

    public void getCommandDescriptions( final CommandDescriptions_
↳descriptions )
    {
        /*
        * We have to specify the action name and its default shortcut.↳
↳This
        * is why it would have been a good idea to specify them in a
        * constant. Now we need to repeat them here.
        */
        final String actionName = "my super mastodon plugin example_
↳action";
        /*
        * We must give an array of shortcuts, even if we only use one.
        */
        final String[] keyboardShortcut = new String[] { "ctrl shift P" }
↳;
        /*
        * The description itself. Make it user-friendly.
        */
        final String description = "Displays a dialog showing the_
↳current number "
                                + "spots in the data.";
        /*
        * And the method to pass the description to Mastodon. Notice it_
↳is
        * somewhat parallel to the way we declare the actions in the
        * `installGlobalActions()` method above.
        */
        descriptions.add( actionName, keyboardShortcut, description );

        /*
        * This is enough to make the action appear in the Preferences
        * dialog, in the Keymap section. For instance if you search for
        * 'example', the action name should appear in the 'Command' column,
        * with the key binding we defined.
        */

        /*
        * Again, if you have more than one action declared in this_
↳plugin
        * class, simply repeat the method call above for each.
        */
    }
}

```



1.18.8 Inserting actions in the menus of Mastodon.

Let's now see how to insert the action in the Mastodon menu. This is done via two methods, the first one to link a menu item to an action, and the second one to give the menu item a user-friendly name.

Here is the method to create a menu item. We will add one that calls to our action there.

```
@Override
public List< MenuItem > getMenuItems()
{
    /*
     * To create the menu item, we use a utility method, that accepts the
     * action name, and a menu path as a list of strings. To have the action
     * executed when the menu item is clicked, it is enough to simply enter
     * the action name.
     *
     * The following will put the action with the name: 'my super mastodon
     * plugin example action' in the menu 'Plugins > Examples'.
     *
     * The name used in this method must match the name of the action, hence
     * we kindly remind how adequate it would have been to declare it as a
     * constant at the top of this class.
     */
}
```

(continues on next page)

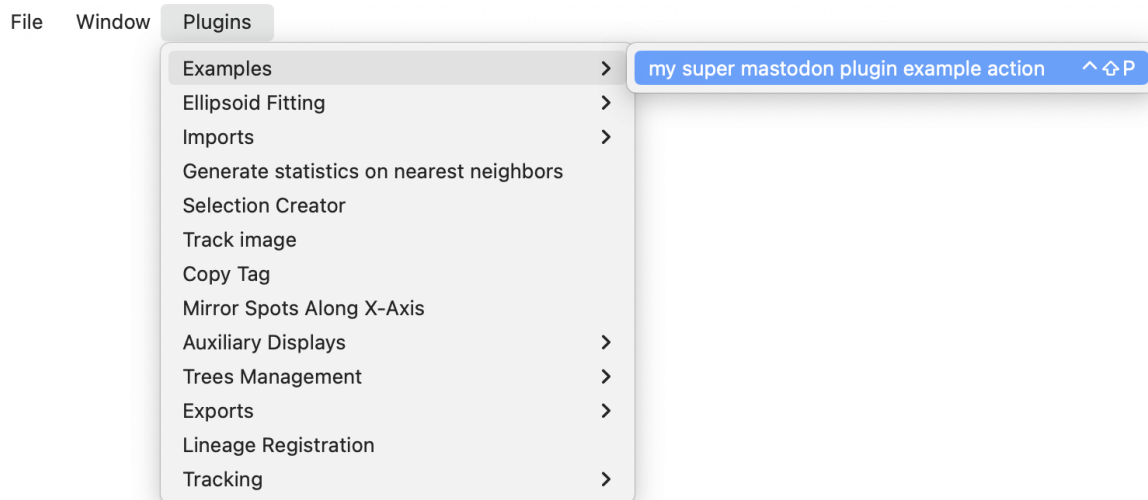
(continued from previous page)

```

    */
    final String actionName = "my super mastodon plugin example action";
    final MenuItem menuItem = MamutMenuBuilder.makeFullMenuItem(
        actionName,
        "Plugins", "Examples" );

    /*
     * And now we can return this menu item in a list of 1 element. And if
     * you had more than one action declared in this plugin, you can create
     * several menu items for them, and return them in this method.
     */
    return Collections.singletonList( menuItem );
}

```



This is great and well, but not pretty. The menu item in Mastodon is at the right place and does the right thing, but is named `my super mastodon plugin example action`, which is the name of the action. The last method we will see allows to customize the name of the menu item, and put something more meaningful to the user.

```

@Override
public Map< String, String > getMenuTexts()
{
    /*
     * This method returns a map String -> String, in which you can put the
     * action name as key, and the desired text as value. For instance:
     */
    final String actionName = "my super mastodon plugin example action";
    return Collections.singletonMap(
        actionName,
        "Show N spots" );

    /*
     * I will refrain from saying something about declaring the action name
     * as constant in the top of this class for convenience. But now you

```

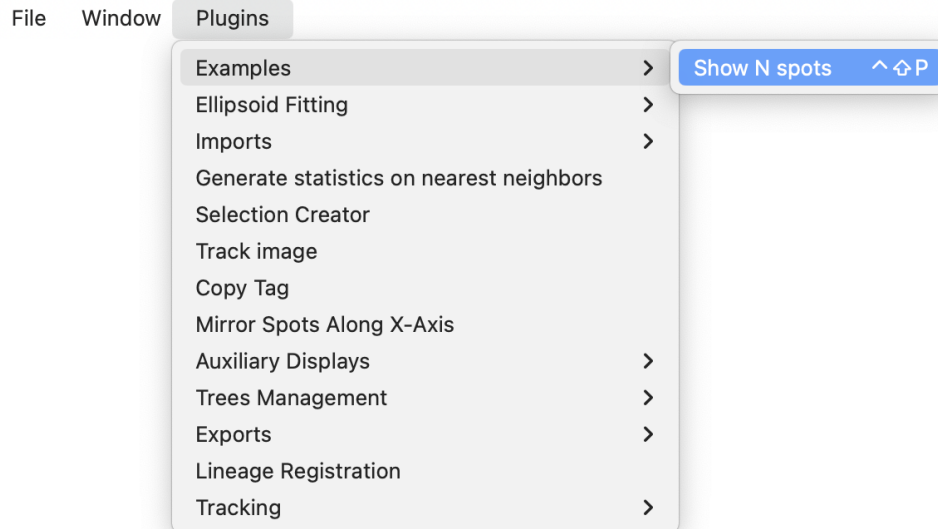
(continues on next page)

(continued from previous page)

```

    * should have in the 'Plugins > Examples' menu an item called 'Show N
    * spots', and that launches the action.
    */
}

```



1.19 Custom simple numerical features in Mastodon.

In this example we show an example of a simple custom numerical feature in Mastodon. We take the simple example of returning the angle of a link in the XY plane. The code corresponding to this part of the documentation is in the same repository as the previous tutorial, in the package:

<https://github.com/mastodon-sc/mastodon-plugin-example/blob/main/src/main/java/org/mastodon/mamut/example/feature/>

1.19.1 The versatility and verbosity of the numerical feature system in Mastodon.

As an introduction, we would like to give some context about the feature system in Mastodon. This is unrelated to the programming tutorial, but maybe useful to understand the complexity we will deal with, and apologize a bit for it. [TrackMate](#) and [MaMuT](#) - our previous works and the most similar to Mastodon - were focused on tracking. But we already had recognized the utility of 'numerical features' for basic analysis and exploration, and the ability to extend analysis by creating and adding new ones to these platforms. In [TrackMate](#) and [MaMuT](#), the numerical feature system has some strong limitations: you can only have features that return a single real number for a data item (spot or link). In Mastodon we wanted to remove this limitation and have features returning any kind of objects, numbers, matrices or text. We also wanted to avoid enforcing a specific implementation. Finally, we wanted to have for Mastodon a reasonable approach when dealing with numerical feature for a very large number of data items. The price to pay for this generality is the length and verbosity of the code required to implement a feature. Now that we want to implement our own feature in Mastodon, we have to pay this price.

Somewhat fortunately, we also recognized this difficulty, and created a few facilities to accelerate feature development for certain cases. For instance, in this example we want to develop a simple feature, that returns the angle of a link in a XY plane, giving the instantaneous direction of displacement for a cell. For this we simply need a feature that

returns a scalar real number for each link. We don't even need to store them, and can compute it on the fly, avoiding the need to create a feature computer and storing the values it returns. We will start with this simple case, that allows for introducing the main concepts of the numerical feature system.

Nonetheless, even our simple example requires two classes to be functional:

1. A class to store feature values and decompose it, the `Feature`,
2. And a class to compute the feature values, the `FeatureComputer`. We will detail them one after the other.

1.19.2 The feature class.

The starting point is the feature class `LinkXYAngleExampleFeature.java` and we will start from there. As for the plugin example this class is heavily commented but we repeat and extend these comments here.

The beginning.

A Mastodon feature is class that associates a value (numerical or not) to a data item (a spot or a link). Such classes implement the `org.mastodon.feature.Feature` interface, with a type that specifies what data item they are defined for. Because we want to compute something on links, our feature implements `Feature < Link >`. We will see below the methods it specifies. So the header of our feature class is:

```
package org.mastodon.mamut.example.feature;

...
// All the imports.

public class LinkXYAngleExampleFeature implements Feature< Link >
{
```

We start by defining a few constants, that are used to declare the 'signature' or specifications of our feature: its name, what type of values it returns, what is their units and dimension, etc.

Our feature needs a name, and we will use it repeatedly. So we declare it now in a constant. The feature name also serves as a unique key, hence the name of the constant. It is a good idea to use something meaningful to the users.

```
private static final String KEY = "Link angle in XY plane";
```

We also have a means to pass some information to the user about what a feature is, and we also store this in a constant.

```
public static final String INFO_STRING = "Example feature that computes "
    + " the angle of a link in the XY plane.";
```

The role of feature projections.

Before going on, we need to speak a little bit about the notion of *projections*.

Since a feature can return any type of data, we needed a way to make use of them in Mastodon in some specific situations, for instance to display feature values in a table, or use them in a color mode. For this we use *projections*. A feature can return a value of any type, but we ask it to be 'decomposable' into scalar, real projections. A feature projection returns a scalar real number for each data item, and a feature can be expressed as a collection of projections. What we see in the Mastodon tables and what we use in color modes are the projections. Of course, it is only useful for numerical features (number, matrices). Other feature types are specific enough to prompt for their own display. In the examples of this repository, we only deal with numerical features.

The way a numerical feature is decomposed into projection is completely up to you. The decomposition does not have to be complete, and it can be redundant. For instance, here we will just compute the angle of a link in the XY plane, but in 3D it requires 2 scalar angles. In such a feature, you could use the polar angle and the azimuthal angle as projections. Or the x, y and z coordinates a unit vector along the link. Or even return the 2 angles and the 3 components. To choose the projections you need to think of the most direct way to use it in your track analysis pipeline.

In concrete implementations, we therefore need to give the specifications of the projections we will use, and pass these projection specifications to the feature specifications (read below). In our case we just have one, and projection specifications just need a projection name, and the dimension of the scalar value they return (in our case, an angle). This is done with the `FeatureProjectionSpec` class, as below. Our feature is made of one real value, so we use only one projection. For the projection name, we can reuse the feature name, since we have only one projection. This part is then just:

```
public static final FeatureProjectionSpec PROJECTION_SPEC
    = new FeatureProjectionSpec( KEY, Dimension.ANGLE );
```

The projection itself will be defined later in the code.

Feature specifications.

In Mastodon, features are discovered and integrated via their specification class. This specification is supposed to return basic information about the feature content, before it is computed. Mastodon then uses it to sort features depending on what they do.

To be integrated into Mastodon, each feature must implement and return a specification class, that inherits from `FeatureSpec< F, O >` where F is the concrete feature class and O is the type of the data item the feature is defined for. This can be done as below, using an inner public static class.

And as for the example plugin we have seen, we rely on the SciJava plugin mechanism for automatic runtime discovery. Concretely this means you need to add the `@Plugin(type = FeatureSpec.class)` line above the class declaration for Mastodon to discover and integrate your class.

```
@Plugin( type = FeatureSpec.class )
public static class Spec extends FeatureSpec< LinkXYAngleExampleFeature, Link >
{
    public Spec()
    {
        super(
            KEY, // 1. The feature name.
            INFO_STRING, // 2. The feature info.
            LinkXYAngleExampleFeature.class, // 3. The feature class.
            Link.class, // 4. The class of the data item.
            Multiplicity.SINGLE, // 5. The multiplicity.
            PROJECTION_SPEC ); // 6... The list of projection specs.
    }
}
```

The specification class itself just need to pass a few arguments via the super constructor. In order:

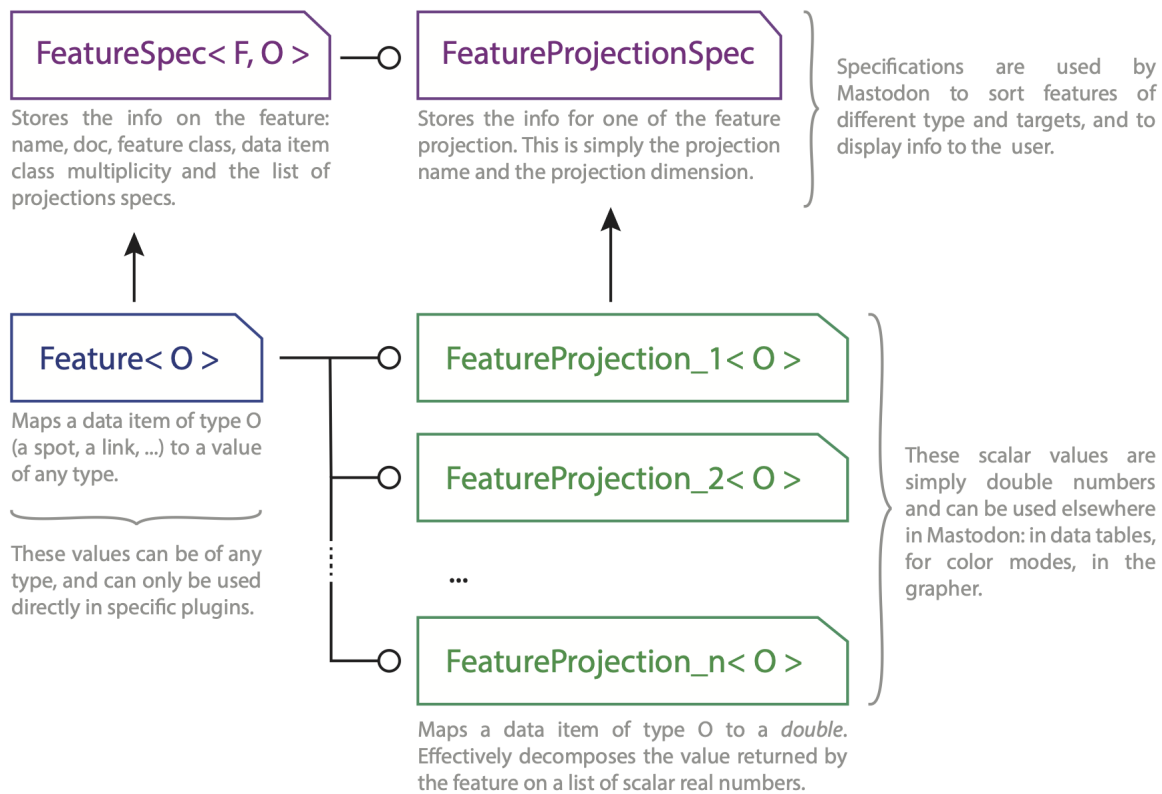
1. The name or key of the feature, which must be unique. By convention we choose user-friendly name, starting with the data item the feature is defined for, such as 'Link angle in XY plane'.
2. The information for users, as a String.
3. The class of the feature. In our case it will just be `LinkXYAngleExampleFeature.class`.
4. The class of the data item the feature is defined for. In our case, `Link.class`.

5. The multiplicity. The multiplicity is an enum that specifies whether the feature is defined once or twice per channels in the image, or is independent from the the number of channels in the image. For instance, for features the measure the mean intensity, we would use `Multiplicity.ON_SOURCES`, to signal that the feature needs to be computed once per channel. In our case, the feature is independent from the image data and the number of channels it contains. For this we use the `Multiplicity.SINGLE` flag.
6. A series or list of feature projection spec, one per projection defined in the feature. Since we have a scalar feature, we will just put the one projection spec we created.

The next part of the code simply creates, stores and returns an instance of this feature specification class.

```
// Now we instantiate and keep an instance of this feature spec class.
public static final Spec SPEC = new Spec();

// And we return it in this method, specified by the 'Feature<O>' interface.
@Override
public Spec getSpec()
{
    return SPEC;
}
```



Implementing the feature projection.

Now we can code and instantiate the projection. Here is the feature projection class. It implements `FeatureProjection< Link >` that specifies a bunch of methods related to getting a scalar value for each link. Because it is compact, we chose to put it in an inner class within the feature class.

In our case, since we do the computation on the fly, this is where the computation logic is.

```
private static final class MyProjection implements FeatureProjection< Link >
{
    /**
     * This should return a key for 'projection'. We use a special class for
     * this, used to facilitate manipulating a collection of projections
     * elsewhere in Mastodon.
     */
    @Override
    public FeatureProjectionKey getKey()
    {
        return FeatureProjectionKey.key( PROJECTION_SPEC );
    }

    /**
     * This method below is used to indicate to Mastodon that a value exist
     * for the specified data item, and *that this value is valid*. In our
     * case, because we compute values on the fly, they always exist and are
     * always valid.
     */
    @Override
    public boolean isSet( final Link link )
    {
        return true; // Always set, always valid.
    }

    // The logic that computes the value.
    @Override
    public double value( final Link link )
    {
        // The source of the link (where it starts from).
        final Spot source = link.getSource();
        final double xs = source.getDoublePosition( 0 );
        final double ys = source.getDoublePosition( 1 );

        // Its target (where it points to).
        final Spot target = link.getTarget();
        final double xt = target.getDoublePosition( 0 );
        final double yt = target.getDoublePosition( 1 );

        final double dx = xt - xs;
        final double dy = yt - ys;

        return Math.atan2( dy, dx );
    }
}
```

(continues on next page)

(continued from previous page)

```

        // The units.
        @Override
        public String units()
        {
            return Dimension.RADIANS_UNITS;
        }
    }

```

Back in the feature class, we need a field to store the instance:

```

/*
 * This is the instance of the projection we will use to return the scalar
 * real values. It is defined below and instantiated in the feature constructor.
 */
private final MyProjection projection;

```

And when we instantiate the projection in the feature class constructor:

```

public LinkXYAngleExampleFeature()
{
    this.projection = new MyProjection();
}

```

This is it for the computation. In our case it is very simple. It is somewhat daunting that such a simple computation requires that many boilerplate code and specifications to be integrated in Mastodon. Again, the feature system ambitions to address a very wide range of situations and means of computations. This generality is the source of this verbosity.

Invalidating values.

The next method in the `Feature` interface is important for features that *store* values. These features have a ‘computer’ class that handles the computation logic. The computer class is called by the user (in the ‘compute features’ dialog), and when after it runs, the feature instance is created. The feature instance is then just used to store and return values.

But additionally, Mastodon has a mechanism to invalidate the values of the data items that are modified. Let’s take for example the feature that stores the mean intensity in a spot. If the user moves a spot to another position, or change its radius, the mean intensity value stored for this spot becomes invalid. When a user modifies a data item in such a way, Mastodon automatically calls the method below, which is meant to *discard* the value the feature stores.

For our example where the computation is done on the fly, we don’t have to do anything.

```

@Override
public void invalidate( final Link link )
{}

```

Returning projections.

The rest of the feature class is dedicated to returning the projections contained in the feature.

```

    /*
     * Here we should return the one projection we have if the specified
     * projection key matches., or 'null' otherwise.
     */
    @Override
    public FeatureProjection< Link > project( final FeatureProjectionKey key )
    {
        if ( projection.getKey().equals( key ) )
            return projection;

        return null;
    }

    /*
     * Here we should return all the projections defined in the feature.
     */
    @Override
    public Set< FeatureProjection< Link > > projections()
    {
        return Collections.singleton( projection );
    }

```

We could think our work is done. Alas! We still need to give Mastodon a way to create the feature. And this is done in the feature computer. So the rest of the tutorial is in the `LinkXYAngleExampleFeatureComputer` class, and here you will see that it is the class that calls the constructor of this feature.

1.19.3 The feature computer class.

We now describe the `LinkXYAngleExampleFeatureComputer`.

This class is part of the first tutorial on writing your own numerical feature for Mastodon. It discusses the computer logic, which will be very simple in our example.

The separation of concerns in Mastodon is quite strong. In the previous class, we have seen the classes that are in charge of *storing* the values of a feature. Though in our example it also contained the logic to compute these values, as we wanted to have a ‘on the fly’ feature. Normally, the logic to compute a feature should be implemented in a feature computer. Such a class is the one discovered by Mastodon, and is in charge of creating the feature and adding values to it. It is normally the important one. This added verbosity and multiplication of classes gave us a wide berth as to how features are defined and calculated. But again, we pay the price now.

However this class will be quite brief. Because for the XY angle the computation is done in the feature class, we just have to instantiate it. To return the feature class to Mastodon, and receive the data required, a feature computer uses the SciJava modifiable private field, annotated with the `@Parameter` tag, that we will explain below. If you never met this technique before in Java, it will feel strange.

The MamutFeatureComputer interface.

The feature computer class itself needs to implement the `MamutFeatureComputer` interface. As for the plugin system, it has a Mamut in its name to indicate that it is specific to the cell lineaging application. And as for the plugin system, you need to annotate it by add the `@Plugin(type = MamutFeatureComputer.class)` line before the class declaration, so that Mastodon can discover the computer at runtime.

```
@Plugin( type = MamutFeatureComputer.class )
public class LinkXYAngleExampleFeatureComputer implements MamutFeatureComputer
{
```

Specifying the output and inputs of the computer.

Let's start immediately with the part that might feel new or strange or like dark magic. A feature computer is expected to receive a set of parameters, such as the image, the track data, etc., and return an output, the feature it computed. But it does not have methods to do so. Everything is done via annotated private field, annotated with a special tag. This system comes from the SciJava core library, and is meant to simplify developing code that process things.

First, we will specify the output of this computer. This is the feature instance we will create. It will be stored in a private field, and we add the `@Parameter` tag to signal Mastodon to look into this field. And we add the extra parameter `type = ItemIO.OUTPUT` to signal that this is the output that Mastodon should take after computation.

Thanks to SciJava, Mastodon will be able to read this field, even if it is private. But very importantly: for this to work, the field must *not* be final. If you get a stange error related to it, this is most of the time because of this.

```
@Parameter( type = ItemIO.OUTPUT )
private LinkXYAngleExampleFeature feature;
```

Second, we can declare the inputs we need in the exact same way. Mastodon will inspect the class and automatically set the input fields you declared with the adequate values. Of course, you have a limited choice in the list of fields you can add to your computer, but they should cover all cases. Our simple example does not need anything, so there is nothing there. The list of possible fields is documented elsewhere.

The computer interface methods.

The FeatureComputer interface has only two methods.

```
/*
 * In this method, the actual feature instance is typically instantiated,
 * without any values.
 */
@Override
public void createOutput()
{
    feature = new LinkXYAngleExampleFeature();
}

/*
 * This method is called after '#createOuput()'. This is where the
 * computation logic should happen, storing the computed values in the
 * feature instance. In our simple case, we have nothing to do.
 */
@Override
```

(continues on next page)

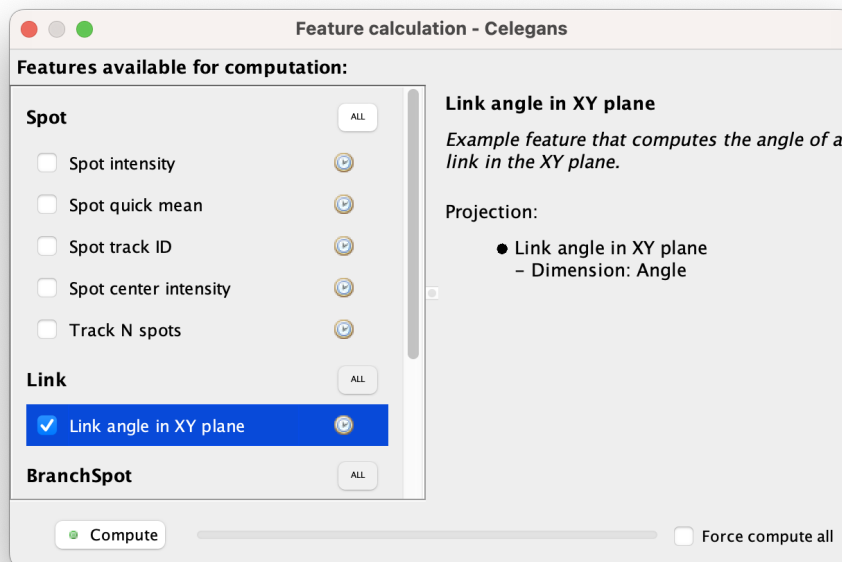
(continued from previous page)

```
public void run()
{}
```

That's it! With this, a new feature should appear in the feature computation dialog. If you check it and trigger a computation, the new feature will appear and be usable throughout Mastodon.

1.19.4 The results.

If you compile this repository and drop it in Fiji, the feature should automatically appear in Mastodon. First in the feature computation dialog:



And after clicking on the compute button, the values will be listed in the data table:

1

2

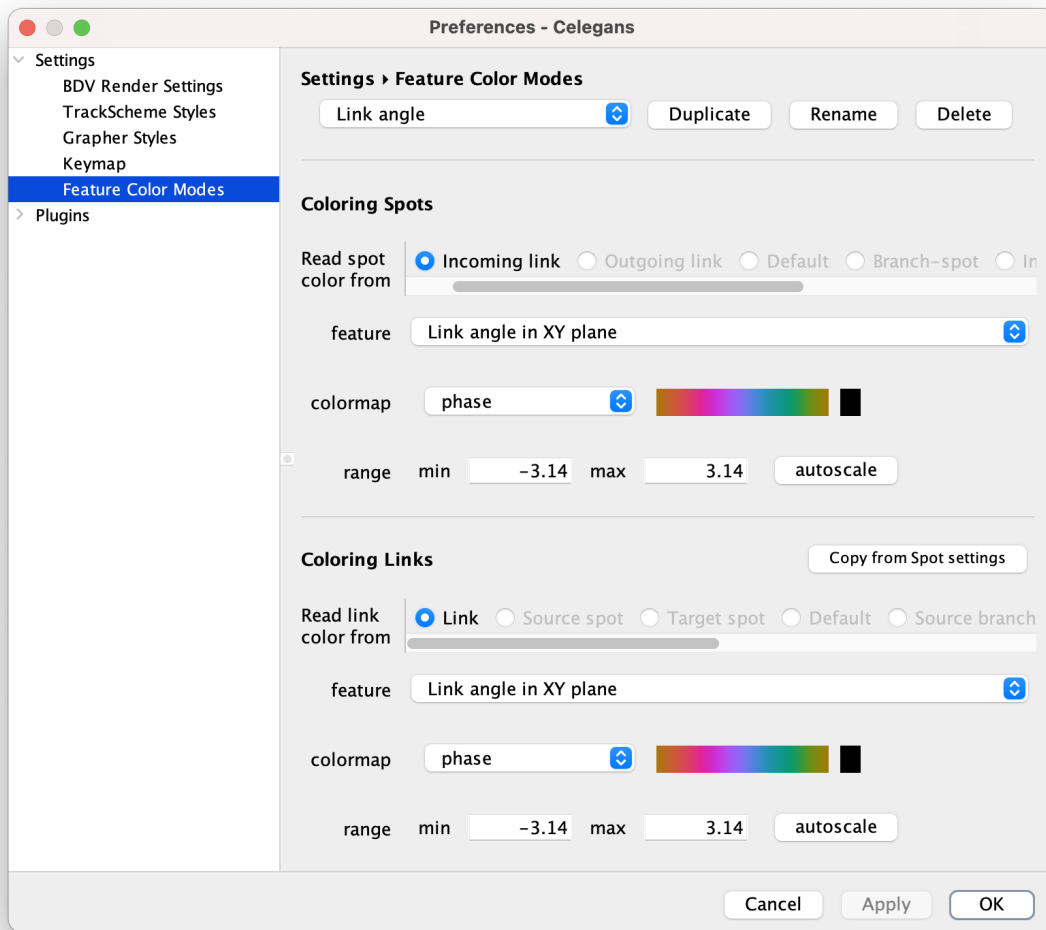
3

context: full graph

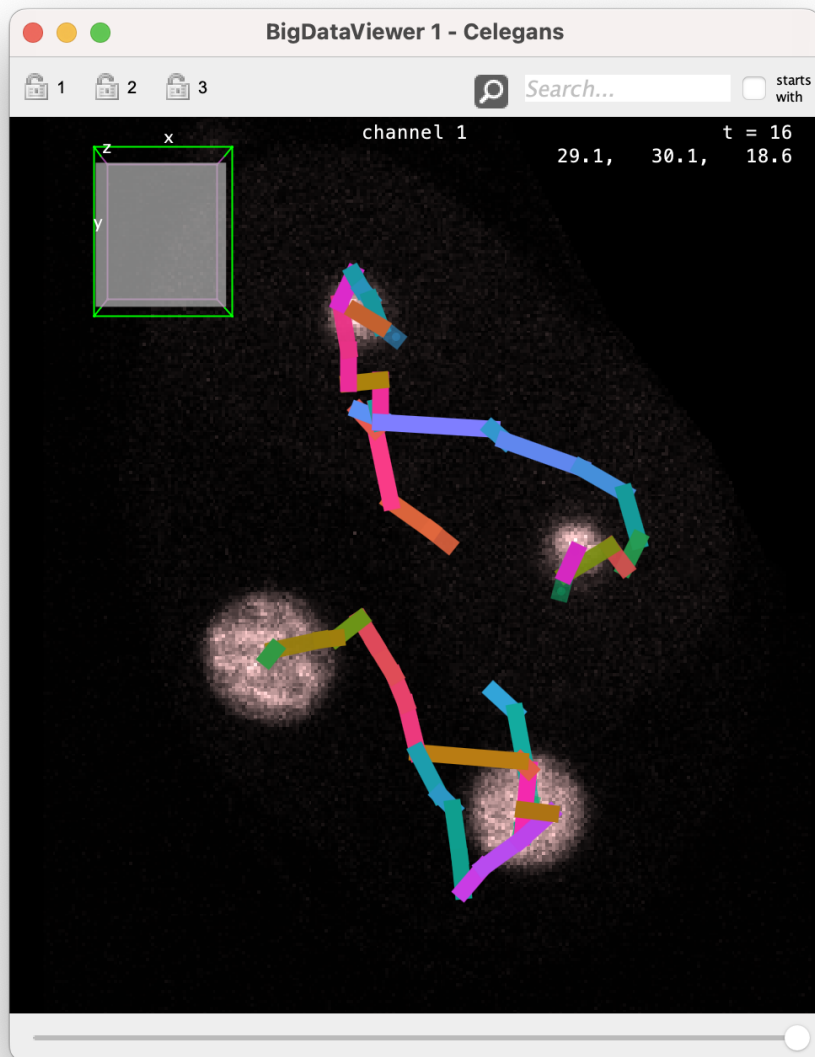
Search...

Spot	Label	ID	Link angle in XY plane (Radians)	Link delt...	Li
Link	0 → 1	0	0.781	1	
	1 → 2	1	1.385	1	
	3 → 4	2	-2.456	1	
	4 → 5	3	-2.518	1	
	5 → 6	4	-1.785	1	
	6 → 7	5	-2.289	1	
	7 → 8	6	0.388	1	
BranchSpot	8 → 9	7	-1.736	1	

The feature values can be used in a color mode to follow cell motion direction over time.



And in the C.elegans dataset, it looks like this:



Because in this case the values are computed on the fly, the resulting link color will be updated live when you move a spot around, as exemplified in the video below.

1.20 Mastodon numerical features.

This chapter describes how the feature values currently available in Mastodon are calculated. We also gives information about their dimension, units, *etc.*

1.20.1 Feature dimensions.

In Mastodon, feature values are expressed when possible in physical units. Each feature projection as a *dimension* (in the physics meaning) from which we compute the *units* of the values. For instance, if a feature value as the dimension LENGTH and the spatial units is m, then the values of this feature will be in m. Mastodon only has physical units for space. For time, the frame interval is always equal to the dummy 1 frame. This is why you will find all units involving time expressed in frames. Each feature projection has a dimension, and the features report what are the dimension of their projections in the feature computation dialog. The table below lists all the feature dimensions currently supported in Mastodon, and give examples of the derived units when the spatial units are m.

Di- men- sion	Nam	Ex- am- ple units	Description
NONE	None	∅	Used for dimensionless quantities, such as frame position, number of things, <i>etc</i>
LENGTH	Leng	μm	For quantities about the length of objects. For instance radius or distance between objects.
POSITIO	Po- si- tion	μm	Dimension for feature that report a position. Different from LENGTH so that for objects with small lengths at large positions, quantities are plotted separately.
TIME	Time	frame	For quantities that report a delay, a duration or the timing of an event. Because Mastodon does not deal with physical units for time, quantities formed with the time dimension always use the frame unit.
VELOCI	Ve- loc- ity	μm/frame	For quantities that report a speed or a velocity.
RATE	Rate	/frame	For quantities that report a change per units of time.
ANGLE	An- gle	Ra- di- ans	Measures of angles. In Mastodon, all angles are in radians.
STRING	NA	∅	For non-numeric features.
INTENS	In- ten- sity	Counts	For quantities based on pixel values. For instance the mean intensity within a spot.
INTENS	Inten	Counts	For quantities based on pixel intensity squared. For instance the variance of the mean within a spot.
QUALIT	Qual- ity	∅	This dimension is used by spot detectors. There is a special feature called Detection quality, that stores for each spot they detect automatically a measure of quality or confidence in their detection.
COST	Cost	∅	This dimension is used by spot linking algorithms. There is a special feature called Link cost used in the estimation phase. It stores for each link the cost that the linker computes for it in the estimation phase. These costs are then used in the association phase to retrieve the best set of links.

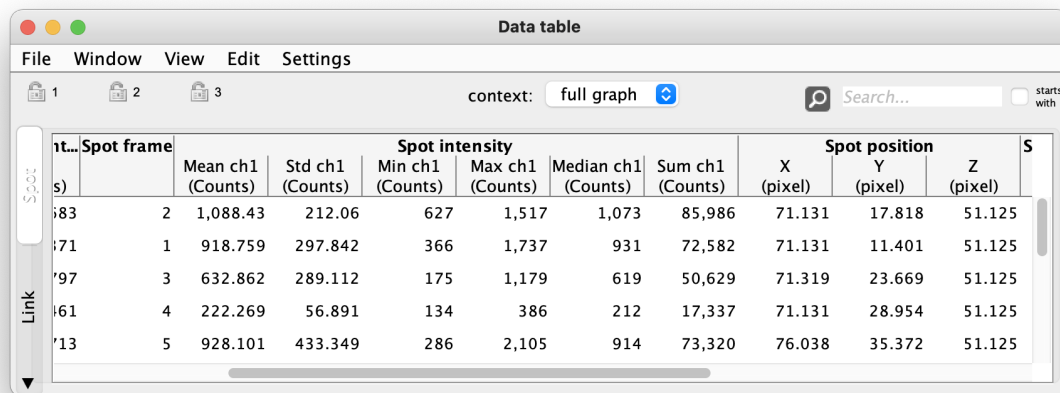
1.20.2 Spot features.

Spot intensity.

This feature has six projections per channel:

- Mean
- Std
- Min
- Max
- Median
- Sum

These 6 projections are multiplied by the number of channels or sources in the image. You will find the projection names appended by *ch 1*, *ch 2*, *etc*, as exemplified below on an image with one channel.



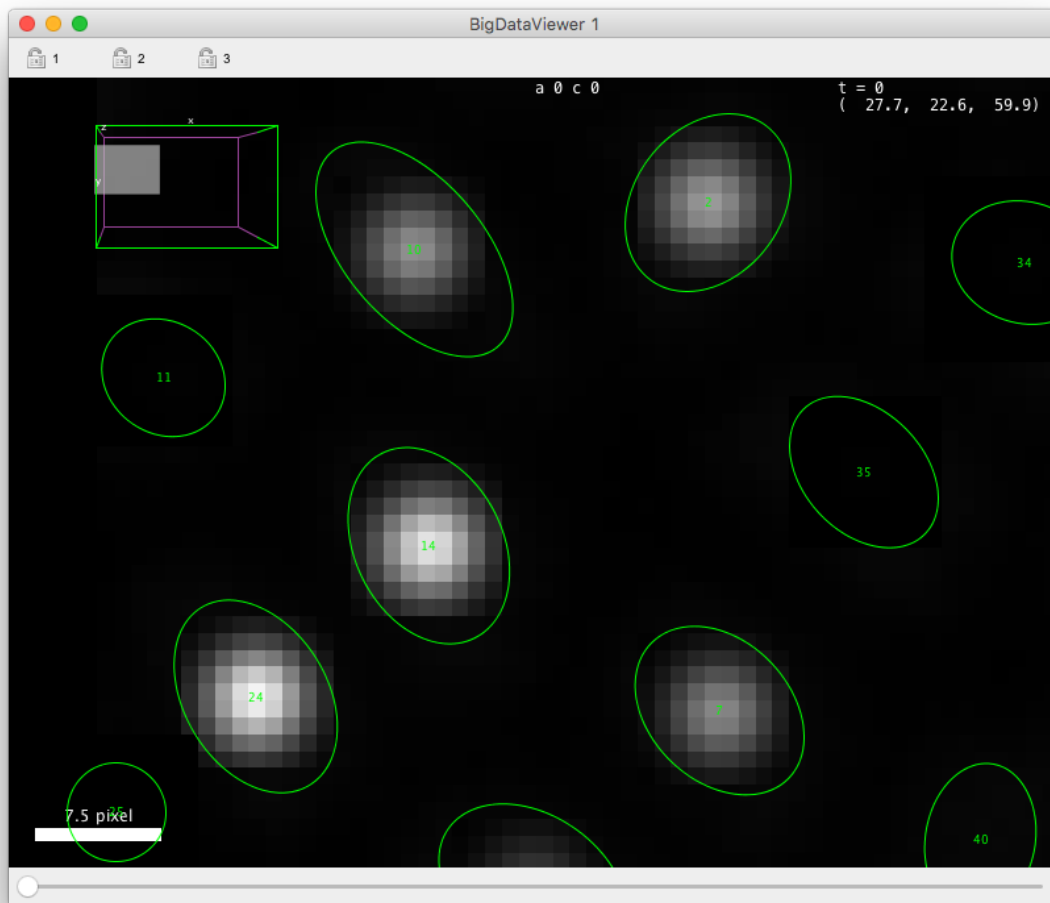
The screenshot shows a 'Data table' window with a menu bar (File, Window, View, Edit, Settings) and a toolbar with icons for file operations, a context menu set to 'full graph', a search bar, and a 'starts with' checkbox. The table has columns for 'Spot frame', 'Spot intensity' (Mean ch1, Std ch1, Min ch1, Max ch1, Median ch1, Sum ch1), 'Spot position' (X, Y, Z), and 'S'. The data is as follows:

Spot frame	Mean ch1 (Counts)	Std ch1 (Counts)	Min ch1 (Counts)	Max ch1 (Counts)	Median ch1 (Counts)	Sum ch1 (Counts)	X (pixel)	Y (pixel)	Z (pixel)	S
83	1,088.43	212.06	627	1,517	1,073	85,986	71.131	17.818	51.125	
71	918.759	297.842	366	1,737	931	72,582	71.131	11.401	51.125	
97	632.862	289.112	175	1,179	619	50,629	71.319	23.669	51.125	
61	222.269	56.891	134	386	212	17,337	71.131	28.954	51.125	
13	928.101	433.349	286	2,105	914	73,320	76.038	35.372	51.125	

The values are floating point numbers, with the dimension **INTENSITY**. These projections give the mean, max, min, ... intensity at over all the pixels inside the spot ellipsoid. Like all features related to measuring pixel intensity, they take a long time to compute over large models.

1.20.3 Spot center intensity.

Computes the intensity at the center of spots by taking the mean of pixel intensity weighted by a gaussian. The gaussian weights are centered on the spot, and have a sigma value equal to the minimal radius of the ellipsoid divided by 2. The image below illustrates how these weights look like inside spots and what pixels contribute the average reported by this feature.



1.20.4 Spot quick mean.

This feature compute the mean intensity of the pixel inside spots using the highest level in the scale pyramid (lowest resolution) to speedup calculation. It exists to offer a quiker way of reporting spot intensity for large models when multiple resolution level exist in the source image. It is recommended to use the ‘Spot intensity’ feature described above when the best accuracy is required

1.20.5 Other spot features.

Feature name	Projections	Description
Spot frame	<i>idem</i>	The spot frame.
Spot N links	<i>idem</i>	The total number of links, incoming and outgoing, of the spot.
Spot position	X & Y & Z	The spot center position, in physical units.
Spot radius	<i>idem</i>	The spot radius in physical units. For spots that are ellipsoids, returns a radius using the geometric mean of the spot ellipsoid radiuses. This approximation is such that the sphere with the reported radius and the spot ellipsoid have the same volume.
Spot track ID	<i>idem</i>	The ID of the track the spot belongs to. Track IDs are positive integer numbers starting from 0.

1.20.6 Link features.

Feature name	Projections	Description
Link target IDs	Source spot id & Target spot id	Stores the IDs of the two spots the link connects to. In Mastodon, the links are oriented: the source and target are not equivalent. By convention in Mastodon, the source spot is always the first in time, and the target the last in time.
Link displacement	<i>idem</i>	The distance between the source and target spots of the links, in physical units.
Link velocity	<i>idem</i>	The velocity at the time of the link. It is calculated as the link displacement divided by the frame interval between the source and target spots (in frame units).

1.20.7 Track features.

'Track' is the vocable we use in Mastodon for the weakly connected components of the graph. A track is made of all the links and spots that can be reached by jumping across links in any direction. In a lineage, a track corresponds to a single cell and all its daughters, grand-daughters, etc. Track features are value that are defined for a whole track. An example would be the number of spots in a track. In Mastodon, there is no special place to store track feature values. Track feature values are stored in the spots of the tracks, and listed in spot features. By convention, their name starts with **Track** and spot features starts with **Spot**.

Feature name	Projections	Description
Track N spots	<i>idem</i>	The number of spots in a track.

1.20.8 Branch-spot features.

Because branch-spots link to spots in the core graph, the branch-spot features all relate to the hierarchy or neighborhood in the branch-graph.

Feature name	Projections	Description
Branch N successors	<i>idem</i>	Reports the number of successors of a branch spot. That is: how many branches emerge from this branch-spot. The branch-spot of a cell that divides will have a value of 2 for this feature. The end of a track will have a value of 0. The beginning of a track, 1.
Branch depth	<i>idem</i>	Report the hierarchy level of a branch. The hierarchy of a branch is how many ancestors a branch has. For instance, the first branch of a track has a level of 0. After one cell division, the two daughter branches have a level of 1, <i>etc.</i> This feature value is used to build the hierarchy graph.

1.20.9 Branch link features.

The branch-links represent a full branch from start to finish. Their features relate mainly to the branch size and extension in time.

Feature name	Projections	Description
Branch Spots	N <i>idem</i>	The number of spots in a branch. The count does not include the spots at the beginning of the branch, but includes the last one. This way the sum of this feature values for all the branches of a track equals the number of spots in the track.
Branch duration and displacement	Displacement	Measure the displacement of a branch, that is: the euclidean distance between the first and last spot of the branch.
	Duration	The time difference between the first and last spot of the branch.

1.21 The graph data structure of Mastodon.

The *mastodon-graph* Java package can be used to implement directed graphs with small memory footprint. Vertices and edges are not stored as individual objects. Instead vertex and edge data is laid out in a primitive `byte[]` array and accessed via proxy objects. This page describes some internals of the *trackmate-graph* package.

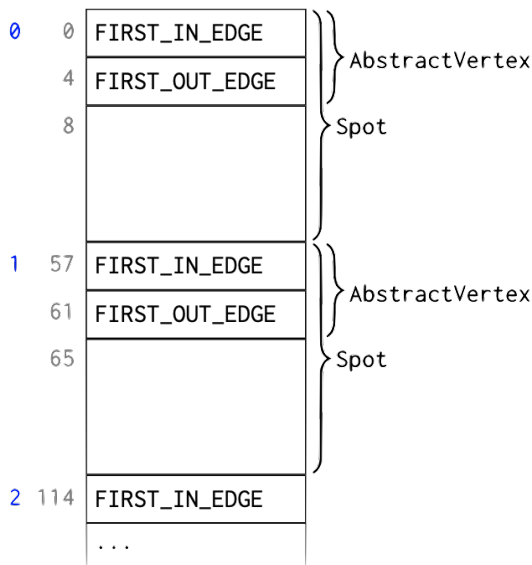
1.21.1 Memory layout.

Each vertex and each edge maps to a contiguous portion of a `byte[]` array. The size of a portion (*elements*) is fixed for a particular `AbstractVertex` or `AbstractEdge` subclass. Each *element* starts with a fixed part that represents the graph structure and then additional payload used by the subclass to describe some vertex attributes, *etc.*

There is one `byte[]` array that stores all vertices, and one `byte[]` array that stores all edges. References between these arrays are in the form of *element indices*. Whether these are indices refer to elements in the vertex or in the edge memory array is clear from the context.

Vertex layout.

The following diagram illustrates the layout of vertices in a `byte[]` array:



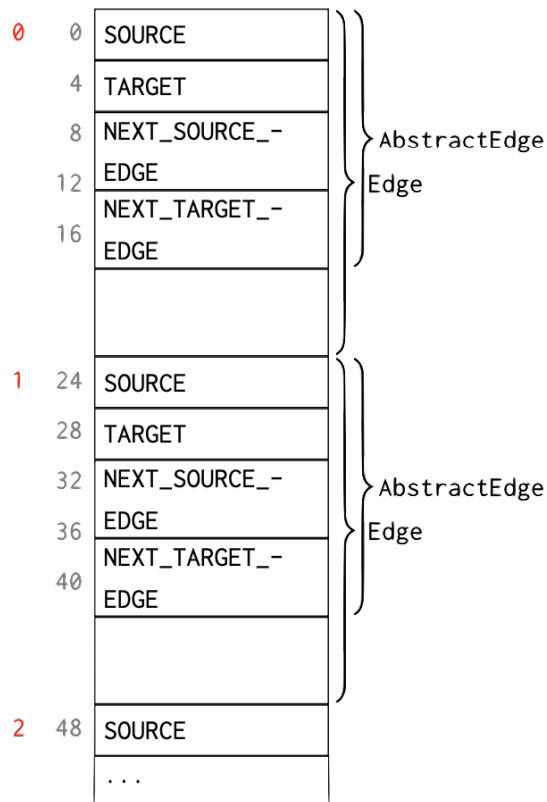
In the left-most column, *element indices* are shown (in blue), followed by byte indices (in grey). In the example, each element of the `AbstractVertex` subclass `Spot` requires 57 bytes to store. The data for the i th `Spot` starts at byte $i * 57$. The fixed `AbstractVertex` part comprises element indices `FIRST_IN_EDGE` and `FIRST_OUT_EDGE`, occupying 4 bytes each. The remaining 49 bytes are `Spot` attributes.

`FIRST_IN_EDGE` is the element index (in the edge memory array) of the first *incoming* edge, *i.e.*, an edge pointing to this vertex. The remaining incoming edges of the same vertex are stored as a linked list in the edge memory as described below. If this vertex does not have any incoming edges `FIRST_IN_EDGE` is -1.

Similarly, `FIRST_OUT_EDGE` is the element index of the first *outgoing* edge, *i.e.*, an edge starting from this vertex. The remaining outgoing edges of the same vertex are stored as a linked list in the edge memory as described below. If this vertex does not have any outgoing edges `FIRST_OUT_EDGE` is -1.

Edge layout.

The following diagram illustrates the layout of edges in a `byte[]` array:

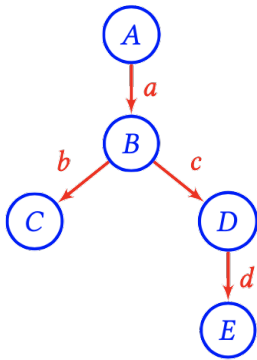


In the left-most column, *element indices* are shown (in red), followed by byte indices (in grey). In the example, each element of the `AbstractEdge` subclass `Edge` requires 24 bytes to store. The data for the i th `Edge` starts at byte $i * 24$. The fixed `AbstractEdge` part comprises element indices `SOURCE`, `TARGET`, `NEXT_SOURCE_EDGE`, and `NEXT_TARGET_EDGE`, occupying 4 bytes each. The remaining 8 bytes are `Edge` attributes.

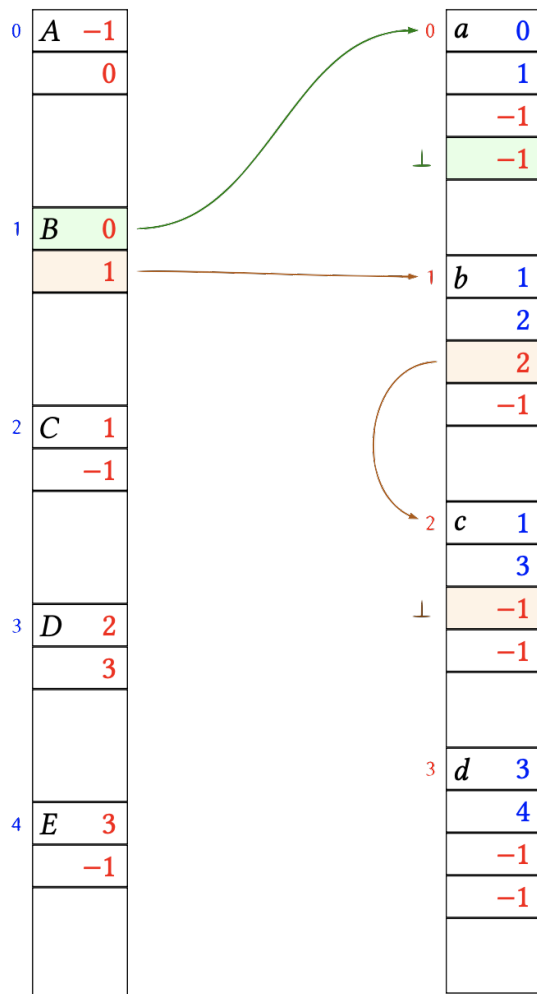
- `SOURCE` is the element index (in the vertex memory array) of the vertex from which this edge starts.
- `TARGET` is the element index (in the vertex memory array) of the vertex to which this edge points.
- `NEXT_SOURCE_EDGE` is the element index (in the edge memory array) of the next outgoing edge of the source vertex, , the next edge that has the same `SOURCE`. If there is no such edge then `NEXT_SOURCE_EDGE` is -1.
- `NEXT_TARGET_EDGE` is the element index (in the edge memory array) of the next incoming edge of the target vertex, , the next edge that has the same `TARGET`. If there is no such edge then `NEXT_TARGET_EDGE` is -1.

Example

Consider the following example graph comprising vertices A, B, C, D, E and edges a, b, c, d .



This is laid out in memory as follows

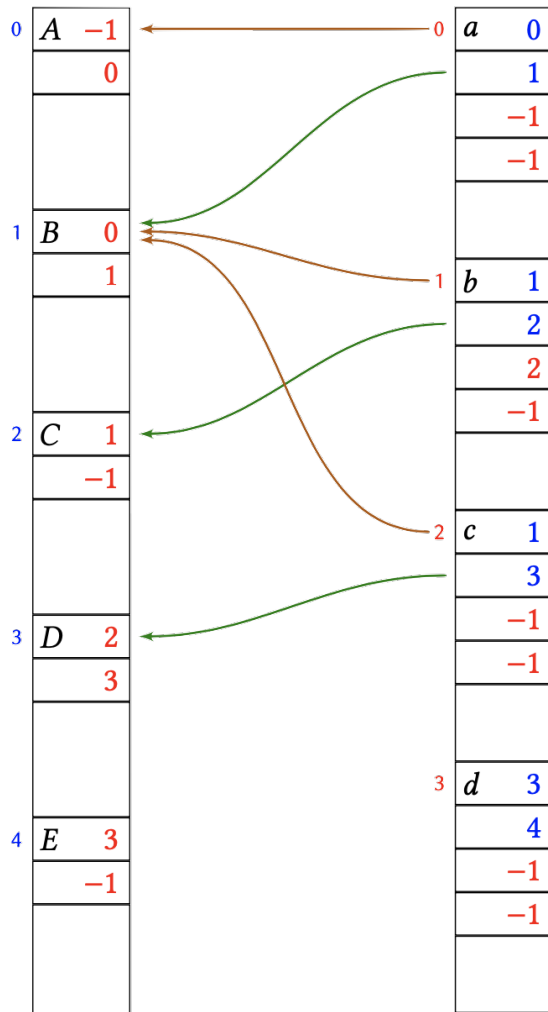


The links between vertex B and its adjacent edges a, b, c have been highlighted. Let's look at that in more detail: Vertex B is stored at element index 1 in the vertex memory array. B has one incoming edge a . The edge a is stored at

element index 0 in the edge memory array. Therefore the FIRST_IN_EDGE field of B is 0. Apart from a , the vertex B has no further incoming edges. Therefore, the NEXT_TARGET_EDGE field of a is -1, i.e., the list of edges entering B terminates here.

B has two outgoing edges b, c . The edge b is stored at element index 1 in the edge memory array. Therefore the FIRST_OUT_EDGE field of B is 1. The next outgoing edge of B is c which is stored at element index 2. Therefore, the NEXT_SOURCE_EDGE field of b is 2. After c , the vertex B has no further outgoing edges. Therefore, the NEXT_SOURCE_EDGE field of c is -1, i.e., the list of edges leaving B terminates here.

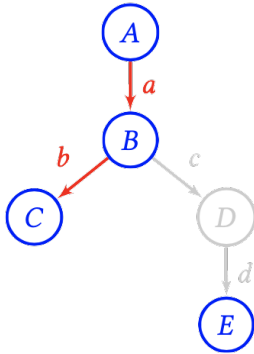
Below is the same memory layout again, this time highlighting the references from edges a, b, c back to the vertex memory array.



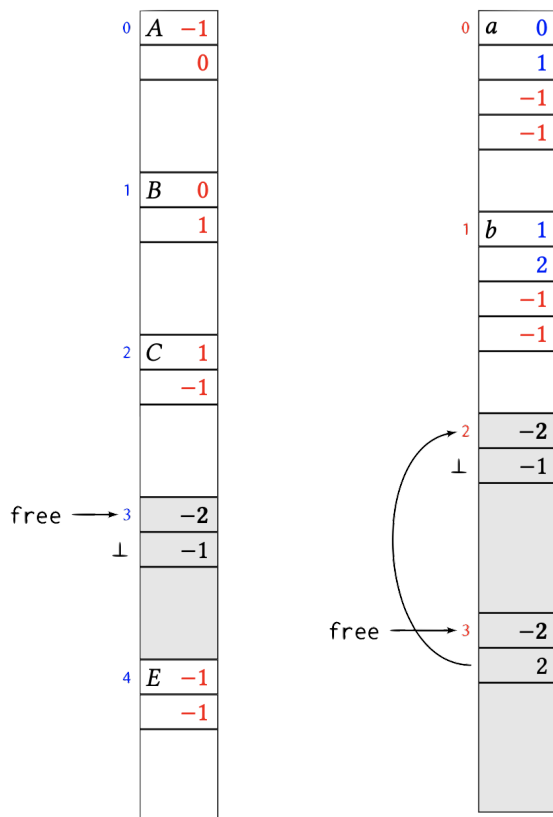
For example, edge c is leaving vertex B (index 1) and entering vertex D (index 3). Therefore the SOURCE field of c is 1, and the TARGET field of c is 3.

1.21.2 Free-list of unallocated elements.

The vertex and edge memory arrays can only ever grow. When elements are released, they are simply marked as free for re-use. Assume that in the above example vertex D is deleted, as well as its adjacent edges c , d , leaving this:



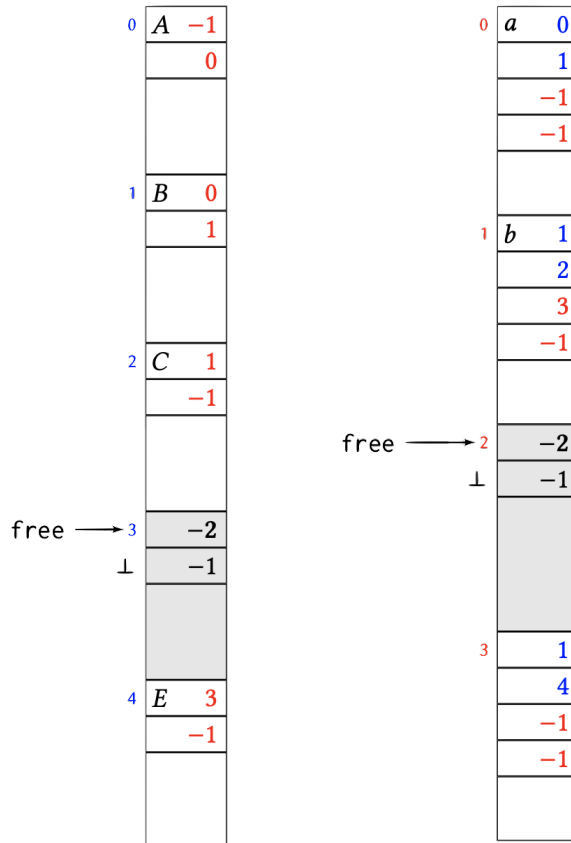
After removing c , d , D the memory layout looks like this:



The element 3 in the vertex memory array as well as elements 2 and 3 in the edge memory array have been marked as free. This is done by putting the magic number “-2” into the first 4 bytes of the element. In vertices and edges the first 4 bytes are always occupied by a (positive) index or a 1 index list terminator. Therefore, occupied and free blocks can not be confused. The next 4 bytes of a freed element are the index of the next freed element in the (same) memory array, or -1 if there is no next freed element. Each memory array remembers the index `free` of the first freed element.

Newly freed elements are enqueued at **free**, that is, at the head of the free-list. So in the above example, edge element 2 was freed first, followed by edge element 3.

The next edge element will be allocated at head of the free-list and the **free** index move to the next element. For example, assume that a new edge is created from *B* to *E*:



If **free** is -1, then no more freed elements are available and the underlying memory array has to grow to fit newly added elements.

1.22 Containment in Convex Polytopes using *k*-D trees.

Several important features of Mastodon rely on the fast retrieval of data items (spots and links) close to specific 3D positions. For instance we need to do so when you move the mouse close to the drawing of a spot in a view, to retrieve the spot in question. Or to determine what spots must be painted in a BDV view, depending on the zoom level, position, rotation and field of view. There exists several well established algorithms and techniques to do that, in the case where the bounding-box in which we need to retrieve data items is a 3D rectangle aligned with the X, Y, Z axes of the dataset.

In our case it is not. In BDV you can rotate the view around an arbitrary angle. The XY view plane does not match an orthogonal plane of the dataset. Also the planes that make the bounds of the field of view are not aligned with these axes. In our case, the field of view is a **Convex Polytope**. It is a portion of 3D space delimited by a set of planes. Think of an ideal diamond. Each facet of the diamond would be one of the planes. The interior of the diamond would be the convex polytope. Our goal is to know what are the points that are inside this volume so that we can paint them without losing time painting the ones not in the field of view.

At the time of the development of Mastodon, there was no published algorithm for the fast retrieval of points in a convex

polytope. Tobias derived such an algorithm in 2016, and it is detailed in this chapter. To the best of our knowledge this is unpublished:

Containment in Convex Polytopes using kD trees

1.23 Scripting functions

This section documents only the main scripting functions of Mastodon. They are grouped in three main classes listed below. See [the scripting tutorial](#) to learn how to script Mastodon in Fiji.

1.23.1 Mamut

public class **Mamut**

Main gateway for scripting Mastodon.

This should be the entry point to create a new project or open an existing one via the [`open\(String\)`](#) and [`newProject\(String\)`](#) static methods. Once an instance is obtained this way, a Mastodon project can be manipulated with the instance methods.

The gateways used in scripting are called Mamut and TrackMate. We chose these names to underly that this application offer functionalities that are similar to that of the MaMuT and TrackMate software, but improved. Nonetheless, all the code used is from Mastodon and allows only dealing with Mastodon projects.

Author

Jean-Yves Tinevez

Static methods

These methods need to be called on the class object *org.mastodon.mamut.Mamut* itself. They return an instance that can be used to manipulate the associated Mastodon project.

newProject

public static final *Mamut* **newProject**(String *bdvFile*, Context *context*)

Creates a new Mastodon project analyzing the specified image data.

Parameters

- **bdvFile** – a path to a BDV XML file. It matters not whether the image data is stored locally or remotely.
- **context** – an existing, non-null Context instance to use to open the project.

Throws

- [`IOException`](#) – when an error occurs trying to locate and open the file.
- [`SpimDataException`](#) – when an error occurs trying to open the image data.
- [`FormatException`](#) – when an error occurs with the image file format.

Returns

a new *Mamut* instance.

newProject

public static final *Mamut* **newProject**(String *bdvFile*)

Creates a new Mastodon project analyzing the specified image data.

A new Context is created along this call.

Parameters

- **bdvFile** – a path to a BDV XML file. It matters not whether the image data is stored locally or remotely.

Throws

- *IOException* – when an error occurs trying to locate and open the file.
- *SpimDataException* – when an error occurs trying to open the image data.
- *FormatException* – when an error occurs with the image file format.

Returns

a new *Mamut* instance.

open

public static final *Mamut* **open**(String *mamutProject*)

Opens an existing Mastodon project and returns a *Mamut* instance that can manipulate it.

A new Context is created along this call.

Parameters

- **mamutProject** – the path to the Mastodon file.

Throws

- *IOException* – when an error occurs trying to locate and open the file.
- *SpimDataException* – when an error occurs trying to open the image data.
- *FormatException* – when an error occurs with the image file format.

Returns

a new *Mamut* instance.

open

public static final *Mamut* **open**(String *mamutProject*, Context *context*)

Opens an existing Mastodon project and returns a *Mamut* instance that can manipulate it.

Parameters

- **mamutProject** – the path to the Mastodon file.
- **context** – an existing, non-null Context instance to use to open the project.

Throws

- *IOException* – when an error occurs trying to locate and open the file.
- *SpimDataException* – when an error occurs trying to open the image data.

- `FormatException` – when an error occurs with the image file format.

Returns

a new *Mamut* instance.

Methods

These methods manipulate a Mastodon project using an instance returned by the static methods above.

clear

public void **clear**()

Clears the content of the data model. Can be undone.

computeFeatures

public void **computeFeatures**(*String... featureKeys*)

Computes the specified features.

Parameters

- **featureKeys** – the names of the feature computer to use for computation. It matters not whether the feature is for spots, links, ...

computeFeatures

public void **computeFeatures**(boolean *forceComputeAll*, *String... featureKeys*)

Computes the specified features, possible forcing recomputation for all data items, regardless of whether they are in sync or not.

Parameters

- **forceComputeAll** – if `true`, will force recomputation for all data items. If `false`, feature values that are in sync won't be recomputed.
- **featureKeys** – the names of the feature computer to use for computation. It matters not whether the feature is for spots, links, ...

createTag

public void **createTag**(*String tagSetName*, *String... labels*)

Creates a new tag-set and several tags for this tag-set.

Parameters

- **tagSetName** – the tag-set name.
- **labels** – the list of labels to create in this tag-set.

createTrackMate

public *TrackMateProxy* **createTrackMate**()

Creates and returns a new *TrackMateProxy* instance. This instance can then be used to configure tracking on the image analyzed in this current *Mamut* instance.

It is perfectly possible to create and configure separately several *TrackMateProxy* instances. Tracking results will be combined depending on the instances configuration.

Returns

a new *TrackMateProxy* instance.

deleteSelection

public void **deleteSelection**()

Deletes all the data items (spots and tracks) currently in the selection.

detect

public void **detect**(double *radius*, double *threshold*)

Performs detection of spots in the image data with the default detection algorithm (the DoG detector).

Parameters

- **radius** – the radius of spots to detect, in the physical units of the image data.
- **threshold** – the threshold on quality of detection below which to reject detected spots.

echo

public void **echo**()

Prints the content of the data model as two tables as text in the logger output.

echo

public void **echo**(int *nLines*)

Prints the first N data items of the content of the data model as two tables as text in the logger output.

Parameters

- **nLines** – the number of data items to print.

getLogger

public Logger **getLogger**()

Returns the logger instance to use to send messages and errors.

Returns

the logger instance.

getModel

public Model **getModel**()

Returns the data model manipulated by this *Mamut* instance.

Returns

the data model.

getSelectionModel

public SelectionModel<Spot, Link> **getSelectionModel**()

Returns the selection model manipulated by this *Mamut* instance.

Returns

the selection model.

getWindowManager

public *WindowManager* **getWindowManager**()

Returns the *WindowManager* gateway used to create views of the data used in this *Mamut* instance.

Returns

the *WindowManager* gateway.

info

public void **info**()

Prints a summary information to the logger output.

infoFeatures

public void **infoFeatures**()

Prints summary information on the feature computers known to Mastodon to the logger output.

infoTags

public void **infoTags**()

Prints summary information on the tag-sets and tags currently present in the current Mastodon project.

link

public void **link**(double *maxLinkingDistance*, int *maxFrameGap*)

Performs linking of existing spots using the default linking algorithm (the Simple LAP linker).

Parameters

- **maxLinkingDistance** – the max linking distance (in physical unit) beyond which to forbid linking.
- **maxFrameGap** – the max difference in frames for bridging gaps (missed detections).

redo

public void **redo**()

Redo the last changes. Can be called several times.

resetSelection

public void **resetSelection**()

Clears the current selection.

save

public boolean **save**()

Saves the Mastodon project of this instance to a Mastodon file.

This method will return an error if a Mastodon file for the project has not been specified a first time with the [saveAs\(String\)](#) method.

Returns

true if saving happened without errors. Otherwise an error message is sent to the Logger instance.

saveAs

public boolean **saveAs**(String *mastodonFile*)

Saves the Mastodon project of this instance to a new Mastodon file (it is recommended to use the .mastodon file extension).

The file specified will be reused for every following call to the [save\(\)](#) method.

Parameters

- **mastodonFile** – a path to a writable file.

Returns

true if saving happened without errors. Otherwise an error message is sent to the `Logger` instance.

select

public void **select**(*String expression*)

Sets the current selection from a selection creator expression.

Such an expression can be:

```
mamut.select( "vertexFeature( 'Track N spots' ) < 10" )
```

Check [the selection creator tutorial](#) to learn how to build such expressions. An error message is sent to the logger if there is a problem with the evaluation of the expression.

Parameters

- **expression** – a selection creator expression.

setLogger

public void **setLogger**(*Logger logger*)

Sets the logger instance to use to send messages and errors.

Parameters

- **logger** – a logger instance.

setTagColor

public void **setTagColor**(*String tagSetName, String label, int R, int G, int B*)

Sets the color associated with a tag in a tag-set. The color is specified as a RGB triplet from 0 to 255.

Parameters

- **tagSetName** – the name of the tag-set containing the target tag.
- **label** – the tag to modify the color of.
- **R** – the red value of the RGB triplet.
- **G** – the green value of the RGB triplet.
- **B** – the blue value of the RGB triplet.

tagSelectionWith

public void **tagSelectionWith**(String *tagSetName*, String *label*)

Assigns the specified tag to the data items currently in the selection.

Parameters

- **tagSetName** – the name of the tag-set to use.
- **label** – the name of the tag in the tag-set to use.

undo

public void **undo**()

Undo the last changes. Can be called several times.

1.23.2 TrackMateProxy

public class **TrackMateProxy**

The tracking gateway used in scripting to configure and execute tracking in Mastodon scripts.

Author

Jean-Yves Tinevez

Methods

info

public void **info**()

Prints the current tracking configuration.

infoDetectors

public void **infoDetectors**()

Prints information on the collection of detectors currently usable in Mastodon.

infoLinkers

public void **infoLinkers**()

Prints information on the collection of linkers currently usable in Mastodon.

resetDetectorSettings

public void **resetDetectorSettings**()

Resets the detection settings to their default values.

resetLinkerSettings

public void **resetLinkerSettings**()

Resets the linking settings to their default values.

run

public boolean **run**()

Executes the tracking with current configuration.

Returns

true if tracking completed successful. An error message will be printed otherwise.

setDetectorSetting

public void **setDetectorSetting**(String key, Object value)

Configures one parameter of the current detector. The parameter key and value must be valid for the detector set with *useDetector(String)*, as shown in *infoDetectors()*.

Parameters

- **key** – the key of the parameter.
- **value** – the value to set for this parameter.

setLinkerSetting

public void **setLinkerSetting**(String key, Object value)

Configures one parameter of the current link. The parameter key and value must be valid for the linkset with *useLinker(String)*, as shown in *infoLinkers()*.

Parameters

- **key** – the key of the parameter.
- **value** – the value to set for this parameter.

useDetector

public void **useDetector**(*String detector*)

Configures this tracking session to use the specified detector. Prints an error message if the name is unknown.

Parameters

- **detector** – the name of the detector, as returned in *infoDetectors()*.

useLinker

public void **useLinker**(*String linker*)

Configures this tracking session to use the specified linker. Prints an error message if the name is unknown.

Parameters

- **linker** – the name of the linker, as returned in *infoLinkers()*.

1.23.3 WindowManager

public class **WindowManager**

Main GUI class for the Mastodon Mamut application.

It controls the creation of new views, and maintain a list of currently opened views. It has a *getProjectManager()* instance that can be used to open or create Mastodon projects. It has also the main app-model for the session.

Author

Tobias Pietzsch, Jean-Yves Tinevez

Methods

closeAllWindows

public void **closeAllWindows**()

Close all opened views and dialogs.

computeFeatures

public void **computeFeatures**()

Displays the feature computation dialog.

createBigDataViewer

public MamutViewBdv **createBigDataViewer**()

Creates and displays a new BDV view, with default display settings.

createBigDataViewer

public MamutViewBdv **createBigDataViewer**(Map<String, Object> *guiState*)

Creates and displays a new BDV view, using a map to specify the display settings.

The display settings are specified as a map of strings to objects. The accepted key and value types are:

- 'FramePosition' → an int[] array of 4 elements: x, y, width and height.
- 'LockGroupId' → an integer that specifies the lock group id.
- 'SettingsPanelVisible' → a boolean that specifies whether the settings panel is visible on this view.
- 'BdvState' → a XML Element that specifies the BDV window state. See `ViewerPanel.stateToXml()` and `ViewerPanel.stateFromXml(org.jdom2.Element)` for more information.
- 'BdvTransform' → an `AffineTransform3D` that specifies the view point.
- 'NoColoring' → a boolean; if true, the feature or tag coloring will be ignored.
- 'TagSet' → a string specifying the name of the tag-set to use for coloring. If not null, the coloring will be done using the tag-set.
- 'FeatureColorMode' → a String specifying the name of the feature color mode to use for coloring. If not null, the coloring will be done using the feature color mode.
- 'ColorbarVisible' → a boolean specifying whether the colorbar is visible for tag-set and feature-based coloring.
- 'ColorbarPosition' → a `Position` specifying the position of the colorbar.

Parameters

- **guiState** – the map of settings.

createBranchBigDataViewer

public MamutBranchViewBdv **createBranchBigDataViewer**()

Creates and displays a new Branch-BDV view, with default display settings. The branch version of this view displays the branch graph.

createBranchBigDataViewer

public MamutBranchViewBdv **createBranchBigDataViewer**(Map<String, Object> *guiState*)

Creates and displays a new Branch-BDV view, using a map to specify the display settings.

Parameters

- **guiState** – the settings map.

See also: `.createBigDataViewer(Map)`

createBranchTrackScheme

```
public MamutBranchViewTrackScheme createBranchTrackScheme()
```

Creates and displays a new Branch-TrackScheme view, with default display settings. The branch version of this view displays the branch graph.

createBranchTrackScheme

```
public MamutBranchViewTrackScheme createBranchTrackScheme(Map<String, Object> guiState)
```

Creates and displays a new Branch-TrackScheme view, using a map to specify the display settings.

Parameters

- **guiState** – the settings map.

See also: `.createTrackScheme(Map)`

createGrapher

```
public MamutViewGrapher createGrapher()
```

Creates and displays a new Grapher view, with default display settings.

createGrapher

```
public MamutViewGrapher createGrapher(Map<String, Object> guiState)
```

Creates and displays a new Grapher view, using a map to specify the display settings.

The display settings are specified as a map of strings to objects. The accepted key and value types are:

- 'FramePosition' → an `int[]` array of 4 elements: x, y, width and height.
- 'LockGroupId' → an integer that specifies the lock group id.
- 'SettingsPanelVisible' → a boolean that specifies whether the settings panel is visible on this view.
- 'NoColoring' → a boolean; if `true`, the feature or tag coloring will be ignored.
- 'TagSet' → a string specifying the name of the tag-set to use for coloring. If not `null`, the coloring will be done using the tag-set.
- 'FeatureColorMode' → a `@link String` specifying the name of the feature color mode to use for coloring. If not `null`, the coloring will be done using the feature color mode.
- 'ColorbarVisible' → a boolean specifying whether the colorbar is visible for tag-set and feature-based coloring.
- 'ColorbarPosition' → a `Position` specifying the position of the colorbar.
- 'GrapherTransform' → a `org.mastodon.views.grapher.datagraph.ScreenTransform` specifying the region to initially zoom on the XY plot.

Parameters

- **guiState** – the map of settings.

createHierarchyTrackScheme

public MamutBranchViewTrackScheme **createHierarchyTrackScheme**()

Creates and displays a new Hierarchy-TrackScheme view, with default display settings.

createHierarchyTrackScheme

public MamutBranchViewTrackScheme **createHierarchyTrackScheme**(Map<String, Object> guiState)

Creates and displays a new Hierarchy-TrackScheme view, using a map to specify the display settings.

Parameters

- **guiState** – the settings map.

See also: `.createTrackScheme(Map)`

createTable

public MamutViewTable **createTable**(Map<String, Object> guiState)

Creates and displays a new Table or a Selection Table view, using a map to specify the display settings.

The display settings are specified as a map of strings to objects. The accepted key and value types are:

- 'TableSelectionOnly' → a boolean specifying whether the table to create will be a selection table of a full table. If `true`, the table will only display the current content of the selection, and will listen to its changes. If `false`, the table will display the full graph content, listen to its changes, and will be able to edit the selection.
- 'FramePosition' → an `int[]` array of 4 elements: x, y, width and height.
- 'LockGroupId' → an integer that specifies the lock group id.
- 'SettingsPanelVisible' → a boolean that specifies whether the settings panel is visible on this view.
- 'NoColoring' → a boolean; if `true`, the feature or tag coloring will be ignored.
- 'TagSet' → a string specifying the name of the tag-set to use for coloring. If not `null`, the coloring will be done using the tag-set.
- 'FeatureColorMode' → a `@link String` specifying the name of the feature color mode to use for coloring. If not `null`, the coloring will be done using the feature color mode.
- 'ColorbarVisible' → a boolean specifying whether the colorbar is visible for tag-set and feature-based coloring.
- 'ColorbarPosition' → a `Position` specifying the position of the colorbar.

Parameters

- **guiState** – the map of settings.

createTable

```
public MamutViewTable createTable(boolean selectionOnly)
```

Creates and display a new Table or Selection Table view with default settings.

Parameters

- **selectionOnly** – if `true`, the table will only display the current content of the selection, and will listen to its changes. If `false`, the table will display the full graph content, listen to its changes, and will be able to edit the selection.

Returns

a new table view.

createTrackScheme

```
public MamutViewTrackScheme createTrackScheme()
```

Creates and displays a new TrackScheme view, with default display settings.

createTrackScheme

```
public MamutViewTrackScheme createTrackScheme(Map<String, Object> guiState)
```

Creates and displays a new BDV view, using a map to specify the display settings.

The display settings are specified as a map of strings to objects. The accepted key and value types are:

- 'FramePosition' → an `int[]` array of 4 elements: x, y, width and height.
- 'LockGroupId' → an integer that specifies the lock group id.
- 'SettingsPanelVisible' → a boolean that specifies whether the settings panel is visible on this view.
- 'TrackSchemeTransform' → a `ScreenTransform` that defines the starting view zone in TrackScheme.
- 'NoColoring' → a boolean; if `true`, the feature or tag coloring will be ignored.
- 'TagSet' → a string specifying the name of the tag-set to use for coloring. If not `null`, the coloring will be done using the tag-set.
- 'FeatureColorMode' → a `@link String` specifying the name of the feature color mode to use for coloring. If not `null`, the coloring will be done using the feature color mode.
- 'ColorbarVisible' → a boolean specifying whether the colorbar is visible for tag-set and feature-based coloring.
- 'ColorbarPosition' → a `Position` specifying the position of the colorbar.

Parameters

- **guiState** – the map of settings.

editTagSets

public void **editTagSets**()

Displays the tag-set editor dialog.

forEachBdvView

public void **forEachBdvView**(*Consumer*<? super MamutViewBdv> *action*)

Executes the specified action for all the currently opened BDV views.

Parameters

- **action** – the action to execute.

forEachBranchTrackSchemeView

public void **forEachBranchTrackSchemeView**(*Consumer*<? super MamutBranchViewTrackScheme> *action*)

Executes the specified action for all the currently opened Branch-TrackScheme views.

Parameters

- **action** – the action to execute.

forEachBranchView

public void **forEachBranchView**(*Consumer*<? super MamutBranchView<?, ?, ?>> *action*)

Executes the specified action for all the currently opened branch-graph views.

Parameters

- **action** – the action to execute.

forEachGrapherView

public void **forEachGrapherView**(*Consumer*<? super MamutViewGrapher> *action*)

Executes the specified action for all the currently opened Grapher views.

Parameters

- **action** – the action to execute.

forEachTableView

public void **forEachTableView**(*Consumer*<? super MamutViewTable> *action*)

Executes the specified action for all the currently opened Table views.

Parameters

- **action** – the action to execute.

forEachTrackSchemeView

public void **forEachTrackSchemeView**(Consumer<? super MamutViewTrackScheme> *action*)

Executes the specified action for all the currently opened TrackScheme views.

Parameters

- **action** – the action to execute.

forEachView

public void **forEachView**(Consumer<? super MamutView<?, ?, ?>> *action*)

Executes the specified action for all the currently opened views.

Parameters

- **action** – the action to execute.

getBdvWindows

public List<MamutViewBdv> **getBdvWindows**()

Exposes currently open BigDataViewer windows.

Returns

a List of MamutViewBdv.

openOnlineDocumentation

public void **openOnlineDocumentation**()

Opens the online documentation in a browser window.

DOCUMENTATION TOOLS.

- search

C

clear() (Java method), 130
 closeAllWindows() (Java method), 137
 computeFeatures() (Java method), 137
 computeFeatures(boolean, String) (Java method), 130
 computeFeatures(String) (Java method), 130
 createBigDataViewer() (Java method), 138
 createBigDataViewer(Map) (Java method), 138
 createBranchBigDataViewer() (Java method), 138
 createBranchBigDataViewer(Map) (Java method), 138
 createBranchTrackScheme() (Java method), 139
 createBranchTrackScheme(Map) (Java method), 139
 createGrapher() (Java method), 139
 createGrapher(Map) (Java method), 139
 createHierarchyTrackScheme() (Java method), 140
 createHierarchyTrackScheme(Map) (Java method), 140
 createTable(boolean) (Java method), 141
 createTable(Map) (Java method), 140
 createTag(String, String) (Java method), 130
 createTrackMate() (Java method), 131
 createTrackScheme() (Java method), 141
 createTrackScheme(Map) (Java method), 141

D

deleteSelection() (Java method), 131
 detect(double, double) (Java method), 131

E

echo() (Java method), 131
 echo(int) (Java method), 131
 editTagSets() (Java method), 142

F

forEachBdvView(Consumer) (Java method), 142
 forEachBranchTrackSchemeView(Consumer) (Java method), 142
 forEachBranchView(Consumer) (Java method), 142
 forEachGrapherView(Consumer) (Java method), 142
 forEachTableView(Consumer) (Java method), 142

forEachTrackSchemeView(Consumer) (Java method), 143
 forEachView(Consumer) (Java method), 143

G

getBdvWindows() (Java method), 143
 getLogger() (Java method), 132
 getModel() (Java method), 132
 getSelectionModel() (Java method), 132
 getWindowManager() (Java method), 132

I

info() (Java method), 132, 135
 infoDetectors() (Java method), 135
 infoFeatures() (Java method), 132
 infoLinkers() (Java method), 135
 infoTags() (Java method), 133

L

link(double, int) (Java method), 133

M

Mamut (Java class), 128

N

newProject(String) (Java method), 129
 newProject(String, Context) (Java method), 128

O

open(String) (Java method), 129
 open(String, Context) (Java method), 129
 openOnlineDocumentation() (Java method), 143
 org.mastodon.mamut (package), 128

R

redo() (Java method), 133
 resetDetectorSettings() (Java method), 136
 resetLinkerSettings() (Java method), 136
 resetSelection() (Java method), 133
 run() (Java method), 136

S

`save()` (*Java method*), [133](#)
`saveAs(String)` (*Java method*), [133](#)
`select(String)` (*Java method*), [134](#)
`setDetectorSetting(String, Object)` (*Java method*), [136](#)
`setLinkerSetting(String, Object)` (*Java method*), [136](#)
`setLogger(Logger)` (*Java method*), [134](#)
`setTagColor(String, String, int, int, int)` (*Java method*), [134](#)

T

`tagSelectionWith(String, String)` (*Java method*), [135](#)
`TrackMateProxy` (*Java class*), [135](#)

U

`undo()` (*Java method*), [135](#)
`useDetector(String)` (*Java method*), [137](#)
`useLinker(String)` (*Java method*), [137](#)

W

`WindowManager` (*Java class*), [137](#)