



TrackMate documentation.

Jean-Yves Tinevez

August 9, 2016

This document constitutes the [TrackMate](#) documentation. It is divided in four parts, that group sections by interest.

- The first part contains three tutorials, aimed at end-users. They are meant to guide users with the TrackMate plugin as a single-particle tracking Fiji plugin. They cover the three applications of the plugin:
 - automated single-particle tracking;
 - manual curation and correction of tracking results;
 - manual and semi-automatic tracking.
- The second part contains technical information. It documents the use of some TrackMate components and gives their exact definition, accuracy and performance. This part is meant to guide the user in choosing the right algorithm for their application.
- The third part is aimed at advanced users, that want to analyze the tracking results in another software package, or use a scripting language to configure and run TrackMate programmatically.
- The last part is made of seven tutorials aimed at Java developers and document how to extend TrackMate with custom modules. The seven tutorials document a specific TrackMate module (detection, particle-linking, *etc*) but are best read first in order, as they introduce developers to the annotation mechanism used for automatic plugin discovery.

Contents

I. Tutorials.	7
1. Using TrackMate for automated single-particle tracking.	7
1.1. Introduction.	7
1.2. The test image.	7
1.3. Starting TrackMate.	7
1.4. The start panel.	8
1.5. Choosing a detector.	10
1.6. The detector configuration panel.	11
1.7. The detection process.	12
1.8. Initial spot filtering.	13
1.9. Selecting a view.	14
1.10. Spot filtering.	15
1.11. Selecting a simple tracker.	17
1.12. Configuring the simple LAP tracker.	18
1.13. Our first tracking results.	19
1.14. Configuring a not so simple tracker.	19
1.15. Filtering tracks.	21
1.16. The end or so.	22
1.17. Wrapping up.	23
2. Manual editing of tracks using TrackMate.	23
2.1. Introduction.	23
2.2. The test image: Development of a <i>C.elegans</i> embryo.	24
2.3. Generating a sub-optimal segmentation.	24
2.4. Generating irrelevant tracks.	26
2.5. Launching TrackScheme.	26
2.6. TrackScheme in a nutshell.	27
2.7. Getting rid of bad tracks.	28
2.8. Spot editing with the HyperStack Displayer.	29
2.8.1. With the mouse.	29
2.8.2. With the keyboard.	30
2.9. Adding missed spots.	31
2.10. Editing tracks: creating links.	31
2.10.1. By drag & drop.	31
2.10.2. Using selection and right-click menu.	32
2.10.3. Creating several links at once.	33
2.11. Editing tracks: deleting links.	33
2.12. Wrapping up.	33

3. Manual and semi-automated tracking with TrackMate.	34
3.1. Setting up.	34
3.2. Creating spots one by one.	35
3.3. Create and removing single links.	36
3.4. The auto-linking mode.	37
3.5. Tracks are updated live.	38
3.6. Track and spot features are updated live.	38
3.7. Step-wise time browsing for sparse annotations.	39
3.8. The semi-automatic tracking tool.	40
3.9. Keyboard shortcuts for manual editing of tracks in the main view.	41
II. Technical documentation.	42
4. TrackScheme manual.	42
4.1. Moving around in TrackScheme.	42
4.2. Configuring TrackScheme look.	43
4.3. Exporting TrackScheme display.	45
4.4. Managing a selection in TrackScheme.	46
4.5. TrackScheme info-pane and feature plots.	47
4.6. Editing tracks with TrackScheme.	47
4.6.1. Linking spots with the popup menu item.	47
4.6.2. Triggering re-layout and style refresh.	48
4.6.3. Linking spots with drag and drop.	49
4.6.4. Removing spots and links.	49
4.6.5. Editing track names and imposing track order.	49
4.6.6. Editing spot names and imposing branch order.	49
5. Description of TrackMate algorithms.	50
5.1. Spot detectors.	51
5.1.1. Spot features generated by the spot detectors.	51
5.1.2. Laplacian of Gaussian particle detection (LoG detector).	51
5.1.3. Difference of Gaussian particle detection (DoG detector).	52
5.1.4. Downsample LoG detector.	52
5.1.5. Handling the detection of large images with the Block LoG detector.	53
5.2. Spot analyzers.	53
5.2.1. Mean, Median, Min, Max, Total intensity and its Standard Deviation.	53
5.2.2. Contrast & Signal/Noise ratio.	53
5.2.3. Estimated diameter.	54
5.3. Spot trackers or particle-linking algorithms.	54
5.3.1. LAP trackers.	54
5.3.2. Linear motion tracker.	57

6. Particle-linking algorithms accuracy.	60
6.1. The ISBI 2012 single particle challenge.	60
6.2. Current TrackMate version accuracy against the ISBI dataset.	60
6.2.1. Scenarios.	61
6.2.2. Example images from the challenge dataset.	61
6.2.3. Accuracy measurements.	62
6.2.4. Parameter used.	63
6.2.5. Results.	63
6.3. Comments.	68
7. Spot detectors performance.	69
7.1. The test environment.	69
7.2. Processing time for a 2D image as a function of its size.	70
7.3. Processing time for a 3D image as a function of its size.	71
7.4. Processing time for a 2D image as a function of the spot radius.	71
7.5. Processing time for a 3D image as a function of the spot radius.	72
7.6. Choosing between DoG and LoG based on performance.	73
III. Interoperability.	74
8. Importing and analyzing TrackMate data in MATLAB.	74
8.1. Installation of TrackMate functions for MATLAB.	74
8.2. The simple case of linear tracks.	74
8.3. Importing the spot feature table.	77
8.4. Importing the edge track table.	80
8.5. Importing TrackMate data as a MATLAB graph.	84
8.6. Other MATLAB functions for TrackMate.	91
8.7. Application examples and links.	91
9. Scripting TrackMate in Python.	92
9.1. A full example.	92
9.2. Loading and reading from a saved TrackMate XML file.	95
9.3. Export spot, edge and track numerical features after tracking.	98
9.4. Manually creating a model.	101
IV. Extending TrackMate.	106
10. How to write your own edge feature analyzer algorithm for TrackMate.	106
10.1. Introduction.	106
10.2. TrackMate modules.	106
10.3. Basic project structure.	107
10.4. Core class hierarchy.	108
10.5. Feature analyzers specific methods.	108

10.6. Multithreading & Benchmarking methods.	111
10.7. The core methods.	112
10.7.1. isLocal().	112
10.7.2. process(Collection< DefaultWeightedEdge > edges, Model model).	113
10.8. Making the analyzer discoverable.	113
11. How to write your own track feature analyzer algorithm for TrackMate.	115
11.1. Introduction.	115
11.2. Track analyzers.	115
11.3. Track feature analyzer header.	115
11.4. Declaring features.	116
11.5. Accessing tracks in TrackMate.	117
11.6. Calculating the position of start and end points.	118
11.7. Wrapping up.	118
11.8. How to disable a module.	119
12. How to write your own track feature analyzer algorithm for TrackMate.	120
12.1. Introduction.	120
12.2. Spot analyzers and spot analyzer factories.	120
12.3. The spot analyzer factory.	120
12.4. The spot analyzer.	121
12.5. Using SciJava priority to determine order of execution.	123
12.6. Wrapping up.	123
13. How to write your own viewer for TrackMate.	124
13.1. Introduction.	124
13.2. A custom TrackMate view.	124
13.3. The ViewFactory.	125
13.4. The TrackMateModelView interface.	125
13.4.1. Methods.	125
13.4.2. Display settings.	126
13.4.3. Listening to model changes.	127
13.4.4. Listening to selection changes.	127
13.5. A simple event logger.	127
13.6. Controlling the visibility of your view with the SciJava visible parameter.	130
14. How to write custom actions for TrackMate.	131
14.1. Introduction.	131
14.2. The TrackMateActionFactory interface.	131
14.2.1. SciJava parameters recapitulation.	131
14.2.2. Action factory methods.	131
14.3. The TrackMateAction interface.	133
14.4. Wrapping up.	134

15. How to write your own detection algorithm for TrackMate.	135
15.1. Introduction.	135
15.2. The SpotDetector interface.	136
15.2.1. A detector instance operates on a single frame.	136
15.2.2. A SpotDetector <i>can be</i> multithreaded.	136
15.2.3. Detection results are represented by Spots.	137
15.2.4. A dummy detector that returns spiraling spots.	137
15.3. The SpotDetectorFactory interface.	141
15.3.1. Getting the raw image data.	141
15.3.2. Getting detection parameters through a configuration panel.	142
15.3.3. Checking the validity of parameters.	142
15.3.4. Saving to and loading from XML.	143
15.3.5. Instantiating spot detectors.	144
15.3.6. The code for the dummy spiral generator factory.	145
15.4. Wrapping up.	148
16. How to write your own particle-linking algorithm for TrackMate.	148
16.1. Introduction.	148
16.2. Simple, undirected graphs.	148
16.3. Graphs in TrackMate.	149
16.4. Particle-linking algorithms in TrackMate.	150
16.5. A dummy example: drunken cell divisions.	150
16.6. The factory class.	154
16.7. Wrapping up.	155

Part I.

Tutorials.

1. Using TrackMate for automated single-particle tracking.

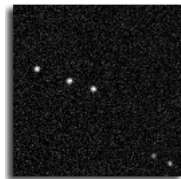
1.1 Introduction.

This tutorial is the starting point for [TrackMate](#) users. It explains how it works by walking you through a simple case, using an easy image.

The [TrackMate](#) plugin provides a way to automatically segment spots or roughly spherical objects from a 1D, 2D or 3D image, and track them over time. It follows the classical single-particle tracking scheme, where the detection step and the particle-linking step are separated. Therefore each step is handled in the user interface by a specific panel, and you will go back in forth through them. Also, TrackMate works like a fishing net with small holes: it will find as much spots as it can, even the ones you are not interested. So there is a step to filter them out before tracking. In these views, TrackMate resembles a bit to the Spot Segmentation Wizard of [Imaris™](#).

1.2 The test image.

The test image we will use for this tutorial has now a link in Fiji. You can find it in [File](#) > [Open Samples](#) > [Tracks for TrackMate \(807K\)](#), at the bottom of the list.



This is 128x128 stack of 50 frames, uncalibrated. It is noisy, but is still a very easy use case: there is at most 4 spots per frame, they are well separated, they are about the same size and the background is uniform. It is such an ideal case that you would not need TrackMate to deal with it. But for this first tutorial, it will help us getting through TrackMate without being bothered by difficulties.


Also, if you look carefully, you will see that there are two splitting events - where a spot seems to divide in two spots in the next frame, one merging event - the opposite, and a gap closing event - where a spot disappear for one frame then reappear a bit further. TrackMate is made to handle these events, and we will see how.

1.3 Starting TrackMate.

With this image selected, launch TrackMate from the menu [Plugins](#) > [Tracking](#) > [TrackMate](#) or from the [Command launcher](#). The TrackMate GUI appears next to the image, displaying the starting dialog panel.

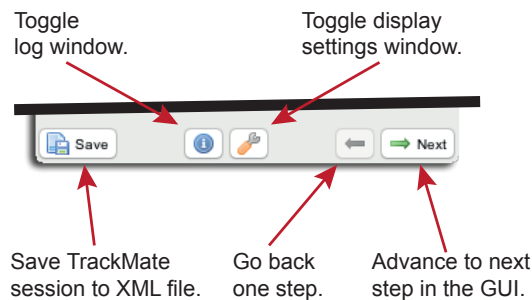
But first, just a few words about its look. The user interface is a single frame - that can be resized - divided in a main panel, that displays context-dependent dialogs, and a permanent bottom panel containing the four main buttons depicted on the right.

The → **Next** button allows to step through the tracking process. It might be disabled depending on the current panel content. For instance, if you do not select a valid image in the first panel, it will be disabled. The ← **Previous** button steps back in the process, without executing actions. For instance, if you go back on the segmentation panel, segmentation will not be re-executed.

The  **Save** button creates a XML file that contains all of the data you generated at the moment you click it. Since you can save at any time, the resulting file might miss tracks, spots, etc. You can load the saved file using the menu item `Plugins >> Tracking >> Load a TrackMate file`. It will restore the session just where you saved it.

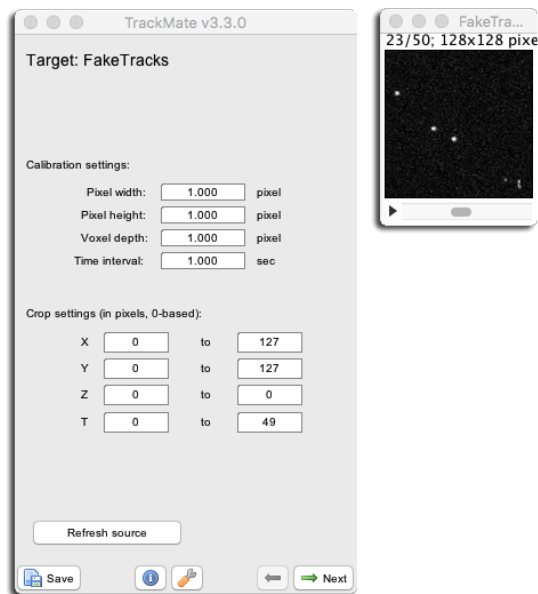
Now is a good time to speak strategy when it comes to saving/restoring. You can save at anytime in TrackMate. If you save just before the tracking process, you will be taken there with the data you generated so far upon loading. TrackMate saves a link to the image file (as an absolute file path) but not the image itself. When loading a TrackMate file, it tries first to retrieve and open this file in ImageJ. So it is a good idea to pre-process, crop, edit metadata and massage the target image first in Fiji, then save it as a .tif, then launch TrackMate. Particularly if you deal with a multi-series file, such as Leica .lif files.

The advantage of this approach is that you load in TrackMate, and everything you need will be loaded and displayed. However, if you need to change the target file or if it cannot be retrieved, you will have to open the TrackMate XML file and edit its 4th line.



1.4 The start panel.

This first panel allows you to check the spatial and temporal calibration of your data. It is very important to get it right, since everything afterwards will be based on physical units and not in pixel units (for instance μm and minutes, and not pixels and frames). In our case, that does not matter actually, since our test image has a dummy calibration (1 pixel = 1 pixel).



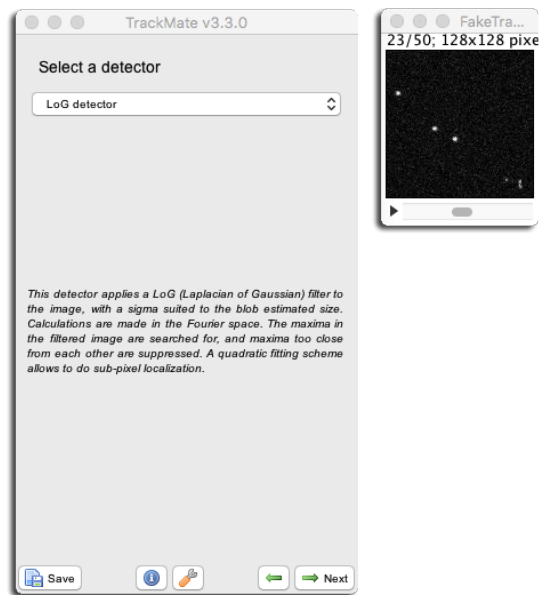
What is critical is also to check the dimensionality of the image. In our case, we have a 2D time-lapse of 50 frames. If metadata are not present or cannot be read, ImageJ tends to assume that stack always are Z-stack on a single time-point.

If the calibration or dimensionality of your data is not right, it is best changing it in the image metadata itself, using `Image >> Properties` (`(Shift) + P`). Then press the **Refresh source** button on the TrackMate start panel to grab changes.

You can also define a sub-region for processing: if you are only interested in finding spots in a defined region of the image, you can use any of the ROI tools of ImageJ to draw a closed area on the image. Once you are happy with it, press the **Refresh source** button on the panel to pass it to TrackMate. You should see that the **X Y** start and end values change to reflect the bounding box of the ROI you defined. The ROI needs not to be a square. It can be any closed shape. If you want to define the min and max **Z** and / or **T**, you have to edit manually the fields on the panel.

Defining a smaller area to analyze can be very beneficial to test and inspect for correct parameters, particularly for the segmentation step. In this tutorial, the image is so small and parse that we need not worrying about it. Press the `→` **Next** button to step forward.

1.5 Choosing a detector.

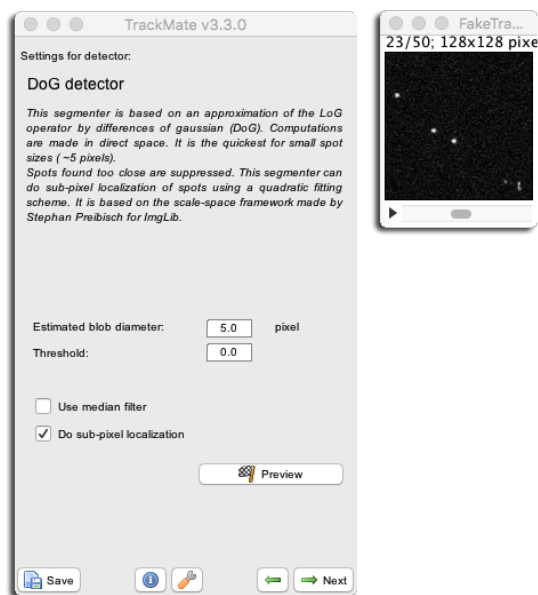


You are now offered to choose a detection algorithm ("detector") amongst the currently implemented ones. The choice is actually quite limited. Apart from the Manual annotation, you will find four detectors, but they are all based on the LoG filter [1]. They are described in detail later, but here is what you need to know.


- The LoG detector applies a plain Laplacian of Gaussian filter on the image. All calculations are made in the Fourier space, which makes it optimal for intermediate spot sizes, between 5 and 20 pixels in diameter.
- The DoG detector uses the difference of Gaussians approach to approximate a LoG filter by the difference of two Gaussians. Calculations are made in the direct space, and it is optimal for small spot sizes, below 5 pixels.
- The Downsample LoG detector uses the LoG detector, but downsizes the image by an integer factor before filtering. This makes it optimal for large spot sizes, above 20 pixels in diameter, at the cost of localization precision.
- The Block LoG detector splits the image in small blocks to limit memory usage in the case of large input image.

In our case, let us just use the DoG detector.

1.6 The detector configuration panel.




The LoG-based detectors fortunately demand very few parameters to tune them. The only really important one is the Estimated blob diameter. Just enter the approximate size of the spots you are looking to tracks. Careful: you are expected to enter it in physical units. In our dummy example, there is no calibration (1 pixel = 1 pixel), so it does not appear here.

There are extra fields that you can configure also. The Threshold numerical value aims at helping dealing with situation where a gigantic number of spots can be found. Every spot with a quality value below this threshold value will not be retained, which can help saving memory. You set this field manually, and check how it fares with the  **Preview** button button.

You can check **Use median filter**: this will apply a 3x3 median filter prior to any processing. This can help dealing with images that have a marked salt and pepper noise which generates spurious spots.

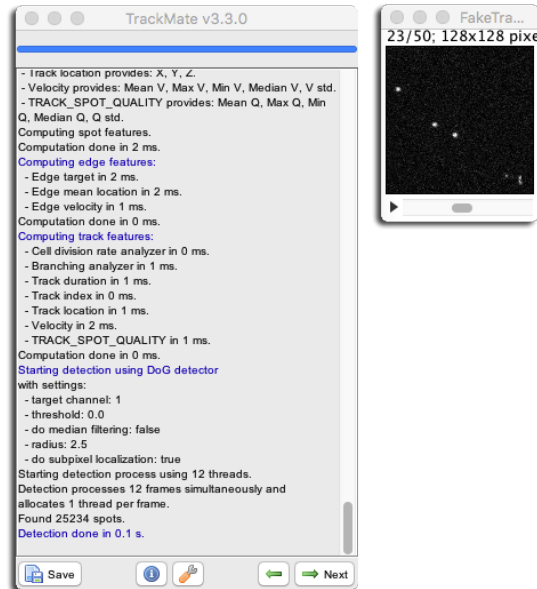
We hope that TrackMate will be used in experiments requiring **Sub-pixel localization**, such as following motor proteins in biophysical experiments, so we added schemes to achieve this. The one currently implemented uses a quadratic fitting scheme (made by Stephan Salfeld and Stephan Preibisch) based on [David Lowe SIFT work](#) [2]. It is not as accurate as the algorithms typically used in super-localization, but has the advantage of being very quick, compared to the segmentation time itself.

Finally, there is a  **Preview** button that allows to quickly check your parameters on the current data. After some computations, you can check the results overlaid on the image. Most likely, you will see plenty of spurious spots that you will be tempted to discard by adjusting the **Threshold** value. This is a very good approach for large problems. Here, we care little for that, just leave the threshold at 0.

The two others automated detectors share more or less the same fields in their own configuration panel. The Downsampled LoG detector simply asks for an extra down-sampling integer factor. In our case, the spots we want to track are about 5 pixels in diameter, so this is what we

enter in the corresponding field. We don't need anything else. The **Sub-pixel localization** option adds a very little time penalty so we can leave it on.

1.7 The detection process.



Once you are happy with the segmentation parameters, press the **Next** button and the segmentation will start. The TrackMate GUI displays the log panel, that you will meet several times during the process. It is basically made of a text area that recapitulates your choices and send information on the current process, and of a progress bar on top. You can copy-paste the text if you want to keep track of the process somewhere. You can even add comments as text in it: it is editable, and everything you type there is saved in the XML file, and retrieved upon loading. You can access the log panel anytime, by clicking on the **log** button at the bottom of the TrackMate window.

Should the process be long enough, you should be able to see that the **Next** button turned into a **Cancel** button. If you press it while the detection is running, TrackMate will finish the detection of the current frames, and stop. You could now go on with the spots it found, or go back and restart.

TrackMate takes advantage of multi-core computers, which seems to be the standard nowadays. If the source image has more time-points than CPU cores, it will segment one time-frame per core available. On computers with many cores, the progress bar will seem to move in a bulky way: if you have 16 cores, 16 time-points will be segmented at once, and it is likely that they will be finished approximately on the same time. So don't be worried if the progress bar does not move in the beginning for large images. If you have more cores than time-points, then TrackMate will allocate cores differently and give more cores to each time-point. For instance, if you have 12 CPU cores and only 4 time-points, each time-point will get 3 cores for calculation.

On our dummy image, this is clearly something we need to worry about, and the segmentation should be over in a few seconds. Typically, this is the step that takes the most time. Once the segmentation is done, the → **Next** button is re-enabled.

1.8 Initial spot filtering.

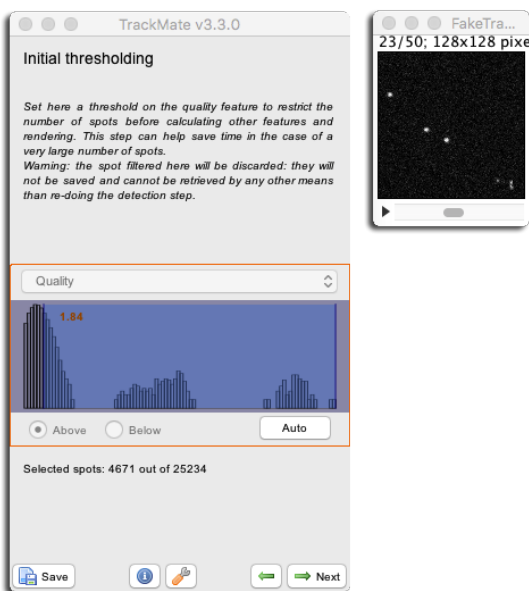
Here is a difficult step to explain, particularly because we do not need at all now. If the explanations following in this paragraph seem foggy, please feel free to press the → **Next** button and skip to the next paragraph. This one is all related to performance, memory and disk usage in difficult cases.

TrackMate uses generic segmentation algorithms for which there is only a little number of parameters to specify. The price to pay then, is that you can get a lot of undesired spots as an outcome. And in some cases, a really large amount of those.

This is why there are spot features and feature filters. In the next steps, each spot will have a series of numerical features calculated using its location, radius and the image it is found in, such as the mean pixel intensity. You will be able to define filters on these features, to retain only the ones that are relevant to your study.

But for a very large number of spots - let's say: more than 1 million of them - performance issues can kick in. Those millions of spots will be stored in the model, and saved in the TrackMate file, in case you want to step back and change the filters, because for instance you realized you are not happy with the end results (you can do that). Some visualization tools - the 3D displayer for instance - will generate the renderings for those millions of spots at once and hide or show them depending on the filter values, because it is too expensive to recreate the renderings while tuning the filter values.

To deal with that, we added a first filter prior to any other step, that uses the **Quality** value. The quality value is set by the detector, and is an arbitrary measure of the likelihood of each spot to be relevant. This panel collects all the quality values for all spots, and display their histogram (Y-scale is logarithmic). You can manually set a threshold on this histogram by clicking and dragging in its window. All spots with a quality value below this threshold will be **discarded**. That is: they will be deleted from the process, not saved in the file, they won't be displayed, nor their features will be calculated. Which is what we want when meeting a gigantic number of spurious spots. Note that this step is **irreversible**: if you step back, you will be taken to the detector configuration panel, ready to generate a new detection from scratch.



In our case, we see from the histogram that we could make sense of this step. There is a big peak at low quality (around a value of 1.2) that contains the majority of spots and is most likely represent spurious spots. So we could put the threshold around let's say 5.5 and probably ending in having only relevant spots. But with less than 10 000 spots, we are very far from 1 million so we need not to use this trick. Leave the threshold bar close to 0 and proceed to the next step.

1.9 Selecting a view.

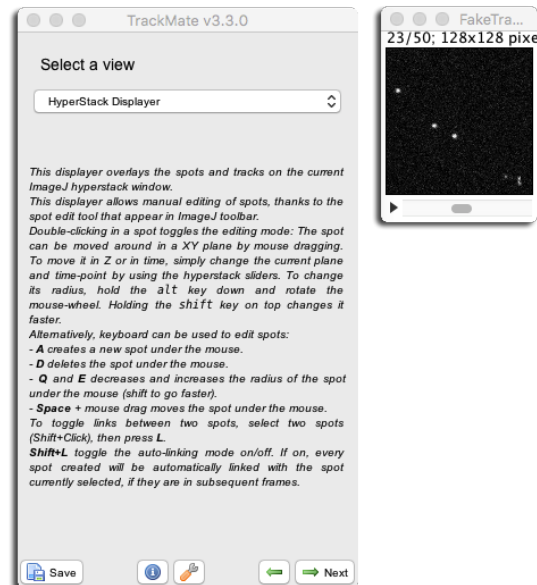
Here, you can choose between the two visualization tools that will be used to display the tracking results. The first one, HyperStack displayer, simply reuses ImageJ stack window and overlay the results non-destructively over the image. Choosing the 3D viewer will open a new 3D viewer window, import that image data in it, and will display spots as 3D spheres and tracks as 3D lines.

Honestly, choose the HyperStack displayer. Unless you have a very specific and complicated case that needs to inspect results in 3D immediately, you do not need the 3D viewer. The HyperStack displayer is simpler, lighter, allow to manually edit spots, and you will be able to launch a 3D viewer at the end of the process and still get the benefits.

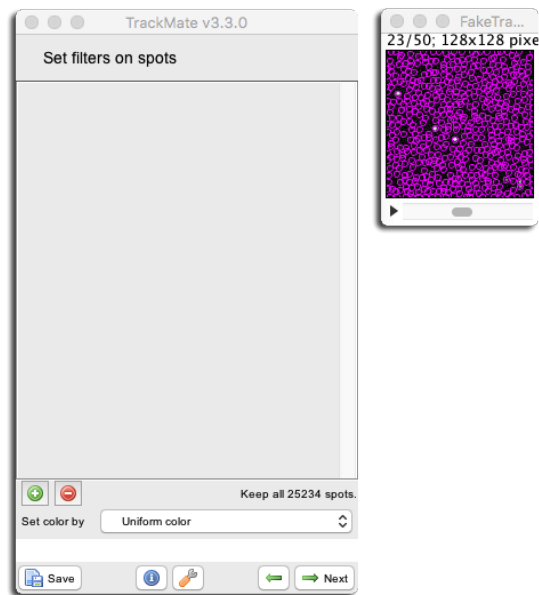
When you press the → **Next** button, two process starts:

- the features of all spots (well, those you left after the initial filtering step) are calculated;
- the displayer selected does everything it needs to prepare displaying them.

So nothing much. Let's carry on.





1.10 Spot filtering.



The moment this panel is shown, the spots should be displayed on the ImageJ stack. They take the shape of purple circles of diameter set previously. As promised, there are quite a lot of them, and their vast majority are irrelevant. If you did not remove the irrelevant one in the initial thresholding step, you should get an overlay that resembles the image to the right.


Trying to do particle linking on all these spots would be catastrophic, and there would be no hope to make sense of the data as it is now. This is why there is this spot filtering step, where you can use the features we just calculated to select the relevant spots only.

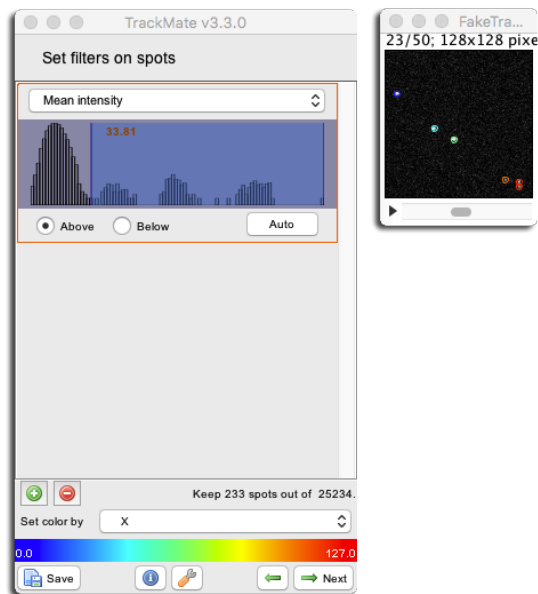
The spot filtering panel is divided in two. The upper part, which is empty now, contains the filter you define, in the shape of histograms. We will come back to them soon. The bottom part contains the  and  buttons that allow to respectively add or remove a feature filter, and a combo-box to set the display color of the spots.

Let us try it to play with it to find the best feature to filter out spurious spots.

By default, when the combo-box is on `Uniform color`, all spots are magenta. By clicking on it, you see that you can select amongst all the possible features calculated. For instance, if you select `X`, the spots will be colored according to their `X` position. A colored bar below the combo-box indicates the range the color gradient corresponds to.

`X` does not seem to be a good feature to select relevant spots. We know that `Quality` should be, by construction, but let us pick `Mean intensity`. By scrolling through the time slide you should be able to see that now all the spurious spots have a blue to turquoise color, whether the real one stands forward in red or yellow.

We will therefore add a filter based on this feature. Click the green  button. A small orange box should appear in the upper part, containing the histogram for a given feature. Click on the orange box combo-box to select `Mean intensity`. You should have something similar to the image below.



We note that the histogram has a very desirable shape: a massive peak at low intensity represent most of the spots. There are other smaller peaks at higher intensity, and fortunately, they are very well separated from the large peak.

To move the threshold, simply click and drag inside the histogram window. Notice how the overlay is updated to display only the remaining spots after filtering.

Interlude: A word on the GUI: We put effort into having a GUI that can be navigated almost solely with the keyboard. Any of the small filter panel can be controlled with the keyboard. For instance: give the histogram the focus by either pressing the Tab key or clicking into it.

- The floating threshold value should turn from orange to dark red. You can now type a numerical value (including decimals using the dot '.' as separator); wait two seconds, and the threshold value will be updated to what you just typed.
- Or use the arrow keys: the left and right arrow keys will change the threshold value by 10%, the up and down arrow will set it to the max and min value respectively.

A filter can be set to be above or below the given threshold. You change this behavior using the radio buttons below the histogram window. In our case, we want it to be above of course. The **Auto** button uses Otsu method [3] to determine automatically a threshold. In our case, we will put it manually around 33.

You can inspect the data by scrolling on the hyperstack window and check that only mostly good spots are retained. This is an easy image. The spots you have filtered out are not discarded; they are simply not shown and they will not be taken into account when doing particle linking. In a later stage, you can step back to this step, and retrieve the filtered out spots by removing or changing the filters.

You can stack several filters by simply clicking on the green + button. TrackMate will retain the spots that satisfy to all (logical *and*) the criteria set by the filters.

Press **Next** when you are ready to build tracks with these spots.

1.11 Selecting a simple tracker.

The next panel let you choose amongst available particle-linking algorithms, or "trackers".

The apparent profusion of choices should not disorient you, for it just that: an appearance. We chose to focus on the Linear Assignment Problem (LAP) in the framework first developed by Jaqaman *et al.* [4].

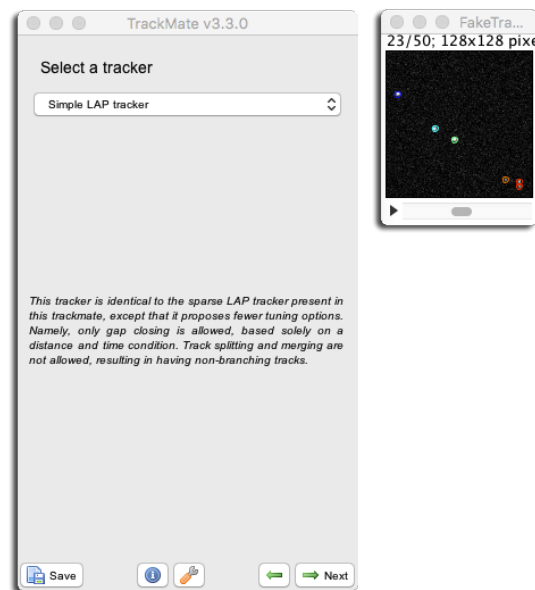
The first two LAP trackers are based on LAP, with important differences from the original paper described [here](#). We focused on this method for it gave us a lot of flexibility and it can be configured easily to handle most cases. You can tune it to allow *splitting events*, where a track splits in 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 segmentation fails, ...) but the track manages to recuperates and connect with reappearing spots later.

These LAP algorithm exists in TrackMate 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.

There is also a 3rd tracker, the [Nearest neighbor search](#) tracker. This is the most simple tracker you can build, and it is mostly here for demonstration purposes. Each spot in one frame is linked to another one in the next frame, disregarding any other spot, thus achieving only a very local optimum. You can set a maximal linking distance to prevent the results to be totally pathological, but this is as far as it goes. It may be of use for very large and easy datasets: its memory consumption is very limited (at maximum only two frames need to be in memory) and is quick (the nearest neighbor search is performed using [Kd-trees](#) [6]).

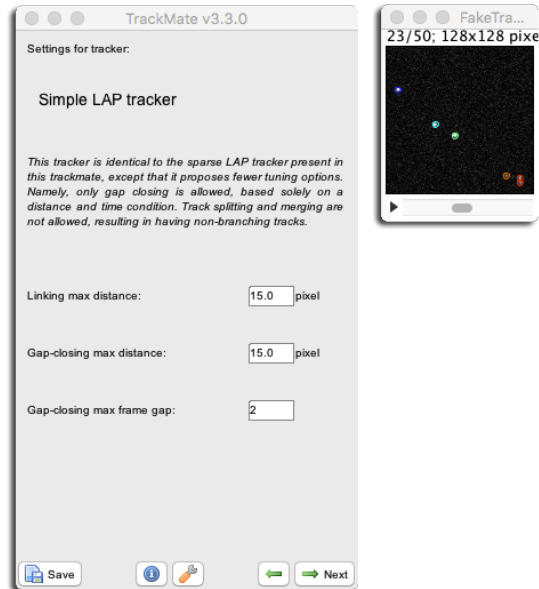
The 4th tracker, Linear motion LAP tracker is well suited for particles that move with a roughly constant velocity, this velocity being different for each particle. Its compared accuracy and relevance for various scenario is discussed later.



Then of course, there is the option to skip the automated tracking using Manual tracking. Right now, in our first trial, let us pick the Simple LAP tracker.

1.12 Configuring the simple LAP tracker.

As promised, there is only three configuration fields.



- The first one defines the maximal allowed linking distance. This value limits the spatial search range for candidate matching spots. If you know the maximal displacement of your object between two frame, you can put this value here. Theoretically, a too large value will demand more computation time. In our case, seeing the size of the dataset, this does not matter at all. This distance must be expressed in physical units, but again, you don't see it there for there is no spatial calibration on our image.
- The second field defines the maximal distance for gap-closing. Two track segments will not be bridged if the last spot of the first segment is further away than the first spot of the second segment. In our dummy example stack, there is spot disappearance at the frame number 45, top left. So the spot on frame 44 and the spot on frame 46 must not be separated by more than the distance you set there to have a chance to be linked.
- The third field also deal with the detection of gap-closing events, and sets the maximal *frame interval* between two spots to be bridged. Careful, the time is set in frame interval, here we do not want the physical time. In our case, since the only disappearance event we have last one frame, we can simply put this value to 2 frames duration. But actually it does not matter, as you can see by experimenting.

Press → **Next** to start the tracking computation.

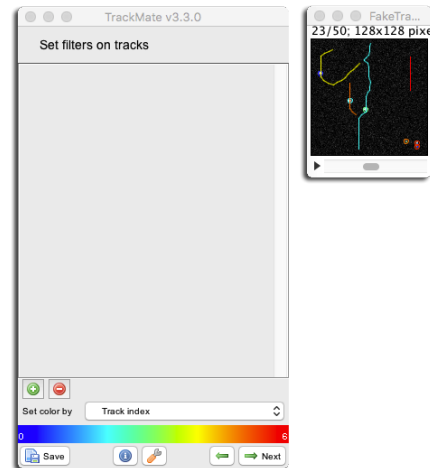
1.13 Our first tracking results.

You are now shown the log panel, where the tracking process is logged. Since our dataset is very small, it should complete very quickly. Press → **Next** again to see the results. They should look like this:

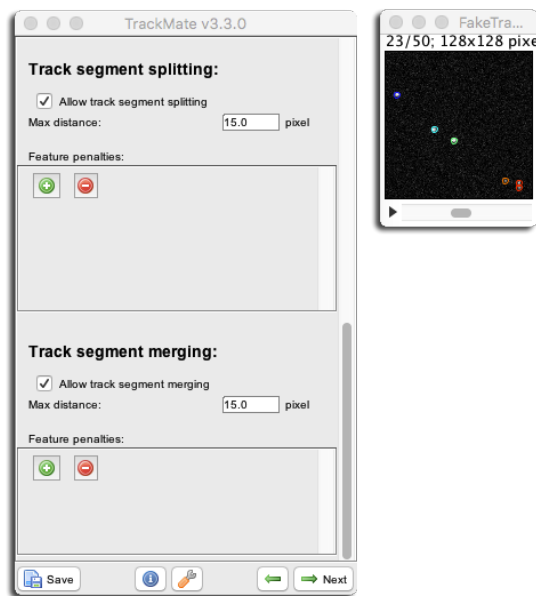
Basically, the tracker held its promises: there is 6 tracks (the two immobile spots at the bottom left part of the image contributed a track each). These tracks are not branching. The red track indeed contains a gap closing event, that did not generate a track break. That would have been different if we would have used the Nearest neighbor search tracker: as it cannot deal with gap-closing events, we would have 7 tracks.

The track colors are yet meaningless; there are just used to facilitate separating different tracks visually.

Now, we would like the shape of these tracks to change. We see that the yellow track is actually branching from the blue one at frame 10. The same goes for the orange track, which branches from the green one at frame 17, and merges to the blue one at frame 27. To deal with that, we need to change of tracker. So go two steps back using the ← **Previous** button and go back to the tracker choice panel. There, select the LAP tracker and move to its configuration panel.




1.14 Configuring a not so simple tracker.




Look at the configuration panel. It is quite more complex than for the simple tracker, obviously, and it is the price for flexibility. Since it is quite long, the panel has to be scrolled to its bottom to venture on all fields. However, this apparent complexity is not that difficult to harness. If you look carefully, you will see that the main panel is made of 4 quasi-identical panel. Each one deals with one event type:

The first one deals with the **frame-to-frame linking**. It consists in creating small track segments by linking spots in one frame to the spots in the frame just after, not minding anything else. That is of course not enough to make us happy: there might be some spot missing, failed detection that might have caused your tracks to be broken. But let us focus on this one now. Linking is made by minimizing a global cost (from one frame to another, yet). The base cost of linking a particle with another one is simply the squared distance between the two particles, which is the adequate cost definition to retrieve particle motion when it exhibits Brownian motion [5]. Following the proposal of Jaqaman *et al.* [4], we also consider the possibility for a particle *not* to make any link, if is advantageous for the global cost. The sum of all costs are minimized to find the set of link for this pair of frame, and we move to the next one. As for the simple tracker, the Max distance field helps preventing irrelevant linking to occur. Two spots separated by more than this distance will never be considered for linking. This also saves some computation time and complexity.

The Feature penalties let you tune the linking cost using some measures of spot similarity. Typically in the single particle tracking framework, you cannot rely on shape descriptors to identify a single object across multiple frames, for spots are "shapeless": they are just described by a X, Y, Z, T position tuple. Yet, you might know your Biology better. For instance, you might be in the case where the mean intensity of a spot is roughly conserved along time, but vary even slightly from one spot to another. Or it might be the spot diameters, or its intensity distribution. Feature penalties allow you to penalize links between spots that have feature values that are different. Since the case you study might be anything, you can pick any feature to build your penalties.

If you want to use feature penalties for frame-to-frame linking, simply press the green  + button in the sub-panel. A combo-box will appear, in which you can choose the target feature. The text field to its right allows specifying the penalty weight. Feature penalties will *change* the base cost. We will not go in the details here (particularly because we are not going to use feature penalties in this tutorial), but basically, two spots with different features will have a linking cost higher than if the selected features values were the same. The weight allows you to specify how much you want to penalize a specific feature difference. A weight of 10 is already very penalizing.

In our case, given the sparsity of spots, we do not need help from the features at all. Remove any penalties you might have added, using the red  - button.

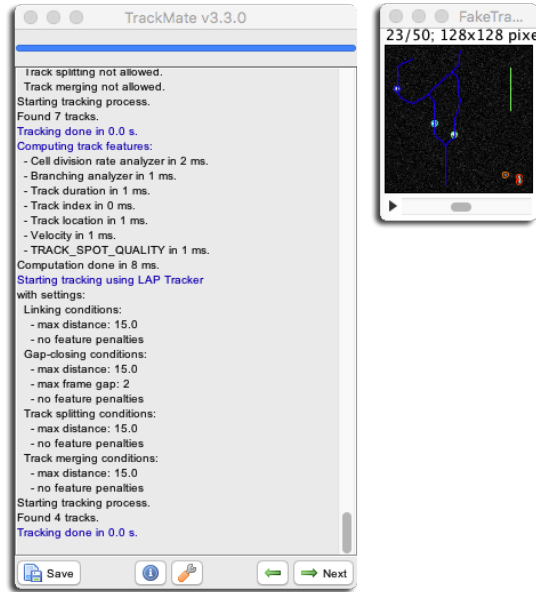
The three other sub-panels deal with the second pass of the linking algorithm, where you take track segments created above and relink them. This **gap-closing** part is already known to you, it is the same as we saw in the previous section: you have to specify a maximal distance, and a maximal frame separation. You can also specify feature penalties, like for frame-to-frame linking. They will be computed on the last spot of the first segment and the first spot of the 2nd segment you are trying to bridge.

The 3rd and 4th panels deal respectively with track splitting events and track merging events.

The mechanisms at play are the same that for the gap-closings: track segments are bridge together depending on the penalties and on the max distance allowed. Track splitting and merging are only allowed from one frame to the next one.

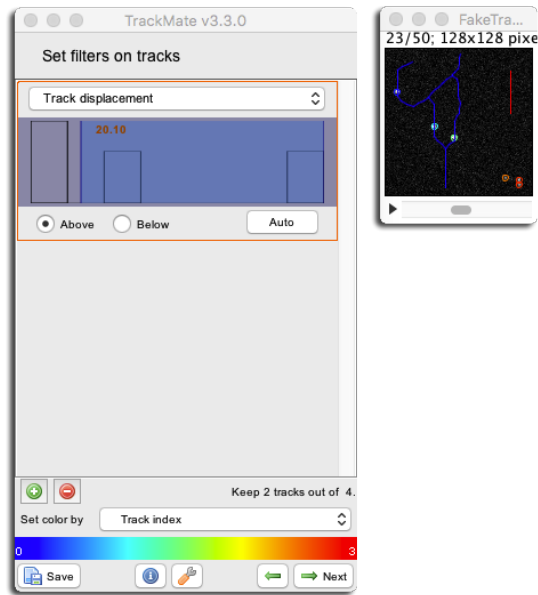
For **track splitting**, the middle of a segment is offered to bridge to the start of another segment. For **track merging**, the end of a segment is offered to bridge to the middle of another one. A check box sets whether you want to forbid or allow any of these events.

As an exercise, try to find the parameters the will fuse the central track segments in a single large track, with two splitting events and a merge event. You should obtain the track layout pictured below.



1.15 Filtering tracks.

The next panel is just the equivalent of the spot filtering step we met before, but this time we use track features. The filter principles are the same: you simply add filters, choosing a target feature, until you are happy with the remaining tracks. As for the spots, the tracks are not really deleted; they are just hidden and you can retrieve them by switching back to this panel and delete the filters.



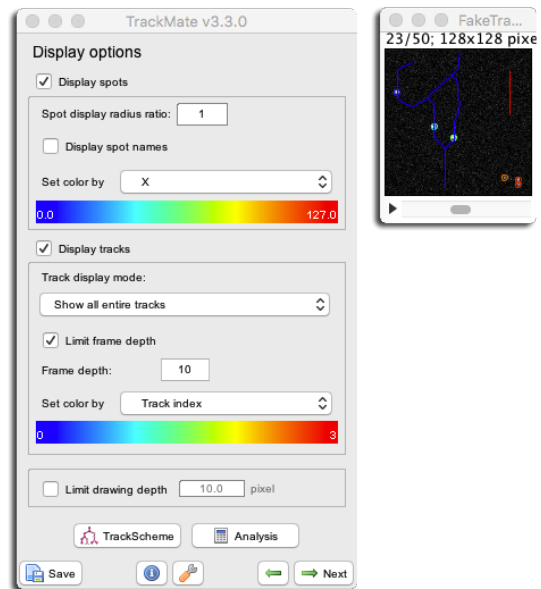
Here, we have a total of 4 tracks. The two immobile spots of the bottom left contribute one track each, that we can barely see because they do not move much. Let us say that we want to get rid of them. There are several ways to do that, but the simple is simply to add a filter on track displacement, as pictured above.

1.16 The end or so.

We are now close to the end of a typical workflow for a tracking problem. The panel you see now is the one that recapitulates display option. You can set spot color by feature, hide them, show their name, etc... Find out what they do, display options are pretty much self-explanatory.

The [TrackScheme](#) button launches a module that allow manually editing tracks, and performing analysis on them. It is the subject of another tutorial.

If you press **Next**, you will see that there is still two panels after this one. The first one allows to plot any kind of feature as a function of another one. TrackMate deals with 3 kind of features: spot, link and track feature, depending on where it makes sense to compute them. For instance, instantaneous velocity is computed over a link (between two spots linked in a track), so you will find it on the Links tab. The **+** and **-** buttons



allow you add several features on the Y-axis, and they will be pooled on the same graph or not, depending on the dimensionality of the features.

The last panel is the Action chooser panel, that allows you to execute simple actions on your data, such as exporting, copying, re-calculating feature, etc...

If you are happy with the results, you can save them now. Loading the resulting file again in TrackMate will bring you to this panel, where you can inspect those results conveniently

1.17 Wrapping up.

That is the end of this introductory tutorial. As you can see, it is quite long. Hopefully that does not mean that TrackMate itself is complicated. We detailed what you could do for the tracking part (the analysis and editing part is still to be seen), but if you recapitulate what we changed from the default, that was pretty simple:

- we set the segmentation diameter to 5;
- we added a spot filter on mean pixel intensity;
- we picked a not-so-simple tracker and allowed for splitting and merging;
- we added a track filter on displacement.

Now that you know how the plugin works, you should be able to reach the end result in less than 30 seconds...

2. Manual editing of tracks using TrackMate.

2.1 Introduction.

This tutorial show how to manually edit, correct and create spots and tracks in [TrackMate](#). You might want to use manual editing to correct mistakes of automated detection or tracking, to do a full manual annotation of a dataset, or to create a "ground-truth" data.

Manual annotation is seldom the most adequate alternative: depending on the size of the target data, it can take an important amount of time and energy, is not objective, and is not reproducible. But sometimes you have to bite the bullet, whether because a detector does not exist for your kind of images, or because it is quicker to manually correct error than to come with the ultimate, flawless algorithm.


Also, tracking is difficult in bio-imaging: images have often by construction a very low SNR, and there is a very wide range of variability amongst experiment types. TrackMate includes generic tracking and segmentation algorithms, and therefore does not exploit the specificity of each problem. It is likely that there are going to be some defects on the difficult use cases you will use it on, and these defects should not stop your science. So there should be a way to manually correct and edit the tracking results. We tried to make it as convenient, easy and quick as possible in TrackMate, should your science requires it.

It is a good idea to be already familiar with the automated segmentation in TrackMate, following the [Getting started with TrackMate](#) tutorial. Here, we will use an incorrect automated

segmentation result, and correct it manually. It is perfectly possible to skip the automated part and to do the whole process manually.

2.2 The test image: Development of a *C.elegans* embryo.

Download the target image here: [Celegans-5pc-17timepoints.zip \(34 MB\)](#). It is a rather large file, so it is zipped. Unzip it somewhere; you should end up having a single tiff file, *Celegans-5pc-17timepoints.tif* of about 100 MB.

Open it in Fiji. You will get a stack, made of 41 Z-slices over 17 time-points, each image being 240 x 295. As you can see in  (Shift + P), it has a spatial and temporal calibration.


The context is the following: We used a *C.elegans* strain named [AZ212](#) that has its histone H2B coupled to the eGFP. The nuclei can therefore be seen in the 488 nm excitation fluorescence channel. The movie started just after the first cell division, so you can see on the first frame two blob-like spots in the center of the egg. On the top-right part of the egg, there is also two smaller spots that are the polar bodies. One will remain at a fixed place, the other one will be pushed around as the cells divide. The movie has 17 time-points that span the first 34 minutes of the *C.elegans* embryo development.

[We](#) were trying to assess the impact of phototoxicity on development in this movie. We used a rather strong laser power, which explains the relatively good quality of the image, and the fact that this egg's development is slowed down compared to a classical development at 21°C. On this movie, we manually erased another egg that was lying on the top left-corner of the image, which makes it suitable only for educational purposes.

We want to reconstruct the cell lineage from this movie. Ideally, we will end up in having four tracks: two for the the two cells present on the first frame, and two for the polar bodies. The cell tracks will be branching, following cell division.

This tutorial uses the following strategy: we will use an inadequate set of segmentation parameters to simulate defects in segmentation. Then we will use a tracking algorithm that does not take into account the possibility for a cell to divide, and will not use the spot feature to make linking robust, thus generating linking defects. Finally, we will learn how to correct these defects manually.

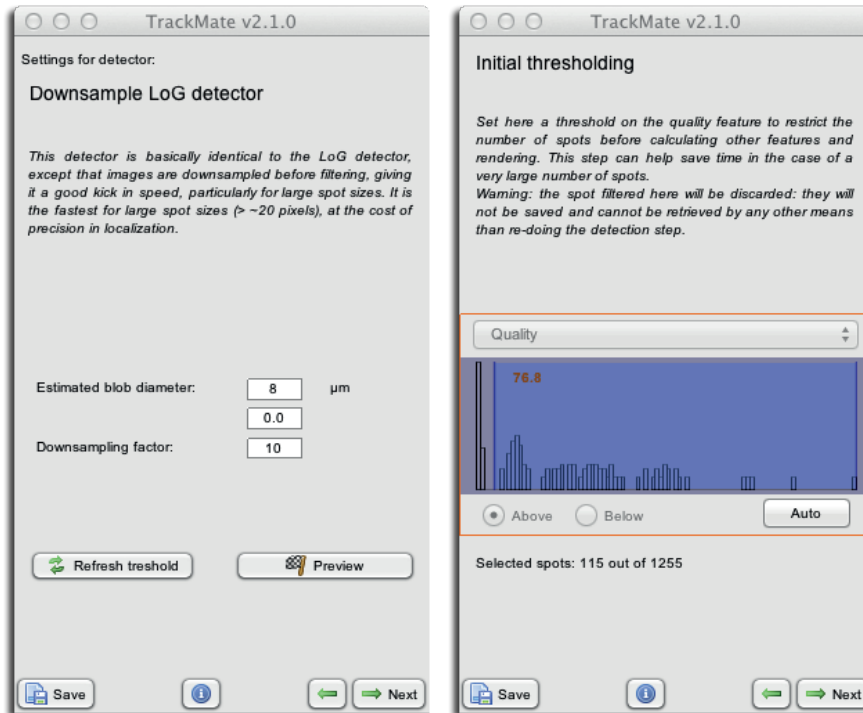
2.3 Generating a sub-optimal segmentation.

Launch TrackMate  and select the *C.elegans* stack as a target. Check on the first panel that all the spatial calibration is OK. The pixel size is about 200 nm in XY, 1 µm in Z, and each frame is separated by 2 minutes.

Select the Downsampled LoG detector. This choice actually makes sense: the nuclei are about 8 µm in diameter, and with a sampling of 200 nm, that makes 40 pixels wide nuclei. It is already advantageous to use the down-sampled version of the LoG segmenter above 20 pixels: segmenting 3D data over time takes already quite some time. Having to track large objects allow to downsample them, making the data to iterate over smaller, which speed up the process.

This comes at a cost: the localization precision. To simulate segmentation defects, we will make it very bad. In the segmenter configuration panel, choose a down-sampling factor of 10

(a factor of 4 would have been wiser), and a target nuclei radius of 8 μm , as depicted below.



The segmentation should take you no more than a minute, even on a standard machine, a considerable improvement over a standard detector. But at what cost!

On the Initial thresholding panel, we see that it is easy to separate spurious spots using the Quality feature only. There is a big and sharp peak at the left of the histogram. By moving the slider around you can get the remaining number of spot after filtering. If we put the threshold around 70, just above the first sharp peak, we see that we are left with about 115 spots. Now: We have 17 time-points, each of them containing at most 4 cells and two polar bodies (check the raw movie). So 115 remaining spots seems to be correct, therefore the threshold set at 70 seems right.

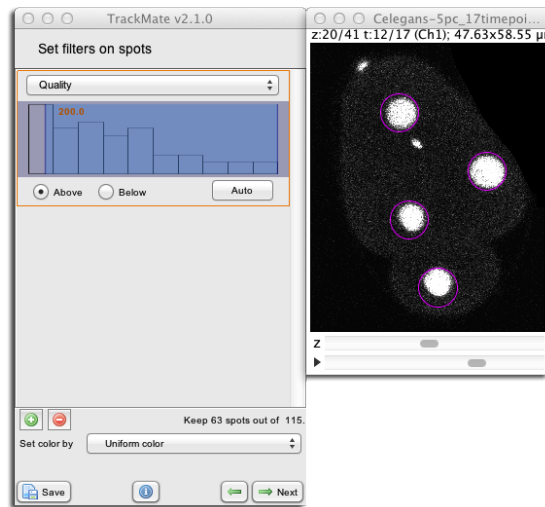
Now move to the next panel. On the *Displayer choice* panel you must pick the HyperStack Displayer. It is the only one that allows for manual editing.

On the *Spot filters* panel, the results of the detection will appear. It is likely that our results are the same if you used the same numerical values as above. And you can see that there are still a lot of spurious spots amongst the remaining 115. We were wrong when we selected the initial threshold.

Anyway, let's correct it now. Just add a filter on Quality, and take a value of 200; 63 spots should remain, and the spurious ones should disappear.

Almost all polar bodies are incorrectly detected, and the localization of cells is bad. These are expected defects given our choice of detection algorithm and the parameters we have used. Here, the results are not so bad, unfortunately for this tutorial. We could fix them right now, before tracking. You can actually edit the results any time after the first panel of TrackMate. But let us exploit these defects for our training purpose, by having them generating additional

linking defects.



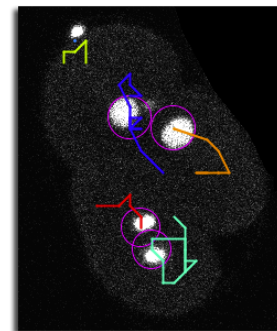
2.4 Generating irrelevant tracks.

Normally, TrackMate can robustly handle track splitting events, representing *e.g.* cell division. Though this happens in this movie, we choose to dismiss this possibility in the automated tracking part.

In the *Tracker choice* panel, select the Simple LAP tracker. Leave the default values for the parameters, and press **Next**. You should end up in having something similar to the image to the right.

A quick assertion shows that for nuclei, the individual track branches are not so bad. Each of the retained spot could use some fine tuning, but the tracks are not completely pathological, disregarding the fact that splitting events are missed. The polar bodies tracking results are hopeless.


This is what we will now manually correct.



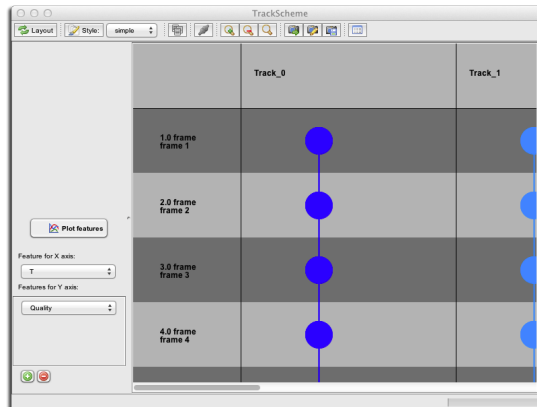
2.5 Launching TrackScheme.



Move to the *Display options* panel, skipping the track filtering part.

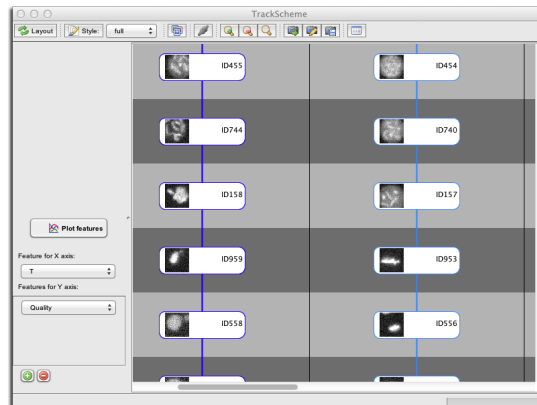
[TrackScheme](#) is a TrackMate tool for the visualization and editing of tracks. It displays a kind of "track map", where a track is laid on a pane, arranged vertically over time, as a Parisian subway train map. Tracks are displayed hierarchically, discarding the spatial location of each spot. Each spot is laid out going through time from top to bottom. It is a great tool particularly to study and edit lineages.

[TrackScheme](#) also allows manually editing the tracks. Press the  **TrackScheme** button on the last panel. By default, the tracks are displayed as colored circles joined by lines. Each

circle represent a spot, and the lines represent a link connecting two dots. The selection in TrackScheme is share across TrackMate, so if you select one circle, it will be highlighted in the HyperStack viewer as well (circled in green).

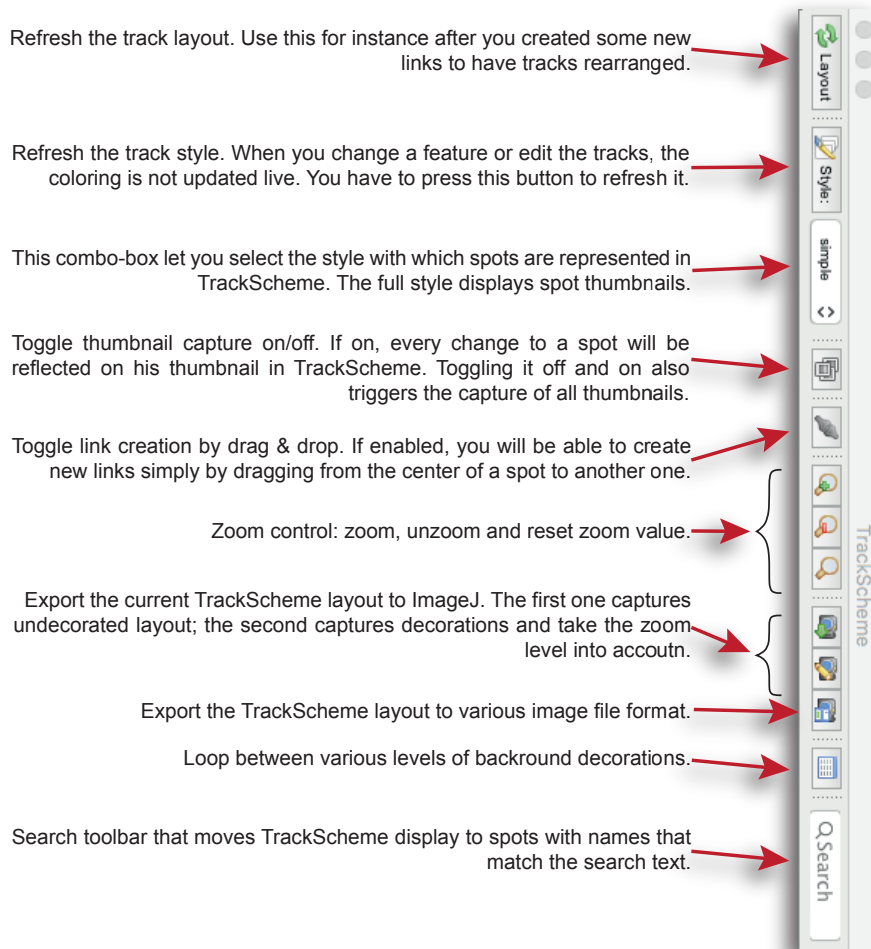



TrackScheme launches with a simple style: each spot is represented with a circle. You can get more information by changing the style. Next to the  **Style** button in the TrackScheme toolbar, there is a combo-box in which you can select either `simple` or `full`. Select `full`. Each spot is now represented by a rounded rectangle, with the default name printed on the right. Go back in the TrackScheme toolbar. Next to the style combo-box, there is a greyed-out button showing three small images . Press it; after some time, each spot in TrackScheme will contain a thumbnail of the spot taken in the raw image. This is very handy to quickly detect detection problems.



2.6 TrackScheme in a nutshell.


You can do quite some things using TrackScheme, notably track analysis. This is not the focus of this tutorial, we will simply be focusing on the track editing features. However, here is a brief description of what the toolbar buttons do.



We will be mainly using the  **Redo layout** button.

2.7 Getting rid of bad tracks.

We will first start by removing all the bad spots and tracks. We decide not to keep the tracks generated by the polar bodies, and only to keep the tracks that follow the two nuclei.

- Move to the Track_3 column in TrackScheme. You can see that it is following the static polar body.
- We wish to select it at once. There is two way you can do it:
 - Draw a selection rectangle around the whole track representation.
 - Select one spot or link in the track. Right-click anywhere on TrackScheme: a menu appears, in which you will find  **Select whole track**.
- Notice in the displayer that the selected track appear with a green and thick line, so as to highlight it.

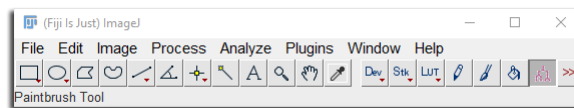
- To delete all of it, simply press the `Delete` key in TrackScheme, or use the right-click menu to do so.

Do the same for Track_1, since we do not care for polar bodies.

Press the **Redo layout** button when you are done. There should be four tracks remaining. Notice that their color changed as you deleted some of them. Their default color-map goes from blue to red and is re-adjusted every time the track number changes.

2.8 Spot editing with the HyperStack Displayer.

We now wish to correct for segmentation mistakes, that caused some nuclei to be missed. Creating new spots is made directly in the HyperStack displayer. First, make sure the TrackMate tool (represented by a blue track over a green background) is selected in the ImageJ toolbar, as pictured below:



The HyperStack displayer let you edit spots in two ways:

2.8.1 With the mouse.

Moving an existing spot.

- `Double-click` *inside* a spot on the displayer to select it for editing. It becomes green with a dashed line.
- Click and drag inside the selected spot to move its center around. To move it Z or in time, simply move the sliders at the bottom of the window and the spot will follow (shortcuts: `<` & `>` for Z, `Alt`+`>` & `Alt`+`<` to move in time).
- When you are happy with its new location, double-click anywhere to leave the editing mode. You should notice that its thumbnail in TrackScheme gets updated.

Creating a new spot.

- Double-click on the image outside of any spot.
- A new spot is created, and is selected for edition.
- The previous remarks apply to change its location.

Deleting an existing spot.

- Select a spot by single-clicking inside it. It turns green.
- Press the `Delete` key.

Changing the radius of a spot.

- Select a spot for editing by double-clicking inside it.
- By holding the **Alt** key, rotates the wheel button. This will change the spot's radius.
- Holding **Shift** + **Alt** changes its radius faster.
- **Double-click** anywhere when you are happy with the new radius. The spot thumbnail in TrackScheme gets updated.

2.8.2 With the keyboard.

I have found using the mouse clicks sub-optimal and painful for the carpal bones when editing a lot of spots. Using a combination of the mouse and keyboard proved to be more efficient. For this to work, the HyperStack window must be selected.

Moving an existing spot.

- Lay the mouse over the target spot (you do not need to select it).
- Hold the **Space** key.
- Move the mouse around. The target spot follows the mouse location until you release the **Space** key.

Creating a new spot.

- Lay the mouse anywhere on the image.
- Press the **A** key.
- A new spot is added at the mouse location.

By default, the new spot has the radius of the last spot edited with the mouse. So if you want to set the default radius, just double-click inside a spot that has the desired radius, then double-click again to leave editing mode. From now on, the radius of this spot will be used by default.

Deleting an existing spot.

- Lay the mouse over the target spot.
- Press the **D** key.
- The target spot is deleted.

Changing the radius of a spot.

- Lay the mouse over the target spot.
- Press the **E** key to increase its radius, **Q** to diminish it.
- **Shift** + **Q** and **Shift** + **E** change the radius by a bigger amount.

2.9 Adding missed spots.

You now have the tools to correct the mistakes our crude detection made. Unfortunately for this exercise, there is just one: At the last time-point the leftmost cell just divided, but one of the daughter cell has been missed.

Notice that when you add spots, they also appear in the TrackScheme window as a magenta cells, on the far right in a lane without track number (under the column `Unlaid spots`). As long as you do not incorporate them in a track, they will remain there.

Also, if you feel courageous, you can improve the look of your TrackScheme layout by adjusting the spot radii. There is however an automated way to do so, which will see later.

You should end up in having a corrected segmentation, where every nuclei correspond to one spot in TrackMate, and it at the right location. Time yourself doing so, so that you learn how much you have to invest on manually correcting segmentation results, and decide if it is acceptable.

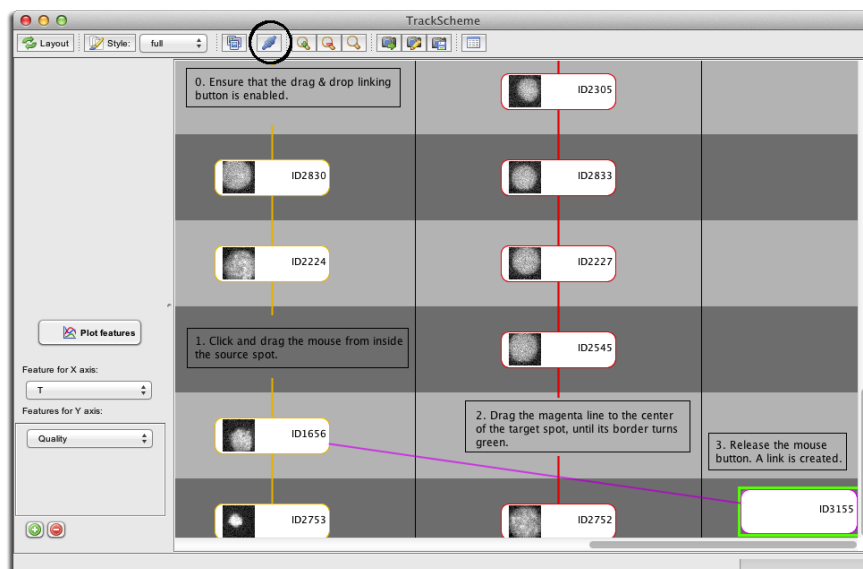
2.10 Editing tracks: creating links.


Now we want to connect the lonesome spots to the track they belong to. This is all about creating links, and there are two ways to do that.

2.10.1 By drag & drop.

You can create a link between two cells in TrackScheme simply by enabling the linking button on the TrackScheme toolbar, and dragging a line between the source cell and the target cell.

This is pictured below, where the fore-to-last cell of the track 4 is connected to the new spot. For visibility on this screenshot, we brought the target cell closer to the lane of the track 4. You can normally find it either on the far right of the panel.



Press the  **Redo layout** button to see the arranged result. The first spot is now incorporated in the right track.

2.10.2 Using selection and right-click menu.

The cell in TrackScheme cannot be easily moved. When the source and target cells are far away, it might be better to use another way to create links. We do this using the shared selection.

In TrackScheme, find the first spot of Track_4, in frame 9. When you click on the corresponding cell in TrackScheme, it gets highlighted in green. In the HyperStack viewer, the displayed slice and time points are changed to display the spot, also highlighted in green.



We want to link this cell to the mother cell in Track_0, frame 8, just before it divided. To do so,

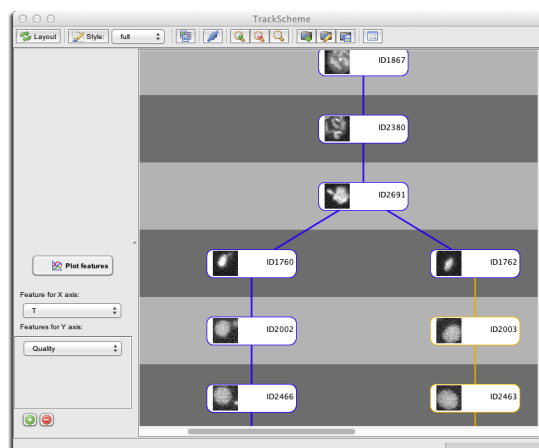
- In the HyperStack displayer, move to the frame 8.
- Hold the Shift key.
- Click on the mother cell.

It gets highlighted in the displayer, and in TrackScheme as well. You now have two cells in the selection.

To create a link between the two,

- Right-click anywhere in TrackScheme.
- In the menu that pops-up, select Link 2 spots.


The newly created link is displayed in magenta. Note that the track arrangement is not changed; you need to press the  **Redo layout** button to rearrange the tracks. After doing so, you should now see a branching track, as picture below. Notice that the track colors are out of sync. The colors are not automatically updated when changing a track layout. You have to click the  **Style** button in the TrackScheme toolbar to do so. Do so.



2.10.3 Creating several links at once.

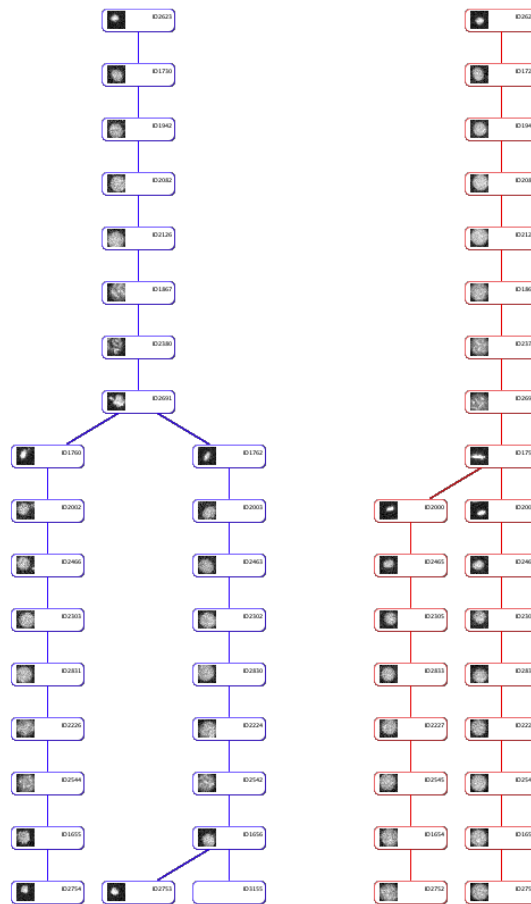
Using **Shift** + **click**, we can put several cells in the selection, and create the links between each pair. We don't have the need for it, but this is a good way to create a single track from several solitary spots: Just select them all (dragging a selection box or **Shift** + **click**) and select the **Link N spots** menu item.

2.11 Editing tracks: deleting links.

We do not have much to say here. The tracks we generated had missing links, but no spurious ones. So we do not need to remove any. But here is how to do it: In TrackScheme, select the target link by clicking on it; it gets highlighted in the displayer as well. Press the **Delete** key to remove it. Removing a link often splits a track in two new tracks. To have them properly re-arranged, press the  **Redo layout** button.

2.12 Wrapping up.

Plus or minus the localization errors and some incorrect cell radii, you now have the full lineage in 3D of this short movie. This concludes this tutorial on manual editing in TrackMate. Here is a picture of the final results:



3. Manual and semi-automated tracking with TrackMate.

The previous TrackMate tutorial - [Manual editing of tracks using TrackMate](#) is dedicated to manually correcting the results of an automated process. This small tutorial here shows how to do a fully manual annotation, from scratch, with [TrackMate](#).

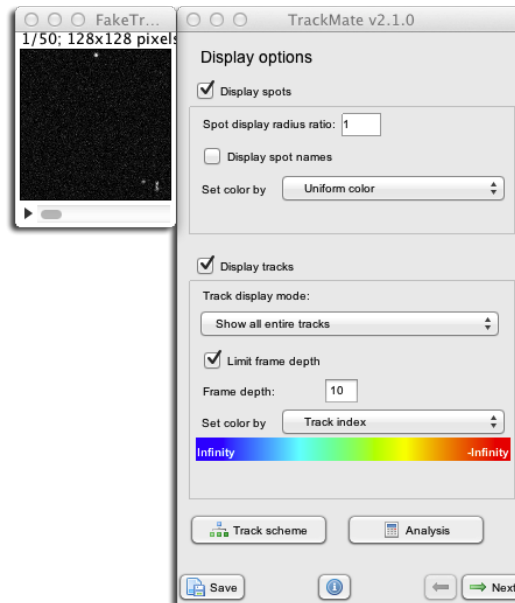
3.1 Setting up.

We will use the same, simple dataset that for [Getting started with TrackMate](#). You can find it in `File >> Open Samples >> Tracks for TrackMate (807K)`.

As for the TrackMate plugin, you could start it up normally, selecting `Plugins >> Tracking >> TrackMate` in the menu, and then when offered to select a detector and a tracker, always pick the manual one. That would work well, but we offered another entry point that has a simpler GUI dedicated to manual tracking. Pick the `Plugins >> Tracking >> Manual tracking with TrackMate` menu item.

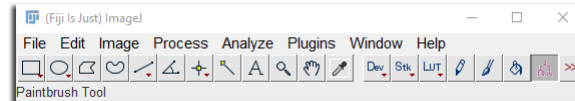
You should should get the layout pictured below. Notice that we are already displaying the Display options panel of the classic GUI, and that the **← Previous** button is disabled at the

bottom. Notice also that the color scales for both spot and track features display a dummy range.



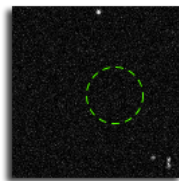
3.2 Creating spots one by one.

The main view (the one that re-uses the HyperStack viewer of ImageJ) can readily edit the tracks. You just have to make sure that the TrackMate tool is selected in the ImageJ toolbar:



With this tool selected, you can now make the image window active and use the mouse or the keyboard to create spots. Here are the commands for the mouse:

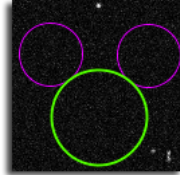
- **Double-click** anywhere in the image to create a spot and enter the edit mode. The edited spot is highlighted with a green, dashed circle, as pictured below:



- To leave the edit mode, **Double-click** again anywhere. The spot is then added to the data model.
- To edit it again, **Double-click** inside the spot. Its outline is now dashed; you are back in the edit mode.

- While in the edit mode, you can move the edited spot around by clicking inside the spot and dragging it around. The spot will follow you if you change the time or the Z slider, and it will be added to the right plane upon leaving the edit mode.
- You can also change its radius by using `Alt + Mouse wheel`. Using `Shift + Alt + Mouse wheel` changes the spot radius faster.

This is how you edit the data with the mouse. You can also use the keyboard:



- To create (or add) a spot, press `A` with the mouse at the desired location. By default, the new spot will have the radius of the last spot you edited with the double-click mode. So if you want to have all spots of a certain radius, edit a spot by double-clicking inside it, set its radius using `Alt + Mouse wheel`, and leave the edit mode. This will "capture" the spot radius and apply it anywhere after.
- To move a spot around, press `Space` with the mouse over the target spot. Then move the mouse around. No need for mouse clicks.
- To delete a spot, press the `D` key with the mouse over the target spot.
- To change a spot radius, press `Q` and `E` over the target spot. `Shift + Q` and `Shift + E` change the radius by a larger amount.

And that's it for spot creation.

3.3 Create and removing single links.

All we have done so far was to create single spots, that are not part of any tracks. Tracks are created on the fly when you link several spots together. You can do it in [TrackScheme](#), as explained elsewhere. Here is how to do it directly on the image. To go on, create a few spots above the bright blob of the source image. We need at least a couple of them in consecutive frames.

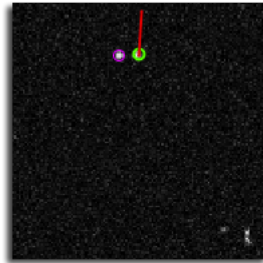
We need to add spots to the selection The selection in TrackMate is a very useful tool for inspection, particularly because it is shared amongst all the possible views of a session, including e.g. [TrackScheme](#). When you click in a spot, the selection is made of this spot, and all views are centered on the target spot.

To create a link, we need exactly two spots to be in the selection. To add or remove a spot from the selection, use `Shift + click`. Selected spots are highlighted with a green, thick circle. To empty the selection, click on an empty (no spot) part of the image.

Once you have two spots in the selection, you can create a link between them by simply pressing the `L` key. It should be immediately displayed, as on the example below.

As you can see, there is nothing that prevents you from creating a link over many frames, between any two spots. A spot can have several link arriving or departing from it. The only impossible things is to create a link between two spots that belong to the same frame.

Removing a link is done the same way: Select exactly two spots that are connected by a link, and press the **L** key. The link will be removed.

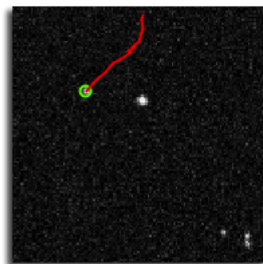


3.4 The auto-linking mode.

Creating long tracks this way would be tedious, as you would always have to select a spot before creating a link. There is way to simplify this.

Press **Shift+L** to toggle the auto-linking mode on/off. When the mode is on, new spots will be automatically created to the spot in the selection (if there is only one). Then the selection is set to be the newly created spot, which allows you to quickly trace tracks by moving through frames and pressing the **A** key over the desired location. When the auto-linking mode is activated, spots can not be deselected by clicking in an empty are of the image anymore.

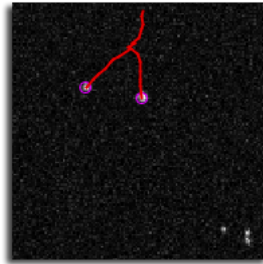
Let's apply this to our data. First create a spot over the bright blob at the top of the first frame, and roughly adjust its radius. Make sure the selection contains this spot, and only it (it must be highlighted in green), and press **Shift+L** to toggle the auto-linking mode on. Then move the second frame and place the mouse over the new spot location. Press **A**; a spot is created AND it is linked to the first spot by a track normally painted in red. Repeat until you reach about the frame 15 (the track branches, at some point, you have to decide what way you want to go). You should get - rather quickly - something like the picture on the right.



Tracks created this way do not have to be linear. You can create branching segments simply by remembering that in the auto-linking mode, links are created between the last selected spot.

Therefore, to create the branch that goes on the right, go back on the frame 9 (the frame just before the branching happens) and click into the spot that's there to select it. Then move

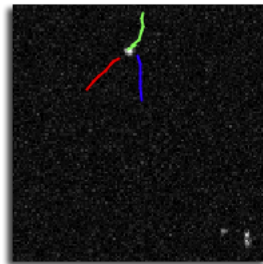
to the next frame and create the spots that belong to the right branch, just like you did before. These spots will be added to the same track, and should get a inverted Y-branch like pictured below.



3.5 Tracks are updated live.

Note that you do not have to worry about what track a spot belongs to when creating a link. Tracks are automatically managed on the fly. If you now create a second link between a pair of spots that are not connected with anything, a new track will be created automatically, and the color of the first ones will change.

The same is valid when you delete a link or a spot. For instance, let's create 3 tracks out of our inverted Y. Go to the frame 9, and delete the spot that is at the crossing. You now have three tracks.



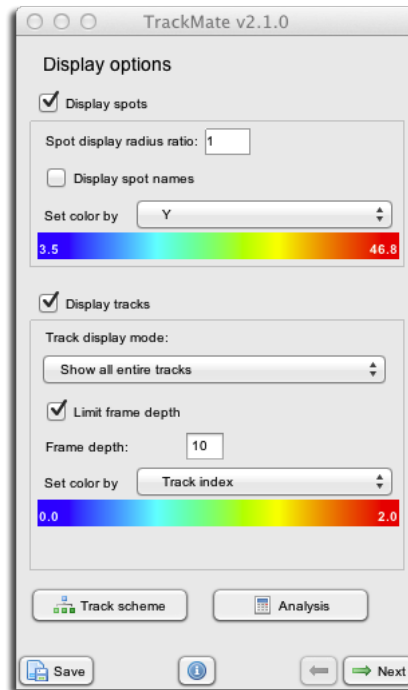
3.6 Track and spot features are updated live.

TrackMate uses computes and uses some numerical features for its spots, edges and tracks. You can use these features to color the TrackMate objects.

For instance this is what happened in the previous section, when you deleted a spot and created 3 tracks out of one. On the GUI panel, the tracks were configured to have their color set by the track index. When you removed the spot, the track index was recalculated and used to give the track a color that ranges from blue to red.

The track and spot colors are refreshed immediately in the HyperStack displayer. Note, though, that the color range in the GUI has not been updated. It still displays $-\infty \rightarrow \infty$. This is by construction, to alleviate a bit the load when editing large models. If you want to refresh the color range, you have to click directly on it, and it will be properly repainted. You can see below what we get if we pick the feature Y for spot coloring, and the

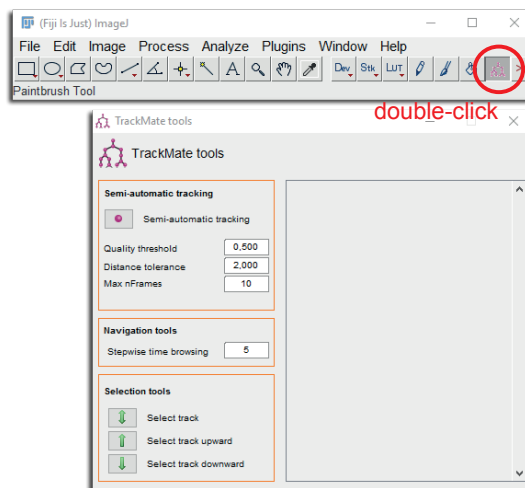
track index, after refreshing.



3.7 Step-wise time browsing for sparse annotations.

For images with many time-points, it might be desirable to generate a quick manual annotation of it by skipping some frames in the tracks. For instance, if the particles you track do not move too much from one frame to another, some frames may be skipped without confusion. Two keyboard shortcuts allow to jump from time-step to time-step: **F** and **G** will step in time by multiple of 5 frames (by default). Note that time jumps to multiples of 5: with **F** and **G** you always access the frames 1, 6, 11, *etc* regardless of your current frame number. This ensures that if you pick this technique to annotate an image, all the spots you add will be in the same frames, which is handy for visualization.

The time step can be changed in the TrackMate tools window. This is a special window useful for manual annotation, that you access by double-clicking on the TrackMate icon in the ImageJ toolbar. This window has several uses, described below. The step size for time-step browsing is defined with the Stepwise time browsing numeric field.



3.8 The semi-automatic tracking tool.

TrackMate includes a tool that can automatically find spots and automatically link them to build a track. This is extremely handy to annotate images for which the automated detection in bulk yields too much spurious spots. This tool is configured in the TrackMate tool option panel, pictured above.

We are interested in the Semi-automatic tracking panel. The bottom panel has just convenience buttons that allow you to select tracks or parts of tracks from the current selection (great to delete faulty tracks at once), and the right panel is a log.

The semi-automatic tracking tool itself works as follows: It takes the single spot in the selection, and uses its radius to build a neighborhood of this spot, but in the next frame. It then searches this neighborhood for a bright blob with a similar radius. If the found spot is close enough and has a quality high enough, it is linked to the first spot. The process is then repeated, until no suitable spot can be found or until there is no time-point to inspect anymore.

The Quality threshold and the Distance tolerance settings allow to tune the tool. A quality threshold of 0.5 means that the spot found must have a quality of at least 50% the quality of the first spot to be accepted. A distance tolerance of 2 means that it must not be further away than twice the radius of the previous spot to be accepted. Max nFrames sets a limit to the process. Even if successful each frame, the semi-automatic tracking will stop the specified number of frames have been iterated. When the process stops, the reason is printed in the log window.

Let's put this in practice. Go to the frame 16 (or wherever you stopped annotating in the previous section), and select the last spot of the right track. You can start the semi-automatic tracking by either clicking on the purple button on the tool panel, or by pressing the **Shift** + **A** key. The tracking process is updated live. How far can it go really depends on the radius you set for the first spot, so results may vary.

3.9 Keyboard shortcuts for manual editing of tracks in the main view.

We recapitulate here the keyboard shortcuts we used above. Note that they are only functional when the main view is active and the TrackMate tool selected in the ImageJ toolbar.

--decrease radius
 --increase radius
 - create a spot
 - +Shift: auto tracking
 - remove a spot
 step-wise time browsing backward/forward
 remove selection

Esc	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	
~	!	@	#	\$	%	^	&	*	()	-	=	→ Delete
Tab	Q	W	E	R	T	Y	U	I	O	P	{	}	
Caps Lock	A	S	D	F	G	H	J	K	L	:	"	Enter	
Shift	Z	X	C	V	B	N	M	<	>	?	,	/	Shift
Ctrl		Alt							Alt		Menu	Ctrl	


change radius faster/slower
 hold space inside a spot:
 move the spot around
 - link two spots in the selection
 - +Shift: toggle auto-linking mode
 +Shift: remove spots from current frame and copy spots from previous frame

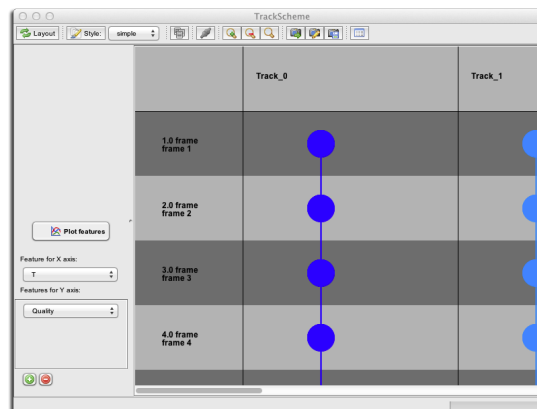
Part II.

Technical documentation.

4. TrackScheme manual.

TrackScheme displays a kind of "track map", where a track is laid on pane, arranged vertically over time, as a Parisian subway train map. In TrackScheme, the image data as well as the spatial location of spots are completely discarded in favor of a hierarchical layout that highlights how cells divide in time.

In the display settings panel of TrackMate, click on the  **TrackScheme** button. A new window should appear, and depending on what you tracked, its content resembles this.



In TrackScheme, tracks are arranged from left to right and time runs from top to bottom. At this time we just have a single track, with two branches. The cell we tracked divides immediately after the first time-point, which is represented in TrackScheme by a fork going down. Each branch below this fork represents the annotation of a daughter cell. However, all the spots and links for these two daughter cells still belong to the same track, as they are connected *via* the mother cell.

Though this view is very synthetic, there is a lot you can do with TrackMate.

4.1 Moving around in TrackScheme.

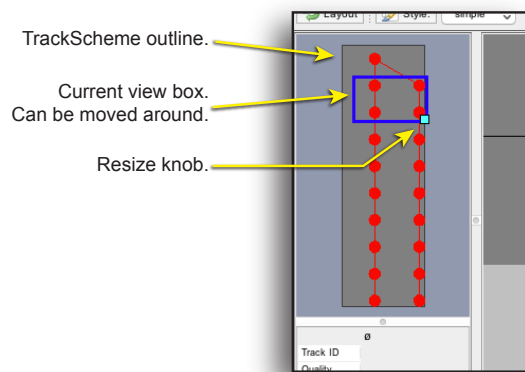
Moving around is done classically with the mouse, and the panning is triggered by holding down the `Space` key:

- `Mousewheel` scrolls up and down.
- `Shift` + `Mousewheel` scrolls left and right.
- `Space` + `Mousedrag` pans the view, à la ImageJ. If you pull the mouse out of the TrackScheme window, it will scroll in the direction of the mouse cursor.
- `Space` + `Mousewheel` is used for zooming.

The keyboard can also be used:

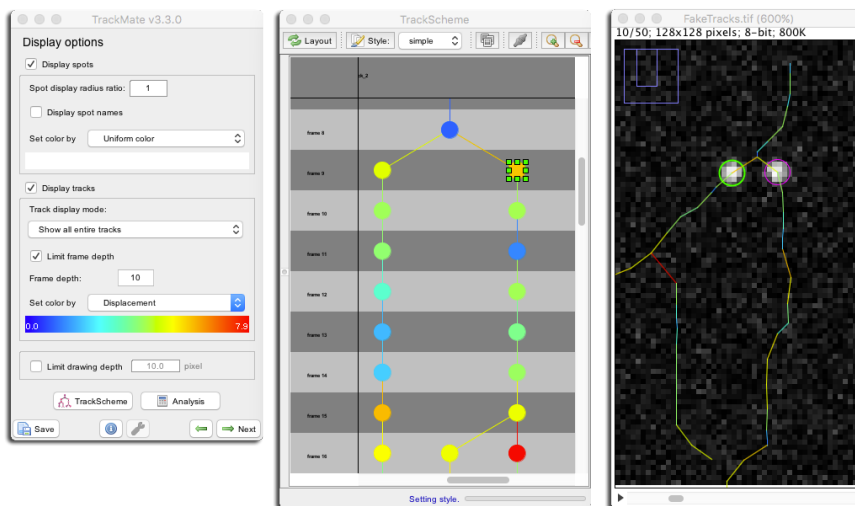
- The numeric keypad numbers , , , , , , and are used to move as on a compass.
- zoom in.
- zoom out.
- restores the zoom to its default level.


The top-left part of the TrackScheme window shows the outline of the graph. The blue square represents the current view and can be resized and moved around.

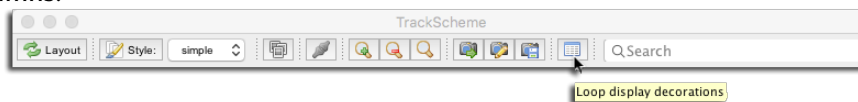



4.2 Configuring TrackScheme look.

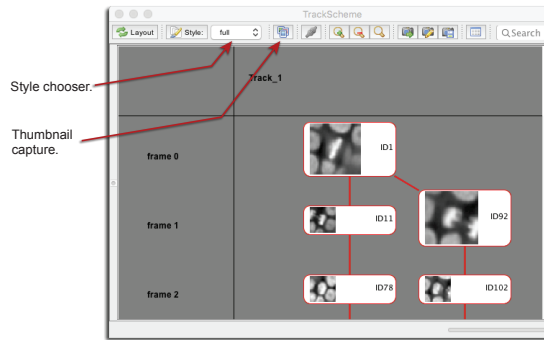
Though TrackScheme is a view of the annotation data, it completely and purposely ignores some the display settings you can set on the main GUI window, such as the track display mode and the global visibility of spots and tracks. The color it chooses for the links and spots representation is also peculiar: The spot color by feature mode is ignored, even for the circles that represent spot. They take their color from the track color mode, and use the color of the incident link. For instance, if you pick the Displacement feature in the **track color mode**, you will get this:



Tracks have a name, and are arranged in columns, separated by a vertical black line. TrackScheme arranges the annotations line by line, and each line represents a time-point. The row header tells you what time-point you are looking at. The background color of each row alternates to highlight different frames. If you find the background too crowded, you can disable the alternating color by clicking on the  **Display decoration button** on the toolbar. The second mode disables track columns and rows alternating colors; the third mode re-enables track columns.

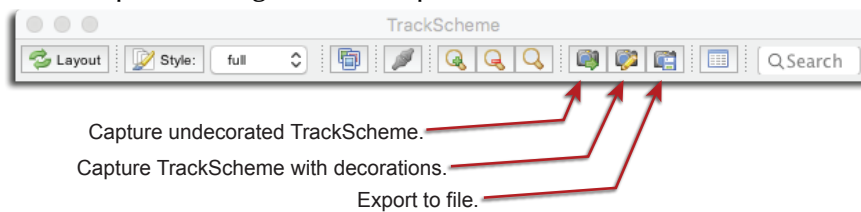





Finally, there is two  **Styles** for the spot display. The *simple* style only displays them as round spots. The *full* style displays them as rounded boxes, with each spot name apparent. In the *full* style, small thumbnails can be captured and displayed in TrackScheme for all spots. Just next to the menu, there is a thumbnail button. If activated, thumbnails are collected from all spots, using the image source they were created on. Thumbnails are captured around the spot location, using their radius plus a tolerance factor. Interestingly, the **Spot display radius ratio** is used to define the size of the thumbnail. For instance, with a display factor of 2, you can obtain the layout below. Notice that the spot boxes can be resized manually to better display thumbnails.



4.3 Exporting TrackScheme display.

The hierarchical layout of the lineages provided by TrackScheme can be useful for communications. It can be exported using the three export buttons in the toolbar.



- The  **Capture undecorated TrackScheme** button will generate a view of TrackScheme and open it in Fiji. The background is set to white and the zoom level is set to the default, regardless of what the actual zoom is in TrackScheme. Once this image is in Fiji, you can modify it, save it, *etc* using the tools in Fiji.
- The  **Capture TrackScheme with decorations** button does the converse. It captures a snapshot of the TrackScheme window as is, and uses the current zoom level to generate an image.
- The  **Export to...** opens file browser on which you can pick the export file format and its location. Many file formats are supported:
 - PNG image file with/without transparent background.
 - PDF or SVG file, that can later be edited with *e.g.* Illustrator.
 - As a HTML page, though the layout is somewhat simplified.
 - The now deprecated VML file format (replaced by SVG).
 - As text, but this only saves a minimal amount of information.
 - The MXE file format is a specialized XML format, that can be parsed with classical XML parsers.
 - And all the common image formats (PNG, JPEG, GIF, BMP).

4.4 Managing a selection in TrackScheme.

TrackScheme is useful to build a selection and query its properties. As we said above, TrackScheme does not abide any visibility setting. Spots and links are always visible, which is useful to build a selection. Spots and links are added to the current selection in a classical way:

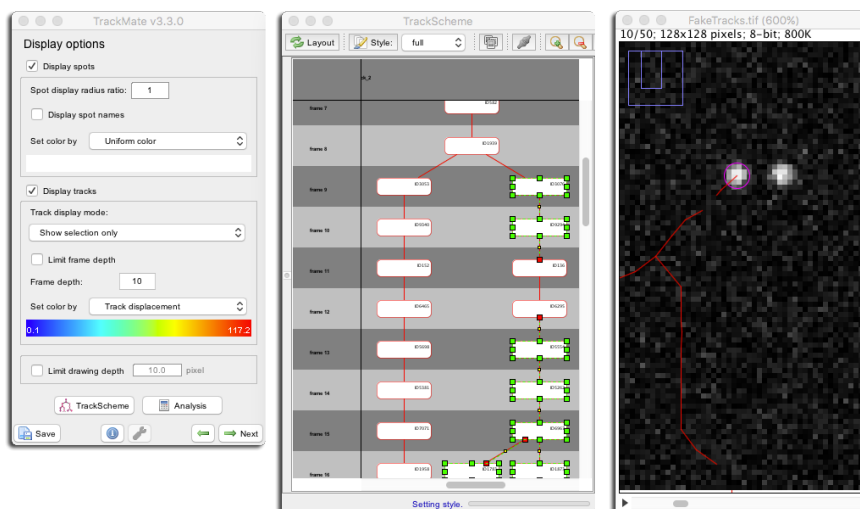
- **Left-Click** on a spot or link to set the selection with this spot or link. The selection is cleared before.
- **Left-Click** outside a spot to clear the selection.
- **Shift + Left-Click** on a spot or link to add or remove this spot or link to the selection.
- **Mousedrag** to select multiple spots and links in a selection box. Hold **Shift** to add them to the current selection.

Adding to this, several items in the **Right-click** popup menu help selecting part of tracks. If you **Right-click** on a spot or **Right-click** outside a spot with a non-empty selection, you can:

- **Select whole track** will include all the spots and the links of the tracks the selection belongs to.
- **Select track downwards** walks from the spots and links in the selection, and add the spots and links that connect from them, forward in time (downward in TrackScheme).
- **Select track upwards** does the same, but backward in time.

Selections are very useful for visualization within a crowded annotation. For instance, select one of the two branches in our single track. With the default track display mode, the selection is drawn on the MaMuT viewer as a thick green light that extends fully in time. The eighth track display mode is called **Show selection only** and just does that. It displays in the main view only the spots and links in the selection, with their proper color settings, and abide to the frame and depth limit settings.

For instance, you can use it to only display a series of disjoint parts of a tracks:



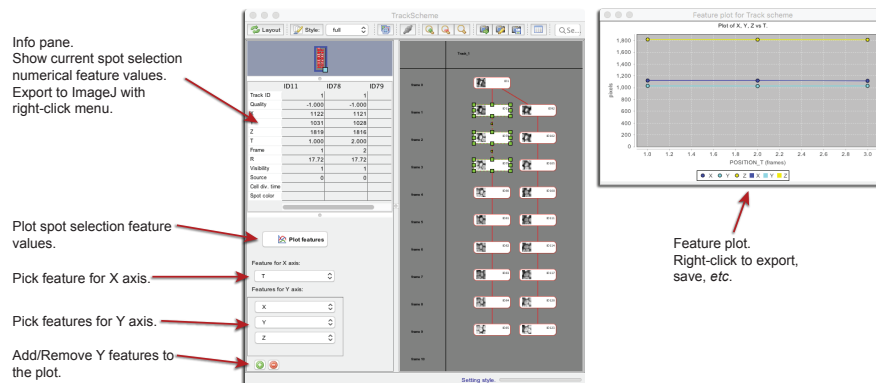
4.5 TrackScheme info-pane and feature plots.

Another use of the selection is to display, plot and export information on its content. The left side bar of TrackScheme has two small panels dedicated to this, in addition to the outline panel in the top left.

The info pane in the middle left takes the shape of a table, that displays the numerical feature values of the spot selection as a table. Spots are arranged as columns and feature as lines. This table can be exported to an ImageJ table with the **Right-click** popup menu.

The bottom left part of the spot feature plotter. The **Feature for X axis** drop down menu lets you choose what will be the feature used for the X axis. **Feature for Y axis** menus work the same way. Y-axis features can be added and removed using the **+** add and **-** remove buttons.

To generate the plot, click the **Plot features** button. A graph should appear on which you can interact a bit. **Mousedrag** towards the bottom right direction will zoom the plot, and **Mousedrag** towards to up right direction will reset the zoom. The **Right-click** menu lets you configure the plot, save it to an image file and export it as an ImageJ table.

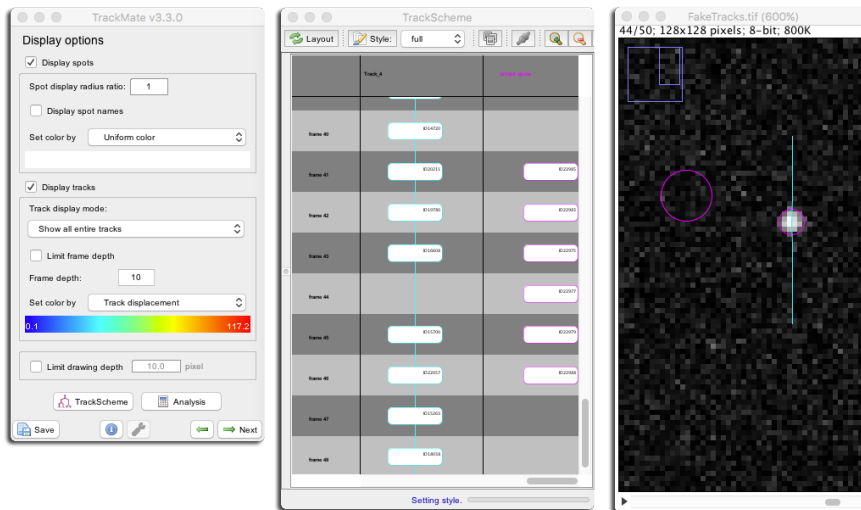


4.6 Editing tracks with TrackScheme.

The main application of TrackScheme is to edit annotations in conjunction with creating and moving spots on another view. Let's do this now.

4.6.1 Linking spots with the popup menu item.

If you have a TrackScheme window opened while you create spots on the source image, you should see them appearing in TrackScheme, under a special column on the right called Unlaid spots. The TrackScheme window should then resemble this:



Normally, TrackScheme only displays the spots that belong in a track. Lonely spots that are not linked to anything when you launch TrackScheme are not shown. The spots you create after TrackScheme are however stacked under this special column. From there, you can attach them to an existing track or create a new one.


Here is a way to do it. In TrackScheme using **Mousedrag** select all the spots in the unlaidd column. **Right-click** somewhere in TrackScheme to make the pop-up menu appear. One of the menu item should be something like **Link x spots**. Choose this one. Each spots is then linked to the next one, frame by frame, and the links should appear in TrackScheme and in other views. You just created a new track.

4.6.2 Triggering re-layout and style refresh.

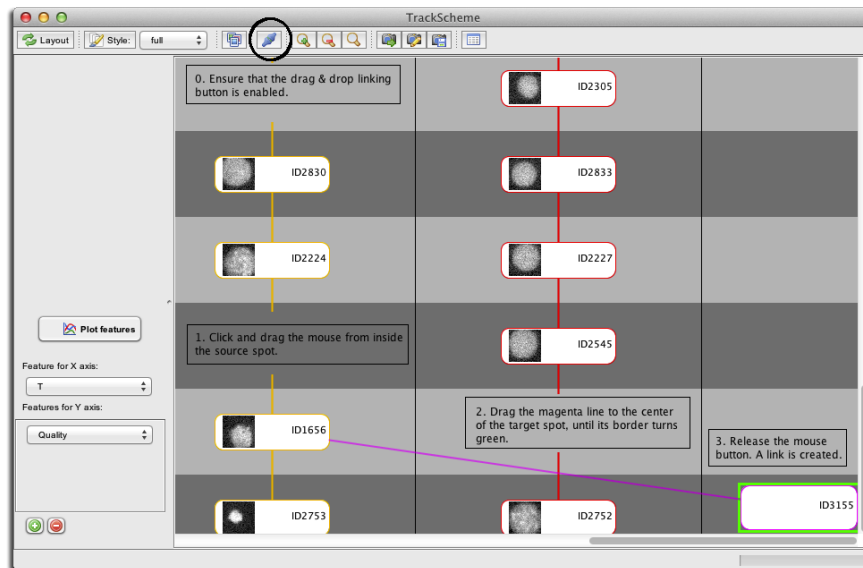
Notice that the TrackScheme display of this new track is somewhat unsatisfactory. The first track may have changed color in the main view, but this change did not happen in TrackScheme. Plus, the new track does not have its own column, and the color of some of its spots might be wrong. The reasons for this are:

- We changed the annotation and these changes affected the numerical features that color the tracks. For instance, if you picked the `Track_index` numerical feature for track coloring, there is now two tracks instead of one. The feature update is seen immediately in the main view, but for performance reason, TrackScheme as well as the color line on the main GUI window have to be refreshed manually. To do so, click on the **Style** button in the TrackScheme toolbar, and directly on the track color line on the main GUI window.
- The changes we made affected the track hierarchy, but the re-layout is not triggered automatically by such changes. To do so, press the **Layout** button in TrackScheme toolbar. This will reorganize TrackScheme with a proper layout. Since in TrackScheme, spots can be moved around at will, this is also a good way to reorder things.

4.6.3 Linking spots with drag and drop.

Another way to create single links is to enable the drag-and-drop linking mode. In the TrackScheme toolbar, click on the grayed-out  **Toggle linking** button.


Now move over any cell in one track. As you do, the cell gets highlighted with a green square. If you click and drag from this cell, a new link (in magenta) will emerge. Release it on any cells to create a link between the source and the target.



4.6.4 Removing spots and links.

The last you link you added may have strongly perturbed our annotation, particularly if you did what was on the screenshot above. Correct it by removing the last link. Simply select it press `Delete`. The same key will remove everything in the selection.

4.6.5 Editing track names and imposing track order.

Tracks are ordered from left to right alphanumerically with their name. To change a track name, `Double-click` on it in the column header part. Track names should be made of a single line with a combination of any character. You can change the track order by changing their name. If you call the first one 'B' and the second one 'A', then click the  **Layout** button, they will be permuted.

4.6.6 Editing spot names and imposing branch order.

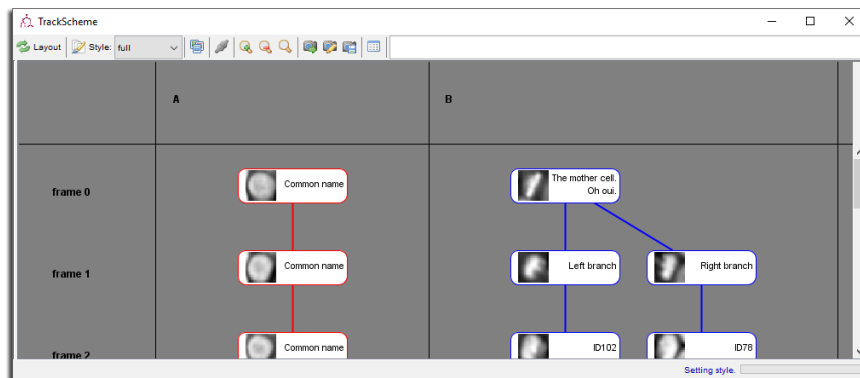
Spots also have a name, that you can see either in the main view by checking the **Display spot names**, either in TrackScheme by using the full display style. They are all called ID## by default, which is not very informative.

To edit a spot name in TrackScheme, **Double-click** on the spot. It should be replaced by an orange box in which you can type the spot name. Press **Shift + Enter** to validate the new name, or **Escape** to cancel the change. Spot names may be several lines long, but their display might then not be very pleasing.

You can also set the name of several spots at once. For instance, select a whole track and **Right click** (outside of a spot) to bring the popup menu. There is an item called Edit X spot names. The closest spot is changed to an edit box. When you validate the new name, all the selected spots get this new name.

Apart from their use to mark some biological meaning to the annotations, spot names have several purposes. There is a search box in TrackScheme toolbar that centers the view on spots with name matching the text you enter there. Press **Enter** to loop over all the matching spots.

Spot names are also used to decide in what order to lay out track branches. For instance, in a track with a cell division, you can force one branch to be the laid left or the right by setting the name of the spot just after the division. Sister cells are laid out from left to right alphanumerically, like for tracks.



5. Description of TrackMate algorithms.

This section documents the current components of [TrackMate](#). TrackMate has a modular design, meaning that it is made of different modules that each have a specific role. Developers can build their own module and re-used the other ones and the GUI to achieve a quick development. The module types are (in the order you meet them when executing the plugin):

- **Spot detectors.** Taking your image data, they detect spots in them.
- **Spot analyzers.** Each spot can receive a wide range of features, calculated from their location, radius and the image data. For instance: max intensity in spot, rough morphology, etc... They are then used to filter out spurious spots and retain only good ones for the subsequent tracking step.
- **Views.** Display the segmentation and tracking results overlaid on your image data.
- **Spot trackers,** or particle-linking algorithms. Take the filtered spots and link them together to build tracks.

- **Edge analyzers.** Like for spot analyzers, but operate links between spots. Can be used to report instantaneous velocity, link direction, etc...
- **Track analyzers.** Like for spot analyzers, but operate on whole track. Can be used to report track mean velocity, track displacement, etc... They are also used to filter spurious tracks.
- **Actions.** Miscellaneous actions you can take on the complete result of the tracking process. It can be used to copy the track overlay to another image, launch a 3D viewer, export the results to a simple format, generate a track stack, etc...

We describe here the best we can the current modules that are shipped with TrackMate.

5.1 Spot detectors.

5.1.1 Spot features generated by the spot detectors.

Behind this barbaric name stands the part responsible for spot detection in your image. The algorithm currently implemented are very generic and naive. They will most likely fail for complicated case such as touching objects, very weak SNR, *etc.* The three of them present are all based on Laplacian of Gaussian filtering, which we describe below.

Detectors can very much vary in implementation and in the technique they rely on, but they must all at least provide the following common spot features:

- **X, Y, and Z:** the spot coordinates in space. Coordinates are expressed in physical units (μm , ...).
- **R** the spot radius, also in physical units. The current detectors only set this radius value to be the one specified by the user. More advanced detectors could retrieve each spot radius from the raw image.
- **Quality:** The implementation varies greatly from detector to detector, but this value reflects the quality of automated detection. It must be a positive real number, large values indicating good confidence in detection result for the target spot. This sole feature is then used in the initial filtering step, where spots with a quality lower than a specified threshold are purely and simply discarded.

The two other time features - **T** and **Frame number** - are set by TrackMate itself when performing detection on all the time-points of the target raw data. T is expressed in physical units, and the Frame number - starting from 0 - is simply the frame the spot was found in.

5.1.2 Laplacian of Gaussian particle detection (LoG detector).

The LoG detector is the best detector for Gaussian-like particles in the presence of noise[7]. It is based on applying a LoG filter on the image and looking for local maxima [1]. The Laplacian of Gaussian result is obtained by summing the second order spatial derivatives of the gaussian-filtered image, and normalizing for scale:

$$\begin{aligned}
\text{in 1D:} \quad & \text{LoG}_\sigma = -\sigma^2 \left(\frac{\partial^2}{\partial X^2} \right) * G_\sigma * I \\
\text{in 2D:} \quad & \text{LoG}_\sigma = -\sigma^2 \left(\frac{\partial^2}{\partial X^2} + \frac{\partial^2}{\partial Y^2} \right) * G_\sigma * I \\
\text{in 3D:} \quad & \text{LoG}_\sigma = -\sigma^2 \left(\frac{\partial^2}{\partial X^2} + \frac{\partial^2}{\partial Y^2} + \frac{\partial^2}{\partial Z^2} \right) * G_\sigma * I
\end{aligned}$$

where G_σ is the Gaussian filter operator with a standard deviation σ and I the source image.

The value of σ is tuned according to the particle radius, which is entered by the user: $\sigma = r/\sqrt{n}$ where n is the dimensionality of the source image (1 for 1D, *etc*) and r the radius of the spot. In practice, filtering is made using real numbers, in the Fourier space to speed up calculation. This makes this detector ideal for spot size between 5 and 20 pixels roughly. The LoG kernel generation can handle anisotropic physical calibration (pixel sizes different in X, Y and Z), so that r and σ are in physical units. Local maxima in the filtered image yields spot detections. Each detected spot is assigned a quality value by taking the local maxima value in the filtered image. Spot with a quality lower than the value the detector is configured with are discarded immediately. If requested, the location of retained spots is refined using a quadratic fitting scheme derived from Lowe 2004 [2].

5.1.3 Difference of Gaussian particle detection (DoG detector).

This detector reproduce the LoG detector logic described above, but use an approximation for the filtering step. Given d an approximate expected particle diameter, determined upon inspection, two Gaussian filters are produced with standard deviation σ_1 and σ_2 :

$$\begin{aligned}
\sigma_1 &= 1/(1 + \sqrt{2}) \times d \\
\sigma_2 &= \sqrt{2} \times \sigma_1
\end{aligned}$$

The image is filtered using these two Gaussian filters, and the result of the second filter (largest sigma) is subtracted from the result of the first filter (smallest sigma). This yields a smoothed image with sharp local maxima at particle locations. Spots are otherwise handled as for the LoG detector.

5.1.4 Downsample LoG detector.



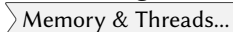
The Downsample LoG detector is made to handle large spots. If the sought spots span more than 20 pixels in their smallest dimension, it is beneficial to down-sample the image prior to detection. The down-sampling operation has a negligible cost compared to the gain achieved by spot detection in a much smaller image. This detector handles the process of down-sampling the source image by an integer factor provided by the user, applying the LoG detector on the resulting image, and mapping the physical coordinates of the spots obtained back in the

source image referential. For 3D images with a pixel size larger in Z than in X and Y, the down-sampling is tempered in Z to approach a quasi isotropic calibration in the down-sampled image. The down-sampling factor should be chosen so that spots have a diameter of at least 5 pixels in the down-sampled image.

5.1.5 Handling the detection of large images with the Block LoG detector.

The detection of spots in large images can require a lot of memory. With the LoG detector working on a 16-bit unsigned integer image, the source is first copied on 32-bit float numbers (doubling its size), then the Fourier convolution requires another image placeholder using 64-bit complex numbers (quadrupling its size). In the end, the detection requires an extra memory space six times larger than the one occupied by the source image. If several time-points are processed in parallel in multi-core computers, the required space is multiplied by the number of time-points processed concurrently.

The Block LoG detector limits memory consumption by processing small blocks of the input image sequentially. The number of blocks are specified by the user, and the partition generates several XY blocks. This privileges classical microscopy images, where the extent of an image in X and Y is much larger than in Z, and where the point-spread function is much larger along Z than along X and Y. Each block is processed independently with the LoG detector, and the resulting spot collections are pooled together. Spots lying on block borders might generate spurious detection, being detected in two separate blocks. To avoid this, spots found within the radius of another spot are discarded.

Several blocks might be processed in parallel, if the number of cores allocated to Track-Mate is larger than 1. To avoid this, set the Parallel threads settings in the    menu to 1.

5.2 Spot analyzers.

Spot features, such as Max intensity, Estimated diameter, *etc*, are calculated for all spots just after the initial filtering step. They are then used to select spots, based on filters set to retain only spots with a given feature below or above a specified threshold. Initial filtering is a good way to limit spot feature calculation time on spots known to be spurious. The current features however are made so that their calculation is cheap computationally.

5.2.1 Mean, Median, Min, Max, Total intensity and its Standard Deviation.

The plain statistical estimates are simply calculated from all the values for pixels within the physical radius from the spot center.

5.2.2 Contrast & Signal/Noise ratio.

This contrast followed [contrast](#) definition:

$$C = \frac{I_{in} - I_{out}}{I_{in} + I_{out}}$$

where I_{in} is the mean intensity inside the spot volume (using the physical radius), and I_{out} is the mean intensity in a ring ranging from its radius to twice its radius.

The spots SNR is computed as

$$SNR = \frac{I_{in} - I_{out}}{std_{in}}$$

where std_{in} is the standard deviation computed within the spot.

These two values depend greatly on the correctness of the radius of each spot. Negative values might be caused by incorrect radius.

5.2.3 Estimated diameter.

This feature estimates an optimal diameter for each spot, based on contrast calculation. The mean pixel intensity is calculated in 20 concentric, tangent rings, centered at the target spot location, and with radiuses ranging from a 10th of the spot radius to twice its radius. The contrast at a given radius is defined as the difference between the mean intensity of a ring with inner radius the radius sought, and the previous ring. The estimated diameter is then defined as the radius that maximizes this contrast. A quadratic interpolation is performed to improve diameter accuracy.

5.3 Spot trackers or particle-linking algorithms.

5.3.1 LAP trackers.

The Linear Assignment Problem (LAP) trackers implemented here follow a stripped down version of the renowned method contributed by Jaqaman and colleagues [4]. We repeat here the ideas found in the reference paper, then stresses the differences with the nominal implementation.

In TrackMate, the LAP framework backs up two instances of a tracker:

- the Simple LAP tracker;
- the LAP tracker.

The first one is simply a simplified version of the second: it has less settings and only deal with particle that do not divide nor merge, and ignores any feature penalty (see below).

All the linking costs for these two trackers are based on the particle-to-particle square distance. If this tracker had to be summarized in one sentence, it would be the following: **The Simple LAP tracker and the LAP tracker are well suited for particle undergoing Brownian motion.** Of course, they will be fine for a non-Brownian motion as long as the particles are not too dense.

Particle-linking happens in two step: track segments creation from frame-to-frame particle linking, then track segments linking to achieve gap closing. The mathematical formulation used for both steps is linear assignment problem (LAP): a cost matrix is assembled contained all possible assignment costs. Actual assignments are retrieved by solving this matrix for minimal total cost. We describe first how cost matrices are arranged, then how individual costs are calculated.

Cost matrix for frame-to-frame linking. In the first step, two consecutive frames are inspected for linking. Each spot of the first frame is offered to link to any other spot in the next frame, or not to link. This takes the shape of a $(n + m) \times (n + m)$ matrix (n is the number of spots in the frame t , m is the number of spots in the frame $t + 1$), that can be divided in four quadrants.

- The top-left quadrant (size $n \times m$) contains the costs for linking a spot i in the frame t to any spot j in the frame $t + 1$.
- The top-right quadrant (size $n \times n$) contains the costs for a spot i in the frame t not to create a link with next frame (yielding a segment stop).
- The bottom-left quadrant (size $m \times m$) contains the costs for a spot j in the frame $t + 1$ not to have any link with previous frame (yielding a segment start).
- The bottom-right quadrant (size $m \times n$) is the auxiliary block mathematically required by the LAP formalism. A detailed explanation for its existence is given in the supplementary note 3 of the Jaqaman paper [4]. This quadrant is built by taking the transpose of the top-left quadrant, and replacing all non-blocking costs by the minimal cost.

Solving LAP. To solve this LAP, we rely on the Munkres & Kuhn algorithm [8], that solves the problem in polynomial time ($O(n^3)$). The algorithm returns the assignment list that minimizes the sum of their costs.

The frame-to-frame linking described above is repeated first for all frame pairs. This yields a series of non-branching track segments. A track segment may be start or stop because of a missing detection, or because of a merge or split event, which is not taken into account at this stage. A second step where track segments are offered to link between each other (and not only spots) is need, and described further down.

Calculating linking costs. In calculating costs, we deviate slightly from the original paper from Jaqaman *et al.* [4]. In the paper, costs depend solely on the spot-to-spot distance, possibly weighted by the difference in spot intensity. Here, we offer to the user to tune costs by adding penalties on spot features, as explained below.

The user is asked for a maximal allowed linking distance (entered in physical units), and for a series of spot features, alongside with penalty weights. These parameters are used to tune the cost matrices. For two spots that may link, the linking cost is calculated as follow:

- The distance between the two spots D is calculated.
- If the spots are separated by more than the max allowed distance, the link is forbidden, and the cost is set to ∞ (*i.e.* the blocking value). If not,
- For each feature in the map, a penalty p is calculated as $p = 3 \times W \times \frac{|f_1 - f_2|}{f_1 + f_2}$ where W is the factor associated to the feature in the map. This expression is such that:
 - there is no penalty if the two feature values f_1 and f_2 are the same;
 - with a factor of 1, the penalty is 1 is one value is the double of the other;
 - the penalty is 2 if one is 5 times the other one.

- All penalties are summed, to form $P = (1 + \sum p)$.
- The cost is set to the square of the product: $C = (D \times P)^2$

If the user feeds no penalty, the costs are simply the distances squared.

Calculating non-linking costs. The top-right and bottom-left quadrant of the frame-to-frame linking matrix contain costs associated with track segment termination or initiation (a spot is not linking to a spot in the next or previous frame). Each of these two blocks is a square matrix with blocking value everywhere, except along the diagonal for which an alternative cost is computed. Following Jaqaman [4], this cost is set to be

$$C_{\text{alt}} = 1.05 \times \max(C)$$

where C is the costs of the top-left quadrant.

Cost calculation & Brownian motion. Without penalties and with a maximal linking allowed distance, the returned solution is the one that minimizes the sum of squared distances. This actually corresponds to the case where the motion of spots is governed by [Brownian motion](#). See for instance Crocker and Grier [5].

By adding feature penalties, we aim at favoring linking particles that "resemble" each other. In brute single particle linking problems, spots are generally all the same, and they only differ by position. However, there is a variety of problems for which these feature penalties can add robustness to the tracking process.

For instance, we originally developed [TrackMate](#) for semi-automated lineaging of *C.elegans* embryos, using a strain fluorescent in the nucleus. Cells that are dividing have a fluorescence distribution which is very different from non-dividing cells, and this can be exploited for robust tracking.

Track segment linking. In a second step, the track segments built above are offered to link between each other. Jaqaman and colleagues proposes to exploit again the LAP framework for this step. A new cost matrix is generated, but this time the following events are considered:

- The end of a track segment is offered to link to any other track segment start. This corresponds to gap-closing events, where a link is created typically over two spots separated by a missed detection.
- The start of a track segment is offered to link to the spots in the central part (not start, not end) of any other track segment. This corresponds to splitting events, where a track branches in two sub-tracks.
- The end of a track segment is offered to link to the spots in the central part of any other track segment. This corresponds to merging events, where two tracks merges into one.
- A spot part of any track segment is offered not to create any link.

The second cost matrix has a shape that resembles the first cost matrix, calculated for frame-to-frame linking, and which is best described in the original article.

As before, we modified the way costs are calculated, and re-used the feature penalties framework described above. Also, the user must provide on top a maximal time-difference to link, over which linking will be provided. Careful: this maximal time is expressed in physical units and not in number of frames.

Main differences with the Jaqaman paper. The nominal implementation of the paper remains the one developed under MATLAB by Khuloud Jaqaman *et al.* [4]. The software is called **u-track** and can be found on [Khuloud Jaqaman homepage](#). TrackMate was initially developed to simplify *C.elegans* lineaging. It therefore just bundles a stripped down version of this framework. The notable differences are:

- The LAP framework is generic: Jaqaman and colleagues proposed a framework to approximate multiple-hypothesis tracking solutions using linear assignment problems. One just need to provide the link cost matrix. **TrackMate** properly implements the LAP framework, but the cost matrix calculation - which is specific to each problem - is much more simpler than in **u-track**.
For instance, in TrackMate all link costs are based on the square distance between two spots (weighted or not by feature differences, see above), which make it tailored for Brownian motion. In **u-track**, the user is proposed with different motion types, including a linear motion whose parameters are determined on the fly. See for instance CD36 tracking, in the supplementary note 7 of the Jaqaman paper.
- In **u-track**, merging and splitting of tracks are used to describe two particles that temporally overlap spatially. These events' costs are weighted by the two particle intensities to properly catch the apparent increase in intensity due to the overlap. In **TrackMate**, we use splitting events to describe cell divisions, as we developed it initially to deal with *C.elegans* lineages. However it seems that Jaqaman and colleagues used it the same way to investigate CD36 dissociation and re-association.
- In **TrackMate**, distance and time cutoffs are specified manually by the user. In **u-track** they are derived for each particle automatically, providing self adaptation.

5.3.2 Linear motion tracker.

The linear motion tracker can deal specifically with linear motion, or particle moving with a roughly constant velocity. This velocity does not need to be the same for all particles. You can find it in TrackMate tracker selection under the name Linear motion LAP tracker.

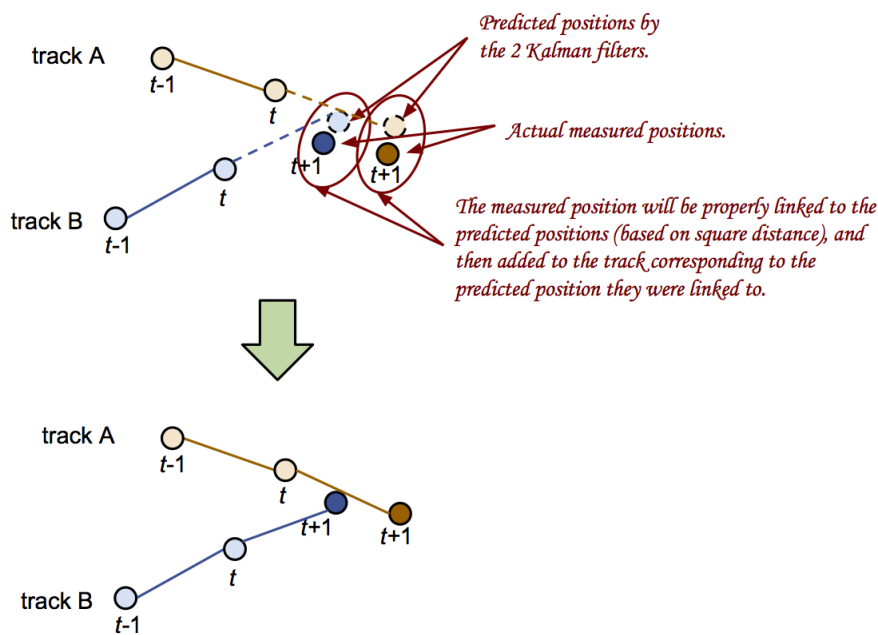
Though it deals with a completely different motion model compared to the LAP trackers in TrackMate, it reuses the Jaqaman LAP framework, and it is similar to a tracker proposed in the Jaqaman paper as well: See the CD36 tracking, in the supplementary note 7 of the Jaqaman paper. But again, the version in TrackMate is simplified compared to what you can find in **u-track**.

Principle. The linear motion tracker relies on the [Kalman filter](#) to *predict* the most probable position of a particle undergoing constant velocity movement.

Tracks are initiated from the first two frames, using the classical LAP framework with the Jaqaman cost matrix (see above), using the square distance as cost. The user can set what is the maximal distance allowed for the initial search with the `Initial search radius` setting.

Each track initiated from a pair of spots is used to create an instance of a Kalman filter. There are as many Kalman filters as tracks. In the next frames, each Kalman filter is used to generate a prediction of the most probable position of the particle. All these predictions are stored.

Then, all the predicted positions are linked against the actual spot positions in the frame, using again the Jaqaman LAP framework, with the square distance as costs. The user can set how far can be an actual position from a predicted position for linking with the `Search radius` setting.



Now of course, after linking, some Kalman filters might not get linked to a found spot. This event is called an occlusion: the predicted position did not correspond to an actual measurement (spot). The good thing with Kalman filters is that they are fine with this, and are still able to make a prediction for the next frame even with a missing detection. If the number of successive occlusions is too large, the track is considered terminated. The user can set the maximal number of successive occlusions allowed before a track is terminated with the `Max frame gap` setting.

Conversely, some spots might not get linked to a track. They will be used to initiate a new track in the next frame, as for the tracker initiation described above.

It is important to note here that the cost functions we use is the square distance, like for the Brownian motion, but from the predicted positions to the actual detections. Because the prediction positions are made assuming constant velocity, we are indeed dealing with an adequate cost function for linear motion. But since we are linking predicted positions to measured positions with the square distance cost function, we do as if the predicted positions deviate

from actual particle position with an error that follows the gaussian distribution. This is a reasonable assumption and this is why this tracker will be robust.

Implementation. The code can be found on [GitHub](#). We now repeat the section above in pseudo-language. When you see the word **link** below, this means:

- Take all the source detections in frame t and the target detections in frame $t + 1$.
- Compute the costs for all possible physical assignment (potential links) between source and target detections and store them in the cost matrix.
- Solve the LAP associated to this matrix.
- Create a link for each assignment found.

The particle linking algorithm would read as follow:

- Initialization:
 - Link all the detections of frame 0 to the detections of frame 1, just based on the square distance costs (for instance).
 - From each of the m links newly created, compute a velocity. This velocity is enough to initialize m Kalman filters.
 - Initialize m tracks with the found detections and links, and store the associated Kalman filters.
- Track elongation:
 - For each Kalman filter, run the prediction step. This will generate m predicted positions.
 - Link the m predicted positions to the n detections in frame 2, based on square distance.
 - Target detection that have been linked to a predicted position are added to the corresponding track.
 - The accepted target detection is used to run the update step of the Kalman filter.
 - Loop to next frame.
- Track termination:
 - Some of the m predicted position might not find an actual detection to link to. In that case, we have an occlusion. The algorithm must decide whether it has to terminate the track or to bridge over a gap.
 - If the number of successive occlusions for a Kalman filter is below a certain limit (typically 2 to 10), the track is not terminated, and the filter goes back to the track elongation step. Hopefully, from the new prediction a target particle will be found, and the detection in frame t will be linked to a detection in frame $t + 2$ (or $t + 3$ etc).
 - Otherwise, the track is terminated and the Kalman filter object is dropped.

- Track initiation:
 - Conversely, some detections in frame $t + 1$ might not be linked to a predicted position. In this case, these orphan detections are stored to initiate a new track. But for this, other orphans detections are needed in frame $t + 2$.
 - This step is identical to the initiation step, but for subsequent frames. It requires to store orphan detections in current and previous frames.
 - In frame $t + 2$, priority must be given to detections linked to the predicted positions by the Kalman filters over orphan detections of frame $t + 1$. So when you deal with frame $t + 2$, you perform first the track elongation step, get a list of orphan detections in frame $t + 2$, and then combine it to the orphan detections in frame $t + 1$ to initiate new Kalman filters.

6. Particle-linking algorithms accuracy.

The problem with tracking algorithms is that they always give an answer.

This answer can be completely irrelevant, even non-physical, and there is no built-in flags that would indicate something wrong. The best way to avoid basing your downstream analysis on faulty tracking results is to know in what situation the tracker works the best, and what are its limitations. This is the aim of this page for the trackers and detectors shipped with [TrackMate](#).

6.1 The ISBI 2012 single particle challenge.

In 2011-2012, an [ISBI Grand Challenge](#) was organized for the Single-Particle Tracking algorithms. For this challenge, images were numerically simulated to serve as dataset for single-particle tracking algorithms. Simulations relied on several particle and motion models, used in turn as ground truth for the quantification of the accuracy of tested algorithms.

We took this challenge with TrackMate, which was in early version 1.1 at the time of the challenge. The results and the methodology to compute the accuracy of a tracking algorithms were published [9] thereafter. In its early version, TrackMate scored roughly in the middle rankings for most scenarios.

6.2 Current TrackMate version accuracy against the ISBI dataset.

Since then TrackMate improved and from version 2.7.x it ships a new tracker that can deal specifically with linear motion. We can now assess its accuracy again, using the ISBI challenge data. The people behind [lcy](#) maintains the website that hosts the challenge data, and made it available for download. The section compares the particle-linking accuracy for the 3 classes of tracking algorithms available in TrackMate:

- The LAP framework derived from Jaqaman *et al.* [4].
- The linear motion tracker based on Kalman filter.
- The plain Nearest neighbor tracker for reference.

6.2.1 Scenarios.

The testing dataset cover four scenarios, that are detailed in the challenge paper [9]. We survey briefly here what is their particle and motion models they are based on the following scenarios:

Scenario name	Particle shape	Motion type
MICROTUBULE	Slightly elongated shape to mimic MT tip staining.	Roughly constant velocity motion.
RECEPTOR	Spherical.	Tethered motion: switch between Brownian and directed motion with random orientation for the later.
VESICLE		Brownian motion.
VIRUS		Switch between Brownian and directed motion with fixed orientation for the later.

For each scenario, images covers several particle density:

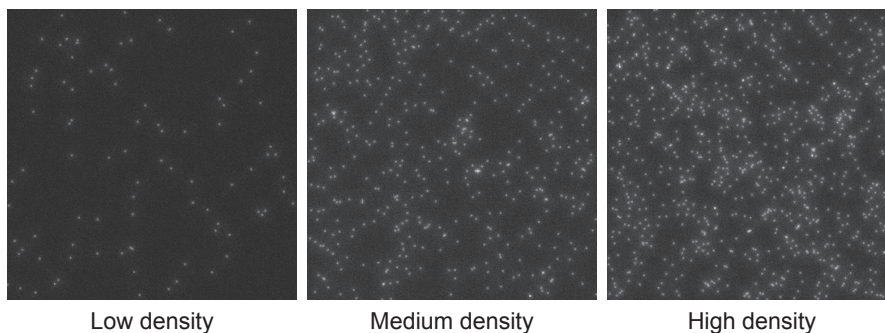
- low: 60-100 / frame
- mid: 400-500 / frame
- high: 700-1000 / frame

to check how a tracking algorithm behaves when particles get very dense. Also, particles SNR spans several values: 1, 2, 4, 7 (plus 3 for the RECEPTOR scenario). As said on the [challenge page](#): "SNR=4 is a critical level at which methods may start to break down".

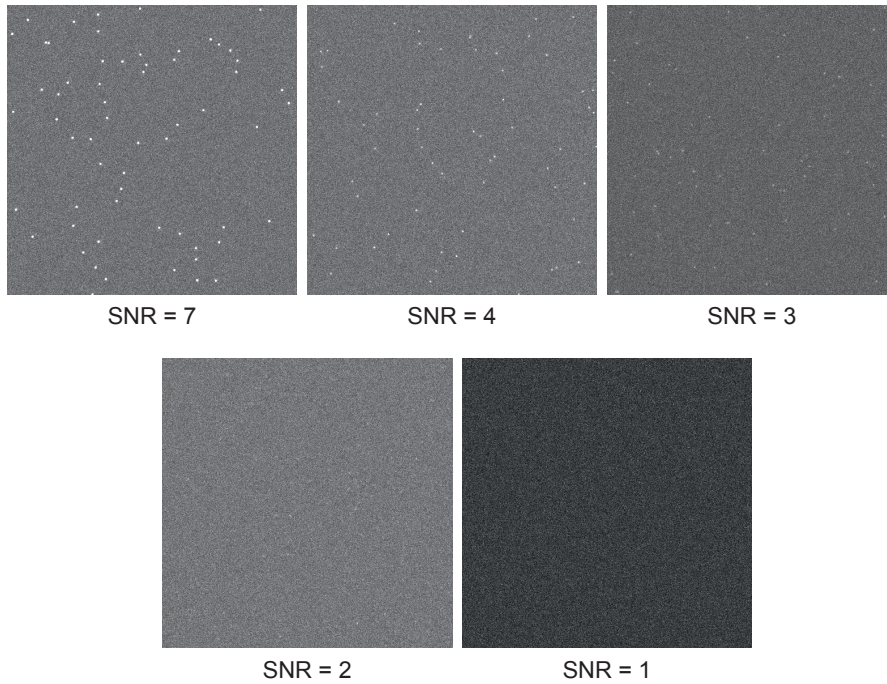
6.2.2 Example images from the challenge dataset.

Below are shown typical images taken from the challenge.

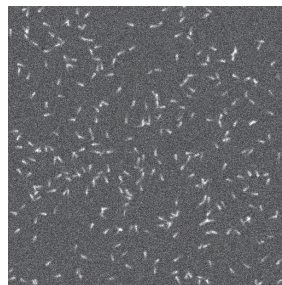
Varying particle density. Variation in the particle density, illustrated in the case of the VESICLE scenario with SNR = 7. Contrast stretched to the 0-150 8-bit range.



Varying particle SNR. Varying SNR in the RECEPTOR scenario dataset. Contrast stretched to the 0-50 8-bit range.



The MICROTUBULE scenario particle shape. An excerpt from the MICROTUBULE scenario, with SNR = 4, and density = medium. The particles have an elongated shape, to mimic microtubule tip staining.



6.2.3 Accuracy measurements.

For each scenario and condition, the method returns numerous values that characterizes the accuracy of a tracking algorithm. They are detailed on [this technical paper](#). We plot below only three of them:

- The **Jaccard similarity between tracks**, that quantifies how well the tracks returned by the algorithm match the ground truth. This value assesses the accuracy of the

[spot tracker](#) you pick in TrackMate. It ranges from 0 (terrible) to 1 (found tracks = ground truth).

- The **Jaccard similarity between detections**, that quantifies how well the particle detected by the detection algorithm match the ground truth. It depends strongly on the [spot detector](#) you pick in TrackMate, and ranges from 0 to 1 like the above quantity.
- The **RMSE of detection positions** that quantifies how precise is the location of the detected particles. The smaller the better.

We fully relied on the [Icy software](#) to compute these values. TrackMate ships an action that exports tracking results to the XML format imposed by the ISBI challenge, which code can be found [here](#). We generated these files for all the conditions of a scenario, and used the [Icy ISBI challenge scoring plugin](#) to yield metrics. We then used [MATLAB](#) to plot them.

6.2.4 Parameter used.

Unless otherwise specified below, we always used the LoG detector as a spot detector, with an estimated particle diameter of 2, and used sub-pixel accuracy. For SNR below 4, this detector was completely confused and the detection results are dominated by noise. We did not make anything special to improve its sensitivity below this limit. When the histogram of detection quality returned by the detector was not bimodal, we pick a quality threshold that yielded approximately the expected number of particles in the sequence. The three spot trackers were configured as indicated in the table below. Finally, for SNR<4, we filtered out tracks that had less than 4 detections.

Spot tracker	Parameter	Value
<i>Linear motion tracker</i>	Initial search radius	10
	Search radius	7
	Max frame gap	3
<i>LAP Brownian motion</i>	Max linking distance	7
	Max gap-closing distance	10
	Max frame gap	3
<i>Nearest neighbor</i>	Max search distance	10

6.2.5 Results.

The results for each scenario is presented and commented below. Since we used the same detector for all scenarios, all the measures have a common shape for their dependence on SNR.

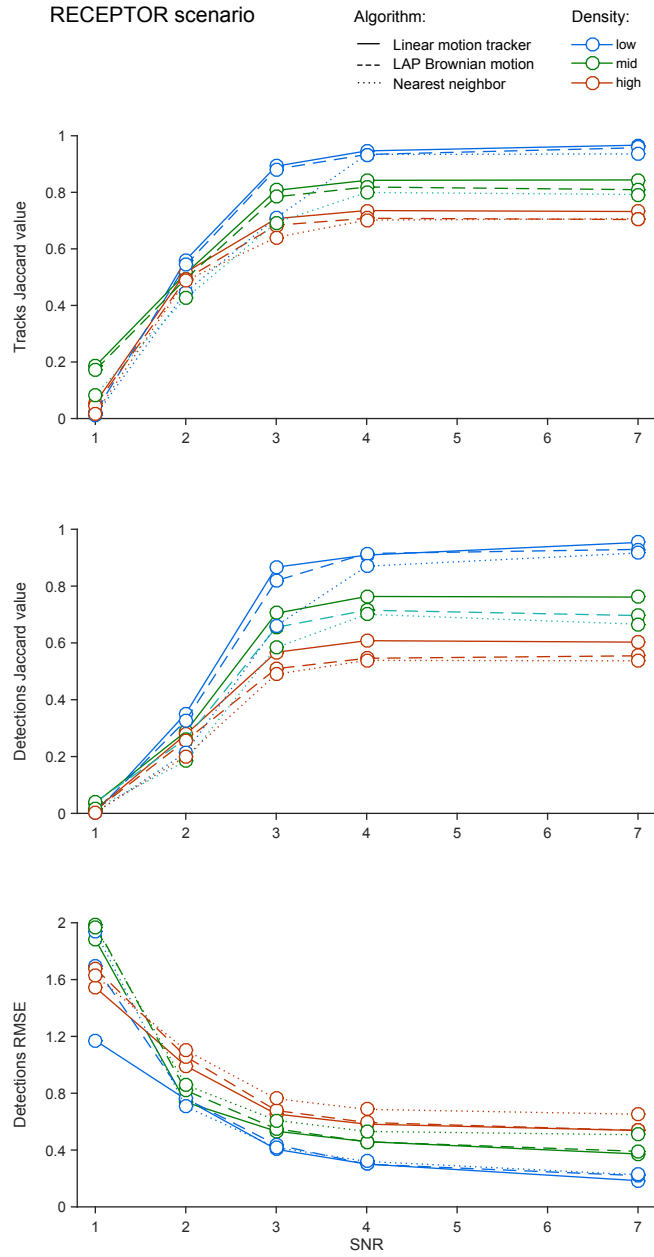
Basically, accuracy is the same for $SNR \geq 3$. Below $SNR = 2$ included, the detector is unable to reliably finds all the particles. For a SNR of 2, it still finds a subset of correct particles, amongst the brightest. At a SNR of 1, detection results are dominated by spurious detections, and the particle linking algorithm performance does not matter anymore. All are equally bad, since they track the wrong particles.

Microtubule scenario. This scenario probes how well the TrackMate algorithms fare against a roughly constant velocity motion model. Unsurprisingly, the linear motion tracker performs the best and resists well against high density of particles.

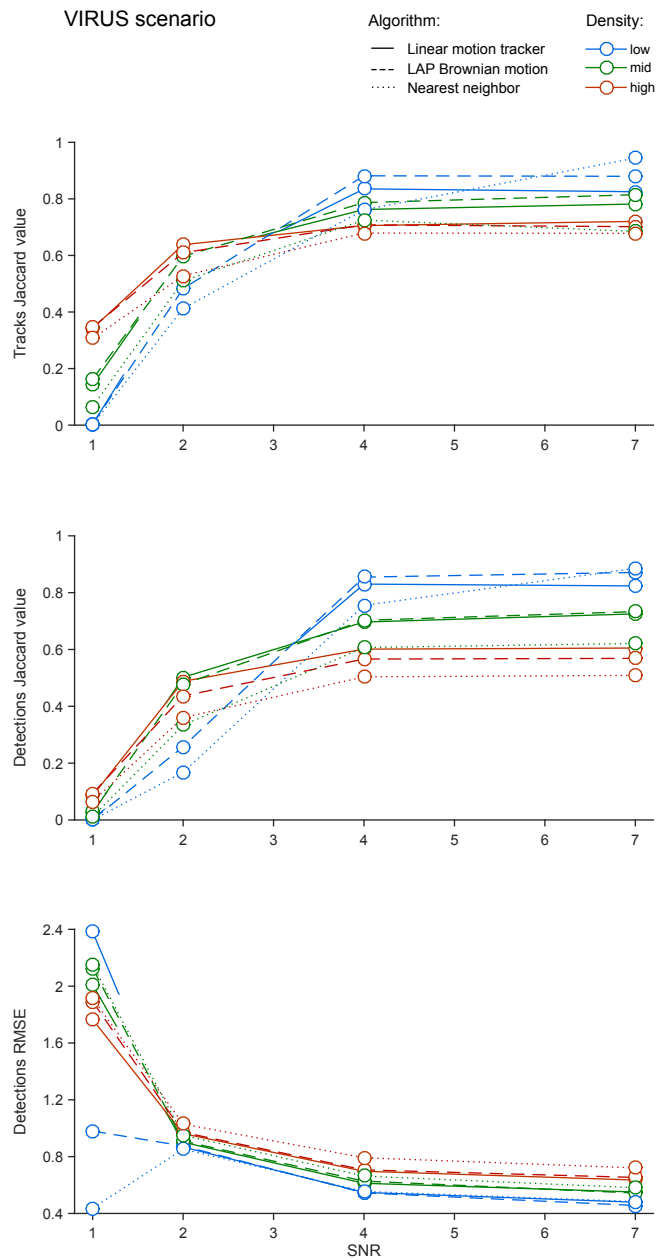
The LAP tracker does not perform well, even in the ideal case, as it expects the average particle position to be constant at least on short timescales, when it does not. Therefore, its performance approaches the worst-case scenario given by the nearest-neighbor search algorithm.

The RMSE on particle position is the worst here over the 4 scenarios. This is the obvious consequence of the particle shape, which is asymmetric and elongated, when the LoG detector expects bright blobs. Still, this does not affect tracking results.

in all conditions.



Vesicle scenario. The motion model of this scenario is the pure Brownian motion. Unsurprisingly the LAP tracker behaves the best as it models precisely this situation. The linear motion tracker is confused by the constant direction changed generated by the random motion, and is superseded even by the nearest neighbor search.



6.3 Comments.

These results serve as a base to help end-users picking the right algorithm for their problems, and maybe encourage developers to implement their own. As said before, a deeper interpretation of these metrics in the general case is found in the original paper [9]. Here are a few things specific to the current version of TrackMate.

The parameters and strategy used for this accuracy assessment are pretty basic and unlab-

orated. This way, results give the 'raw' accuracy, before a user exploits the deeper specificity of their problem. The two sections below quickly list what we could have done and could not have done even if we wanted to improve results.

What was in the challenge that TrackMate did not exploit. We saw that at low SNR, the detection step dominates and its inability to robustly detect faint particles is the cause for low scores. Here we did not try to improve the detection results via pre-processing. One could have denoised the image, or averaged succeeding frames to improve the SNR. Other strategies rely on a denoising pre-processing step to improve detection accuracy [10].

What is in TrackMate that we could not exploit for the challenge. The LAP trackers were the first trackers to be shipped with TrackMate. They are a stripped down version of. They are based solely on square distance costs (Brownian motion), but they can be modulated by a penalty factor based on numerical features. For instance, the cost to link two particles can be penalized if their mean intensity is too different.

The challenge data does offer that possibility: all particles have roughly the same shape and intensity (modulo some minor variations in SNR). An exception is the MICROTUBULE scenario, where particles have an elongated shape. Therefore, we could have computed an orientation for them, and penalize two particles with different orientation (assuming - correctly - that orientation remains roughly the same between two frames).

7. Spot detectors performance.

We report here performance metrics in the shape of execution time for the spot detectors natively shipped with TrackMate. They serve as a basis for end users to pick an adequate spot detector when concerned by detection time. We focus on comparing the LoG detector and DoG detector.

7.1 The test environment.

The computer used for these tests is the following:

```
Mac-Pro  
mi-2010  
Processor 2 x 2.66 GHz 6-Core Intel Xeon  
Memory 24 GB 1333 MHz DDR3 ECC  
Software Mac OS X Lion 10.7.5 (11G63)
```

For these tests we used Fiji running on Java 1.6. The detectors instantiated were set to use only 1 thread, not to confuse metrics with concurrent programming issues. Unless indicated, the median filter and the sub-pixel localization were not used.

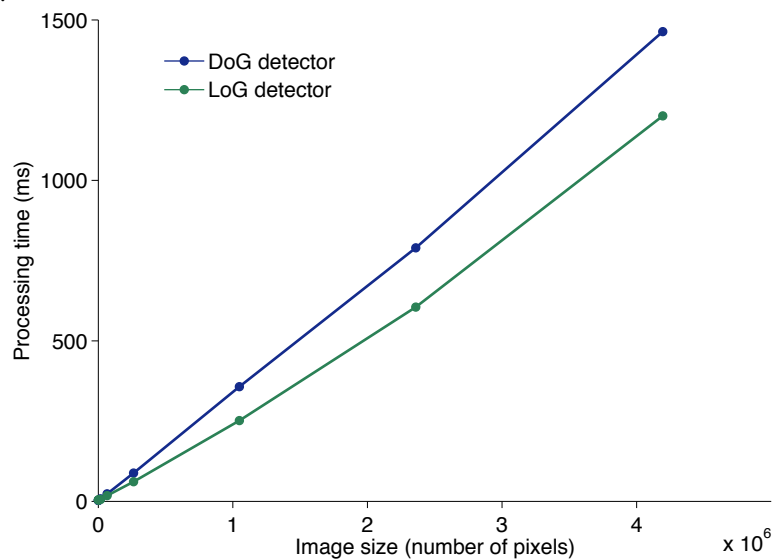
7.2 Processing time for a 2D image as a function of its size.

For a uint16 image, varying its size, containing 200 gaussian spots of radius 3 (everything is in pixel units).

N (pixels)	Image size	DoG detector time (ms)	LoG detector time (ms)
256	16 × 16	3.0	4.8
1024	32 × 32	2.95	4.3
4096	64 × 64	3.95	4.35
16384	128 × 128	7.9	5.85
65536	256 × 256	23.35	17.7
262144	512 × 512	88.2	61.15
1048576	1024 × 1024	357.3	251.65
2359296	1536 × 1536	789.85	605.4
4194304	2048 × 2048	1463.4	1201.1

For the DoG detector, unsurprisingly, we find that the execution time is proportional to the number of pixels, following approximately $t(\text{ms}) = 3.4 \cdot 10^{-4} \times N_{\text{pixels}}$. This is expected as all calculations are done in direct space.

The LoG detector operates in Fourier space, and because of the Fourier transform implementation we use, the images are padded with 0s to reach a size equal to a power of 2. This does not show here as all but one tests are made with such a size. Still, the execution time slightly deviates from the linear case, and shows a quadratic shape. The best linear fit yields a low in $t(\text{ms}) = 2.8 \cdot 10^{-4} \times N_{\text{pixels}}$, showing that the LoG detector is slightly quicker than the DoG detector.

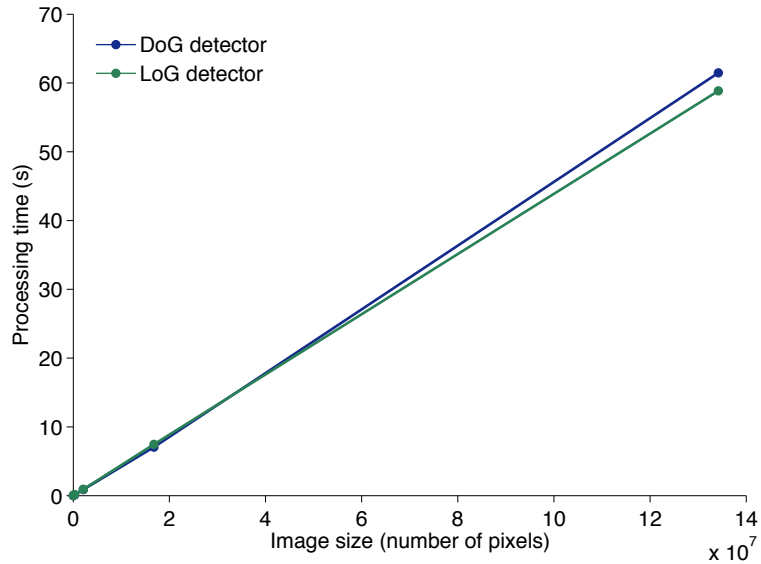


7.3 Processing time for a 3D image as a function of its size.

N (pixels)	Image size	DoG detector time (ms)	LoG detector time (ms)
4096	16 × 16 × 16	8.7	24.7
32768	32 × 32 × 32	23.5	38.5
262144	64 × 64 × 64	129.3	159.2
2097152	128 × 128 × 128	875.1	936.3
16777216	256 × 256 × 256	7054.0	7462.4
134217728	512 × 512 × 512	61477.2	58860.6

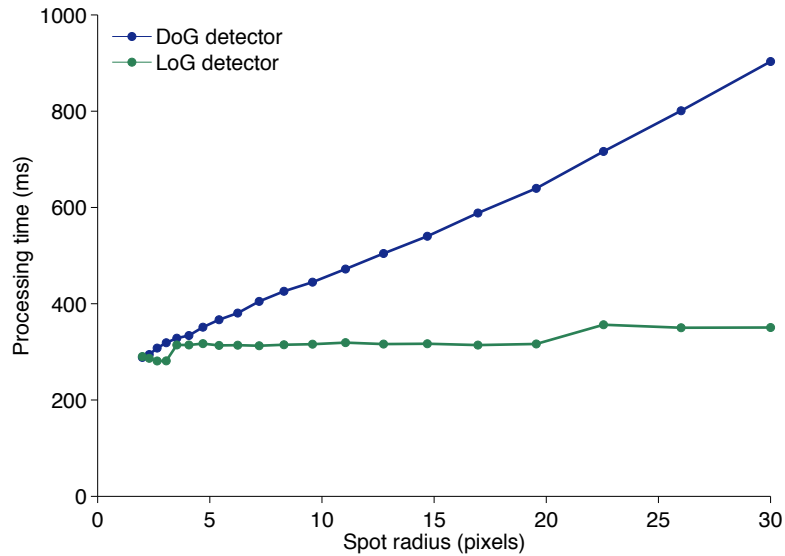
And again, the processing time is found to be linear with the number of pixels. The linear fit is slightly steeper, however: $t(\text{ms}) = 4.6 \cdot 10^{-4} \times N_{\text{pixels}}$, which we attribute to the 3D kernel overhead.

Interestingly, the LoG detector seems to become the slowest at intermediate size, which we attribute to overhead in 3D iteration with the ImageJ data model, where Z-slices are stored as individual arrays, accessed in a list.



7.4 Processing time for a 2D image as a function of the spot radius.

We used a 1024x1024 uint16 image, with 200 gaussian spots, the size of which we varied. The detector was tuned to this radius.



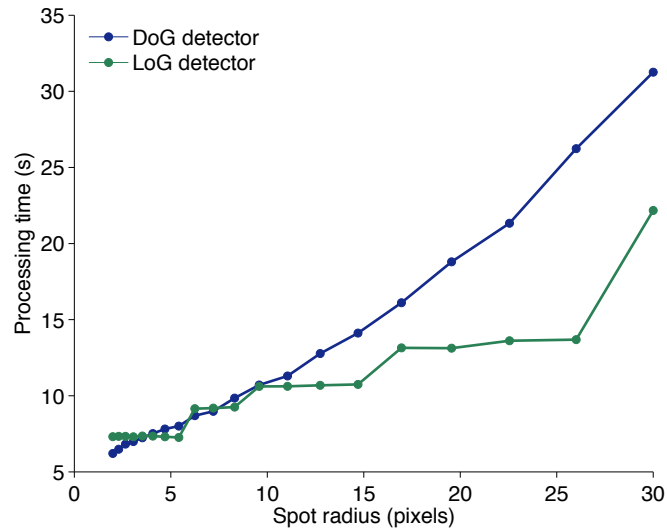
We find that for the DoG detector, the processing time to increase linearly with the specified radius, following approximately $t(\text{ms}) = 20.5 \times \text{radius} + 260$. As the difference-of-gaussians is calculated in the direct space, a marked increase is expected as there is more pixels to iterate over. Without optimization, we should however have found the time to be increasing with the square of the radius, and find the same dependence that for the image size. Thanks to the clever implementation of gaussian filtering¹, this is avoided.

The LoG detector shows a near-constant processing time, which makes it desirable for spots larger than 2 pixels in radius. This is due to the way we compute the convolution which is explained below.

7.5 Processing time for a 3D image as a function of the spot radius.

This time we used a 256x256x256 3D image, but with otherwise the same parameters.

¹[Gauss3 code](#)



The processing time increases, but this time deviates slightly from linearity in the DoG case. We retrieve the 3D kernel overhead we had for the 3D images.

The LoG performance clearly highlights the 0-padding used because of the Fourier transform: Indeed, the processing time increase in a step-wise manner. We use the Fourier transform to compute the convolution by the LoG kernel. But for the implementation we use, the kernel image (and the source image as well) are padded by 0 until their size reaches a power of 2 (128, 256, 512, etc). Whenever the required kernel size is smaller than this power of 2, its size is increased to this value. Because ultimately the processing time depends on the number of pixels, we see a constant processing time until the kernel size imposes a larger power of 2.

7.6 Choosing between DoG and LoG based on performance.

This stepwise evolution makes it slightly harder to choose between LoG and DoG detectors based on performance. As a crude rule of thumb we will remember that

- The LoG detector outperforms the DoG detector in 2D for radiuses larger than 2 pixels.
- The LoG detector outperforms the DoG detector in 3D for radiuses larger than 4 pixels.

Part III.

Interoperability.

8. Importing and analyzing TrackMate data in MATLAB.

We document here the functions shipped with Fiji that allows for importing TrackMate data in MATLAB. These functions use amongst other things the MATLAB classes `table` and `digraph`, introduced respectively with versions R2013b and R2015b, so a recent version of MATLAB is required to use them. Also, end-users should have some notions of MATLAB and basic understanding of mathematical graphs to take full advantage of this documentation.

For most tracking applications, importing tracking results is easy, as long as the tracks are linear tracks. If the tracks do not have split or merge events, then single tracks can be represented by a linear array, for instance containing the particle positions or indices. Things are much more complicated when a track can divide in two or more components, or inversely merge with another track. A track cannot be represented anymore by linear arrays. This is the case for TrackMate, as its data model permits split and merge events.

Here we document the various ways of importing TrackMate data in MATLAB, how to deal with complex tracks and show examples of basic analysis and visualization of tracks in MATLAB.

8.1 Installation of TrackMate functions for MATLAB.

The Fiji application ships up to date MATLAB functions that - amongst other things - deal with TrackMate import. They are contained in the `scripts` folder of the `Fiji.app` installation:

```
ls ~/Applications/Fiji.app/scripts
```

We are interested in the five `trackmate*.m` and `importTrackMateTracks.m` functions. To make them available in your MATLAB sessions, you need to add the `scripts` folder to your MATLAB path. To do so, open the MATLAB path editor, and add the `scripts` folder to it. The way to do it is best explained on the [MATLAB documentation website](#).

Now check that the new functions are available from MATLAB:

```
>> which trackmateGraph

/Users/tinevez/Development/Matlab/functions/jy/...
trackmate/trackmateGraph.m
```

8.2 The simple case of linear tracks.

In the case where your application only uses for linear tracks, you don't need to deal with the aforementioned complexity and can rely on a simple array data structure.

TrackMate has an action that exports data to a simplified XML file containing the track. For in the action menu and look for the action called **Export tracks to XML file**. This action will create a XML file with a simple nomenclature, that resembles this:

```

<?xml version="1.0" encoding="UTF-8"?>
  <Tracks nTracks="1959" spaceUnits="pixels" frameInterval="1.0"
timeUnits="frames" generationDateTime="Wed, 22 Jun 2016 16:33:25"
from="TrackMate v3.3.0">
  <particle nSpots="10">
    <detection t="0" x="1710.2" y="1015.7" z="1613.8" />
    <detection t="1" x="1711.7" y="1017.4" z="1607.9" />
    <detection t="2" x="1708.3" y="1015.6" z="1610.0" />
    <detection t="3" x="1707.4" y="1012.3" z="1614.9" />
    ...
  </particle>
  ...
</Tracks>

```

Tracks are organized as `<particle>` element, and in each element, with T, X, Y and Z listed with their physical value.

This simple file format emerged from the [ISBI Grand Challenge](#) on single-particle tracking. It is here to allow end-users to write simple import filters for other languages. But it is limited to linear tracks. Running the **Export tracks to XML file** action on data wither split and merge events will **silently fail** and generate a XML file that will generate errors later.

Importing such a file in MATLAB is done via the `importTrackMateTracks` function:

```

>> file_path_tracks = 'Video_1_Tracks.xml';
>> tracks = importTrackMateTracks(file_path_tracks);
>> n_tracks = numel( tracks );
>> fprintf('Found %d tracks in the file.\n', n_tracks)

```

Found 55 tracks in the file.

In MATLAB the tracks are stored in a cell array of matrices.

```

>> tracks( 1 )

```

ans =

```

[48x4 double]

```

Each track is a $N \times 4$ matrix of double, one line per detection. On a single line, the spot data is arranged as [T X Y Z]:

```

>> tracks{1}(5, :)

```

ans =

```

4.0000  80.7978  76.6681  0

```

By default, time is reported in frame number, so it is an integer. Also, spots have always a Z coordinate, even if the tracking was made in 2D. The function `importTrackMateTracks` has two switches to change this behavior. For instance:

```

>> clipZ = true; % Remove Z coordinates, if you know you can.
>> scaleT = true; % Use physical time for T.
>> tracks = importTrackMateTracks(file_path_tracks, clipZ, scaleT);
>> tracks{1}(5, :)

```

```
ans =
```

```
0.2400 80.7978 76.6681
```

The data file we used for this section has a physical frame interval, so now T is in seconds. And since we clipped the Z coordinate, the tracks are made of $N \times 3$ matrices now.

The metadata is stored in the file, and can be accessed as a secondary output of the `importTrackMateTracks` function.

```

>> [ tracks, md ] = importTrackMateTracks(file_path_tracks, ...
    clipZ, scaleT);
>> md

```

```
md =
```

```

    spaceUnits: 'µm'
    timeUnits: 'sec'
  frameInterval: 0.0600
           date: 'Wed, 22 Jun 2016 16:55:20'
    source: 'TrackMate v3.3.0'

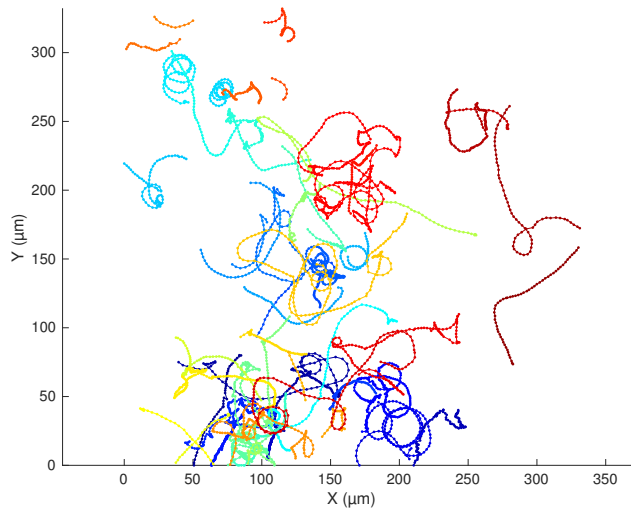
```

The data we used for this section came from the tracking of *Helicobacter pylori*, a pathogenic bacteria responsible for ulcers amongst other things. Their movement resembles this:

```

>> figure
>> hold on
>> c = jet(n_tracks);
>> for s = 1 : n_tracks
>>     x = tracks{s}(:, 2);
>>     y = tracks{s}(:, 3);
>>     plot(x, y, '-.', 'Color', c(s, :))
>> end
>> axis equal
>> xlabel( [ 'X (' md.spaceUnits ')' ] )
>> ylabel( [ 'Y (' md.spaceUnits ')' ] )

```



8.3 Importing the spot feature table.

There are many limitations with the later approach. The first one being its inability to cope with complex tracks, as we said earlier. The second one being that we did not import the spot features (mean intensity, radius, etc) other than their position. To do so we have to move to more complex functions that directly interact with TrackMate data files.

The XML files are the ones in which TrackMate saves its tracking session, when you press the Save button on the GUI. Their first lines resemble this:

```
<?xml version="1.0" encoding="UTF-8"?>
  <TrackMate version="3.3.0">
    ...
```

They are made of several XML elements, the most important one being the Model:

```
<Model spatialunits="pixel" timeunits="sec">
```

The example file we use here comes from the lineage of a *C.elegans* embryo over the first 3 hours of development post first anaphase. It is made of four tracks: two for the polar bodies (PB1 and PB2), one for the AB lineage and one for the P1 lineage. The AB and P1 lineages take the shape of two tracks. In these tracks there are several cell divisions (5 for the AB lineage) so they are not linear and the approach of the previous section will fail.

```
>> % C.elegans lineage TrackMate data file.
>> file_path = '10-03-17-3hours.xml';
```

The function `trackmateSpots` will import the visible spots of this file as a MATLAB table. The table class is somewhat recent in MATLAB, so you need at least MATLAB v2013b to have this function working.

The function basic syntax is the following:

```
>> [ spot_table, spot_ID_map ] = trackmateSpots( file_path );
```

We will speak about the second output argument `spot_ID_map` later. We retrieve it now, for the functions that load a TrackMate file in MATLAB can take quite some time to run, so you want to run them only once.

There is another optional input argument we did not speak about here. You can pass as a second argument a cell array of strings containing the names of the features you want to import. If this list is empty or if the second argument is not present, all spot features are retrieved. Read the help of the `trackmateSpots` to learn more about its full syntax.

The table returned contains all the spot features by default, and spots are listed by frame order:

```
>> spot_table( 1 : 6, { 'ID', 'name', 'FRAME' } )
```

```
ans =
```

ID	name	FRAME
0	'AB'	0
1	'PB1'	0
2	'P1'	0
3	'PB2'	0
47360	'PB1'	1
47361	'PB2'	1

It is best to read the table documentation to take full advantage of it. Nonetheless, here is a few things we can do with it. The lineage was created with TrackMate, and the spot features we measured are imported with name and units:

```
>> % List a subset of features, there are 32 of them.
>> feature_subset = [ 1, 2, 13, 17, 22, 23, 24 ];
>>
>> [ spot_table.Properties.VariableNames(feature_subset)
>>   spot_table.Properties.VariableDescriptions(feature_subset)
>>   spot_table.Properties.VariableUnits(feature_subset) ]
```

```
ans =
```

```
Columns 1 through 3
```

'ID'	'name'	'ESTIMATED_DIAMETER'
'Spot ID'	'Spot name'	'Estimated diameter'
''	''	'µm'

```
Columns 4 through 7
```

'MEAN_INTENSITY'	'POSITION_X'	'POSITION_Y'	'POSITION_Z'
'Mean intensity'	'X'	'Y'	'Z'
'Counts'	'µm'	'µm'	'µm'

Values can be accessed directly with the feature name:

```
>> t = spot_table.FRAME;  
>> t(10:16)
```

ans =

```
2  
2  
2  
3  
3  
3  
3
```

Values can then be used to slice through the table:

```
% All spots in frame 10:  
>> index = (t == 10);  
>> spot_table(index, feature_subset)
```

ans =

ID	name	ESTIMATED_DIAMETER	MEAN_INTENSITY
51336	'P2'	5.5835	1943.2
51337	'AB.a'	6.9765	1374
51339	'EMS'	5.0021	1223.4
51340	'AB.p'	7.1779	1183.3
51334	'PB1'	2.482	1949.4
51335	'PB2'	4.4596	1729.9

POSITION_X	POSITION_Y	POSITION_Z
55.868	36.022	18
18.094	15.977	19
42.042	24.312	19
31.325	37.146	22
10.353	9.9564	16
25.437	23.386	15

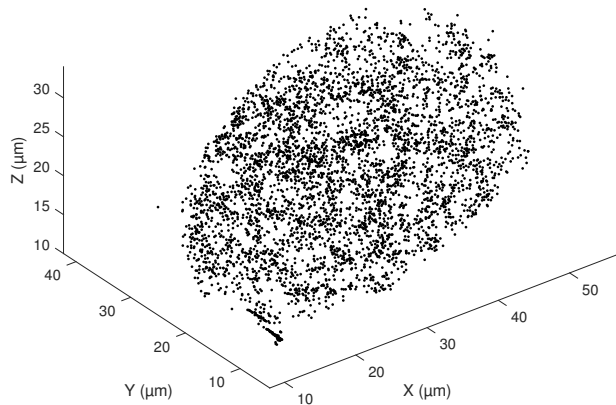
You can redisplay the spot cloud in MATLAB:

```
>> x = spot_table.POSITION_X;  
>> y = spot_table.POSITION_Y;  
>> z = spot_table.POSITION_Z;  
>>  
>> figure
```

```

>> plot3(x, y, z, 'k.')
>> axis equal
>> units = char(spot_table.Properties.VariableUnits(22));
>> xlabel( [ 'X (' units ')' ] )
>> ylabel( [ 'Y (' units ')' ] )
>> zlabel( [ 'Z (' units ')' ] )

```



Notice that there is no time information on this plot, nor track information. We just imported the spots so far, and there is not yet data on how spots are linked.

8.4 Importing the edge track table.

Edges - or links - are what assemble these spots in tracks. Each edge represents a link from a spot (the source spot) to another (the target spot). In TrackMate, edges are directed and oriented towards time: the target has spot always a FRAME value strictly larger than the source spot. Beyond this, there is no restrictions. A spot can be the source or target of many links or none.

Edges have features too. These are values that make sense only for links, such as velocity. The edge features are imported using a function similar to the one for spots:

```

>> edge_map = trackmateEdges( file_path );

```

However here the data is not directly returned as a table but as a map of table, one for each track:

```

>> % What is the output class?
>> class( edge_map )

```

ans =

```

containers.Map

```

```

>> % The track names are used as keys in the map:
>> track_names = edge_map.keys

```



```
track_names =
    'AB'    'P1'    'PB1'    'PB2'
```

```
>> % How many tracks?
>> n_tracks = numel( track_names )
```

```
n_tracks =
    4
```

The values of the map are edge feature tables:

```
>> ab_edges = edge_map('AB');
>> class( ab_edges )
>> ab_edges(1:6, 1:5)
```

```
ans =
```

```
table
```

```
ans =
```

SPOT_SOURCE_ID	SPOT_TARGET_ID	DISPLACEMENT
3.8376e+05	3.8381e+05	3.5395
3.8534e+05	3.8543e+05	0.7715
2.2182e+05	2.0921e+05	1.3444
3.5751e+05	3.5323e+05	1.0961
2.1763e+05	2.2603e+05	0.49616
2.6826e+05	2.9371e+05	0.3138

EDGE_TIME	EDGE_X_LOCATION
119	23.568
143	32.317
109	13.347
167	34.781
105	38.949
121	22.476

Feature names and units are imported too:

```
>> [ ab_edges.Properties.VariableNames
>>   ab_edges.Properties.VariableDescriptions
>>   ab_edges.Properties.VariableUnits ]
```

ans =

Columns 1 through 3

'SPOT_SOURCE_ID'	'SPOT_TARGET_ID'	'DISPLACEMENT'
'Source spot ID'	'Target spot ID'	'Displacement'
'no unit'	'no unit'	' μm '

Columns 4 through 6

'EDGE_TIME'	'EDGE_X_LOCATION'	'EDGE_Y_LOCATION'
'Time (mean)'	'X Location (mean)'	'Y Location (mean)'
'min'	' μm '	' μm '

Columns 7 through 9

'EDGE_Z_LOCATION'	'LINK_COST'	'VELOCITY'
'Z Location (mean)'	'Link cost'	'Velocity'
' μm '	'no unit'	' $\mu\text{m}/\text{min}$ '

The `trackmateEdges` has a similar syntax to the `trackmateSpots` function for optional input arguments. It is detailed in its help section. There are two key features in these edge tables: the `SPOT_SOURCE_ID` and the `SPOT_TARGET_ID`. They are the ones with which we can rebuild tracks in `TrackMate`:

```
>> track_spot_IDs = cell( n_tracks, 1 );
>> for s = 1 : n_tracks
>>
>>     track_name = track_names{s};
>>     edge_table = edge_map( track_name );
>>     track_spot_IDs{ s } = unique( [
>>         edge_table.SPOT_SOURCE_ID
>>         edge_table.SPOT_TARGET_ID
>>     ] );
>>
>> end
```

We now have the IDs of the spots that are in specified tracks. The problem is that these IDs are spot IDs, and we have a spot table in which the table row does not match the spot ID. This is where the second output argument of the `trackmateSpots` function is useful. `spot_ID_map` is a map that links spot IDs to row number in the spot table. It is used as follow:

```
>> % Retrieve the spot with ID:
>> spot_ID = 3087;
>> r = spot_ID_map( spot_ID )
```

r =

12

```
>> spot_table( r, feature_subset )
```

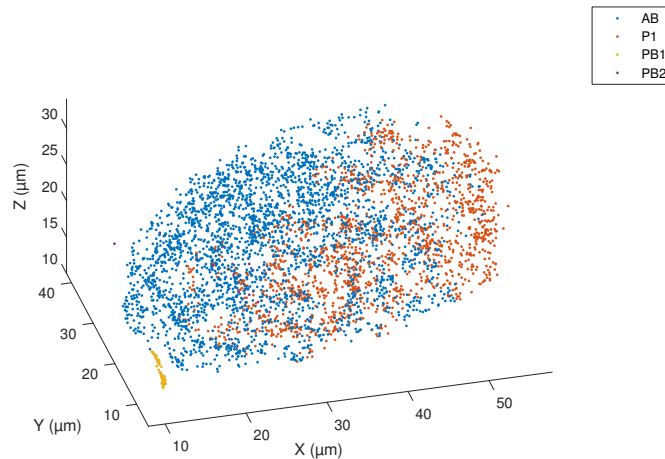
```
ans =
```

ID	name	ESTIMATED_DIAMETER	MEAN_INTENSITY
3087	'AB'	6.5642	1187.2

POSITION_X	POSITION_Y	POSITION_Z
30.266	28.282	19

We can use it to retrieve the position of the spots in each track:

```
>> figure
>> hold on
>> for s = 1 : n_tracks
>>
>>     track_name = track_names{ s };
>>     track_spot_ID = track_spot_IDs{ s };
>>
>>     % To extract several values all at once, we have to play
>>     % with cell arrays and the value map method:
>>     rows = cell2mat(spot_ID_map.values(num2cell(track_spot_ID)));
>>
>>     x = spot_table.POSITION_X( rows );
>>     y = spot_table.POSITION_Y( rows );
>>     z = spot_table.POSITION_Z( rows );
>>
>>     % Plot the tracks by coloring spots.
>>     plot3( x, y, z, '.', 'DisplayName', track_name)
>>
>> end
>>
>> xlabel( [ 'X (' units ')' ] )
>> ylabel( [ 'Y (' units ')' ] )
>> zlabel( [ 'Z (' units ')' ] )
>> view(-15, 30)
>> axis equal
>> legend toggle
```



8.5 Importing TrackMate data as a MATLAB graph.

We are still missing one piece of information on the latest plot, which is the connectivity between spot. We can retrieve it from the edges we imported in the previous section, and accessing the `SPOT_SOURCE_ID` and the `SPOT_TARGET_ID` features. A better solution is to directly import the whole data as a MATLAB graph.

The graph is the ultimate solution to represent complex tracks, manipulate and inspect them for finer analysis. A mathematical graph is a data structure made of vertices (in our case, spots) connected by edges (in our case links between spots). Graphs have numerous uses, which fostered the advent of graph theory and its applications. With a graph data structure, you get the tools to iterate, partition, edit and investigate the data like you could never do with linear data structures.

MATLAB offers two main graph classes, one for undirected graphs (the direction of edges do not matter) and directed graphs (edges are directed). We rely of course on the later, which is named `digraph`. It was introduced in MATLAB R2015b, so you need at least this version for what follows.

The function `trackmateGraph` imports the whole TrackMate data as a MATLAB `digraph`. Doing so, spot and edge features are imported as well, so this function can replace the two preceding ones. It offers optional arguments to import a subset of features, as for `trackmateSpots` and `trackmateEdges`, plus an extra optional flag for verbosity. They are all documented in the help section of the function.

```
>> % Import all features and be verbose during import.
>> G = trackmateGraph( file_path, [], [], true );
```

```
Importing spot table. Done in 20.6 s.
Importing edge table. Done in 6.3 s.
Building graph. Done in 0.2 s.
```

The graph structure stores the spots and links features in tables:

```
>> % Spots.
>> G.Nodes(1:5, feature_subset)
```

ans =

ID	name	ESTIMATED_DIAMETER	MEAN_INTENSITY
0	'AB'	4.5477	1129.4
1	'PB1'	6.3325	2418.9
2	'P1'	2.7192	1142.6
3	'PB2'	4.5137	671.54
47360	'PB1'	4.5293	757.12

POSITION_X	POSITION_Y	POSITION_Z
34.136	30.564	21
10.221	11.61	16
43.266	32.549	20
9.9233	27.587	22
10.32	11.511	16

>> % Edges.

>> G.Edges(1:6, 1:5)

ans =

EndNodes	SPOT_SOURCE_ID	SPOT_TARGET_ID
1 8	0	47363
2 5	1	47360
3 7	2	47362
4 6	3	47361
5 9	47360	3084
6 10	47361	3085

DISPLACEMENT	EDGE_TIME
2.2946	1
0.14034	1
2.8798	1
18.875	1
0.19847	3
6.8934	3

These tables have the same shape that the tables imported by trackmateSpots and trackmateEdges, except for the edge table, whose first column EndNodes is a $N \times 2$ array that stores the source and target indices of nodes in the spot table. Careful: these indices are **row numbers** in the spot table.

```

>> % Access one edge:
>> i_edge = 28; % 28th edge from AB to AB.p
>> edge = G.Edges(i_edge, 1:5);
>> source = G.Nodes( edge.EndNodes(1), feature_subset )

```

```
source =
```

ID	name	ESTIMATED_DIAMETER	MEAN_INTENSITY
27303	'AB'	4.7838	1457.9
	POSITION_X	POSITION_Y	POSITION_Z
	26.396	25.801	21

```
>> target = G.Nodes( edge.EndNodes(2), feature_subset )
```

```
target =
```

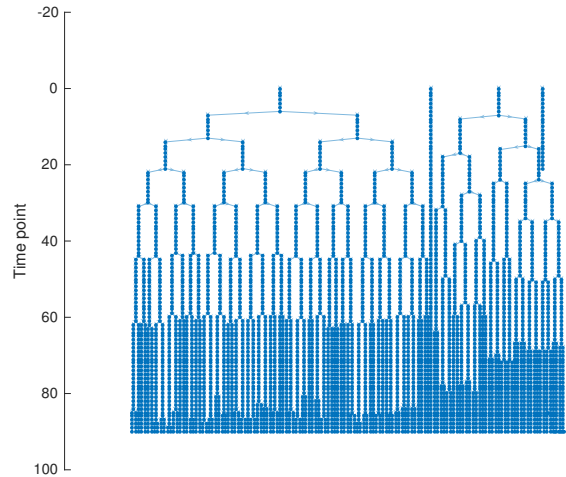
ID	name	ESTIMATED_DIAMETER	MEAN_INTENSITY
31404	'AB.p'	4.5622	834.89
	POSITION_X	POSITION_Y	POSITION_Z
	27.488	29.737	23

We can now rely on MATLAB facilities to lay out the graph. For instance you can create a graph display that resembles TrackScheme using the `layered` option of the `plot` function, and setting the nodes Y coordinates to the spot frame:

```

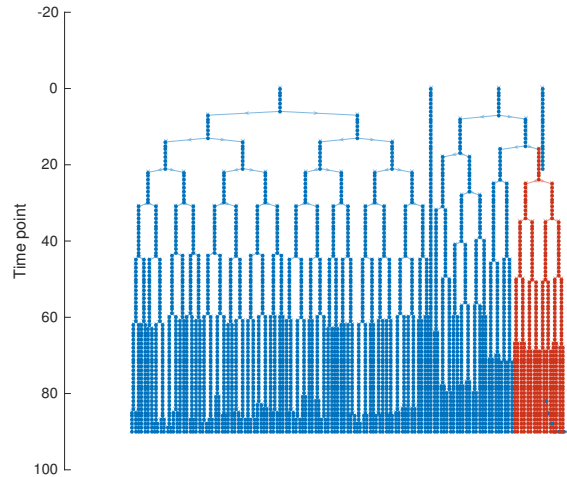
>> figure
>> hp = plot(G, 'layout', 'layered');
>> set(hp, 'YData', G.Nodes.FRAME);
>> set(gca, 'YDir', 'reverse', 'XColor', 'none')
>> ylabel('Time point')
>> box off

```



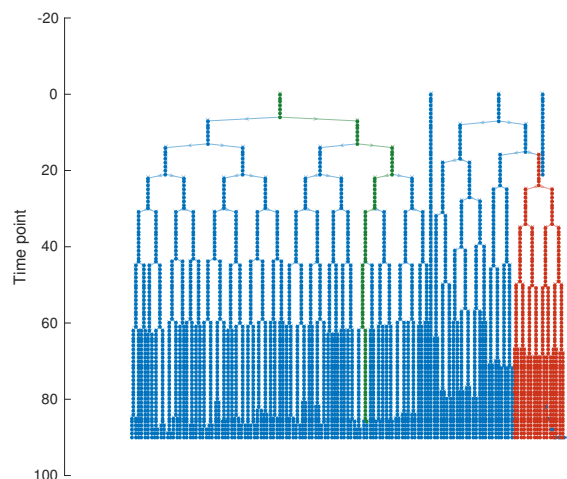
As said above, there is a rich collection of tools offered to manipulate a graph. Here is a few examples. For instance, suppose we want to color on this graph all the descendant of the MS cell.

```
>> % Find the first occurrence of MS using spot names.
>> index_MS = find( strcmp( G.Nodes.name, 'MS'), 1 );
>>
>> % Use depth first iterator to have all its descendant. This will
>> % work since our graph is a directed graph. The following
>> % instruction generates a table with nodes indices (when they are
>> % met for the first time) and edge indices (when they are
>> % traversed).
>> t = dfsearch(G, index_MS, { 'discovernode', 'edgetonew' } );
>>
>> % We have to prune NaNs if we want a separate list of nodes and
>> % edges.
>> v = t.Node;
>> v = v( ~isnan(v) );
>> e = t.Edge;
>> e = e( ~isnan(e(:,1)), : );
>>
>> % Highlight them in the plot:
>> % Nodes
>> coll = [ 0.8 0.2 0.1 ];
>> highlight( hp, v, 'NodeColor', coll)
>> % Edges
>> highlight( hp, e(:,1), e(:,2), 'EdgeColor', coll)
```



Now let's find a path in the graph, from AB to AB.araapp

```
>> index_AB1 = find( strcmp( G.Nodes.name, 'AB'), 1 );
>> index_AB2 = find( strcmp( G.Nodes.name, 'AB.araapp'), 1 );
>>
>> path_AB = shortestpath( G, index_AB1, index_AB2 );
>> col2 = [ 0.1 0.5 0.2 ];
>> % The highlight function can color edges of a path automatically.
>> highlight( hp, path_AB, 'NodeColor', col2, 'EdgeColor', col2 )
```



We can also layout the graph using the spot coordinates for the nodes in the plot. Unfortunately, the plot function of MATLAB allows for specifying on the X and Y coordinates. If we want to reproduce the tracks in 3D with cell fate, we have to generate our own plotting function. Here is a procedure adapted from the work of John Gilbert:

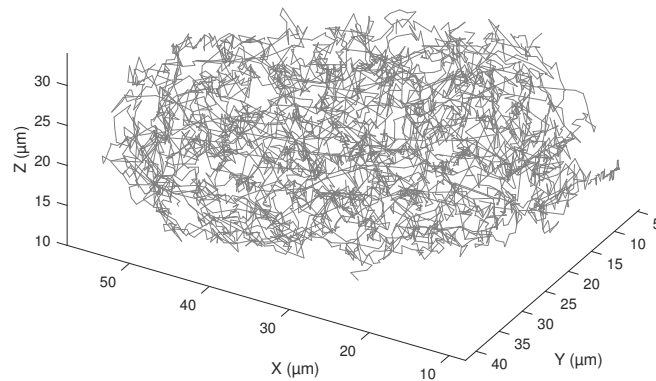
```
>> % Get X, Y, Z coordinates.
>> x = G.Nodes.POSITION_X;
>> y = G.Nodes.POSITION_Y;
>> z = G.Nodes.POSITION_Z;
>>
```



```

>> % Get links source and target.
>> s = G.Edges.EndNodes( : , 1 );
>> t = G.Edges.EndNodes( : , 2 );
>>
>> % We intercalate NaNs between node pairs to have a line for
>> % each edge.
>> n_nodes = numel(s);
>> X = [ x(s) x(t) NaN( n_nodes, 1) ]';
>> Y = [ y(s) y(t) NaN( n_nodes, 1) ]';
>> Z = [ z(s) z(t) NaN( n_nodes, 1) ]';
>> X = X(:);
>> Y = Y(:);
>> Z = Z(:);
>>
>> figure
>> plot3( X, Y, Z, '-', 'Color', [ 0.5 0.5 0.5 ] )
>> xlabel( [ 'X (' units ')' ] )
>> ylabel( [ 'Y (' units ')' ] )
>> zlabel( [ 'Z (' units ')' ] )
>> view(-150, 30)
>> axis equal

```



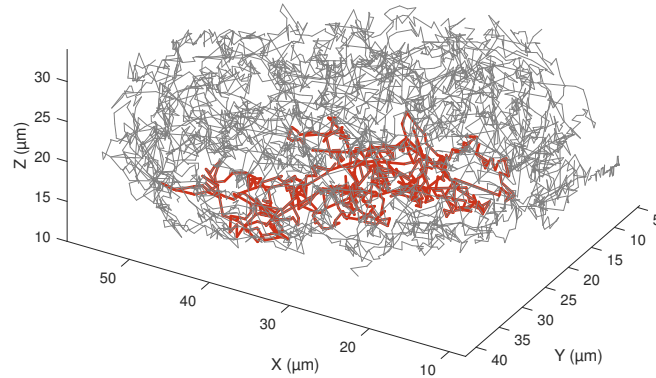
We can also reuse the edges of the MS descendant calculated above to repaint these descendants in another color on this 3D plot:

```

>> % The variable e stores MS descendants edges.
>> s = e(:,1);
>> t = e(:,2);
>>
>> % Same procedure otherwise:
>> n_nodes = numel(s);
>> Xms = [ x(s) x(t) NaN( n_nodes, 1) ]';
>> Yms = [ y(s) y(t) NaN( n_nodes, 1) ]';
>> Zms = [ z(s) z(t) NaN( n_nodes, 1) ]';
>> Xms = Xms(:);
>> Yms = Yms(:);
>> Zms = Zms(:);
>>

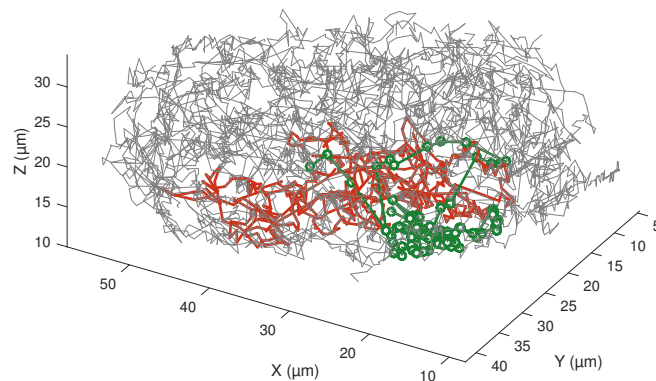
```

```
>> hold on
>> plot3( Xms, Yms, Zms, '-', 'Color', col1, 'LineWidth', 2)
```



Now if we want to color a path, we have to access the edges of this path, which we cannot do with the simple `shortestpath` function. We have to rely on `shortestpathtree`, which returns a digraph with the relevant edges only:

```
>> path_AB2 = shortestpathtree( G, index_AB1, index_AB2 );
>>
>> % We get edges from the created digraph:
>> s = path_AB2.Edges.EndNodes(:,1);
>> t = path_AB2.Edges.EndNodes(:,2);
>>
>> % Same procedure otherwise:
>> n_nodes = numel(s);
>> Xab = [ x(s) x(t) NaN( n_nodes, 1) ]';
>> Yab = [ y(s) y(t) NaN( n_nodes, 1) ]';
>> Zab = [ z(s) z(t) NaN( n_nodes, 1) ]';
>> Xab = Xab(:);
>> Yab = Yab(:);
>> Zab = Zab(:);
>>
>> hold on
>> plot3( Xab, Yab, Zab, 'o-', 'Color', col2, 'LineWidth', 2, ...
>>       'MarkerFaceColor', 'w')
```



8.6 Other MATLAB functions for TrackMate.

The three sections above presented the core of interoperability between TrackMate and MATLAB. The main function is `trackmateGraph` but `trackmateSpots` and `trackmateEdges` can be used advantageously when the whole graph is not required for analysis. The logic used in these three functions can be reproduced and translated to languages other than MATLAB.

There are two supplemental functions that do not import the track data but are useful to probe the metadata stored in a TrackMate file. `trackmateImageCalibration` is able to read the physical calibration of the image on which TrackMate operated:

```
>> cal = trackmateImageCalibration( file_path );
>> cal.x
```

```
ans =
```

```
start: 0
end: 348
size: 349
value: 0.1985
units: 'µm'
```

The function `trackmateFeatureDeclarations` is used to probe what features are declared in the TrackMate file:

```
>> [ spot_fd, edge_fd, track_df ] = ...
>> trackmateFeatureDeclarations( file_path );
```

They are returned as 3 maps, one for spot, edge and track feature declarations. These maps use the feature names as keys:

```
>> edge_fd( 'VELOCITY' )
```

```
ans =
```

```
key: 'VELOCITY'
name: 'Velocity'
shortName: 'V'
dimension: 'VELOCITY'
isInt: 0
units: 'µm/min'
```

8.7 Application examples and links.

There are some specialized tools in MATLAB that can exploit TrackMate results. For instance, here is a MATLAB class that performs [analysis](#). It is hopefully well documented in this [MATLAB tutorial](#).

9. Scripting TrackMate in Python.

[TrackMate](#) can be used out of the GUI, using a scripting language that allows making calls to Java. The most simple way to get started is to use the [Script Editor](#) of Fiji, which takes care of the difficult and boring part for you (such as path). The examples proposed on this page all use Jython, but can be adapted to anything.

Since we are calling the internals of TrackMate, we must get to know a bit of its internal design. There are three main classes to interact with in a script:

- Model ([fiji.plugin.trackmate.Model](#)) is the class in charge of storing the data. It cannot do anything to create it. It can help you follow manual modifications you would made in the manual editing mode, interrogate it, ... but it is conceptually just a data recipient.
- Settings ([fiji.plugin.trackmate.Settings](#)) is the class storing the fields that will configure TrackMate and pilot how the data is created. This is where you specify what is the source image, what are the detector and tracking algorithms to use, what are the filters to use, *etc.*
- TrackMate ([fiji.plugin.trackmate.TrackMate](#)) is the class that does the actual work. In scripts, we use it to actually perform the analysis tasks, such as generating spots from images, linking them into track, *etc.* It reads configuration information in the Settings object mentioned above and put the resulting data in the model.

So getting a working script is all about configuring a proper Settings object and calling `exec*` methods on a TrackMate object. Then we read the results in the Model object.

9.1 A full example.

Here is an example of full tracking process, using the easy image found in the [first tutorial](#). The following (Jython) script works as following:

- It fetches the image from the web.
- It configures settings for segmentation and tracking.
- The model is instantiated, with the settings and imp objects.
- The TrackMate class is instantiated with the model object.
- Then the TrackMate object performs all the steps needed.
- The final results is displayed as an overlay.

```
from fiji.plugin.trackmate import Model
from fiji.plugin.trackmate import Settings
from fiji.plugin.trackmate import TrackMate
from fiji.plugin.trackmate import SelectionModel
from fiji.plugin.trackmate import Logger
from fiji.plugin.trackmate.detection import LogDetectorFactory
from fiji.plugin.trackmate.tracking.sparselap import SparseLAPTrackerFactory
from fiji.plugin.trackmate.tracking import LAPUtils
```

```

from ij import IJ, WindowManager
import fiji.plugin.trackmate.visualization.hyperstack.HyperStackDisplayer as
    ↳ HyperStackDisplayer
import fiji.plugin.trackmate.features.FeatureFilter as FeatureFilter
import sys
import fiji.plugin.trackmate.features.track.TrackDurationAnalyzer as TrackDurationAnalyzer

# Get currently selected image
#imp = WindowManager.getCurrentImage()
imp = IJ.openImage('http://fiji.sc/samples/FakeTracks.tif')
imp.show()

#-----
# Create the model object now
#-----

# Some of the parameters we configure below need to have
# a reference to the model at creation. So we create an
# empty model now.

model = Model()

# Send all messages to ImageJ log window.
model.setLogger(Logger.IJ_LOGGER)

#-----
# Prepare settings object
#-----

settings = Settings()
settings.setFrom(imp)

# Configure detector - We use the Strings for the keys
settings.detectorFactory = LogDetectorFactory()
settings.detectorSettings = {
    'DO_SUBPIXEL_LOCALIZATION' : True,
    'RADIUS' : 2.5,
    'TARGET_CHANNEL' : 1,
    'THRESHOLD' : 0.,
    'DO_MEDIAN_FILTERING' : False,
}

# Configure spot filters - Classical filter on quality
filter1 = FeatureFilter('QUALITY', 30, True)
settings.addSpotFilter(filter1)

```

```

# Configure tracker - We want to allow merges and fusions
settings.trackerFactory = SparseLAPTrackerFactory()
settings.trackerSettings = LAPUtils.getDefaultLAPSettingsMap()
# almost good enough
settings.trackerSettings['ALLOW_TRACK_SPLITTING'] = True
settings.trackerSettings['ALLOW_TRACK_MERGING'] = True

# Configure track analyzers - Later on we want to filter out tracks
# based on their displacement, so we need to state that we want
# track displacement to be calculated. By default, out of the GUI,
# not features are calculated.

# The displacement feature is provided by the TrackDurationAnalyzer.

settings.addTrackAnalyzer(TrackDurationAnalyzer())

# Configure track filters - We want to get rid of the two immobile
# spots at the bottom right of the image. Track displacement must
# be above 10 pixels.

filter2 = FeatureFilter('TRACK_DISPLACEMENT', 10, True)
settings.addTrackFilter(filter2)

#-----
# Instantiate plugin
#-----

trackmate = TrackMate(model, settings)

#-----
# Process
#-----

ok = trackmate.checkInput()
if not ok:
    sys.exit(str(trackmate.getErrorMessage()))

ok = trackmate.process()
if not ok:
    sys.exit(str(trackmate.getErrorMessage()))

#-----
# Display results
#-----

selectionModel = SelectionModel(model)
displayer = HyperStackDisplayer(model, selectionModel, imp)

```

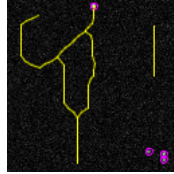
```

displayer.render()
displayer.refresh()

# Echo results with the logger we set at start:
model.getLogger().log(str(model))

```

The results should look like this:



And the ImageJ log window should display information resembling this:

```

Starting detection process using 4 threads.
Detection processes 4 frames simultaneously and allocates 1 thread per frame.
Found 22883 spots.
Starting initial filtering process.
Computing spot features.
Computation done in 11 ms.
Starting spot filtering process.
Starting tracking process.
Computing track features:
  - Track duration in 2 ms.
Computation done in 17 ms.
...

```

9.2 Loading and reading from a saved TrackMate XML file.

Scripting is a good way to interrogate and play non-interactively with tracking results. The example below shows how to load a XML TrackMate file and rebuild a full working model from it. That way you could for instance redo a full tracking process by only changing one parameter with respect to the saved one. You might also want to check results without relying on the GUI, *etc.*

For the example below to work for you, you will have to edit line 20 and put the actual path to your TrackMate file.

```

from fiji.plugin.trackmate.visualization.hyperstack import HyperStackDisplayer
from fiji.plugin.trackmate.io import TmXmlReader
from fiji.plugin.trackmate import Logger
from fiji.plugin.trackmate import Settings
from fiji.plugin.trackmate import SelectionModel
from fiji.plugin.trackmate.providers import DetectorProvider
from fiji.plugin.trackmate.providers import TrackerProvider
from fiji.plugin.trackmate.providers import SpotAnalyzerProvider
from fiji.plugin.trackmate.providers import EdgeAnalyzerProvider
from fiji.plugin.trackmate.providers import TrackAnalyzerProvider
from java.io import File
import sys

```

```

#-----
# Setup variables
#-----

# Put here the path to the TrackMate file you want to load
file = File('/Users/tinevez/Desktop/Data/FakeTracks.xml')

# We have to feed a logger to the reader.
logger = Logger.IJ_LOGGER

#-----
# Instantiate reader
#-----

reader = TmXmlReader(file)
if not reader.isReadingOk():
    sys.exit(reader.getErrorMessage())

#-----
# Get a full model
#-----

# This will return a fully working model, with everything
# stored in the file. Missing fields (e.g. tracks) will be
# null or None in python
model = reader.getModel()
# model is a fiji.plugin.trackmate.Model

#-----
# Display results
#-----

# We can now plainly display the model. It will be shown on an
# empty image with default magnification.
sm = SelectionModel(model)
displayer = HyperStackDisplayer(model, sm)
displayer.render()

#-----
# Get only part of the data stored in the file
#-----

# You might want to access only separate parts of the
# model.

spots = model.getSpots()
# spots is a fiji.plugin.trackmate.SpotCollection

```



```

logger.log(str(spots))

# If you want to get the tracks, it is a bit trickier.
# Internally, the tracks are stored as a huge mathematical
# simple graph, which is what you retrieve from the file.
# There are methods to rebuild the actual tracks, taking
# into account for everything, but frankly, if you want to
# do that it is simpler to go through the model:

trackIDs = model.getTrackModel().trackIDs(True)
# only filtered tracks.
for id in trackIDs:
    logger.log(str(id) + ' - ' + str(model.getTrackModel().trackEdges(id)))

#-----
# Building a settings object from a file
#-----

# Reading the Settings object is actually currently complicated.
# The reader wants to initialize properly everything you saved
# in the file, including the spot, edge, track analyzers, the
# filters, the detector, the tracker, etc...
# It can do that, but you must provide the reader with providers,
# that are able to instantiate the correct TrackMate Java classes
# from the XML data.

# We start by creating an empty settings object
settings = Settings()

# Then we create all the providers, and point them to the target
# model:
detectorProvider      = DetectorProvider()
trackerProvider       = TrackerProvider()
spotAnalyzerProvider  = SpotAnalyzerProvider()
edgeAnalyzerProvider  = EdgeAnalyzerProvider()
trackAnalyzerProvider = TrackAnalyzerProvider()

# Now we can flesh out our settings object:
reader.readSettings(settings, detectorProvider, trackerProvider, spotAnalyzerProvider,
    ↪ edgeAnalyzerProvider, trackAnalyzerProvider)

logger.log(str('\n\nSETTINGS:'))
logger.log(str(settings))

# The settings object is also instantiated with the target image.
# Note that the XML file only stores a link to the image.
# If the link is not valid, the image will not be found.

```

```

imp = settings.imp
imp.show()

# With this, we can overlay the model and the source image:
displayer = HyperStackDisplayer(model, sm, imp)
displayer.render()

```

9.3 Export spot, edge and track numerical features after tracking.

This example shows how to extract numerical features from tracking results.

TrackMate computes and stores three kind of numerical features:

- Spot features, such as a spot location (X, Y, Z), its mean intensity, radius *etc.*
- Edge or link features: An edge is a link between two spots. Its feature typically stores the velocity and displacement, which are defined only for two consecutive spots in the same track.
- Track features: numerical features that apply to a whole track, such as the number of spots it contains.

By default, TrackMate only computes a very limited number of features. The GUI forces TrackMate to compute them all, but if you do scripting, you will have to explicitly configure TrackMate to compute the features you desire. This is done by adding feature analyzers to the settings object.

There are some gotchas: some feature analyzers require other numerical features to be already calculated. If something does not work, it is a good idea to directly check the preamble in the source code of the analyzers ([TrackMate feature logic](#)).

Finally, depending on their type, numerical features are not stored at the same place:

- Spot features are simply conveyed by the spot object, and you can access them through `spot.getFeature('FEATURE_NAME')`
- Edge and track features are stored in a sub-component of the model object called the FeatureModel ([FeatureModel.java](#)).

Check the script below to see a working example.

```

from ij import IJ, ImagePlus, ImageStack
import fiji.plugin.trackmate.Settings as Settings
import fiji.plugin.trackmate.Model as Model
import fiji.plugin.trackmate.SelectionModel as SelectionModel
import fiji.plugin.trackmate.TrackMate as TrackMate
import fiji.plugin.trackmate.Logger as Logger
import fiji.plugin.trackmate.detection.DetectorKeys as DetectorKeys
import fiji.plugin.trackmate.detection.DogDetectorFactory as DogDetectorFactory
import fiji.plugin.trackmate.tracking.sparselap.SparseLAPTrackerFactory as
    ↪ SparseLAPTrackerFactory
import fiji.plugin.trackmate.tracking.LAPUtils as LAPUtils

```

```

import fiji.plugin.trackmate.visualization.hyperstack.HyperStackDisplayer as
    ↪ HyperStackDisplayer
import fiji.plugin.trackmate.features.FeatureFilter as FeatureFilter
import fiji.plugin.trackmate.features.FeatureAnalyzer as FeatureAnalyzer
import fiji.plugin.trackmate.features.spot.SpotContrastAndSNRAnalyzerFactory as
    ↪ SpotContrastAndSNRAnalyzerFactory
import fiji.plugin.trackmate.action.ExportStatsToIJAction as ExportStatsToIJAction
import fiji.plugin.trackmate.io.TmXmlReader as TmXmlReader
import fiji.plugin.trackmate.action.ExportTracksToXML as ExportTracksToXML
import fiji.plugin.trackmate.io.TmXmlWriter as TmXmlWriter
import fiji.plugin.trackmate.features.ModelFeatureUpdater as ModelFeatureUpdater
import fiji.plugin.trackmate.features.SpotFeatureCalculator as SpotFeatureCalculator
import fiji.plugin.trackmate.features.spot.SpotContrastAndSNRAnalyzer as
    ↪ SpotContrastAndSNRAnalyzer
import fiji.plugin.trackmate.features.spot.SpotIntensityAnalyzerFactory as
    ↪ SpotIntensityAnalyzerFactory
import fiji.plugin.trackmate.features.track.TrackSpeedStatisticsAnalyzer as
    ↪ TrackSpeedStatisticsAnalyzer
import fiji.plugin.trackmate.util.TMUtils as TMUtils

# Get currently selected image
#imp = WindowManager.getCurrentImage()
imp = IJ.openImage('http://fiji.sc/samples/FakeTracks.tif')
#imp.show()

#-----
# Instantiate model object
#-----

model = Model()

# Set logger
model.setLogger(Logger.IJ_LOGGER)

#-----
# Prepare settings object
#-----

settings = Settings()
settings.setFrom(imp)

# Configure detector
settings.detectorFactory = DogDetectorFactory()
settings.detectorSettings = {
    DetectorKeys.KEY_DO_SUBPIXEL_LOCALIZATION : True,
    DetectorKeys.KEY_RADIUS : 2.5,
    DetectorKeys.KEY_TARGET_CHANNEL : 1,

```

```

    DetectorKeys.KEY_THRESHOLD : 5.,
    DetectorKeys.KEY_DO_MEDIAN_FILTERING : False,
}

# Configure tracker
settings.trackerFactory = SparseLAPTrackerFactory()
settings.trackerSettings = LAPUtils.getDefaultLAPSettingsMap()
settings.trackerSettings['LINKING_MAX_DISTANCE'] = 10.0
settings.trackerSettings['GAP_CLOSING_MAX_DISTANCE']=10.0
settings.trackerSettings['MAX_FRAME_GAP']= 3

# Add the analyzers for some spot features.
# You need to configure TrackMate with analyzers that will generate
# the data you need.
# Here we just add two analyzers for spot, one that computes generic
# pixel intensity statistics (mean, max, etc...) and one that
# computes an estimate of each spot's SNR.
# The trick here is that the second one requires the first one to be
# in place. Be aware of this kind of gotchas, and read the docs.
settings.addSpotAnalyzerFactory(SpotIntensityAnalyzerFactory())
settings.addSpotAnalyzerFactory(SpotContrastAndSNRAnalyzerFactory())

# Add an analyzer for some track features, such as the track mean
# speed.
settings.addTrackAnalyzer(TrackSpeedStatisticsAnalyzer())

settings.initialSpotFilterValue = 1

print(str(settings))

#-----
# Instantiate trackmate
#-----

trackmate = TrackMate(model, settings)

#-----
# Execute all
#-----

ok = trackmate.checkInput()
if not ok:
    sys.exit(str(trackmate.getErrorMessage()))

ok = trackmate.process()
if not ok:
    sys.exit(str(trackmate.getErrorMessage()))

```

```

#-----
# Display results
#-----

model.getLogger().log('Found ' + str(model.getTrackModel().nTracks(True)) + ' tracks.')

selectionModel = SelectionModel(model)
displayer = HyperStackDisplayer(model, selectionModel, imp)
displayer.render()
displayer.refresh()

# The feature model, that stores edge and track features.
fm = model.getFeatureModel()

for id in model.getTrackModel().trackIDs(True):

    # Fetch the track feature from the feature model.
    v = fm.getTrackFeature(id, 'TRACK_MEAN_SPEED')
    model.getLogger().log('')
    model.getLogger().log('Track ' + str(id) + ': mean velocity = ' + str(v) + ' ' + model.
        ↳ getSpaceUnits() + '/' + model.getTimeUnits())

    track = model.getTrackModel().trackSpots(id)
    for spot in track:
        sid = spot.ID()
        # Fetch spot features directly from spot.
        x=spot.getFeature('POSITION_X')
        y=spot.getFeature('POSITION_Y')
        t=spot.getFeature('FRAME')
        q=spot.getFeature('QUALITY')
        snr=spot.getFeature('SNR')
        mean=spot.getFeature('MEAN_INTENSITY')
        model.getLogger().log('\tspot ID = ' + str(sid) + ': x='+str(x)+' , y='+str(y)+' , t='
            ↳ +str(t)+' , q='+str(q) + ' , snr='+str(snr) + ' , mean = ' + str(mean))

```

9.4 Manually creating a model.

TrackMate aims at combining automatic and manual tracking facilities. This is also the case when scripting: a part of the API offers to edit a model extensively. A few code patterns must be followed.

First, every edit must happen between a call to `model.beginUpdate()` and `model.endUpdate()`:

```

model.beginUpdate()
# ... do whatever you want to the model here.
model.endUpdate()

```

The reason for this is that TrackMate caches each modification made to its model. This is required because we can deal with a rather complex content. For instance: imagine you have

a single track that splits in two branches at some point. If you decide to remove the spot at the fork, a complex series of events will happen:

- First, three edges will be removed: the ones that were connected to the spot you just removed.
- Then the spot will actually be removed from the model.
- But then you need to recompute the tracks, because now, you have three tracks instead of one.
- But also: all the numerical features of the tracks are now invalid, and you need to recompute them.
- And what happens to the track name? What track, amongst the 3 new ones, will receive the old name?

Well, TrackMate does that for you automatically, but for the chain of events to happen timely, you must make your edits within this `model.beginUpdate() / model.endUpdate()` code block.

This script just shows you how to use this construct to build and populate a model from scratch. Appending content to a model is done by, sequentially:

- Creating spot objects. You have to provide their x, y, z location, as well as a radius and a quality value for each. At this stage, you don't provide at what frame (or time) they belong.
- This is done by adding the spot to the model, using `model.addSpotTo(Spot, frame)`, frame being a positive integer number.
- Then you create a link, or an edge as it is called in TrackMate, between two spots. You have to provide the link cost: `model.addEdge(Spot1, Spot2, cost)`.

Spot quality and link cost are useful to quantify automatic spot detection and linking. We typically use negative values for these two numbers when doing manual edits. The following script writes the letter T using spots and links.

```
import ij.gui.NewImage as NewImage
import fiji.plugin.trackmate.Settings as Settings
import fiji.plugin.trackmate.Model as Model
import fiji.plugin.trackmate.Logger as Logger
import fiji.plugin.trackmate.Spot as Spot
import fiji.plugin.trackmate.SelectionModel as SelectionModel
import fiji.plugin.trackmate.TrackMate as TrackMate
import fiji.plugin.trackmate.visualization.hyperstack.HyperStackDisplayer as
↳ HyperStackDisplayer
import fiji.plugin.trackmate.visualization.trackscheme.TrackScheme as TrackScheme
import fiji.plugin.trackmate.visualization.PerTrackFeatureColorGenerator as
↳ PerTrackFeatureColorGenerator
import fiji.plugin.trackmate.features.ModelFeatureUpdater as ModelFeatureUpdater
import fiji.plugin.trackmate.features.track.TrackIndexAnalyzer as TrackIndexAnalyzer
import ij.plugin.Animator as Animator
```

```

import math

# We just need a model for this script. Nothing else, since
# we will do everything manually.
model = Model()
model.setLogger(Logger.IJ_LOGGER)

# Well actually, we still need a bit:
# We want to color-code the tracks by their feature, for instance
# with the track index. But for this, we need to compute the
# features themselves.
#
# Manually, this is done by declaring what features interest you
# in a settings object, and creating a ModelFeatureUpdater that
# will listen to changes in the model, and compute the features
# on the fly.
settings = Settings()
settings.addTrackAnalyzer( TrackIndexAnalyzer() )
# If you want more, add more analyzers.

# The object in charge of keeping the numerical features
# up to date:
ModelFeatureUpdater( model, settings )
# Nothing more to do. When the model changes, this guy will be
# notified and recalculate all the features you declared in
# the settings object.

# Every manual edit to the model must be made
# between a model.beginUpdate() and a model.endUpdate()
# call, otherwise you will mess with the event signalling
# and feature calculation.
model.beginUpdate()

# The letter T.

s1 = None
for t in range(0, 5):
    x = 10 + t * 10
    if s1 is None:

        # When you create a spot, you always have to specify its x,
        # y, z coordinates (even if z=0 in 2D images), AND its
        # radius, AND its quality. We enforce these 5 values so as
        # to avoid any bad surprise in other TrackMate component.
        # Typically, we use negative quality values to tag spot
        # created manually.
        s1 = Spot(x, 10, 0, 1, -1)
        model.addSpotTo(s1, t)
    continue

```

```

s2 = Spot(x, 10, 0, 1, -1)
model.addSpotTo(s2, t)
# You need to specify an edge cost for the link you create
# between two spots. Again, we use negative costs to tag
# edges created manually.
model.addEdge(s1, s2, -1)
s1 = s2

# So that's how you manually build a model from scratch.
# The next lines just do more of this, to build something enjoyable.

middle = s2
s1 = s2
for t in range(0, 4):
    x = 60 + t * 10
    s2 = Spot(x, 10, 0, 1, -1)
    model.addSpotTo(s2, t + 5)
    model.addEdge(s1, s2, -1)
    s1 = s2

s1 = middle
for t in range(0, 16):
    y = 20 + t * 6
    s2 = Spot(50, y, 0, 1, -1)
    model.addSpotTo(s2, t + 5)
    model.addEdge(s1, s2, -1)
    s1 = s2

# Commit all of this.
model.endUpdate()
# This actually triggers the features to be recalculated.

# Prepare display.
sm = SelectionModel(model)
color = PerTrackFeatureColorGenerator(model, 'TRACK_INDEX')
# The last line does not work if you did not compute the
# 'TRACK_INDEX' feature earlier.

# The TrackScheme view is a bit hard to interpret.
trackscheme = TrackScheme(model, sm)
trackscheme.setDisplaySettings('TrackColoring', color)
trackscheme.render()

# You can create an hyperstack viewer without specifying any
# ImagePlus. It will then create a dummy one tuned to
# display the model content.
view = HyperStackDisplayer(model, sm)
# Display tracks as comets

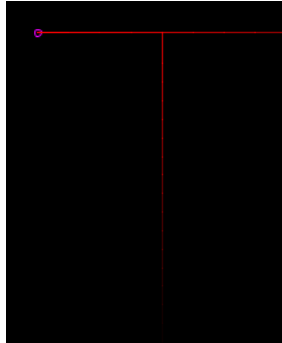
```



```
view.setDisplaySettings('TrackDisplaymode', 1)
view.setDisplaySettings('TrackDisplayDepth', 20)
view.setDisplaySettings('TrackColoring', color)
view.render()
```

```
# Animate it a bit
imp = view.getImp()
imp.getCalibration().fps = 30
Animator().run('start')
```

And here is the resulting image:



Part IV.

Extending TrackMate.

Do you have a tracking or a detection algorithm you want to implement? Of course you can write a whole software from scratch. But at some point you will have to design a model to hold the data, to write code that can load and save the results, visualize them, have even a minimalistic GUI, and allow for the curation of the algorithm outcome. This can be long, tedious and boring, while the part that interests you is just the algorithm you are working on.

We propose using TrackMate as a home for your algorithm. The framework is already there; it might not be perfect but can get your algorithm integrated very quickly. And then you can benefit from other modules, which provide GUI elements, visualization, *etc.* The subject of extending TrackMate is not completely trivial. However, recent advances in the [SciJava](#) package, brewed by the Fiji and ImageJ2 teams considerably simplified the task. It should be of no difficulty for an average Java developer. The following tutorials show how to integrate a module of each kind in TrackMate. They are listed by increasing complexity, and it is a good idea to practice them in this order.

10. How to write your own edge feature analyzer algorithm for TrackMate.

10.1 Introduction.

This page is a tutorial that shows how to integrate your own edge feature analyzer algorithm in TrackMate. It is the first in the series of tutorials dedicated to TrackMate extension, and should be read first by scientists willing to extend TrackMate.

All these tutorials assume you are familiar with Java development. You should be at ease with java core concepts such as object oriented design, inheritance, interfaces, *etc.* Ideally you would even know that maven exists and that it can help you to compile software. Beyond this, the tutorials will provide what you need to know.

Edge feature analyzers are algorithms that can associate one or more scalar numerical features to an edge, or a link between two spots in TrackMate. For instance, the instantaneous velocity is an edge feature (you need two linked spots to compute a displacement and a time interval), which happens to be provided by the algorithm named [EdgeVelocityAnalyzer.java](#).

10.2 TrackMate modules.

TrackMate is extended by writing *modules*. Modules are just the basic algorithms that provide TrackMate with core functionality, that the GUI and API wrap. There are 7 classes of modules:

- detection algorithms
- particle-linking algorithms
- numerical features for spots (such as mean intensity, *etc.*)

- numerical features for links (such as velocity, orientation, *etc.*)
- numerical features for tracks (total displacement, length, *etc.*)
- visualization tools
- post-processing actions (exporting, data massaging, *etc.*)

All of these modules implement an interface, specific to the module class. For instance, an edge analyzer algorithm will implement the [EdgeAnalyzer](#) interface. There are therefore seven interfaces. They do have in common that they all extend the mother module interface called [TrackMateModule](#).

TrackMateModule is used for two basic purpose:

- It itself extends the SciJavaPlugin interface, which will fuel the automatic discovery of new modules. We will discuss this point last.
- It has basic methods for the GUI integration:
 - `getKey()` returns a unique string identifier that is used internally to reference the algorithm. For instance: "EDGE_VELOCITY_ANALYZER"
 - `getName()` returns a string suitable to be displayed in the GUI that named the algorithm. For instance "Edge velocity".
 - `getIcon()` returns an ImageIcon to be displayed in the GUI.
 - `getInfoText()` returns a html string that briefly documents what the algorithm does. Basic html markup is accepted, so you can have something like


```
<html>Plot the number of spots in each frame as a
function of time. Only the <u>filtered</u> spots are
taken into account.</html>
```

These are the methods used to integrate you module within the GUI. According to the class of the module, some might be plainly ignored. For instance, the edge analyzers subject of this tutorial ignore the icon and info text, since they are used silently within the GUI to provide new features.

10.3 Basic project structure.

Before we step into the edge analyzers specific, you want to setup a development environment that will ease TrackMate module development. Rather than listing the requirement, just checkout [this github repository](#), and clone it. It contains the files of this tutorial series and more importantly, is configured to depend on the latest TrackMate version, which will make it available to your code.

Compiling this project with maven will generate a jar, that you will be able to drop in the fiji plugins folder. Your modules will then be automatically detected and integrated in TrackMate. But more on this later.

10.4 Core class hierarchy.

Let's get back on our edge analyzer. For this tutorial, we are going to do something simple, at least mathematically. We will write an edge analyzer that can return the angle (in radians) of a link in the XY plane. Nothing more. So create a package for your new analyzer in our project, for instance `fiji.plugin`.

`trackmate.examples.edgeanalyzer`. In this package, create a class `EdgeAngleAnalyzer` and let it implement the [EdgeAnalyzer](#) interface. You should be getting something like this:

```
package fiji.plugin.trackmate.examples.edgeanalyzer;

import fiji.plugin.trackmate.features.edges.EdgeAnalyzer;

public class EdgeAngleAnalyzer implements EdgeAnalyzer
{
}
```

It is important to note that we provide a blank constructor. This is very important: with the way we use SciJavaPlugin integration, we cannot use the constructor to pass any object reference. If your analyzer needs some objects which are not provided through the interface methods, then you cannot code it with TrackMate directly. However we should cover most use-cases with what we have.

10.5 Feature analyzers specific methods.

Eclipse will immediately complain (if you are using Eclipse) that your class needs to implement some abstract method. A variety of methods popup. We see the general module methods we discussed above, plus some specific to edge analyzers. Actually, most of the new methods are generic for *all* the feature analyzers (spot, track or edge). These methods belong to the [FeatureAnalyzer](#) interface, which `EdgeAnalyzer` extends, of course.

They exist because TrackMate needs to know what your feature analyzer does. Since it computes numerical features, it needs to know what features it computes, their name, their short name (when we want to show them in crowded part of the GUI) and their physical dimension. Indeed, TrackMate wants to know the dimension of the feature you generate, for it was coded in part by a conflicted physicist who does not want angles and velocities to be plotted on the same graph.

These 6 methods are:

- `getFeatures()` returns a list of string that identifies the features the analyzer generate. There can be more than one. This list must contain strings that can be used in a XML file. Historically, we use capitalized strings, in the shape of java constants, such as `DISPLACEMENT`. We call them feature keys.
- `getFeatureNames()` returns a map that links the feature keys to the feature names. For instance in the GUI, we want to display "Displacement" rather than "DISPLACEMENT", so that is what this map is about. It is important that the keys of this map are the keys defined in the list above.

- `getFeatureShortNames()` returns another map with the same rules. We just use its value to display short names of features when this is needed in the GUI. There are no general advice on how to shorten your feature names; just try until it fits.
- `getFeatureDimensions()` returns a last map, that gives a dimension to your features. Physical dimensions are listed in the [Dimension enum](#).
- `getIsIntFeature()` is just about sugar coating. It returns a map that tells what features are integer mapped. For instance, if you have a feature that count things, such as number of neighbors, you should map this feature to true here. This one is actually not *really* useful; there will be no problem, no loss of precision if you do not set it right. It's just about having numbers displayed correctly. We wanted that when there were 2 neighbors, the number of neighbors displayed was "2" and not "2.00000000000001". In our case, we measure an angle, so this feature should map to false.
- `isManualFeature()` returns a single flag that affects **all** the features calculated by this analyzer. Manual features are special features that were introduced in TrackMate v2.3.0. Let's leave that aside for now. Our angle feature is calculated automatically by the code we are just about to write. So this method should return false.

In this tutorial, our analyzer just returns one feature, which is an angle. So a concrete implementation could be:

```
package plugin.trackmate.examples.edgeanalyzer;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.swing.ImageIcon;

import fiji.plugin.trackmate.Dimension;
import fiji.plugin.trackmate.features.edges.EdgeAnalyzer;

public class EdgeAngleAnalyzer implements EdgeAnalyzer
{

    // The string key that identifies our analyzer.
    private static final String KEY = "Edge angle";
    // The only feature we compute here.
    private static final String EDGE_ANGLE = "EDGE_ANGLE";
    private static final List< String > FEATURES = new ArrayList< String >( 1 );
    private static final Map< String, Boolean > IS_INT = new HashMap< String, Boolean >( 1 );
    private static final Map< String, String > FEATURE_NAMES = new HashMap<String, String>(1);
    private static final Map< String, String > FEATURE_SHORT_NAMES = new HashMap< String,
        ↪ String >( 1 );
    private static final Map< String, Dimension > FEATURE_DIMENSIONS = new HashMap< String,
        ↪ Dimension >( 1 );
```

```

// Let's set the feature list, names, short names and dimensions.
static
{
    FEATURES.add( EDGE_ANGLE );
    IS_INT.put( EDGE_ANGLE, false );
    FEATURE_NAMES.put( EDGE_ANGLE, "Link angle" );
    FEATURE_SHORT_NAMES.put( EDGE_ANGLE, "Angle" );
    FEATURE_DIMENSIONS.put( EDGE_ANGLE, Dimension.ANGLE );
}

private long processingTime;

/*
 * TRACKMATEMODULE METHODS
 */

@Override
public String getKey()
{
    return KEY;
}

// Return a user-compliant name for this analyzer.
@Override
public String getName()
{
    return "Edge angle";
}

// We do not use info texts for any feature actually.
@Override
public String getInfoText()
{
    return "";
}

// The same: we don't use icons for features.
@Override
public ImageIcon getIcon()
{
    return null;
}

@Override
public List< String > getFeatures()
{
    return FEATURES;
}

```

```

@Override
public Map< String, String > getFeatureShortNames()
{
    return FEATURE_SHORT_NAMES;
}

@Override
public Map< String, String > getFeatureNames()
{
    return FEATURE_NAMES;
}

@Override
public Map< String, Dimension > getFeatureDimensions()
{
    return FEATURE_DIMENSIONS;
}

@Override
public Map<String, Boolean> getIsIntFeature()
{
    return Collections.unmodifiableMap(IS_INT);
}

@Override
public boolean isManualFeature()
{
    // This feature is calculated automatically.
    return false;
}

```

10.6 Multithreading & Benchmarking methods.

There are also 4 methods which we will skip right now. They are related to the multi-threading aspect of the analyzer. You can code your analyzer to exploit a multithreaded environment, and TrackMate will configure it through the following methods:

```

@Override
public void setNumThreads()
{
    // We ignore multithreading for this tutorial.
}

@Override
public void setNumThreads( final int numThreads )
{
    // We ignore multithreading for this tutorial.
}

```

```

@Override
public int getNumThreads()
{
    // We ignore multithreading for this tutorial.
    return 1;
}

```

There is also

```
public long getProcessingTime()
```

that returns how much milliseconds was spent on computing the features.

10.7 The core methods.

What is really important is the two methods that actually perform the work:

- `isLocal()`
- `process(Collection< DefaultWeightedEdge > edges, Model model)`

Let's see how they would look for our example angle analyzer.

10.7.1 `isLocal()`.

This method simply returns a boolean that states whether the features you compute are *local* ones or not. By local we mean the following: Does your feature value for an edge depends on the other edges? If no, then it is a local feature: it does not affect the other edges. If yes, then it is non local. Note that it applies to all the features provided by an analyzer.

This distinction fosters some optimization in TrackMate. TrackMate does automated and manual tracking. Allowing for both simultaneously in the same software proved difficult to balance, particularly when the goal is to offer good performance when manually correcting large datasets. When a manual modification of the data is made, TrackMate recomputes all the features, so that they are always in sync. But if a single punctual modification is made on an edge, you want to recompute features only for this edge, not for all the others if they are not affected. TrackMate can do that if the feature is local. This is why this method exists.

An example of a local edge feature would be the instantaneous velocity. The velocity of an edge only depends on this edge and not on the rest. You might say that if you modify the position of a spot, all the edges touching this spot will be affected, so it is not local. But no: all the edges touching the spot will be modified, therefore will be marked for update, but the other edges that are not modified will not have their velocity affected. So the velocity is a local feature.

An example of a non-local edge feature would be the distance of an edge to its closest neighbor. If you move an edge, its own feature value will be affected. But this will also affect the closest distance to many other edges. So it is non-local and we *a priori* have to recompute it for all edges.

In our case, we are coding an analyzer that returns the angle of a single edge, regardless of the angles of the other edges. It is therefore a local feature.

10.7.2 process(Collection< DefaultWeightedEdge > edges, Model model).

The method that actually performs the work is the less elaborated. The concrete implementation is provided with edges, the collection of the edge whose features are to be calculated, and model, the TrackMate model that holds all the information you need. There is just one thing to know: Once you computed the numerical value of your feature, you need to store it in the [FeatureModel](#). The feature model is a part of the main model. It works like a 2D Map:

```
final FeatureModel fm = model.getFeatureModel();
Double val = Double.valueOf(3.1451564);
String FEATURE = "MY_AWESOME_EDGE_FEATURE";
fm.putEdgeFeature( edge, FEATURE, val );
```

And for our XY edge angle, here are the methods content:

```
@Override
public void process( final Collection< DefaultWeightedEdge > edges, final Model model )
{
    final FeatureModel fm = model.getFeatureModel();
    for ( final DefaultWeightedEdge edge : edges )
    {
        Spot source = model.getTrackModel().getEdgeSource( edge );
        Spot target = model.getTrackModel().getEdgeTarget( edge );

        final double x1 = source.getDoublePosition( 0 );
        final double y1 = source.getDoublePosition( 1 );
        final double x2 = target.getDoublePosition( 0 );
        final double y2 = target.getDoublePosition( 1 );

        final double angle = Math.atan2( y2 - y1, x2 - x1 );
        fm.putEdgeFeature( edge, EDGE_ANGLE, Double.valueOf( angle ) );
    }
}

@Override
public boolean isLocal()
{
    return true;
}
```

10.8 Making the analyzer discoverable.

Right now, your analyzer is functional. It compiles and would return expected results. Everything is fine.

Except that TrackMate doesn't even know it exists. It sits in his lonely corner and is perfectly useless.

Until TrackMate v2.2.0, there was no other to extend TrackMate than to modify it or fork it, then recompile and redeploy it from scratch. With v2.2.0 we benefited from the effort of the ImageJ2 team who built a very simple and very clever discovery mechanism, that allow to

simply drop a jar in the plugins folder of Fiji and have TrackMate be aware of it. On top of it all, it is plain and simple.

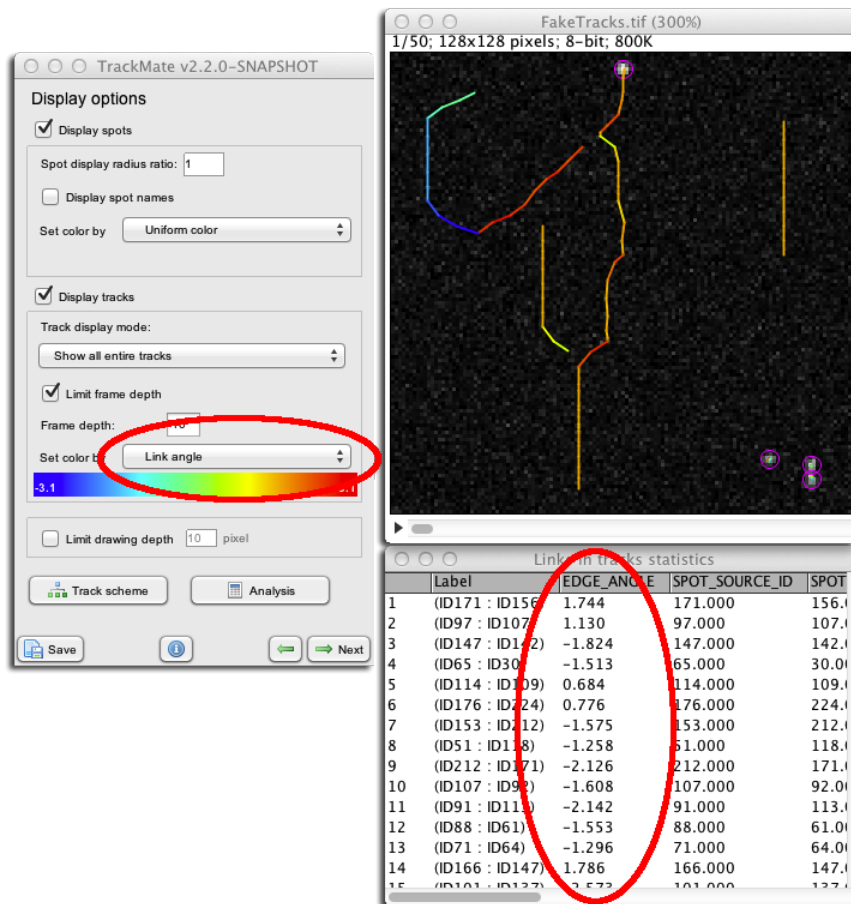
Just add the following line before the class declaration:

```
@Plugin( type = EdgeAnalyzer.class )
public class EdgeAngleAnalyzer implements EdgeAnalyzer
{
...
}
```

and that's it.

To make a TrackMate module discoverable in TrackMate, just annotate its class with `@Plugin(type = TrackMateModuleClassExtending.class)`.

Just the line `@Plugin(type = EdgeAnalyzer.class)` is enough. There are also mechanisms that allow fine tuning of priority, visibility (in the GUI menus), or enabling/disabling, but we will see this later. Right now, just compile your project, and drop the resulting jar in the Fiji plugins folder. Here is what you get:



Great, no? You can find the full source for this example [here](#). It can also be used as a template for your analyzer.

11. How to write your own track feature analyzer algorithm for TrackMate.

11.1 Introduction.

This article is the second in the series dedicated to extending TrackMate with your own modules. Here we focus on creating **feature analyzers**: small algorithms that calculate one or several numerical values for the TrackMate results. The previous section focused on writing edge analyzers: algorithms that allocate a numerical value to the link between two spots. In this section, we will create a **feature analyzer for tracks** that calculate numerical values for whole tracks. To make it simple, and also to answer the request of a colleague, we will make an analyzer that reports the location of the starting and ending points of a track. Actually, we will not learn much beyond what we saw previously. The only little change is that our analyzer will generate six numerical values instead of one. We will use the [SciJava](#) discovery mechanism as before, but just for the sake of it, we will introduce how to **disable** modules.

11.2 Track analyzers.

All the track feature analyzers must implement [TrackAnalyzer interface](#). Like for the [EdgeAnalyzer](#) interface, it extends both

- [FeatureAnalyzer](#) that helps you declaring what you compute,
- and [TrackMateModule](#), that is in charge of the integration in TrackMate.

The only changes for us are two methods specific to tracks:

```
public void process( Collection< Integer > trackIDs, Model model );
```

the does the actual feature calculation for the specified tracks, and

```
public boolean isLocal();
```

that specified whether the calculation of the features for one track affects only this track or all the tracks. For the discussion on local vs non-local feature analyzers, report to the previous section.

11.3 Track feature analyzer header.

Like all TrackMate modules, you need to annotate your class to make it discoverable by TrackMate. It takes the following shape:

```
@Plugin( type = TrackAnalyzer.class )
public class TrackStartSpotAnalyzer implements TrackAnalyzer
{
    // etc...
```

and that's good enough.

11.4 Declaring features.

Declaring the features your provide is done as before. This time, a single analyzer returns 6 values, so you need to declare them. Here is the related code:

```
@Plugin( type = TrackAnalyzer.class )
public class TrackStartSpotAnalyzer implements TrackAnalyzer
{

    private static final String KEY = "TRACK_START_SPOT_ANALYZER";
    private static final String TRACK_START_X = "TRACK_START_X";
    private static final String TRACK_START_Y = "TRACK_START_Y";
    private static final String TRACK_START_Z = "TRACK_START_Z";
    private static final String TRACK_STOP_X = "TRACK_STOP_X";
    private static final String TRACK_STOP_Y = "TRACK_STOP_Y";
    private static final String TRACK_STOP_Z = "TRACK_STOP_Z";
    private static final List< String > FEATURES = new ArrayList< String >( 6 );
    private static final Map< String, String > FEATURE_SHORT_NAMES = new HashMap< String,
        ↪ String >( 6 );
    private static final Map< String, String > FEATURE_NAMES = new HashMap<String, String>(6);
    private static final Map< String, Dimension > FEATURE_DIMENSIONS = new HashMap< String,
        ↪ Dimension >( 6 );

    static
    {
        FEATURES.add( TRACK_START_X );
        FEATURES.add( TRACK_START_Y );
        FEATURES.add( TRACK_START_Z );
        FEATURES.add( TRACK_STOP_X );
        FEATURES.add( TRACK_STOP_Y );
        FEATURES.add( TRACK_STOP_Z );

        FEATURE_NAMES.put( TRACK_START_X, "Track start X" );
        FEATURE_NAMES.put( TRACK_START_Y, "Track start Y" );
        FEATURE_NAMES.put( TRACK_START_Z, "Track start Z" );
        FEATURE_NAMES.put( TRACK_STOP_X, "Track stop X" );
        FEATURE_NAMES.put( TRACK_STOP_Y, "Track stop Y" );
        FEATURE_NAMES.put( TRACK_STOP_Z, "Track stop Z" );

        FEATURE_SHORT_NAMES.put( TRACK_START_X, "X start" );
        FEATURE_SHORT_NAMES.put( TRACK_START_Y, "Y start" );
        FEATURE_SHORT_NAMES.put( TRACK_START_Z, "Z start" );
        FEATURE_SHORT_NAMES.put( TRACK_STOP_X, "X stop" );
        FEATURE_SHORT_NAMES.put( TRACK_STOP_Y, "Y stop" );
        FEATURE_SHORT_NAMES.put( TRACK_STOP_Z, "Z stop" );

        FEATURE_DIMENSIONS.put( TRACK_START_X, Dimension.POSITION );
        FEATURE_DIMENSIONS.put( TRACK_START_Y, Dimension.POSITION );
        FEATURE_DIMENSIONS.put( TRACK_START_Z, Dimension.POSITION );
        FEATURE_DIMENSIONS.put( TRACK_STOP_X, Dimension.POSITION );
    }
}
```

```

    FEATURE_DIMENSIONS.put( TRACK_STOP_Y, Dimension.POSITION );
    FEATURE_DIMENSIONS.put( TRACK_STOP_Z, Dimension.POSITION );
}

/*
 * FEATUREANALYZER METHODS
 */

@Override
public List< String > getFeatures()
{
    return FEATURES;
}

@Override
public Map< String, String > getFeatureShortNames()
{
    return FEATURE_SHORT_NAMES;
}

@Override
public Map< String, String > getFeatureNames()
{
    return FEATURE_NAMES;
}

@Override
public Map< String, Dimension > getFeatureDimensions()
{
    return FEATURE_DIMENSIONS;
}

```

Let's compute them now.

11.5 Accessing tracks in TrackMate.

In the previous article, we went maybe a bit quickly on how to access data in TrackMate. This is not the goal of this series, but here is a quick recap:

All the track structure is stored in a sub-component of the model called the [TrackModel](#). It stores the collection of links between two spots that builds a graph, and has some rather complex logic to maintain a list of connected components: the tracks.

The tracks themselves are indexed by their ID, stored as an `int`, that has no particular meaning. Once you have the ID of track, you can get the spots it contains with

```
trackModel.trackSpots( trackID )
```

and its links (or edges) with

```
trackModel.trackEdges( trackID )
```

Let's exploit this.

11.6 Calculating the position of start and end points.

Well, it is just about retrieving a track and identifying its starting and end points. Here is the whole code for the processing method:

```
@Override
public void process( final Collection< Integer > trackIDs, final Model model )
{
    // The feature model where we store the feature values:
    final FeatureModel fm = model.getFeatureModel();

    // Loop over all the tracks we have to process.
    for ( final Integer trackID : trackIDs )
    {
        // The tracks are indexed by their ID. Here is how to get their
        // content:
        final Set< Spot > spots = model.getTrackModel().trackSpots( trackID );
        // Or .trackEdges( trackID ) if you want the edges.

        // This set is NOT ordered. If we want the first one and last
        // one we have to sort them:
        final Comparator< Spot > comparator = Spot.frameComparator;
        final List< Spot > sorted = new ArrayList< Spot >( spots );
        Collections.sort( sorted, comparator );

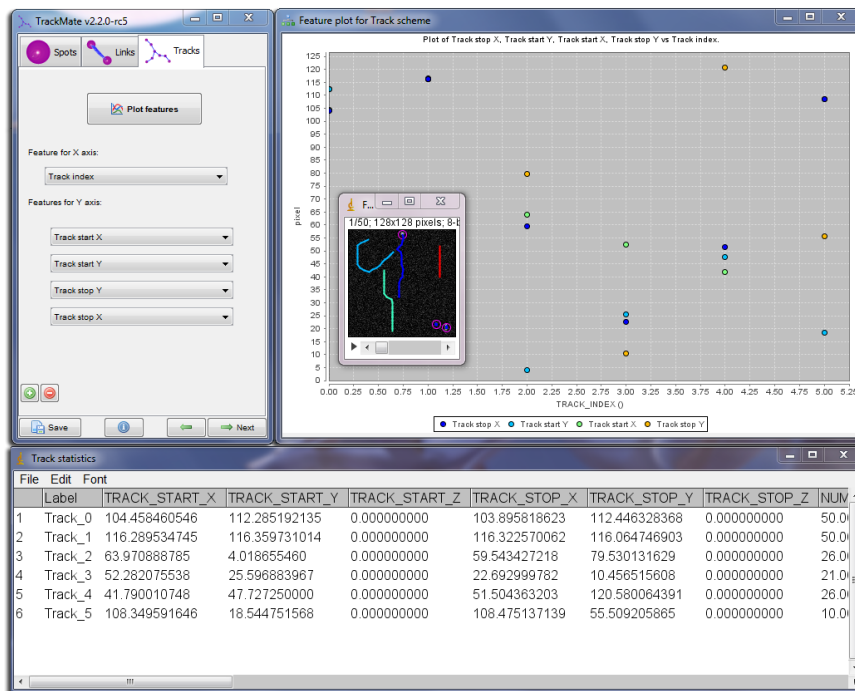
        // Extract and store feature values.
        final Spot first = sorted.get( 0 );
        fm.putTrackFeature(trackID, TRACK_START_X, Double.valueOf(first.getDoublePosition(0)));
        fm.putTrackFeature(trackID, TRACK_START_Y, Double.valueOf(first.getDoublePosition(1)));
        fm.putTrackFeature(trackID, TRACK_START_Z, Double.valueOf(first.getDoublePosition(2)));

        final Spot last = sorted.get( sorted.size() - 1 );
        fm.putTrackFeature(trackID, TRACK_STOP_X, Double.valueOf(last.getDoublePosition(0)));
        fm.putTrackFeature(trackID, TRACK_STOP_Y, Double.valueOf(last.getDoublePosition(1)));
        fm.putTrackFeature(trackID, TRACK_STOP_Z, Double.valueOf(last.getDoublePosition(2)));
    }
}
```

The whole code for the analyzer can be found [here](#).

11.7 Wrapping up.

The new track features get properly integrated along with other native features:



In the next article we will build a spot analyzer and complicate things a bit, by introducing the notion of *priority*. But before this, a short word on how to disable a module.

11.8 How to disable a module.

Suppose you have in your code tree a TrackMate module you wish not to use anymore. The trivial way would be to delete its class, but here is another one what allows us to introduce [SciJava](#) plugin annotation parameters.

The `@Plugin(type = TrackAnalyzer.class)` annotation accepts extra parameters on top of the type one. They all take the shape of a key = value pair, and a few of them allow the fine tuning of the TrackMate module integration.

The first one we will see is the enabled value. It accepts a boolean as value and by default it is true. Its usage is obvious:

If you want to disable a TrackMate module, add the `enabled = false` annotation parameter.

Like this:

```
@Plugin( type = TrackAnalyzer.class, enabled = false )
```

Disabled modules are not even instantiated. They are as good as dead, except that you can change your mind easily. By the way, you can see that the TrackMate source tree has many of these disabled modules...

12. How to write your own track feature analyzer algorithm for TrackMate.

12.1 Introduction.

This third article in the series dedicated to extending [TrackMate](#) deals with spot feature analyzer. This is the last of the three kind of feature analyzers you can create, and it focuses on spots, or detections.

Spot features are typically calculated from the spot location and the image data. For instance, there is a spot feature that reports the mean intensity within the spot radius. You need to have the spot location, its radius and the image data to compute it.

In this tutorial, we will generate an analyzer that is not directly calculated from the image data. This will enable us to skip over introducing [ImgLib2](#) API, which would have considerably augmented the length of this series. But this choice does not only aim at nurturing my laziness: We will make our feature **depend on other features** which will allow us to introduce **analyzers priority**.

But before this, let's visit the spot feature analyzers specificities.

12.2 Spot analyzers and spot analyzer factories.

In the two previous articles we dealt with [edge](#) and [track](#) analyzers. We could make them in a single class, and this class embedded both the code for

- TrackMate integration (feature names, dimensions, declaration, etc...);
- and actual feature calculation.

For spot analyzer, the two are separated.

You must first create a [SpotAnalyzerFactory](#). This factory will be in charge of the TrackMate integration. The interface extends both the [TrackMateModule](#) and the [FeatureAnalyzer](#) interfaces. It is the class you will need to annotate with a [SciJava](#) annotation for automatic discovery.

But it is also in charge of instantiating [SpotAnalyzers](#). As you can see, this interface just extends [ImgLib2 Algorithm](#), so all parameters will have to be passed in the constructor, which can be what you want thanks to the factory. We do not need a return value method, because results are stored directly inside the spot objects. But we will see this later.

Let's get started with our example.

12.3 The spot analyzer factory.

We want to generate an analyzer that will compute for each spot, its intensity relative to the mean intensity of all spots in the same frame. So you get for this feature a value of 1 if its intensity is equal to the mean, etc... We could have our analyzer actually compute the pixel intensity for each spot, take the mean over a frame, then normalize, etc... But, there is an analyzer that already computes the spot intensity and we can re-use it. Check the [SpotIntensityAnalyzerFactory](#).

It is a good idea to reuse this value in our computations, both for the quickness of development and runtime performance. But if we do so, we must ensure that the feature we depend on is available when our new analyzer runs. There is a way to do that, thanks to the notion of **priority**, which we will deal with later.

Right now, let's focus on the factory class itself. There is not much to say: its content resembles all the feature analyzers we saw so far. So I am going to skip over the details and point you to the full source code [here](#).

The one interesting part is the factory method in charge of instantiating the `SpotAnalyzer`:

```
@Override
public SpotAnalyzer< T > getAnalyzer( Model model, ImgPlus< T > img, int frame, int
    ↪ channel )
{
    return new RelativeIntensitySpotAnalyzer< T >( model, frame );
}
```

Since we want to build a feature that does not need the image data, the constructor just skips the image reference. And that's it. We must now move on to the analyzer itself to implement the feature calculation logic.

12.4 The spot analyzer.

As you noted in the above method, each analyzer is meant to operate only on one frame. It can access the whole model, but it is supposed to compute the values for all the spots of a single frame. This permits multithreading: The factory will be asked to generate as many analyzer as there is threads available, and they will run concurrently. And we, as we build our analyzer - do not have to worry about concurrent issues.

A little word about the expected execution context: The TrackMate GUI operates in steps, as you have noted. First the detection step generates spots, then they are filtered, then they are tracked, etc... Therefore, when I said earlier that the whole model is available for calculation, this is not entirely true. When using the GUI, spot numerical features are used to filter spots after they have been detected. So that this stage, there is no tracks yet. There is not even filtered spots. A spot feature cannot depend on these objects, and this is a built-in limitation of TrackMate. So be cautious on what your numerical feature depends.

Before we go into the code, here is quick recap on the TrackMate model API. After the detection step, the spots are stored in a [SpotCollection](#) object. It gathers all the spots, and can deal with their filtering visibility, etc... Spot analyzers are meant to operate only on one frame, so we will need to require the spot of this frame. The target frame is specified at construction time, by the factory.

The [SpotAnalyzer](#) interface is pretty naked. There is nothing specific, and all the logic has to go in the `process()` method. There is no need to have a method to return the results of the computation, for spot objects can store their own feature values, thanks to the `Spot.putFeature(feature, value)` method.

Here is what the `process()` method of the analyzer looks like:

```
@Override
public boolean process()
```

```

{
    /*
     * Collect all the spots from the target frame. In a
     * SpotAnalyzer, you cannot interrogate only visible
     * spots, because spot features are typically used to
     * determine whether spots are going to be visible or
     * not. This happens in the GUI at the spot filtering
     * stage: We are actually building a feature on which
     * a filter can be applied. So the spot features must
     * be calculated over ALL the spots.
     */

    /*
     * The spots are stored in a SpotCollection before they
     * are tracked. The SpotCollection is the product of
     * the detection step.
     */
    final SpotCollection sc = model.getSpots();
    // 'false' means 'not only the visible spots, but all spots'.
    Iterator< Spot > spotIt = sc.iterator( frame, false );

    /*
     * Compute the mean intensity for all these spots.
     */

    double sum = 0;
    int n = 0;
    while ( spotIt.hasNext() )
    {
        final Spot spot = spotIt.next();
        // Collect the mean intensity in the spot radius.
        final double val = spot.getFeature( SpotIntensityAnalyzerFactory.MEAN_INTENSITY );
        sum += val;
        n++;
    }

    if ( n == 0 )
    {
        // Nothing to do here.
        return true;
    }

    final double mean = sum / n;

    /*
     * Make a second pass to set the relative intensity of these
     * spots with respect to the mean we just calculated.
     */
}

```

```

spotIt = sc.iterator( frame, false );
while ( spotIt.hasNext() )
{
    final Spot spot = spotIt.next();
    final double val = spot.getFeature( SpotIntensityAnalyzerFactory.MEAN_INTENSITY );
    final double relMean = val / mean;
    // Store the new feature in the spot
    spot.putFeature( RELATIVE_INTENSITY, Double.valueOf( relMean ) );
}

return true;
}

```

The code for the whole class is [here](#).

12.5 Using SciJava priority to determine order of execution.

Now it's time to discuss the delicate subject of dependency.

As stated above, our new analyzer depends on some other features to compute. Therefore, the analyzers that calculate these other features need to run *before* our analyzer. Or else you will get `NullPointerException`s randomly.

TrackMate does not offer a real in-depth module dependency management. It simply offers to **determine** the order of analyzer execution thanks to the [SciJava](#) plugin **priority parameter**.

For instance, if you check the annotation part of the spot analyzer factory, you can see that there is an extra parameter, `priority`:

```
@Plugin( type = SpotAnalyzerFactory.class, priority = 1d )
```

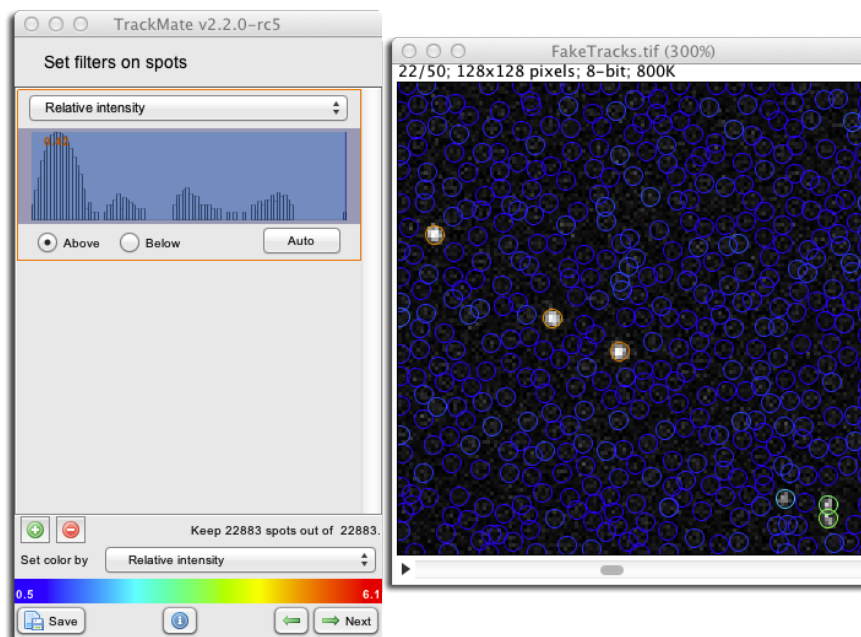
This `priority` parameter accepts a `double` as value and this value determines the order of execution. Careful, the rule is the opposite of what would make sense for a `priority`:

Feature analyzers are executed in order according to **increasing priority**. This means that analyzers with the greatest priority are executed last.

By convention, if your feature analyzer depends on the features calculated by `N` other analyzers, you take the larger priority of these analyzers, and add 1. In our case, we depend on the [SpotIntensityAnalyzerFactory](#), which has a priority of 0 (the default if the parameter is unspecified). So quite logically, we set the priority of our analyzer to be 1. This ensures the proper execution order.

12.6 Wrapping up.

Apart from the discussion on the priority and execution order, there is not much to say. It works!



13. How to write your own viewer for TrackMate.

13.1 Introduction.

Developing a custom view for [TrackMate](#) is *hard* and painful. Of course it must be a graphical representation of the model: the tracking results with all intermediate steps. If you want to build something really useful, it has to be interactive and should allow modifying the model. And be aware that modifications might happen somewhere else. Performance is also critical: since it stands at the user interface, it must be responsive, and possibly deal with large models (millions of detections). One of the main good reason to extend TrackMate is most likely that there is ready some views available.

Still, it is perfectly possible to build something useful without fulfilling all these requirements. And we still hope that someday someone will contribute a view that displays the model in the orthogonal slicer of Fiji.

This tutorial introduces the view interfaces of TrackMate, and since they deal with user interactions, we will also review the TrackMate event system. As for the [SciJava](#) discovery system, we will see how to make a TrackMate module available in TrackMate, but not visible to the user, using the `visible` parameter.

13.2 A custom TrackMate view.

Like for the [spot feature analyzers](#), a TrackMate view is separated in two parts, that each extends a different interface:

- The [TrackMateModelView](#), that is the actual view of the model. All the hard work is done here.
- The [ViewFactory](#) that is a factory in charge of instantiating the view and of the integration in TrackMate. This interface extends the [TrackMateModule](#) interface, so we expect to find there some of the methods we discussed earlier, and the [SciJava](#) annotation.

In this tutorial, we will build something simple. We will limit ourselves to develop a view that simple messages the user every time something happens in TrackMate. For instance, when the spots are detected, how many there are; if he selects spots and edges, how many of them; *etc.* And we will just reuse the Fiji log window for this, which will save us from the full development of a graphical view of the model.

But because this is a bit limited, we will not let the user pick this view as the main one, just after the detection step. A [SciJava](#) parameter will be used to make it invisible in the view selection menu. To make good use of it, we still need some way to launch this view, but this will be the subject of the next tutorial. Right now, we just focus on building the view.

13.3 The ViewFactory.

The factory itself has nothing particular. On top of the TrackMateModule methods, it just has a method to instantiate the view it controls:

```
@Override
public TrackMateModelView create( Model model, Settings settings, SelectionModel
    ↪ selectionModel )
```

You can see that we can possibly pass 3 parameters to the constructor of the view itself: the model of course, but also the settings object, so that we can find there a link to the image object. The [HyperStackDisplayer](#) uses it to retrieve the ImagePlus over which to display the TrackMate data. The selection model is also offered, and the instance passed is the common one used in the GUI, so that a selection made by the user can be shared amongst all views.

13.4 The TrackMateModelView interface.

13.4.1 Methods.

This is where the hard work takes place and there is a lot to say. However, the method you find in this interface are scarce and relate just to general use, and most of them are not mandatory:

- `public void render();`

This is the initialization method for your view. Your view should not show up to the user when it is instantiating, but only when this method is called. This allows TrackMate to properly manage the rendering.

- `public void refresh();`

This method should be in charge of updating the view whenever it is sensible to do so. Careful: it is **not** called automatically when the model has changed. You have to listen to model change yourself, and call this method manually if you want your view to be in sync. However, it **is** called automatically whenever the user changes a display setting (because views are not made to listen to GUI changes). But more on that below.

- `public void clear();`

This one is rather explicit. It ensures a way to clear a view in case it is not kept in sync with the model changes.

- `public void centerViewOn(Spot spot);`

This is a non-mandatory convenience tool that allow centering a view (whatever it means) on a specific Spot. It is called for instance when the user selects **one** spot in the GUI: all the views that implement this method move and pan to show this spot.

- The three methods related to display settings:

```
public Map< String, Object > getDisplaySettings();
public void setDisplaySettings( String key, Object value );
public Object getDisplaySettings( String key );
```

are explained below.

- `public Model getModel();`

exposes the model this view renders.

- `public String getKey();`

Returns the unique key that identifies this view. Careful: this key **must** be the same that for the ViewFactory that can instantiates this view. This is used to save and restore the views present in a TrackMate session.

13.4.2 Display settings.

It should be possible to configure the look and feel of your view, or even to set what is visible or not. This is made through display settings, and 3 methods are used to pass them around:

```
public Map< String, Object > getDisplaySettings();
public void setDisplaySettings( final String key, final Object value );
public Object getDisplaySettings( final String key );
```

Display settings are passed using a pair of key (as String) / value (as Object, that should be cast upon the right class).

The TrackMate GUI allows the user to edit a limited series of display settings that ought to be common to all views. These are the settings you can tune on the antepenultimate panel of the GUI (spot visible or not, color by feature, etc...). If you feel like it, your view can just ignore them. Otherwise, their keys and desired classes are defined in the [TrackMateModelView](#) interface. Check the static fields there.

Everytime the user changes a setting in the GUI, the new setting value is passed with the `setDisplaySettings()` method, then the `refresh()` method is called as well.

13.4.3 Listening to model changes.

You don't *have to* keep your view in sync with the model. You can make something useful that would just capture a snapshot of the model as it is when you launch the view and be happy about it. But, TrackMate is about allowing both automatic and manual annotation of the image data, so most likely a very useful view will echoes the changes made to the model. Ideally it would even *enable* these changes to be made. But this is out of the scope of this tutorial.

If you want to listen to changes made to the model, you have to register as a listener to it. This is made through

```
Model.addModelChangeListener( YourViewInstance );
```

and then you get a new method:

```
public void modelChanged( final ModelChangeEvent event )
```

The event itself can report 5 types of changes:

- The spots detection is done. In the GUI, this is sent just after the detection step, before the initial filtering step.
- The spots are filtered reversibly. This is sent everytime you change anything on the spot filtering panel (a new filter, a threshold value, etc..).
- The tracking step is done. That just follows the tracking step in the GUI.
- The tracks are filtered. Like for the spots.
- The model is *modified*. By modification, we mean an incremental, manual modification of the model. The user might have deleted a spot, or moved it in space, or changed its size, or add an edge between two spots, *etc*. In that case, the [ModelChangeEvent](#) instance can be interrogated to know what was changed, deleted, added, *etc*.

13.4.4 Listening to selection changes.

The TrackMate GUI shares a common instance of [SelectionModel](#) that stores the selection the user made. This is convenient when exploring the tracking results. Your view can be kept in sync with the selection changes by implementing the [SelectionChangeListener](#) interface. It adds a single method:

```
public void selectionChanged(SelectionChangeEvent event);
```

13.5 A simple event logger.

Let's keep our custom view simple: we will just build an event logger that recycles the IJ logger window to echo what happens to the model. We then of course have to implement the two listener interfaces mentioned above. But the code stays pretty simple: check [here](#) for the details.

As for the factory, nothing fancy:

```

package plugin.trackmate.examples.view;

import ij.ImageJ;
import ij.ImagePlus;

import javax.swing.ImageIcon;

import org.scijava.plugin.Plugin;

import fiji.plugin.trackmate.Model;
import fiji.plugin.trackmate.SelectionModel;
import fiji.plugin.trackmate.Settings;
import fiji.plugin.trackmate.TrackMatePlugIn_;
import fiji.plugin.trackmate.visualization.TrackMateModelView;
import fiji.plugin.trackmate.visualization.ViewFactory;

@Plugin( type = ViewFactory.class )
public class EventLoggerViewFactory implements ViewFactory
{
    private static final String INFO_TEXT = "<html>This factory instantiates an event logger
        ↪ view for TrackMate, that uses the IJ log window to just echo all the event sent by
        ↪ the model.</html>";

    public static final String KEY = "EVENT_LOGGER_VIEW";

    @Override
    public String getInfoText()
    {
        return INFO_TEXT;
    }

    @Override
    public ImageIcon getIcon()
    {
        return null;
    }

    @Override
    public String getKey()
    {
        return KEY;
    }

    @Override
    public String getName()
    {
        return "Event logger view";
    }
}

```



```

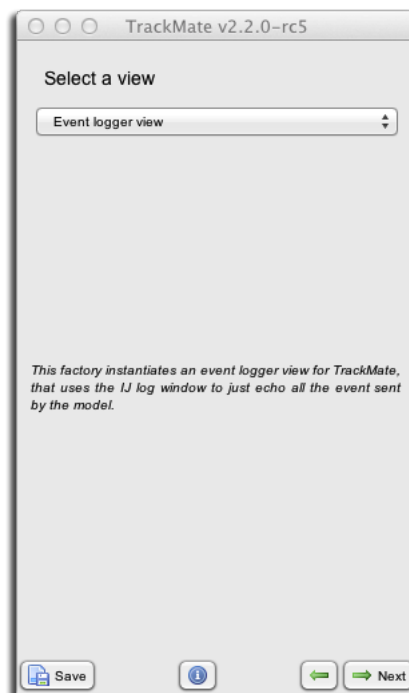
@Override
public TrackMateModelView create( final Model model, final Settings settings, final
    ↪ SelectionModel selectionModel )
{
    return new EventLoggerView( model, selectionModel );
}

/*
 * MAIN METHOD
 */

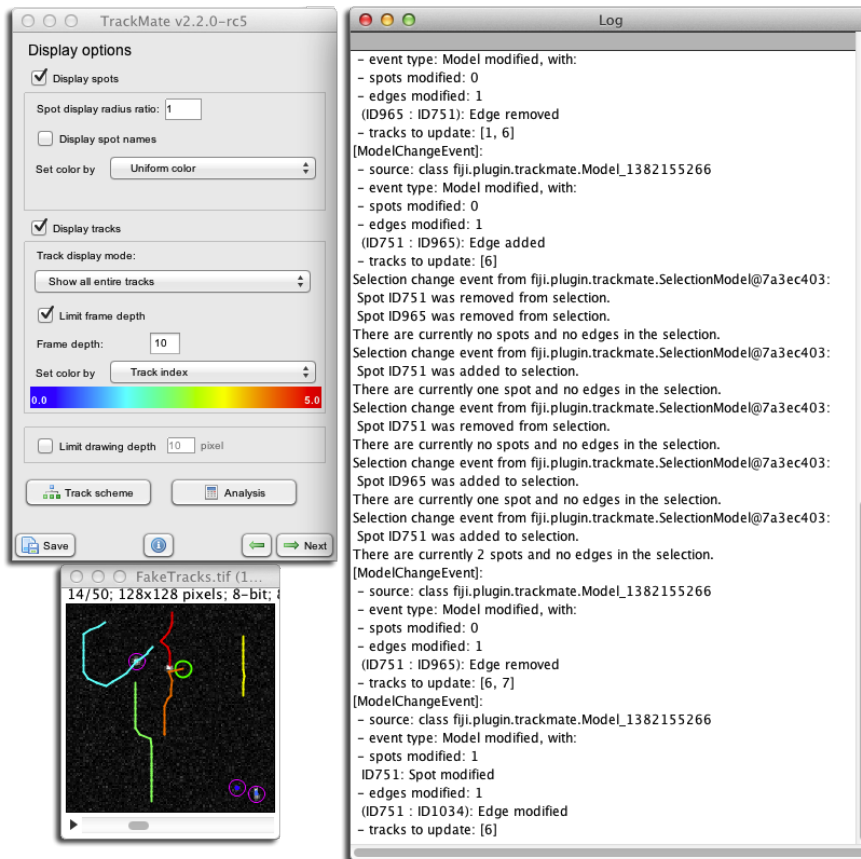
public static void main( final String[] args )
{
    ImageJ.main( args );
    new ImagePlus( "../fiji/samples/FakeTracks.tif" ).show();
    new TrackMatePlugIn_().run( "" );
}
}

```

Just note that the [SciJava](#) annotation mention the ViewFactory class. This is enough to have the view selectable in the GUI menu:



Note that this time, TrackMate good use of the `getName()` and `getInfoText()` methods. And here is what you get after a few manipulations:



13.6 Controlling the visibility of your view with the SciJava visible parameter.

Our view is a good dummy example. It is not that useful, and the info panel of the GUI could be used instead advantageously. We have nothing against it, but maybe we should not let users select it as the main view in the GUI, otherwise they might get frustrated (well, the HyperStack view is *always* used, whatever you choose, so we could not mind, but eh).

There is way to do that, just by tuning the SciJava annotation:

To make a TrackMate module available in TrackMate, but not visible in the GUI menus, use the annotation parameter `visible = false`.

So editing the header of our ViewFactory to make it look like:

```
@Plugin( type = ViewFactory.class, visible = false )
public class EventLoggerViewFactory implements ViewFactory
```

is enough to hide it in the menu. This is different from the `enabled` parameter we saw in the previous section. The factory is instantiated and available in TrackMate; it just does not show up in the menu. But how could we make use of it then? you want to ask. Fortunately, this is just the subject of the next section, on TrackMate actions.

14. How to write custom actions for TrackMate.

14.1 Introduction.

Actions are a simple solution to the problem of adding random features to [TrackMate](#) without having to change the GUI too much. Adding buttons or dialogs or extra panels is cumbersome and it would complexify the GUI, which was meant to remain simple. A TrackMate action is a lazy workaround for this problem. You must keep in mind that it is a placeholder for random feature ideas, and provided a quick and dirty way to test them.

A TrackMate action takes the shape of an item in a drop-down list in the last panel of the GUI. It can do more or less anything, since we pass everything to the action, even a reference to the GUI itself. Thanks to the [SciJava](#) discovery mechanism, we do not have to worry about adding it on the GUI: it will automatically be listed in the action list. The drawback of this simplicity is that you cannot use it to provide elaborated user interaction mechanisms, such as the ones you can find in a view.

In this tutorial, we will use it to launch the event logger we created in the previous section of this series. If you remember, we saw in the last paragraph how to use the `visible = false` parameter the [SciJava](#) annotation to hide it from the view menu. Hereby preventing the user to access it. No problem, we will now build an action that will launch it as a supplementary view.

14.2 The TrackMateActionFactory interface.

Again, the action behavior and its integration in TrackMate are split in two classes. The behavior is described by the [TrackMateAction](#) interface. The integration mechanism is controlled by the [TrackMateActionFactory](#) interface, which extends the [TrackMateModule](#) interface.

14.2.1 SciJava parameters recapitulation.

There is not much to say about the factory itself. It is the class that must be annotated with `@Plugin(type = TrackMateActionFactory.class)`

All the SciJava annotation parameters apply, and they have the following meaning:

- The `enabled = true/false` parameter is used to control whether the action is enabled or not. A disabled action is not even instantiated.
- The `visible = true/false` parameter determines whether the action is listed in the action panel. If false, the action factory is instantiated but the corresponding action will not be listed in the panel, preventing any use.
- The `priority = double` parameter is used here just to determine the order in which the action items appear in the list. High priorities are listed last.

14.2.2 Action factory methods.

As of [TrackMate](#) version 2.2.0 (March 2014), actions are the only TrackMate modules that use the `getIcon()` method. The icon is then displayed in the action list, next to the action name.

That's it for the TrackMateModule part.

The method specific to actions is more interesting:

```
@Override
public TrackMateAction create( final TrackMateGUIController controller )
```

This means that when we create our specific action, we have access to the some of GUI context through the controller. We therefore have to check its [API](#) to know what we can get. It gives us access to:

- The GUI window itself (public TrackMateWizard getGUI()), that we can use as parent for dialogs, wild live GUI editing...
- The trackmate plugin (public TrackMate getPlugin()), hereby the model and settings objects.
- The selection model (public SelectionModel getSelectionModel())
- Even the GUI model (public TrackMateGUIModel getGuimodel())
- And all the providers that manage the modules of TrackMate.

So you can pretty well mess stuff with the controller, but it gives us access to mainly everything. In our case, we do not need much. Here is the code for our simple event logger launcher:

```
package plugin.trackmate.examples.action;

import javax.swing.ImageIcon;

import org.scijava.plugin.Plugin;

import fiji.plugin.trackmate.action.TrackMateAction;
import fiji.plugin.trackmate.action.TrackMateActionFactory;
import fiji.plugin.trackmate.gui.TrackMateGUIController;

@Plugin( type = TrackMateActionFactory.class )
public class LaunchEventLoggerActionFactory implements TrackMateActionFactory
{

    private static final String INFO_TEXT = "<html>This action will launch a new event logger,
        ↔ that uses the ImageJ log window to append TrackMate events.</html>";
    private static final String KEY = "LAUNCH_EVENT_LOGGER";
    private static final String NAME = "Launch the event logger";

    @Override
    public String getInfoText()
    {
        return INFO_TEXT;
    }

    @Override
    public ImageIcon getIcon()
    {
```

```

        return null; // No icon for this one.
    }

    @Override
    public String getKey()
    {
        return KEY;
    }

    @Override
    public String getName()
    {
        return NAME;
    }

    @Override
    public TrackMateAction create( final TrackMateGUIController controller )
    {
        return new LaunchEventLoggerAction( controller.getPlugin().getModel(), controller.
            ↪ getSelectionModel() );
    }
}

```

Nothing complicated.


14.3 The TrackMateAction interface.

This interface is just made of two methods:

```

public void execute(TrackMate trackmate);
public void setLogger(Logger logger);

```

The execute method is the one triggered by the user when he clicks the  **Execute** button. It receives a TrackMate instance that can be of use. In our case, as you saw in the factory class, we got the model and selection model through the controller. The other method is used to pass a logger instance that is specific to the action panel in the GUI. All messages and updates sent to this logger will be shown on the action panel. Here is how this translates simply in a simple launcher:

```

package plugin.trackmate.examples.action;

import plugin.trackmate.examples.view.EventLoggerView;
import fiji.plugin.trackmate.Logger;
import fiji.plugin.trackmate.Model;
import fiji.plugin.trackmate.SelectionModel;
import fiji.plugin.trackmate.TrackMate;
import fiji.plugin.trackmate.action.TrackMateAction;

public class LaunchEventLoggerAction implements TrackMateAction

```

```

{
    private final SelectionModel selectionModel;

    private final Model model;

    private Logger logger;

    public LaunchEventLoggerAction( final Model model, final SelectionModel selectionModel )
    {
        this.model = model;
        this.selectionModel = selectionModel;
    }

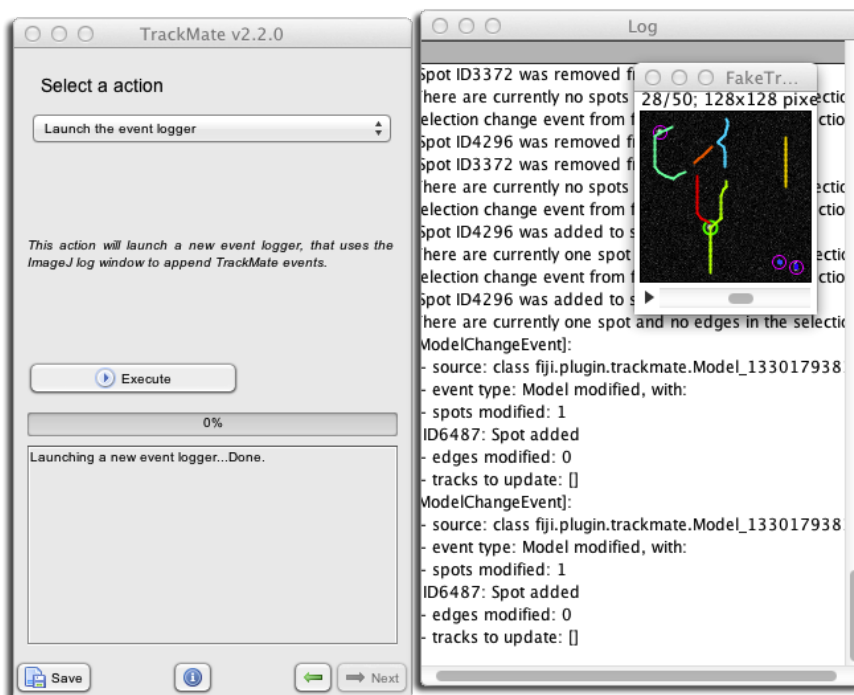
    @Override
    public void execute( final TrackMate trackmate )
    {
        logger.log( "Launching a new event logger..." );
        final EventLoggerView view = new EventLoggerView( model, selectionModel );
        view.render();
        logger.log( " Done.\n" );
    }

    @Override
    public void setLogger( final Logger logger )
    {
        this.logger = logger;
    }
}

```

14.4 Wrapping up.

And here are the results:



You can imagine a lot of applications for Actions. Since they give you access to most of the plugin context, you can basically plug anything there. The one limitation is that it does not fit perfectly in the existing GUI: actions just appear as items in a drop-down list. But in most cases it does not matter much. Actions are very useful to quickly graft a piece of new functionality on TrackMate.

This concludes this tutorial, which was pretty quick and simple. This is unfortunately the last time in this series that things are simple and short. The next tutorial will be about implementing a custom detector, which will turn to be quite complicated for apparently wrong reasons.

15. How to write your own detection algorithm for TrackMate.

15.1 Introduction.

Welcome to the most useful and also unfortunately the hardest part in this tutorial series on how to extend [TrackMate](#) with custom modules.

The detection algorithms in TrackMate are basic: they are all based or approximated from the [Laplacian of Gaussian](#) technique. They work well even in the presence of noise for round or spherical and well separated objects. As soon as you move away from these requirements, you will feel the need to implement your own custom detector.

This is the subject of this tutorial, which will appear denser than the previous ones. Not because implementing a custom detection algorithm is difficult. It *is* difficult, even very difficult if you are not familiar with the [ImgLib2](#) library. But we will skip this difficulty here by

not making a true detector, but just a dummy one that returns detections irrespective of the image content. This involved task is left to your Java and `ImgLib2` skills.

No, this tutorial will be difficult because contrary to the previous ones, we need to do a lot of work even for just a dummy detector. The reason for this comes from our desire to have a nice and tidy integration in `TrackMate`. The custom detector we will write will be a first-class citizen of `TrackMate`, and this means several things: Not only it must be able to provide a proper detection, but it must also

- offer the user some configuration options, in a nice GUI;
- check that the user entered meaningful detection parameters;
- enable the saving and loading of these parameters to XML.

We did not have to care when implementing a [custom action](#), but now we do. Let's get started with the easiest part, the detection algorithm.

15.2 The `SpotDetector` interface.

15.2.1 A detector instance operates on a single frame.

The detection part itself is implemented in a class that implements the [SpotDetector](#) interface. Browsing there, you will see that it is just a specialization of an output algorithm from [ImgLib2](#). We are required to spit out a `List<Spot>` that represents the list of detection (one `Spot` per detection) for a **single frame**. This is important: an instance of your detector is supposed to work on a single frame. `TrackMate` will generate as many instances of the detector per frame it has to operate on. This facilitates development, but also multithreading: `TrackMate` fires one detector per thread it has access to, and this is done without you having to worry about it. `TrackMate` will bundle the outputs of all detectors in a thread-safe manner.

It is the work of the detector factory to provide each instance with the data required to segment a specific frame. But we will see how this is done below.

15.2.2 A `SpotDetector` can be multithreaded.

So `TrackMate` offers you a turnkey multithreaded solution: If you have a computer with 12 cores and 50 frames to segment, `TrackMate` will fire 12 `SpotDetectors` at once and process them concurrently.

But let's say that you have 24 cores and only 6 frames to segment. You can exploit this situation by letting your concrete instance of `SpotDetector` implement the `ImgLib2` [MultiThreaded](#) interface. In that case, `TrackMate` will still fire 6 `SpotDetector` instances (one for each frame), but will allocate 4 threads to each instance, and get an extra kick in speed.

Of course, you have to devise a clever multithreading strategy to operate concurrently on a single frame. For instance, you could divide the image into several blocks and process them in parallel. Or delegate to sub-algorithms that are multithreaded; check for instance the [LogDetector](#) code.

15.2.3 Detection results are represented by Spots.

[Spots](#) are used to represent detection results: one detection = one spot. By convention, a detection algorithm must provide *at least* the following numerical feature to each spot:

- The X, Y, Z coordinates, obviously. What is not that obvious is that TrackMate uses only image coordinates. This means that if your image has a physical calibration in μm (e.g. $0.2 \mu\text{m}/\text{pixels}$ in X,Y), the spot coordinates must be in μm^2 . If you have just a 2D image, use 0 for the Z position, but it must not be omitted.
- A quality value, that reflects the quality of the detection itself. It must be a real, positive number, that reflects how confident your detection algorithm is that the found detection is not spurious. The larger the more confident.
- The spot radius, representing in physical units, the approximate size of the image structure that was detected. TrackMate default detectors do not have an automatic size detection feature, so they ask the user what is the most likely size of the structures they should detect, tune themselves to this size, and set all the radius of the detections to be the one entered by the user.

Any omission will trigger errors at runtime.

15.2.4 A dummy detector that returns spiraling spots.

For this tutorial we will build a dummy detector, that actually fully ignores the image content and just create spots that seem to spiral out from the center of the image. A real detector would require you to hone your [ImgLib2](#) skills; check the [LogDetector](#) code for an example.

Below is the source code for the dummy detector. You can also find it [online](#). Let's comment a bit on this:

The type parameter `< T extends RealType< T > & NativeType< T >`. Instances of `SpotDetector` are parametrized with a generic type `T` that must extends [RealType](#) and [NativeType](#). These are the bounds for all the scalar types based on native types, such as `float`, `int`, `byte`, etc. This is the type of the image data we are to operate on.

The constructor. Since the [SpotDetector](#) interface gives little constraint on inputs, all of them must be provided at construction time in the constructor. Keep in mind that we have one instance per frame, so we must know what frame we are to process.

Normal detectors would be fed with a reference to the image data *for this very single frame*. Here we do not care for image content, so it is not there. But we will speak of this more when discussing the factory.

²The reason behind this is that TrackMate wants to break free of the source data. Keeping all the coordinates in physical units allow exchanging and working on results without having to keep a reference to the original image.

Because TrackMate can also be tuned to operate only on a ROI, the instance receives an [Interval](#) that represent the bounding box **in pixel coordinates** of the ROI the user selected. Here, we just use it to center the spirals.

Because we must store the *physical coordinates* in the spots we create, we need a calibration array to convert pixel coordinates to physical ones. That is the role of the `double[]` `calibration` array, and it contains the pixel sizes along X, Y and Z.

The Algorithm methods. `checkInput()` checks that the parameters passed are OK prior to processing, and returns `false` if they are not. `process()` does all the hard work, and return `false` if something goes wrong. If any of these two methods returns `false`, you are expected to document what went wrong in an error message that can be retrieved through `getErrorMessage()`.

The OutputAlgorithm method. This one just asks us to return the results as a list of spots. It must be a field of your instance, that is ideally instantiated and built in the `process()` method. The `getResult()` method exposes this list.

The Benchmark method. Well, we just want to know how much time it took. Note that all of these are the usual suspects of an `ImgLib2` generic algorithm, so they should not confuse you.

The code itself. Here is the code listing:

```
package plugin.trackmate.examples.detector;

import java.util.ArrayList;
import java.util.List;

import net.imglib2.Interval;
import net.imglib2.type.NativeType;
import net.imglib2.type.numeric.RealType;
import fiji.plugin.trackmate.Spot;
import fiji.plugin.trackmate.detection.SpotDetector;

public class SpiralDummyDetector< T extends RealType< T > & NativeType< T >> implements
    ↳ SpotDetector< T >
{

    private static final double RADIAL_SPEED = 3d; // pixels per frame

    // radians per frame
    private static final double ANGULAR_SPEED = Math.PI / 10;

    // in image units
    private static final double SPOT_RADIUS = 1d;
```

```

/** The width if the ROI. */
private final long width;

/** The height if the ROI. */
private final long height;

/** The X coordinates of the ROI. */
private final long xstart;

/** The Y coordinates of the ROI. */
private final long ystart;

/** The pixel sizes in the 3 dimensions. */
private final double[] calibration;

/** The frame we operate in. */
private final int frame;

/** Holder for the results of detection. */
private List< Spot > spots;

/** Error message holder. */
private String errorMessage;

/** Holder for the processing time. */
private long processingTime;

/*
 * CONSTRUCTOR
 */

public SpiralDummyDetector( final Interval interval, final double[] calibration, final int
    ↪ frame )
{
    // Take the ROI box from the interval parameter.
    this.width = interval.dimension( 0 );
    this.height = interval.dimension( 1 );
    this.xstart = interval.min( 0 );
    this.ystart = interval.min( 1 );
    // We will need the calibration to convert to physical units.
    this.calibration = calibration;
    // We need to know what frame we are in.
    this.frame = frame;
}

/*
 * METHODS
 */

```

```

@Override
public List< Spot > getResult()
{
    return spots;
}

@Override
public boolean checkInput()
{
    // Nothing to test, it's all good.
    return true;
}

@Override
public boolean process()
{
    final long start = System.currentTimeMillis();
    spots = new ArrayList< Spot >();

    /*
     * This dummy detector creates spots that spiral out from the center of
     * the specified ROI. It spits a new spiral every 10 frames.
     */

    final int x0 = ( int ) ( width / 2 + xstart );
    final int y0 = ( int ) ( height / 2 + ystart );

    int t = frame;
    int nspiral = 0;
    while ( t >= 0 )
    {
        final double r = t * RADIAL_SPEED;
        final double phi0 = nspiral * Math.PI / 4;
        final double phi = t * ANGULAR_SPEED + phi0;

        // Spot in pixel coordinates.
        final double x = x0 + r * Math.cos( phi );
        final double y = y0 + r * Math.sin( phi );

        // But we want to create spots in image coordinates:
        final double xpos = x * calibration[ 0 ];
        final double ypos = y * calibration[ 1 ];
        final double zpos = 0d;

        // Create the spot.
        final Spot spot = new Spot( xpos, ypos, zpos, SPOT_RADIUS, 1d / ( nspiral + 1d ) );
        spots.add( spot );

        // Loop to another spiral.
    }
}

```

```

        t = t - 10;
        nspiral++;
    }

    final long end = System.currentTimeMillis();
    this.processingTime = end - start;
    return true;
}

@Override
public String getErrorMessage()
{
    /*
     * If something wrong happens while you #checkInput() or #process(),
     * state it in the errorMessage field.
     */
    return errorMessage;
}

@Override
public long getProcessingTime()
{
    return processingTime;
}
}

```

And that's about it. Now for something completely different, we move to the factory class that instantiates this detector.

15.3 The SpotDetectorFactory interface.

The SpotDetectorFactory concrete implementation is the class that needs to be annotated with the [SciJava](#) annotation. For instance:

```

@Plugin( type = SpotDetectorFactory.class )
public class SpiralDummyDetectorFactory< T extends RealType< T > & NativeType< T >>
    ↪ implements SpotDetectorFactory< T >

```

Note that we have to deal with the same type parameter than for the SpotDetector instance. We skip all the TrackMateModule methods we have seen over and over in this tutorial series. There is nothing new here, they all have the same role. The difficult and interesting parts are linked to what we introduced above. Basically we need to provide a logic for passing the raw image data, for saving/loading to XML, for querying the user for parameters, and checking them.

15.3.1 Getting the raw image data.

Since the TrackMateModule concrete implementation must have a blank constructor, there must be another way to pass required arguments to factories. For SpotDetector factories, this

role is played by the `setTarget` method:

```
@Override
public boolean setTarget( ImgPlus< T > img, Map< String, Object > settings )
```

The raw image data is returned as an [ImgPlus](#), that can be seen as the [ImgLib2](#) equivalent of ImageJ1 [ImagePlus](#). It contains the pixel data for all available dimensions (all X, Y, Z, C, T if any), plus the spatial calibration we need to operate in physical units. The concrete factory must be able to extract from this `ImgPlus` the data useful for the `SpotDetectors` it will instantiate, keeping in mind that each `SpotDetector` operates on one frame.

The second argument is the settings map for this specific detector. It takes the shape of a map with string keys and object values, that can be cast to whatever relevant class. The concrete factory must be able to check that all the required parameters are there, and have a valid class, and to feed to the `SpotDetector` instances. We will see below that the user provides them through a configuration panel.

15.3.2 Getting detection parameters through a configuration panel.

For a proper `TrackMate` integration, we need to provide a means for users to tune the detector they chose. And since `TrackMate` was built first to be used through a GUI, we need to create a GUI element for this task: a configuration panel. The class that does that in `TrackMate` is [ConfigurationPanel](#). It is an abstract class that extends `JPanel`, and that adds two methods to display a settings map and return it.

Each `SpotDetectorFactory` has its own configuration panel, which must be instantiated and returned through:

```
@Override
public ConfigurationPanel getDetectorConfigurationPanel( Settings settings, Model model )
```

The GUI panel has access to the model and settings objects, and can therefore display some relevant information.

This is a difficult part because you have to write a GUI element. GUIs are excruciating long and painfully hard to write, at least if you want to get them right. Check the configuration panel of the [LoG detector](#) for an example.

15.3.3 Checking the validity of parameters.

There is a layer of methods that allows checking for the parameter validity. Normally you don't need to, since you write the configuration panel for the detector you also develop, but I have found this to be useful to find errors early. Parameter checking is done after user edition, loading and saving.

Three methods are at play:

```
public Map< String, Object > getDefaultSettings();
public boolean checkSettings( Map< String, Object > settings );
public String getErrorMessage();
```

The `getDefaultSettings()` method return a new settings map initialized with default values. It must contain all the required parameter keys, and nothing more. The `checkSettings()`

method does the actual parameter checking. It must check that all the required parameters are there, that they have the right class, and that there is no spurious mapping in the map. Should any of these defects be found, it returns `false`. Finally, `getErrorMessage()` should return a meaningful error message if the check failed.

15.3.4 Saving to and loading from XML.

TrackMate tries to save as much information as possible when saving to XML. The save file should contain at the very least the tracking results, but it should also include the parameters that help creating these results. The detection algorithm parameters should therefore be included.

You have to provide the means to save and load this parameters, since they are specific to the detector you write. This is done through the two methods:

```
public boolean marshall( Map< String, Object > settings, Element element );
public boolean unmarshall( Element element, Map< String, Object > settings );
```

Marshalling. Marshalling is the action of serializing a java object to XML. TrackMate relies on the [JDom library](#) to do so, and it greatly simplifies the task.

The settings map that the `marshall` method receives is the settings map to save. You can safely assume it has been successfully checked. The element parameter is a [JDom element](#), and it must contain everything you want to save from the detector, as attribute or child elements. Here is what you must put in it:

- You must at the very least set an attribute that has for key `"DETECTOR_NAME"` and value the `SpotDetectorFactory` key (the one you get with the `getKey()` method). This will be used in turn when loading from XML, to retrieve the right detector you used.
- If something goes wrong when saving, then the `marshall` method must return `false`, and you must provide a meaningful error message for the `getErrorMessage()` method.
- Everything else is pretty much up to you. There is a [helper method in IOUtils](#) that you can use to serialize single parameters. Check the [LogDetectorFactory marshall method](#) for an example.

Unmarshalling. Unmarshalling is just the other way around. You get a map that you must first clear, then build from the JDom element specified. You can safely assume that the XML element you get was built from the `marshall` method of the same `SpotDetectorFactory`. TrackMate makes sure the right `unmarshall` method is called.

There are a few help methods around to help you with reading from XML. For instance, check all the `read*Attribute` of the [IOUtils](#) class. It is also a good idea to call the `checkSettings` method with the map you just built.

Check again the [LogDetectorFactory unmarshall method](#) for an example.

15.3.5 Instantiating spot detectors.

And finally, the method that gives its name to this factory:

```
public SpotDetector< T > getDetector( Interval interval, int frame )
```

This will be called repeatedly by TrackMate to generate as many SpotDetector instances as there is frames in the raw data to segment. The two parameters specify the ROI the user wants to operate on as an [Imglib2 interval](#), and the target frame. So you need to process and bundle:

- this interval and this frame;
- the raw image data and settings map received from the setTarget method in the parameters required to instantiate a new SpotDetector.

Because the dummy example we use in this tutorial is not very enlightening, we copy here the code from the LogDetectorFactory. It shows how to extract parameters from a settings map, and how to access the relevant data frame in a possibly 5D image:

```
@Override
public SpotDetector< T > getDetector( final Interval interval, final int frame )
{
    final double radius = ( Double ) settings.get( KEY_RADIUS );
    final double threshold = ( Double ) settings.get( KEY_THRESHOLD );
    final boolean doMedian = ( Boolean ) settings.get( KEY_DO_MEDIAN_FILTERING );
    final boolean doSubpixel = ( Boolean ) settings.get( KEY_DO_SUBPIXEL_LOCALIZATION );
    final double[] calibration = TMUtils.getSpatialCalibration( img );

    RandomAccessible< T > imFrame;
    final int cDim = TMUtils.findCAxisIndex( img );
    if ( cDim < 0 )
    {
        imFrame = img;
    }
    else
    {
        // In ImgLib2, dimensions are 0-based.
        final int channel = ( Integer ) settings.get( KEY_TARGET_CHANNEL ) - 1;
        imFrame = Views.hyperSlice( img, cDim, channel );
    }

    int timeDim = TMUtils.findTAxisIndex( img );
    if ( timeDim >= 0 )
    {
        if ( cDim >= 0 && timeDim > cDim )
        {
            timeDim--;
        }
        imFrame = Views.hyperSlice( imFrame, timeDim, frame );
    }
}
```



```

    final LogDetector< T > detector = new LogDetector< T >( imFrame, interval, calibration,
        ↪ radius, threshold, doSubpixel, doMedian );
    detector.setNumThreads( 1 );
    return detector;
}

```

15.3.6 The code for the dummy spiral generator factory.

And here is the full code for this tutorial example. It is the ultimate simplification of a Spot-DetectorFactory, and was careful to strip anything useful by first ignoring the image content, second by not using any parameter. You can also find it [online](#).

```

package plugin.trackmate.examples.detector;

import ij.ImageJ;
import ij.ImagePlus;

import java.util.Collections;
import java.util.Map;

import javax.swing.ImageIcon;

import net.imglib2.Interval;
import net.imglib2.meta.ImgPlus;
import net.imglib2.type.NativeType;
import net.imglib2.type.numeric.RealType;

import org.jdom2.Element;
import org.scijava.plugin.Plugin;

import fiji.plugin.trackmate.Model;
import fiji.plugin.trackmate.Settings;
import fiji.plugin.trackmate.TrackMatePlugIn_;
import fiji.plugin.trackmate.detection.SpotDetector;
import fiji.plugin.trackmate.detection.SpotDetectorFactory;
import fiji.plugin.trackmate.gui.ConfigurationPanel;
import fiji.plugin.trackmate.util.TMUtils;

@Plugin( type = SpotDetectorFactory.class )
public class SpiralDummyDetectorFactory< T extends RealType< T > & NativeType< T >>
    ↪ implements SpotDetectorFactory< T >
{

    static final String INFO_TEXT = "<html>This is a dummy detector that creates spirals made
        ↪ of spots emerging from the center of the ROI. The actual image content is not used
        ↪ .</html>";
    private static final String KEY = "DUMMY_DETECTOR_SPIRAL";
    static final String NAME = "Dummy detector in spirals";
}

```

```

private double[] calibration;

private String errorMessage;

@Override
public String getInfoText()
{
    return INFO_TEXT;
}

@Override
public ImageIcon getIcon()
{
    return null;
}

@Override
public String getKey()
{
    return KEY;
}

@Override
public String getName()
{
    return NAME;
}

@Override
public SpotDetector< T > getDetector( final Interval interval, final int frame )
{
    return new SpiralDummyDetector< T >( interval, calibration, frame );
}

@Override
public boolean setTarget( final ImgPlus< T > img, final Map< String, Object > settings )
{
    /*
     * Well, we do not care for the image at all. We just need to get the
     * physical calibration and there is a helper method for that.
     */
    this.calibration = TMUtils.getSpatialCalibration( img );
    // True means that the settings map is OK.
    return true;
}

@Override
public String getErrorMessage()
{

```

```

    /*
     * If something is not right when calling #setTarget (i.e. the settings
     * maps is not right), this is how we get an error message.
    */
    return errorMessage;
}

@Override
public boolean marshall( final Map< String, Object > settings, final Element element )
{
    /*
     * This where you save the settings map to a JDom element. Since we do
     * not have parameters, we have nothing to do.
    */
    return true;
}

@Override
public boolean unmarshall( final Element element, final Map< String, Object > settings )
{
    /*
     * The same goes for loading: there is nothing to load.
    */
    return true;
}

@Override
public ConfigurationPanel getDetectorConfigurationPanel( final Settings settings, final
    ↪ Model model )
{
    // We return a simple configuration panel.
    return new DummyDetectorSpiralConfigurationPanel();
}

@Override
public Map< String, Object > getDefaultSettings()
{
    /*
     * We just have to return a new empty map.
    */
    return Collections.emptyMap();
}

@Override
public boolean checkSettings( final Map< String, Object > settings )
{
    /*
     * Since we have no settings, we just have to test that we received the
     * empty map. Otherwise we generate an error.

```

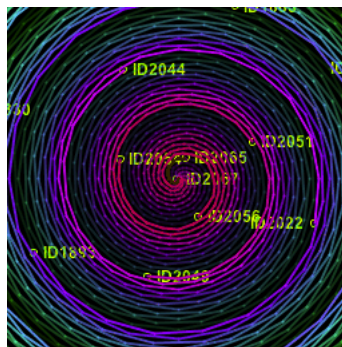
```

    */
    if ( settings.isEmpty() ) { return true; }
    errorMessage = "Expected the settings map to be empty, but it was not: "+settings+'\n';
    return false;
}
}

```

15.4 Wrapping up.

This was a lot of information and a lot of coding for a single piece of functionality. But all of these painful methods make your detector a first-class citizen of TrackMate. "Native" detectors use the exact same logic. Here is what our dummy example looks, after tracking the spots this it creates.



16. How to write your own particle-linking algorithm for TrackMate.

16.1 Introduction.

This last part on particle-linking modules concludes the series of tutorials on extending TrackMate. The most difficult modules to create are spot detectors, which was the subject of the previous section. Particle-linking modules, or trackers, are a little bit less complicated. However, you still need to understand how we store and manipulate links in TrackMate, and this implies very briefly introducing mathematical graphs.

16.2 Simple, undirected graphs.

TrackMate stores the results of the detection step as spots in a [SpotCollection](#). The tracking results are mainly links between these spots so we needed a structure to hold them. We went for the most general one, and picked a mathematical graph.

[Mathematical graphs](#) are mathematical structures that hold objects (**vertices**) and links between them (**edges**, we will use the two terms interchangeably). TrackMate relies specifically on a specialization: it uses an undirected, simple weighted graph.

- Undirected means that a link between A and B is the same as a link between B and A. There is no specific direction and it cannot be exploited. However, you will see that the API offers specific tools that can fake a direction. Indeed, since we deal mainly with time-lapse data, we would like to make it possible to say that we iterate a graph following the time direction.
- Simple is not related to the efforts that must be made to grasp this mathematical field, but to the fact that there can be only 1 or no link between two spots, and that we do not authorize a link going from one spots to this same spot (no loop).
- Weighted means that each link has a numerical value, called weight, associated to it. We use it just to store some of the results of the tracking algorithm, but it has no real impact on TrackMate.

This restrictions do not harm the generality of what you can represent in Life Science with this. You can still have the classical links you find in typical time-lapse experiment:

- Following a single object over time:

A0 - A1 - A2 - A3 - ...

- A cell division:

A0 - A1 -- B2 - B3 - ...
 |
 +- C2 - C3 - ...

- But also anything fusions, tripolar divisions, and a mix of everything in the same model.

16.3 Graphs in TrackMate.

On a side note, this is important if you plan to build analysis tools for TrackMate results. TrackMate philosophy is to offer managing the most general case (when it comes to linking), but your analysis tools might require special use cases.

- For instance, when you are tracking vesicles that do not fuse nor split, you just have a linear data structure (an array of objects for each particle).
- When you follow a cell lineage, you have a [rooted mathematical tree](#).
- And if all cells divide in two daughters, then you have a [rooted binary tree](#).

They all are specialization of the simple graph, and offer special tools that can be very useful. But *TrackMate assumes none of these specializations*. It stores and manipulate a graph.

Since we are Java coders, we use a Java library to manipulate these graphs, and for this we rely on the excellent [JGraphT](#) library. Why a graph? Why not storing a list of successors and a list of predecessors for each spot? Or directly have a track object that would save some time on determining what are the tracks? Well, because a graph is very handy and simple to use when creating links. When you will write your own tracker, and found a link you want to add the model, the only thing you have to do is: `graph.addEdge(A, B)`. You don't have to care

whether A belongs to a track and if yes to what track, you don't need to see the whole graph globally, you can just focus on the local link. Adding a link in the code is always very simple.

Then of course, you still need a way to know how many tracks are there in the model, and what are they made of. But this is the job of TrackMate. It offers the API that hides the graph and deals in track. This is done via a component of the model, the [TrackModel](#). But in the tracker we will not use this. We will be given a simple graph, and will have to flesh it out with spots and links between these spots. When the tracker completes, TrackMate will build and maintains a list of tracks from it.

The price to pay for this simplicity is that - when tracking - it is not trivial to get the global information. For instance, it is easy to query whether a link exists between two spots, but the graph does not see the tracks directly. If you need them, you either have to build them from the graph, either have to maintain them locally. But more on this below.

16.4 Particle-linking algorithms in TrackMate.

We used the term *tracker* since the beginning of this series, but the correct term for what we will build now is particle linking algorithm. Our particles are the visible spots resulting from the detection step, and the links will be the edges of the graph. A tracker could be defined as the full application that combines a particle detection algorithm with a particle linking algorithm.

In TrackMate, particle linking algorithms implements the [SpotTracker](#) interface. It is very simple. As explained in the docs, a SpotTracker algorithm is simply expected to create a new [SimpleWeightedGraph](#) from the SpotCollection given (using of course only the *visible* spots). We use a simple weighted graph:

- Though the weights themselves are not used for subsequent steps, it is suggested to use edge weight to report the cost of a link.
- The graph is undirected, however, some link direction can be retrieved later on using the Spot.FRAME feature. The SpotTracker implementation does not have to deal with this; only undirected edges are created.
- Several links between two spots are not permitted.
- A link with the same spot for source and target is not allowed.
- A link with the source spot and the target spot in the same frame is not allowed. This must be enforced by implementations.

There is also an extra method to pass a instance of [Logger](#) to log the tracking process progresses. That's all.

16.5 A dummy example: drunken cell divisions.

There is already an example online that does [random link creation](#). Let's do something else, and build a tracker that links a spot to any two spots in the next frame (if they exist) as if it would go cell division as fast as it can.

Creating the class yields the following skeleton:

```

package plugin.trackmate.examples.tracker;

import org.jgrapht.graph.DefaultWeightedEdge;
import org.jgrapht.graph.SimpleWeightedGraph;

import fiji.plugin.trackmate.Logger;
import fiji.plugin.trackmate.Spot;
import fiji.plugin.trackmate.tracking.SpotTracker;

public class DrunkenCellDivisionTracker implements SpotTracker
{
    private SimpleWeightedGraph< Spot, DefaultWeightedEdge > graph;

    private String errorMessage;

    private Logger logger = Logger.VOID_LOGGER;

    @Override
    public SimpleWeightedGraph< Spot, DefaultWeightedEdge > getResult()
    {
        return graph;
    }

    @Override
    public boolean checkInput()
    {
        return true;
    }

    @Override
    public boolean process()
    {
        graph = new SimpleWeightedGraph<Spot,DefaultWeightedEdge>(DefaultWeightedEdge.class);
        return true;
    }

    @Override
    public String getErrorMessage()
    {
        return errorMessage;
    }

    @Override
    public void setNumThreads()
    {
        // Ignored. We do not multithreading here.
    }
}

```

```

@Override
public void setNumThreads( final int numThreads )
{
    // Ignored.
}

@Override
public int getNumThreads()
{
    return 1;
}

@Override
public void setLogger( final Logger logger )
{
    // Just store the instance for later use.
    this.logger = logger;
}
}

```

Parameters need to be passed to the class via its constructor. As for detectors, the factory we will build later will be in charge of getting these parameters. Of course, the most important one is the SpotCollection to track. In our case it will be the only one, as our dummy tracker do not have any settings. So we can have a constructor like this:

```

public DrunkenCellDivisionTracker( final SpotCollection spots )
{
    this.spots = spots;
}

```

then we exploit the SpotCollection in the process() method. Our strategy here is to loop over all the frames that have a content, and link each spot to two spots in the next frame - wherever they are - until there is either no source spots or no target spots left. The method looks like this:

```

@Override
public boolean process()
{
    graph = new SimpleWeightedGraph<Spot,DefaultWeightedEdge>(DefaultWeightedEdge.class);

    // Get the frames in order.
    final NavigableSet< Integer > frames = spots.keySet();
    final Iterator< Integer > frameIterator = frames.iterator();

    // Get all the visible spots in the first frame, and put them in a new
    // collection.
    final Iterable< Spot > iterable = spots.iterable( frameIterator.next(), true );
    final Collection< Spot > sourceSpots = new ArrayList< Spot >();
    for ( final Spot spot : iterable )
    {
        sourceSpots.add( spot );
    }
}

```



```

}

// Loop over frames, and link the source spots to spots in the next
// frame.
double progress = 0;
while ( frameIterator.hasNext() )
{
    final Integer frame = frameIterator.next();
    final Iterator< Spot > it = spots.iterator( frame, true );
    SOURCE_LOOP: for ( final Spot source : sourceSpots )
    {
        /*
         * Add the source to the graph, if it is not already done (doing
         * it several time is not a problem: it's backed up by a Set).
         */
        graph.addVertex( source );
        // Finds 2 targets.
        for ( int i = 0; i < 2; i++ )
        {
            if ( it.hasNext() )
            {
                final Spot target = it.next();
                // You must add it as vertex before creating the link.
                graph.addVertex( target );
                // This is how we create a link.
                final DefaultWeightedEdge edge = graph.addEdge( source, target );
                // We get the edge back, and set its weight through:
                if ( null != edge )
                {
                    graph.setEdgeWeight( edge, 3.14 );
                    /*
                     * Edge can be null if a link already exists between
                     * the two spots.
                     */
                }
            }
            else
            {
                break SOURCE_LOOP;
            }
        }
    }

    // Regenerate source list for next frame.
    sourceSpots.clear();
    for ( final Spot spot : spots.iterable( frame, true ) )
    {
        sourceSpots.add( spot );
    }
}

```

```

    progress += 1;
    logger.setProgress( progress / frames.size() );
}
return true;
}

```

So it's not really complicated. Which is good, because the complicated part, completely omitted here, is the one where you have to determine what links to create. This is where you Science should kick in.

16.6 The factory class.

Now that we have the clever part of the code (the one that does the actual linking), we need to deal with TrackMate integration. Like for the detection modules, this is done *via* a factory class, named [SpotTrackerFactory](#). It is completely equivalent to the SpotDetectorFactory we saw in the [previous tutorial](#), so I won't detail the common methods again.

The methods specific to the tracker are:

- `public SpotTracker create(final SpotCollection spots, final Map< String, Object > settings);`

This method instantiates the actual tracker class. You can see that it received the SpotCollection and a settings map. This method is expected to unpack this map and extract the actual parameters need to instantiate the tracker. Note that contrary to the detector factory, TrackMate calls this method only once for a tracking process. It does not generate a tracker per frame. So it is actually simpler than for detection: a tracker instance is expected to solve the tracking problem for the whole model at once. Therefore, there is no need for a `setTarget()` method, like previously.

- `public ConfigurationPanel getTrackerConfigurationPanel(final Model model);`

This method should generate a GUI panel to request tracking parameters from the user. Completely similar to the detection modules.

- `marshall` and `unmarshall`. Save to and retrieve from XML, like previously.
- `public String toString(final Map< String, Object > sm);`

Used to pretty-print the settings map specific to this tracker.

The rest is classic. Here is what it looks like for our tracker:



TrackMate recognize there were two tracks. You did not have to worry about that.

16.7 Wrapping up.

The full code, as well as the code for another tracker example can be found on [github](#). And this concludes flatly our series of tutorials on how to extend TrackMate. Go forth now, and bend it to your needs; it is *your* tool.

References

- [1] Lindeberg, T. *Feature detection with automatic scale selection*. International Journal of Computer Vision 30 (2) (1998) pp 77–116.
- [2] Lowe, D.G. *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, 60, 2 (2004), pp. 91-110.
- [3] Otsu, N., *A threshold selection method from gray-level histograms*, in IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, no. 1, pp. 62-66, Jan. 1979.
- [4] [Jaqaman, K. et al.](#), *Robust single-particle tracking in live-cell time-lapse sequences*, Nat Methods. 2008 Aug;5(8):695-702.
- [5] [Crocker and Grier](#). *Methods of Digital Video Microscopy for Colloidal Studies*, J Colloid Interf Sci (1996) vol. 179 (1) pp. 298-310.
- [6] Bentley, J. L. *Multidimensional binary search trees used for associative searching*, Communications of the ACM, vol. 18, no 9, 1975, p. 509-517.
- [7] [Sage, D. et al.](#), *Automatic tracking of Individual fluorescence Particles: Application to the study of chromosome dynamics*, IEEE Transactions on Image Processing, vol. 14, no. 9, pp. 1372-1383, September 2005.
- [8] Munkres, J. *Algorithms for the assignment and transportation problems*, Journal of the Society for Industrial and Applied Mathematics, 5(1):32–38, March 1957.
- [9] [Chenouard et al.](#), *Objective comparison of particle tracking methods*, Nature Methods, 2014.
- [10] Krull, A., et al., *A divide and conquer strategy for the maximum likelihood localization of low intensity objects*, Opt. Express 22, 210-228 (2014)