LUDWIG-MAXIMILIANS-UNIVERSITÄT
TECHNISCHE UNIVERSITÄT MÜNCHEN

Helmholtz Zentrum München
Institut für Bioinformatik und Systembiologie

Bachelorarbeit
in Bioinformatik

Automated construction of cell lineage trees from time-lapse microscopy data

Oliver Hilsenbeck

Aufgabensteller:	Prof. Dr. Dr. Fabian Theis
Betreuer:	Michael Schwarzfischer
Abgabedatum:	15.09.2011

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

15.09.2011 _____

Oliver Hilsenbeck

Abstract

Hematopoiesis is the process of regenerating all mature blood cells. It is based on hematopoietic stem cells (HSCs), i.e. blood cells that have the ability to give rise to new HSCs (self renewal) or differentiate to more specialized cell types. To maintain a constant number of every cell type, the blood system must be highly regulated. Yet the molecular mechanisms governing differentiation of HSCs are still poorly understood. A deeper biological understanding of this process would allow the development of treatments for severe diseases such as leukemia and anemia.

Continuous imaging of HSCs is an approach that has already proven to provide new insights into hematopoiesis. With this technique researchers can identify progenitor cells of differentiated blood cells and quantitatively analyze cellular components like transcription factors that control asymmetric cell division. Usually, the first step of analysis is to track cells in the imaging data, since this is necessary to further examine the depicted cells. Since blood cells tend to keep moving during experiments, and the generated gray-scale images can have very low contrast, it is sometimes difficult even for experts to identify cells in the images and distinguish them from debris. Therefore, reliable results of high quality are currently mainly generated by manual cell tracking, which is very time consuming. So mostly only a rather small subset of cells in each experiment is tracked, while the majority of cells in the experiment is neglected from further analysis.

In this work, a new software toolkit is presented that enables automated tracking of cells yielding reliable results of high, quantifiable quality. It integrates with the existing work-flow allowing further examination by using already existing tools. To maximize quality, the algorithm avoids unsure decisions and leaves them to the user instead. For that purpose, a powerful and easy to use graphical user interface (GUI) has been developed. The results of automated cell tracking are compared to those generated by manual tracking and are used for a biological analysis of cell movement. We show that the presented software not only helps to eliminate the bottleneck of manual cell tracking but also improves the quality of tracking data significantly. The increase in quantity and quality of available data will make further analysis based on computational methods much more reliable and allow us to tackle new biological questions in the future.

Zusammenfassung

Die Regenerierung von Blutzellen wird als Hämatopoese bezeichnet. Sie basiert auf hämatopoetischen Stammzellen (HSZs), d.h. Blutzellen, die in der Lage sind, neue HSZs zu erzeugen (Selbsterneuerung) oder in stärker spezialisierte Zellen zu differenzieren. Um eine konstante Anzahl aller Zelltypen aufrecht zu erhalten, wird das Blutsystem stark reguliert. Die molekularen Mechanismen, die die Differenzierung von HSZs steuern, sind aber kaum erforscht. Ein besseres biologisches Verständnis dieses Vorgangs würde die Entwicklung von Behandlungsmöglichkeiten für schwere Krankheiten wie Leukämie oder Anämie ermöglichen.

Kontinuierliche Beobachtung von HSZs hat bereits zu neuen Erkenntnissen bezüglich Hämatopoese geführt. Mit dieser Technologie können Forscher die Vorläufer von ausdifferenzierten Blutzellen erkennen und quantitativ zelluläre Komponenten wie Transkriptionsfaktoren, die asymmetrische Zellteilung regulieren, analysieren. Der erste Schritt der Analyse ist üblicherweise das Tracken von Zellen in den Bilddaten. Da Blutzellen die Tendenz haben, sich während den Experimenten zu bewegen, und da die erzeugten Graustufenbilder einen sehr geringen Kontrast haben können, ist es manchmal sogar für Experten schwierig, Zellen in den Bildern zu erkennen und von Fremdkörpern zu unterscheiden. Daher werden verlässliche Ergebnisse von hoher Qualität momentan hauptsächlich durch manuelles Tracken von Zellen generiert, was enorm viel Zeit kostet. Daher wird meistens nur ein kleiner Teil der Zellen in jedem Experiment getrackt, wohingegen die Mehrheit der Zellen im Experiment nicht für weitere Analysen verwendet werden kann.

In dieser Arbeit wird ein neues Software System vorgestellt, das es ermöglicht, automatisch Zellen zu verfolgen und dabei verlässliche Daten von hoher, quantifizierbarer Qualität zu erzeugen. Es ist optimal in unseren Work-Flow integriert und ermöglicht somit die weitere Analyse der Daten mit bereits existierenden Tools. Um die Qualität zu maximieren, vermeidet der Algorithmus unsichere Entscheidungen und überlässt diese stattdessen dem Benutzer. Zu diesem Zweck wurde eine mächtige und einfach zu bedienende grafische Benutzeroberfläche (GUI) entwickelt. Die Ergebnisse der automatischen Verfolgung von Zellen werden mit denen, die durch manuelle Zellverfolgung generiert wurden verglichen, und für eine biologische Analyse von Zellbewegungen benutzt. Wir zeigen, dass die vorgestellte Software nicht nur hilft, den Engpass der manuellen Zellverfolgung zu beseitigen, sondern auch die Qualität der erzeugten Daten erheblich verbessert. Die Verbesserung von Quantität und Qualität der verfügbaren Tracking Daten wird weitere auf Computern basierende Analysen verlässlicher machen und es ermöglichen, neue biologische Fragestellungen zu behandeln.

Contents

1. Introduction

1.1. Motivation

Under homeostatic conditions an adult human regenerates about $2.3 \cdot 10^6$ erythrocytes and $1.2 \cdot 10^5$ leukocytes per second [1]. This process is called hematopoiesis and is necessary, since cells of the blood system are dying constantly and have to be replaced. It relies on hematopoietic stem cells (HSCs) residing in the bone marrow, i.e. blood cells that have the ability to give rise to new HSCs (self renewal) or differentiate into more specialized cell types [2]. To maintain a constant number of every cell type, the blood system must be highly regulated. However, despite its importance and decades of research, the molecular mechanisms governing regulation of hematopoiesis are still poorly understood.

Continuous imaging of living HSCs is an approach that already provided new insights into hematopoiesis [3]. With this technique biological researchers can investigate tissue regeneration on the single cell level at high spatiotemporal resolutions. Thus, different types of cells and their progenitors can be identified, and concentrations of cellular components like transcription factors can be quantitatively analyzed to understand the molecular mechanisms regulating asymmetric cell division [4]. The measured concentrations can also serve as solid basis for mathematical models derived from, for example, systems of differential equations or Boolean networks [5, 6]. It has been shown that such models, especially if they were created using high quality data, can help deepen our understanding of biological processes and make correct predictions about the behavior of the underlying biological systems under different conditions [7].

However, microscopic time-lapse experiments generate large amounts of data. Our experiments at the hematopoiesis group of Timm Schroeder at the Stem Cell Dynamics research unit, Helmholtz Center Muenchen, for example, are typically divided into about 90 fields of view called *positions*. Each contains about 5,000 pictures, which sums up to 450,000 image files that usually have a resolution of $1388 \cdot 1040$ pixels. The amounts of data generated by time-lapse microscopy assays can be even larger [8], making manual examination impossible. Therefore, powerful computational methods for automated analysis are required. The first step of analysis is usually tracking of cells to generate lineage trees, but especially blood cells tend to move around at different speeds, which makes this a challenging task. Furthermore, cell images that have not been obtained by fluorescence microscopy but by bright field, phase contrast or differential contrast microscopy often contain debris like parts of dead cells or dirt. Fig. 1 shows two examples for pictures with particularly low quality.

Therefore, creating software for automated tracking of cells is not a trivial task. Nevertheless, various software toolkits for that purpose have been published [9, 10, 11, 12, 13]. Despite that, when highly reliable tracking data with little or no errors is required, cells are usually still tracked manually [14, 15, 16, 17, 18]. Although some of the available tools are powerful on the algorithmic level, using them instead of manual tracking is often problematic for various reasons that are covered in more detail in section 1.3.

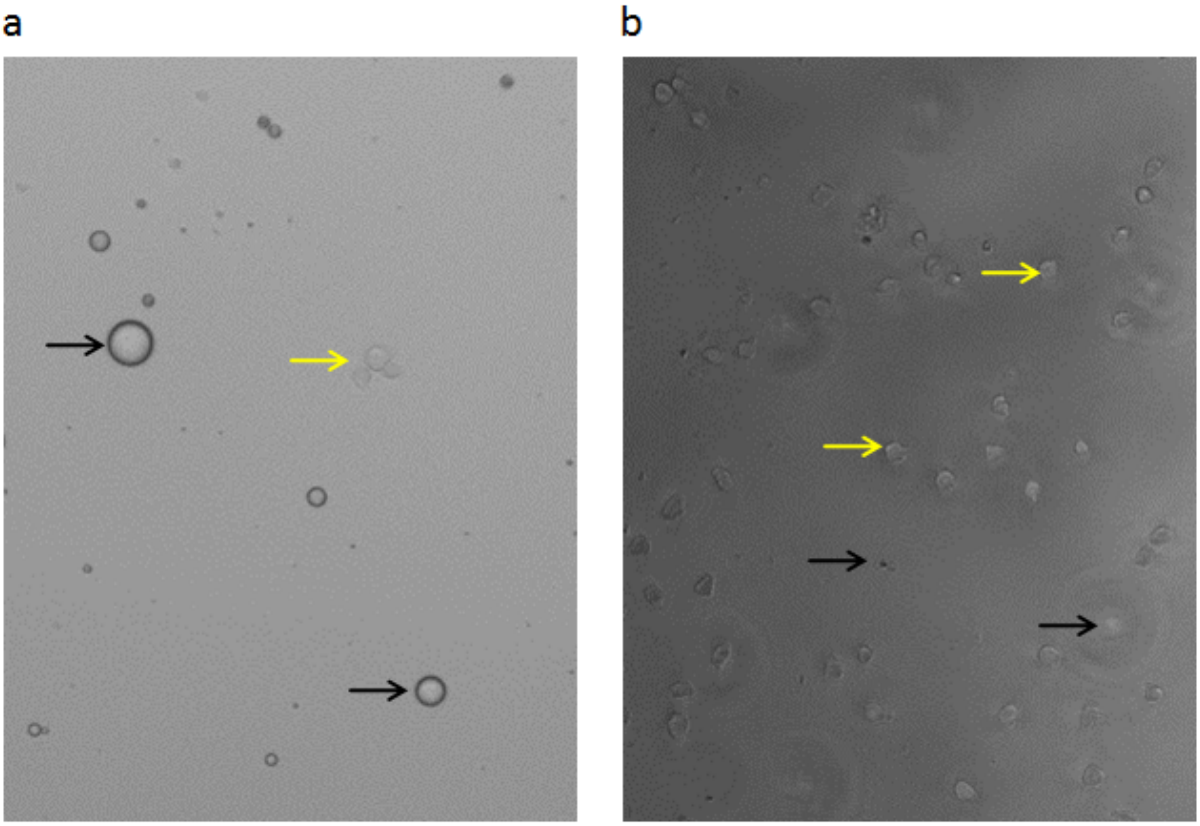

Fig. 1 Cell images with particularly low quality. For clarification some cells are highlighted with yellow arrows and some variants of debris with black arrows. a) Image with changes in contrast and way more debris than cells. b) Image with strong changes in brightness containing debris looking similar to cells (lower black arrow).

1.2. The Problem of Automated Cell Tracking

Teaching computers how to track objects in video image sequences has been a field of intense research for decades, since there is a wide variety of application possibilities for such technologies. Especially in the last years, various very powerful approaches have been published [19, 20] and were used to track, for example, the faces of humans yielding very impressive results [21]. Yet applying these algorithms to the problem of tracking cells in time-lapse microscopy data is problematic.

For example, one widely used technique for tracking objects is provided by the *scale-invariant feature transform* (SIFT) algorithm [19]. SIFT works by first extracting features from a picture and then computing *SIFT descriptors* of the found features that are invariant to changes in location, scale and rotation. The descriptors found in two successive pictures can then be compared and clustered in order to identify and track objects that appear in both images.

SIFT has actually been applied successfully to the problem of automated cell tracking in time-lapse microscopy data [22]. However, this works only well if the obtained images are of very good quality with spatiotemporal resolutions which are high enough to recognize specific features in each cell, because feature detection by SIFT is based on the idea of

looking for areas in the image that correspond to corners. When tracking objects like humans or cars, this makes perfectly sense as they can usually be described very well over different frames by focusing on their corners.

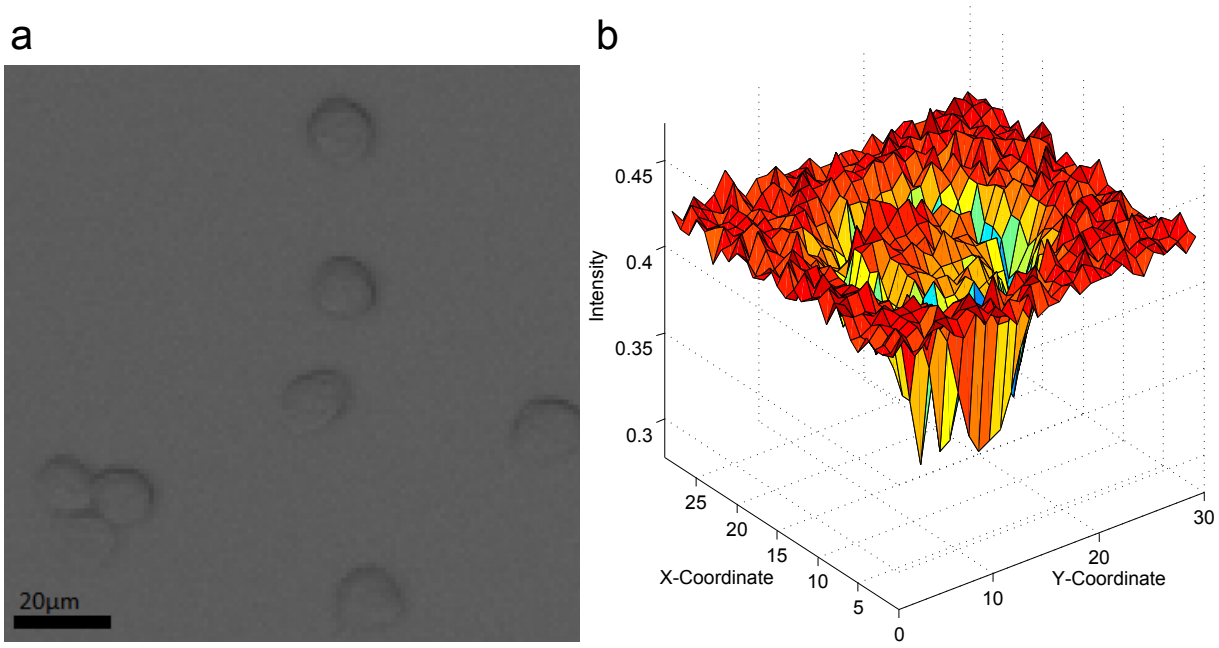

Fig. 2 Example for appearance of cells in images obtained by bright field microscopy. a) Cells as they appear in the image raw data (×3 zoom). The cells have a diameter of about 15 pixels, which corresponds to about 15 µm. b) Intensity values of one cell as heat map after normalization in the $[0, 1]$ interval. Due to low contrast the actual intensity values lay within the $[0.3, 0.45]$ interval, and it can be seen that the interior of the cell is very similar to the background of the picture.

However, especially images obtained by time-lapse microscopy often have low resolution and contrast as demonstrated in Fig. 2, so looking for corners in these images does not work equally well as for real world images as shown in Fig. 3. To measure the similarity of two depicted cells, it usually makes more sense to focus on more abstract features such as cell area, brightness and eccentricity [23].

1.3. Overview of Existing Software

As mentioned before, various software solutions for automated cell tracking have been published. This section gives a short overview of some of them. Usually, the first step of automated cell tracking is *segmentation*, i.e. the identification and classification of relevant objects like cells or beads in the image data. Then it is attempted to correctly assign each identified cell its correct successor in the next image.

CellTracker is one program for automated cell tracking, which has initially been developed by Hailin Shen at The University of Manchester [12]. In 2006, further development has been taken over by the Warwick Systems Biology Centre based at the university of Warwick. Since the most recent version of CellTracker is being distributed without the required version of the MATLAB Component Rumtime, which is not available for download, an older version from 2006 had to be used for testing. After selecting images for

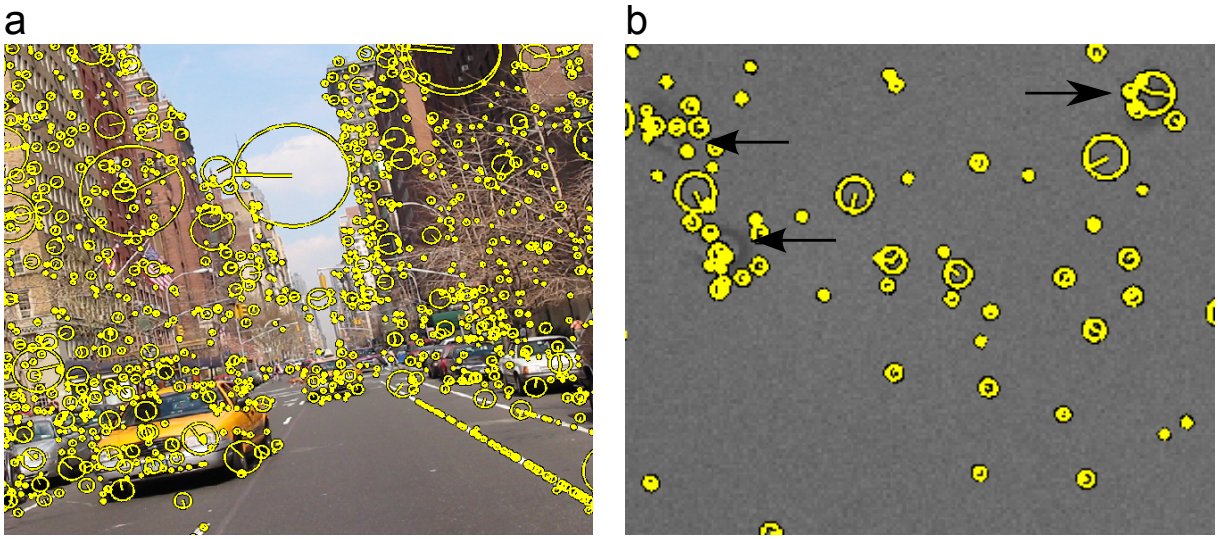

Fig. 3 Positions of SIFT features (yellow circles) detected in (a) real world and (b) bright field microscopy images. a) SIFT features are spread all over the relevant objects, whereas only a few are located in regions between them. By comparing and clustering sets of descriptors in successive frames, objects can be tracked reliably. Original image from [24]. b) Despite parameter optimization, SIFT feature detection does not work equally well for the cell image due to the absence of corners, small cell sizes and low contrast. The descriptors covering the cells (highlighted by black arrows) are unstable and turned out to be very similar for different cells, and a lot of features were detected between cells.

analysis, the program attempts to load all of them into memory at once. Loading a small set of 400 pictures resulted in an increase of memory consumption from about 60MBs to 660MBs. Due to memory limitations, this behavior renders the program useless for analyzing realistic numbers of pictures. Before starting the tracking algorithm, the user first has to mark nuclear and cytoplasmic boundaries. Due to these limitations, the program was not tested any further.

Another program, *TimeLapseAnalyzer* (TLA) [9], is very powerful on the algorithmic level but requires the input images in form of an *AVI* movie. Converting images to an AVI movie with a specific codec is not a trivial task since it requires the usage of advanced third-party programs like *ffmpeg*, and it takes a lot of time. Furthermore, TLA only supports AVI files that are either uncompressed or use one of five supported codecs with lossy compression. This is problematic, since uncompressed videos are very big (in a test converting 100 JPEG files consuming 17.4 MBs of space produced a 140.6 MBs AVI file), whereas lossy compression can have a bad impact on the tracking results. Once a suitable AVI file has been loaded, the user can choose between five predefined *setups* for cell tracking or create his own. Setups specify the algorithmic steps TLA will use for cell tracking and can be defined in a script language using the integrated *Advanced Setup Editor*. TLA has been successfully used to track cells from the same experiment that is shown in Fig. 2 by using the predefined setup for bright field microscopy. However, as TLA first failed to detect about 90% of the cells, it was necessary to adjust some parameters of the underlying algorithms. The results are shown in Fig. 4.

Despite being very powerful on the algorithmic level, TLA is not suitable to be used for automated cell tracking by regular users due to lacking user-friendliness. During testing

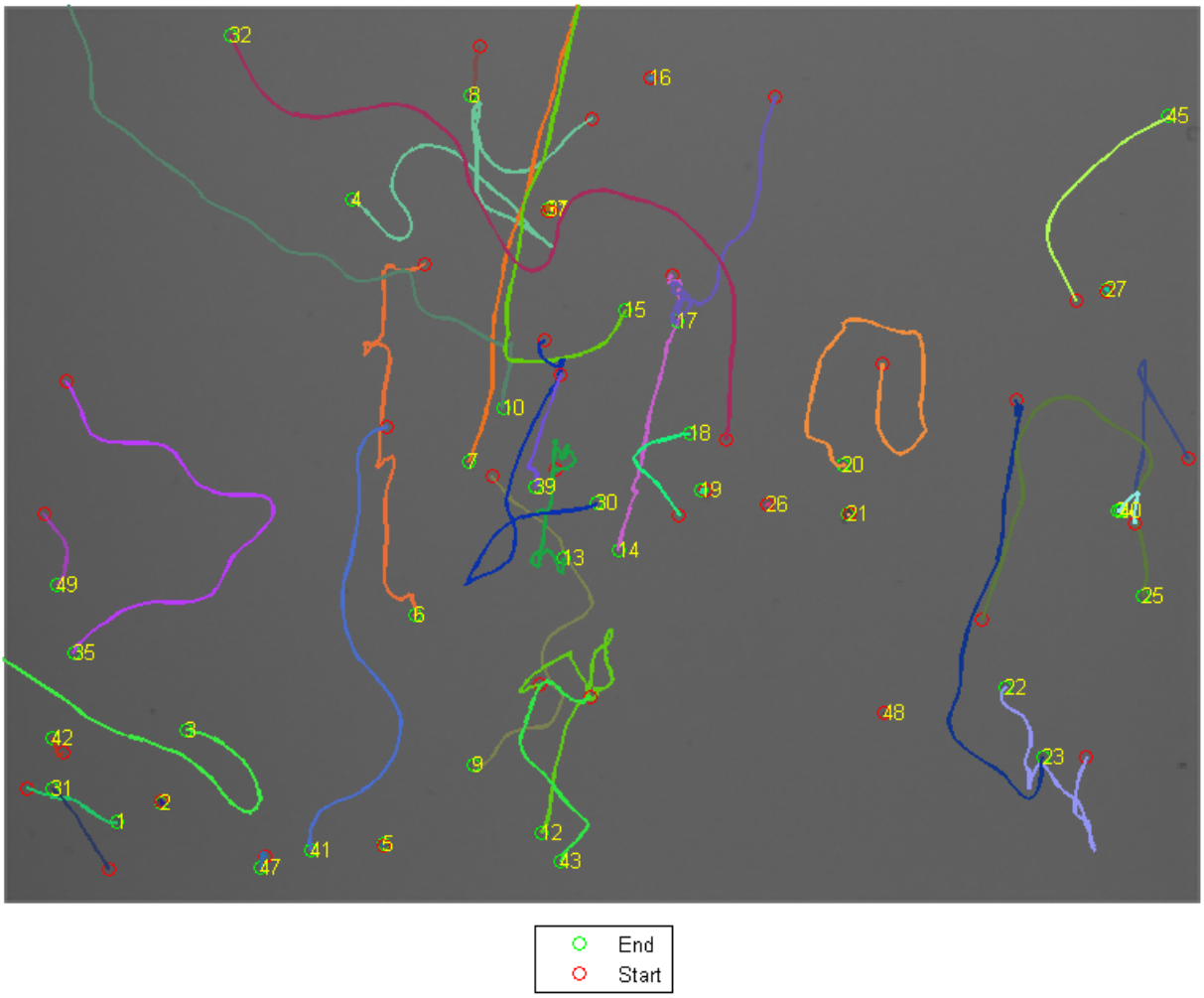

Fig. 4 Automated cell tracking results generated by *TimeLapseAnalyzer* for a subset of images obtained by bright field microscopy. The first image is shown with the starting positions of the detected cells (red circles), their ending positions (green circles) and their trajectories (colored lines).

it turned out, for example, not to be very robust against wrong user input. TLA crashed when it was attempted to load an AVI file using an unsupported codec or when parameters were invalid. In that case it also tends to display cryptic error messages - in one test the same error message was displayed several hundred times at the same time, once for each picture. Furthermore, the provided default setups usually require the adjustment of parameters to work properly for a given experiment. But using the *Advanced Setup Editor* is obviously difficult for users with no deep understanding of the underlying algorithms and no programming experience.

Another powerful tool that can be used for tracking cells is *CellProfiler* [11]. Like TLA, it is very powerful on the algorithmic level and also supports the integration of external tools such as *ilastik* [25] for segmenting images. However, creating and adapting the required *processing pipelines* in CellProfiler to specific experiments is not easy, if the user does not know the underlying algorithms very well.

The issue of lacking user-friendliness observed in TLA and CellProfiler applies to all tested tools. Furthermore, the most important problems are that their output usually contains

no information about the reliability of the tracking results and the missing integration into our work-flow. None of them supports tracking cells moving over different positions, so each position has to be tracked one after another. Since there are no standard file formats, for every tool specific software would be necessary to convert the tracking results into formats that can be read by our existing tools. Finally, mostly no convenient possibility to manually inspect tracking results is provided.

1.4. Aims of this Work

In this work, a new software toolkit for automated cell tracking and lineage tree construction is presented that has been developed according to our needs and with the main focus put on user-friendliness. All software modules come with a powerful GUI that is as robust and intuitively to use as possible. Great care has been taken to ensure that it never freezes, crashes or fills the screen with error messages. It has been developed in close cooperation with people who will be using the software and results have been presented to them on a regular basis in order to get feedback as early as possible.

For maximal performance concerning both runtime and memory requirements, all parts of the software have been implemented using C++ and solely rely on external libraries also written in C or C++. Great care has also been taken to design all algorithms so that they efficiently scale to an arbitrary number of threads without causing notable overhead and maintaining a high level of robustness and stability at the same time. Furthermore, the code can be compiled on other operating systems than *Microsoft Windows*, and it includes a console version with no GUI. This makes it possible, for example, to run the software remotely on a server, for example using a batch-queuing system such as the *Sun Grid Engine*. When working with the console version, all settings have to be passed as command line arguments. To make that easier, the currently set parameters can be exported from the GUI version as a text string that can then be used directly as command line argument of the console version. The current settings can also be exported to a file that can be loaded again at any time. This way parameter sets for different types of experiments can be maintained. To make finding the correct parameters as easy as possible, the GUI includes a powerful preview feature showing segmentation results for the currently set parameters (see section 2.2 for more details).

Segmentation and tracking results are saved in well defined binary formats that have been designed for high performance (see appendix A and B). This enables users to perform the computationally expensive image segmentation part only once for an experiment, for example using the Sun Grid Engine, and then use the results for automated cell tracking several times to test different tracking parameters. This is helpful, since in contrast to the tracking settings, optimal segmentation parameters can be found efficiently using the preview feature. It also makes it easier to integrate external tools for image analysis like *ilastik* [25], whose output just has to be converted into the format used by our software and can then be loaded and used for tracking cells.

When tracking cells, even one error can be sufficient to render complete lineage trees useless for further analysis. This can happen, if two cells are accidentally mistaken for each other during tracking, which causes all daughter cells to be associated with the

wrong progenitors making analysis of the maturation process impossible. The fact that such errors can be very hard to find by manual inspection worsens this problem. Therefore, in this work the main focus is put on avoiding errors rather than automatically generating lineage trees that are as big as possible.

For that reason, the tracking algorithm has been designed to avoid unsure decisions and leave them to the user instead. Concretely, this means that a cell trajectory is interrupted if the correct successor cell in the next frame cannot be detected with high confidence. The required level of certainty can be set by adjusting corresponding thresholds (see section 2.2). Especially in experiments with many cells, this leads to the generation of many rather small lineage trees, which will be called *tree fragments* therefore. A powerful and easy to use GUI has been developed to let users inspect and connect the generated tree fragments. It requires only as little manual interaction as possible and therefore the effort required for tracking cells is reduced to a minimum compared to manual tracking.

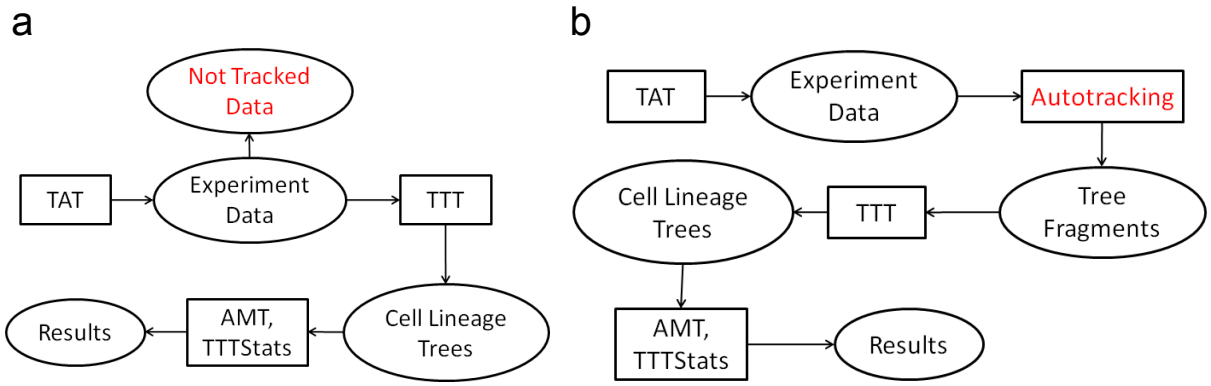

Fig. 5 Our work-flow with and without automated cell tracking. a) Without automated cell tracking a lot of data cannot be made available for analysis within reasonable time. b) Work-flow after integration of automated cell tracking. The time consuming task of clicking on the same cell throughout every time point is now done automatically, only clumping cells and cell divisions have to be examined manually. So, much more cells can be tracked and further analyzed.

The software perfectly fits into our work-flow. The previously mentioned GUI to analyze the tracking results has been integrated into *Timm's Tracking Tool* (TTT), which is the software currently used for manual cell tracking. The cell lineage trees created in TTT that are based on the automatically generated tree fragments can then easily be saved in the binary format already used by TTT, which can be read by other tools that are already part of our work-flow, such as *TTTStats* for statistical analysis or *Aided Manual Tracking* (AMT) [26, 27] for quantitative analysis of fluorescence intensities. The trees generated by automated cell tracking include a confidence level that can be visualized in TTT with an intuitive color scheme to make finding errors as easy as possible. Fig. 5 gives an overview of the current work-flow and how it will ideally look after integrating the new software for automated cell tracking.

To make further development and maintenance as easy as possible and to ensure high quality, all complex modules were designed with tools provided by the *Unified Modeling Language* (UML) [28] before starting with the actual implementation. The created UML diagrams now serve as an excellent documentation of the software design on a more abstract level than, for example, the comments in the source code.

2. Methods

2.1. The Algorithmic Approach

2.1.1. Overview

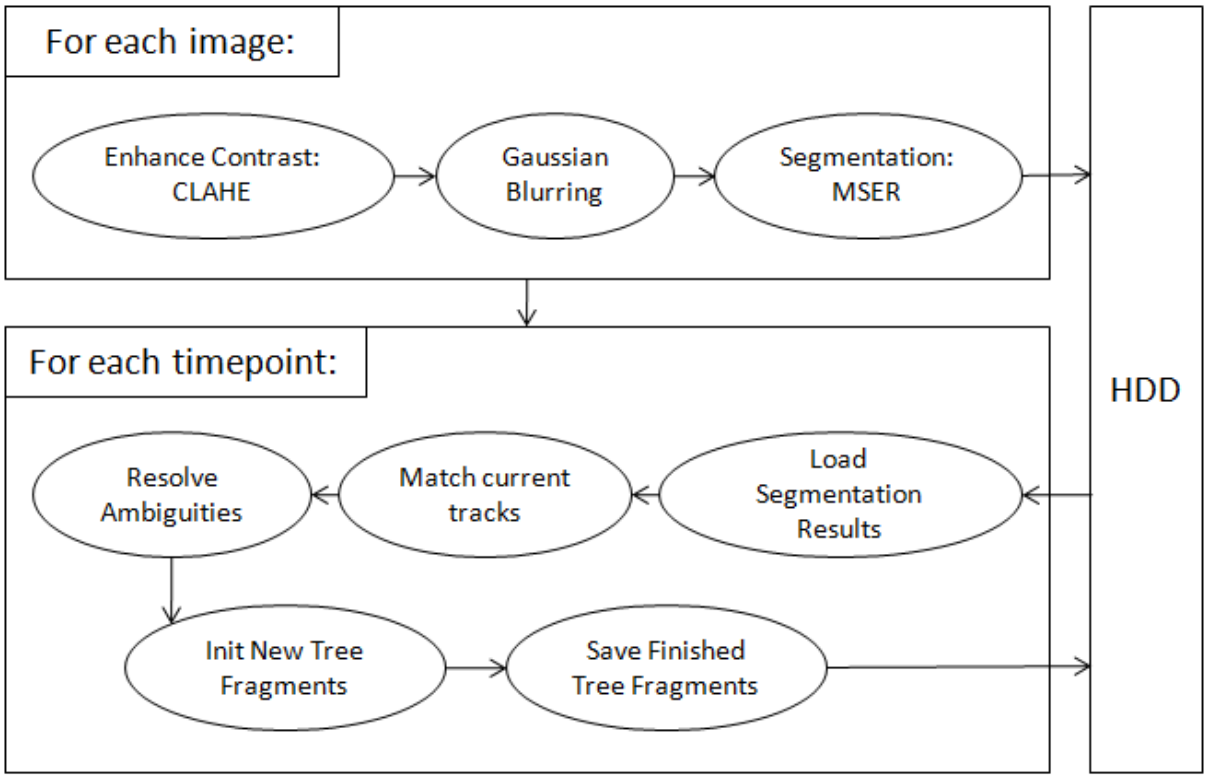

Fig. 6 Overview of the algorithmic approach used in this work. The upper box describes the image processing part. It is done independently for every picture and consists of image preprocessing and segmentation. The lower box illustrates the steps of the tracking algorithm, which is run after segmentation. For each time point, the segmentation results are loaded and it is attempted to identify the correct successors of all cells from the previous time point. After resolving ambiguities that possibly occurred in the previous step, new trees are started for all newly found cells and finished trees are saved.

Fig. 6 gives an overview of the algorithmic approach used in this work. Notably, image processing and cell tracking are carried out separately. The image processing part is done independently for every picture and is therefore perfectly scalable to an arbitrary number of threads (see 2.2 for more details). For technical reasons, multi-threading does not scale equally well for the tracking part. All steps and the corresponding parameters are explained in more detail in the following sections.

2.1.2. Image Enhancement

For images with low contrast, the first step of preprocessing is contrast enhancement using *Contrast Limited Adaptive Histogram Equalization* (CLAHE) [29]. CLAHE is a

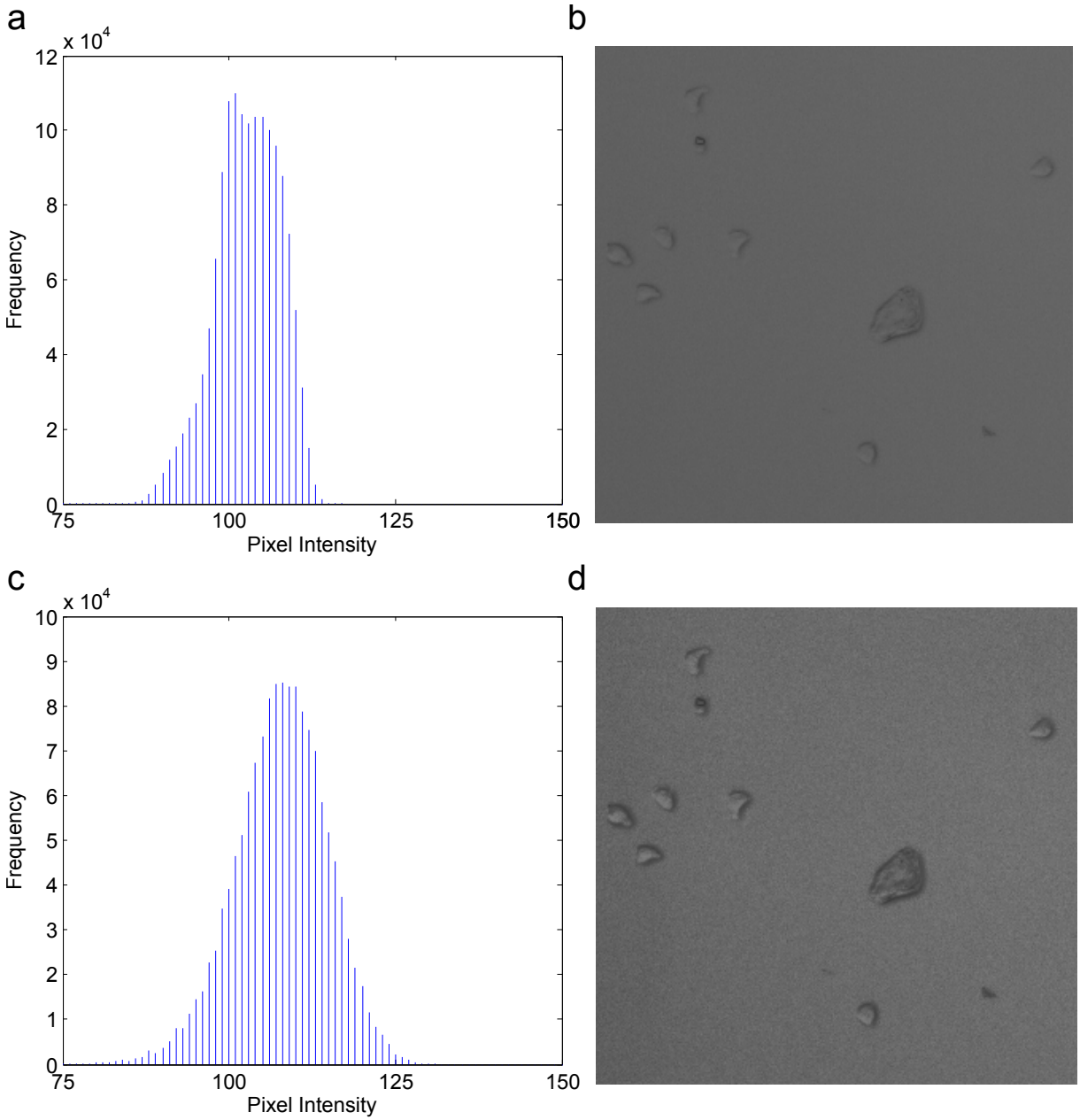

Fig. 7 Excerpts of histograms of pixel intensity values of an image before and after using CLAHE. a) The very thin histogram results in low contrast in the corresponding image (b). c) After using CLAHE, the histogram has become wider and contrast in the corresponding image (d) has increased notably.

special form of image histogram equalization, which is a technique to adjust the contrast by spreading out frequent intensity values. Since the intensity histograms of images can change depending on which region of the image is observed, the results can be improved notably by computing several histograms corresponding to distinct regions of the image rather than only one. This technique is called *Adaptive Histogram Equalization* and improves the *local contrast* of an image. It can, however, cause notable noise amplification, especially in homogeneous regions. To avoid this problem, CLAHE limits the contrast enhancement and thereby effectively reduces noise amplification. The value used as limit for contrast enhancement is specified by the *ClipLimit* parameter. The higher it is, the stronger is the contrast enhancement. Apart from that, the number of regions in x and y

direction, for which a distinct histogram should be computed, and the number of *greybins* can be specified. The number of bins determines how many different intensity values the output image will have, which influences processing time and image quality. In this work, it is always set to 256, since the processing time of CLAHE is only responsible for a tiny fraction of the overall computation time required for segmentation. Fig. 7 illustrates the effects of applying CLAHE to a cell image obtained by bright field microscopy.

Since cell images are mostly noisy, no matter if CLAHE has been used or not, in the next step *Gaussian Blurring* is used to reduce noise. Gaussian Blurring simply averages the intensity value of every pixel with the ones of its neighbors. For weighting the pixel intensities a Gaussian Distribution, characterized by its parameter σ, is used and applied to all pixels within the specified *kernel size*. Thus, the higher σ and the bigger the kernel size are, the stronger is the blurring effect.

2.1.3. Segmentation

In this work, *Maximally Stable Extremal Regions* (MSER) [30], an approach strongly related to thresholding techniques, is used for segmentation. Thresholding methods such as *Otsu's algorithm* [31] have been used successfully for segmentation of cell images generated by time-lapse microscopy [32]. The idea of this approach is to identify regions in the image that are darker (or brighter) than the background by applying a threshold to separate foreground from background pixels. This process is also called *binarization* and the detected regions are often referred to as *blobs*. Using the MSER algorithm, however, is an approach that has not been used in this context before. Though, it has been used successfully to track objects like car license plates or human faces [33] and for segmenting blood smear images [34].

The general idea of MSER is not to use one or a few thresholds for separating foreground from background, but to apply all thresholds that are possible for a given picture. Regions detected by applying different thresholds are compared and if the size of a region remains stable over a range of thresholds, it is called a *maximally stable extremal region*. In other words, MSERs are regions where local binarization is stable over a range of different thresholds. The required minimum size of this range is specified by the *Delta* parameter, which strongly influences the required contrast for a blob in order to be detected. The higher it is, the bigger is the required threshold interval and the higher has to be its contrast to the background. The required degree of stability is specified by the $MaxVariation$ parameter. Apart from that, the minimum and maximum sizes of the blobs to be detected are specified by the $MinArea$ and $MaxArea$ parameters, and the $DarkOnBright$ and $BrightOnDark$ specify if the algorithm should look for *dark on bright* or *bright on dark* spots.

This approach makes MSER very robust against variations of brightness and contrast, which is especially useful in this context. The standard algorithm to detect MSERs runs in quasi-linear time ($\mathcal{O}(n \cdot log(log(n)))$ where n is the number of pixels in the image) and by now, algorithms running in true linear-time have been published (see section 4.1).

The blobs returned by the MSER algorithm are then further analyzed. Regions that are touching the borders of the image are marked as *cutoff* and ignored in the following steps,

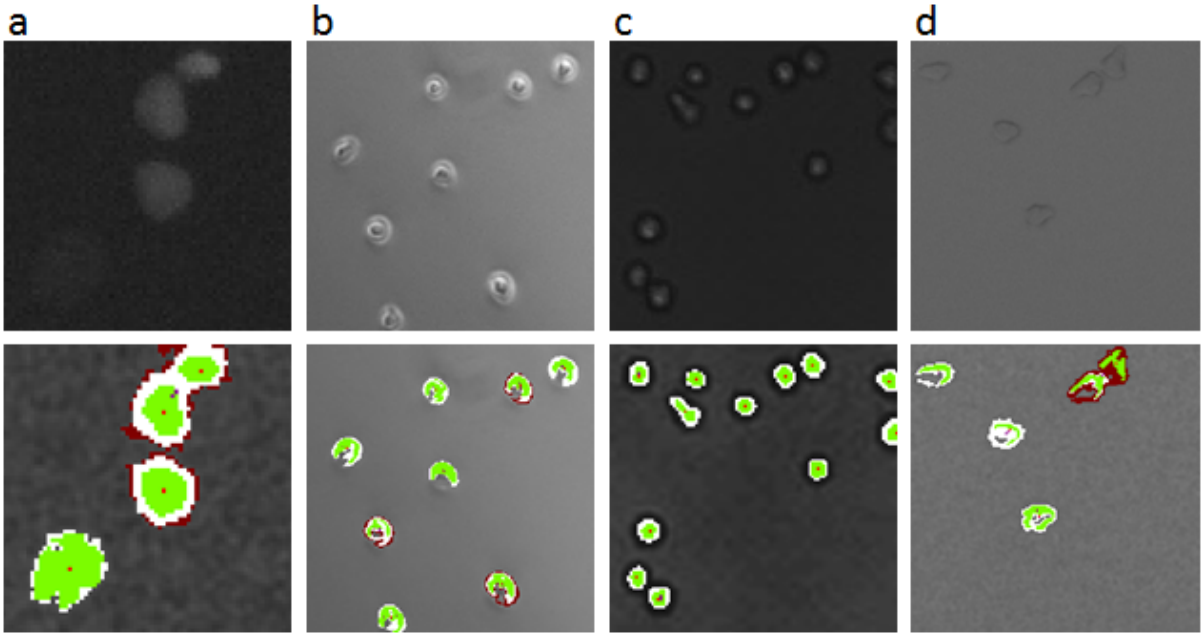

Fig. 8 Various cell images with segmentation results. Detected blobs are colored according to their classification: dark red is used for blobs that are too big or too small, white for blobs that have a valid size but contain smaller blobs that are still valid (i.e. not too small) and green for blobs that have a valid size and contain no child blobs of valid size. These blobs are most likely to correspond to single cells and are marked as *normal*. Centroids of green blobs are shown with a red dot and centroids of all other blobs are with a dot colored in magenta. a) Image obtained by fluorescence microscopy. Although contrast varies greatly, all cells are detected correctly. The two upper cells are very close together but were successfully separated by investigating the tree of detected blobs as described in the text. b) Images obtained by phase-contrast c-d) and bright field microscopy.

since it is mostly not possible to determine their sizes correctly. This is usually no problem as adjacent positions in our experiments are always overlapping. Then the areas of the found blobs are calculated and the ones that are too big or too small according to the corresponding parameters *minCellArea* and *maxCellArea* are marked as such. Sometimes the detected blobs do not cover the whole areas of cells but only sickle-shaped parts of their borders (see Fig. 8b and d). Therefore, the areas of the detected regions can be much smaller than the actual cell areas. This can be confusing for the users, since they have to specify these parameters. For that reason, by default the area of the convex hull of every detected blob is calculated and used for classification.

Another advantage of MSER over simpler thresholding methods is that it does not simply perform a binarization of the pixels into foreground and background. Instead, a hierarchical tree of nested blobs is returned where one blob can contain several other smaller ones which can itself contain even more smaller ones and so on. This situation actually occurs very often in cell images when cells are very close together. In that case MSER typically returns one large blob that contains several smaller blobs, each corresponding to a single cell. By applying the classification based on minimum and maximum cell sizes, this makes it possible to separately detect cells that would typically be returned as one blob with no further information by other segmentation methods. Therefore, in this work no further algorithms like *Watershedding* [35] are used to separate detected blobs.

The minimal size difference of two nested maximally stable extremal regions in order to be detected as distinct nested blobs is specified by the $MinDiversity$ parameter. If the relative area variation of two nested regions is below this threshold, only the most stable one is returned.

Fig. 8 shows that this approach can successfully segment different cell images that were obtained by fluorescence, phase-contrast and bright field microscopy. It can be seen also that the cell areas found by this approach are often smaller and do not perfectly match the actual cell areas, but this is no problem as long as the segmentation results are only used for automated cell tracking. For each picture, the segmentation results are saved to the HDD in a well defined binary format (see appendix A).

2.1.4. Tracking of Cells

The cell tracking algorithm is based on nearest neighbor search, an approach that has already been used successfully for this purpose [36, 37]. During tracking, the current time point is always stored in t and all currently tracked cells are stored in a list called $openTracks$, which is empty in the beginning. As soon as tracking of a cell stops, for example because no successor blob has been found, the corresponding cell is marked as *finished* and removed from $openTracks$.

As shown in Fig. 6, at first the segmentation results for all images at the current time point t are loaded. Then, $matchCurrentTracks$ (see Algorithm 1) tries to identify the correct successor blobs for all currently tracked cells. For that purpose, it iterates over all cells in $openTracks$, predicts their next positions at time point t (line 3) and then finds all blobs within the range specified by $searchRange$. If position prediction is enabled, the last 2 coordinates $\vec{v}_{t-1}$ and $\vec{v}_{t-2}$ of the cell at time points $t-1$ and $t-2$ are used to predict the position of the cell at time point t: $\vec{v}_t = \vec{v}_{t-1} + (\vec{v}_{t-1} - \vec{v}_{t-2}) = 2 \cdot \vec{v}_{t-1} - \vec{v}_{t-2}$. If prediction is disabled, $\vec{v}_t = \vec{v}_{t-1}$ is used instead. In case more than 1 blob has been found, all blobs that are at least $blobThreshold$ µm further away than the blob that is closest to the predicted position are removed from $foundBlobs$ (lines 5 to 14).

Ideally, $foundBlobs$ contains only one blob now, which means that a unique successor blob ($foundBlobs[1]$) for the current cell has been found with the required confidence level. In that case (lines 16 to 23), a new *track point* is created at the position of $foundBlobs[1]$ and appended to the current cell. An initial confidence level for the new track point is obtained by calculating the distance of $foundBlobs[1]$ to the predicted position ($dist$) and comparing it to $searchRange$: $confidence := 1 - (dist/searchRange)$. Since it always holds that $0 \leq dist \leq searchRange$, this yields a value between 0 and 1. The closer the found blob is to the predicted position, the closer $confidence$ is to 1. The current cell is then associated with the found blob by adding it to $foundBlobs[1].possibleTracks$. If this list contains 2 cells now, a cell has already been associated with $foundBlobs[1]$ before and $foundBlobs[1]$ is added to the $ambiguousBlobs$ list, which will be investigated later.

If $foundBlobs$ contains more than one blob, it means that a unique successor blob with the required confidence level could not be found. In that case (lines 24 to 32), the current cell is added to the $ambiguousTracks$ list, which will also be investigated later. In the next step, all found blobs are associated with the current cell by adding them to the

Algorithm 1 Match current tracks

```
1: procedure MatchCurrentTracks(openTracks, searchRange, blobThreshold)
2:     for all currentTrack ∈ openTracks do
3:         predictedPosition := predict next position of currentTrack
4:         foundBlobs := find blobs in searchRange around predictedPosition
   sorted by distance
5:         if |foundBlobs| > 1 then
6:             closestBlob := foundBlobs[1]
7:             distClosestBlob := distance closestBlob to predictedPosition
8:             for i := 2, |foundBlobs| do
9:                 distOfThisBlob := distance foundBlobs[i] to predictedPosition
10:                if distOfThisBlob − distClosestBlob ≥ blobThreshold then
11:                    Remove this and all following blobs from foundBlobs
12:                    break
13:                end if
14:            end for
15:        end if
16:        if |foundBlobs| = 1 then
17:            dist := distance foundBlobs[1] to predictedPosition
18:            confidence := 1 − (dist/searchRange)
19:            currentTrack.addTrackPoint(foundBlobs[1].position, confidence)
20:            Add currentTrack to foundBlobs[1].possibleTracks
21:            if |foundBlobs[1].possibleTracks| = 2 then
22:                Add foundBlobs[1] to ambiguousBlobs
23:            end if
24:        else if |foundBlobs| > 1 then
25:            Add currentTrack to ambiguousTracks
26:            for all curBlob ∈ foundBlobs do
27:                Add curBlob to currentTrack.possibleSuccessorBlobs
28:                Add currentTrack to curBlob.possibleTracks
29:                if |curBlob.possibleTracks| = 2 then
30:                    Add curBlob to ambiguousBlobs
31:                end if
32:            end for
33:        else
34:            openTracks.remove(currentTrack)
35:            Finish currentTrack
36:        end if
37:    end for
38: end procedure
```

currentTrack.possibleSuccessorBlobs list and by adding the current cell to every blob's *possibleTracks* list. Again it is tested, if any blob is now associated with 2 cells and therefore has to be added to the *ambiguousBlobs* list.

However, if *foundBlobs* contains no blob, it means that no blob has been found within the specified search range. In that case, the current cell is simply marked as *finished* and removed from *openTracks* (lines 33 to 35).

Algorithm 2 Resolve Ambiguities

```
1: procedure ResolveAmbiguities(trackThreshold, searchRange)
2:     for all curBlob ∈ ambiguousBlobs do
3:         Sort curBlob.possibleTracks by distance to curBlob
4:         if ¬curBlob.possibleTracks[1].finished() ∧
   ¬ambiguousTracks.contains(curBlob.possibleTracks[1]) then
5:             distClosest := distance curBlob.possibleTracks[1] to curBlob.position
6:             distNext := distance curBlob.possibleTracks[2] to curBlob.position
7:             if distNext − distClosest ≥ trackThreshold then
8:                 if ¬curBlob.possibleTracks[1].hasTrackPointAt(t) then
9:                     confidence := 1 − (distNext/searchRange)
10:                    curBlob.possibleTracks[1].addTrackPoint(curBlob.position, confidence)
11:                end if
12:                if distNext − distClosest > 0 then
13:                    confidence2 := 1 − (trackThreshold/(distNext − distClosest))
14:                    trackpoint := curBlob.possibleTracks[1].getTrackPoint(t)
15:                    trackpoint.setConfidence(min(trackpoint.confidence(), confidence2)
16:                end if
17:                curBlob.possibleTracks.removeFirst()
18:            end if
19:        end if
20:        for all curTrack ∈ possibleTracks do
21:            if curTrack has track point at t then
22:                curTrack.removeTrackpoint(t)
23:            end if
24:            openTracks.remove(currentTrack)
25:            Finish currentTrack
26:            if curTrack ∈ ambiguousTracks then
27:                ambiguousTracks.removeTrack(curTrack)
28:            end if
29:        end for
30:    end for
31:    for all curTrack ∈ ambiguousTracks do
32:        if detectCellDivisions ∧ |curTrack.possibleSuccessorBlobs| = 2 then
33:            (child1, child2) := Create child tracks
34:            openTracks.add(child1, child2)
35:        end if
36:        Finish curTrack
37:        openTracks.remove(curTrack)
38:    end for
39: end procedure
```

Now a unique successor blob with the required confidence should have been found for most cells in *openTracks*. But there are probably also blobs that have been assigned to more than one cell (*ambiguousBlobs*) and cells, for which more than one possible successor blob has been found (*ambiguousTracks*). These blobs and cells are dealt with in the *resolveAmbiguities* function (see Algorithm 2).

It starts by investigating all blobs in *ambiguousBlobs* (lines 2 to 30). For each blob, the associated tracks are sorted by the distance of their predicted position to the blob in ascend-

ing order (line 3). Then it is tested if the closest associated track ($curBlob.possibleTracks[1]$) can be chosen according to the threshold $trackThreshold$ (lines 4 to 19). This is the case, if $curBlob.possibleTracks[1]$ is not finished, is not in $ambiguousTracks$ and if for the distance of the closest track ($distClosest$) and the one of the second closest track ($distNext$) to the current blob the following condition is met: $distNext - distClosest \geq trackThreshold$. If that is the case, it is ensured that the cell has a track point at t and a new confidence level is calculated based on how close the difference of $distClosest$ and $distNext$ is to $trackThreshold$: $confidence2 := 1 - (trackThreshold/(distNext - distClosest))$. Since it holds that $trackThreshold \leq distNext - distClosest$, this yields a value between 0 and 1 that is the closer to 1, the bigger $distNext$ is compared to $distClosest$. The confidence level of the track point is then set to the minimum of its initial confidence level and $confidence2$. After that, all tracks remaining in $curBlob.possibleTracks$ are marked as finished and removed from $openTracks$ and from $ambiguousTracks$.

In the next step, the cells in $ambiguousTracks$ are investigated (lines 31 to 38). If $detectCellDivisions$ is enabled, it is tested if exactly two successor blobs were found. In that case, it is assumed that a cell division has occurred. Despite its simplicity, this primitive approach has successfully detected cell divisions in tests. However, usually the two daughter cells are very close together after the division and if their distance is below $blobThreshold$ or $trackThreshold$, this can cause the algorithm to immediately abort tracking of the daughter cells. This problem underlines the fact that maximizing the size of the tress and minimizing the number of errors are competing aims. All tracks from $ambiguousTracks$ are then marked as finished and removed from $openTracks$.

After that, the tracking algorithm creates new trees for all blobs that were classified as *normal* and were not associated with any currently tracked cells. Then, finished tree fragments, i.e. ones with no cells that are contained in $openTracks$, are saved to disk in a binary file format (see appendix B). Finally, the loaded segmentation results are released from memory and the algorithm goes to the next time point. Approaches to improve the tracking algorithm are discussed in section 4.2.

2.2. Implementation

The presented software solution includes a stand-alone executable for segmentation and automated cell tracking and an interface, which has been integrated into TTT, to further analyze the results. Both modules are described in more detail in the following sections.

2.2.1. Autotracking Software

As mentioned before, the program for tracking is available with and without a GUI. Its design is based on a *multi-tier architecture*, which is an approach that is often used to keep software of arbitrary complexity robust and maintainable. In a multi-tier architecture, parts of the software (*packages*) are grouped into a hierarchy of *tiers*, where each one can only access tiers below it.

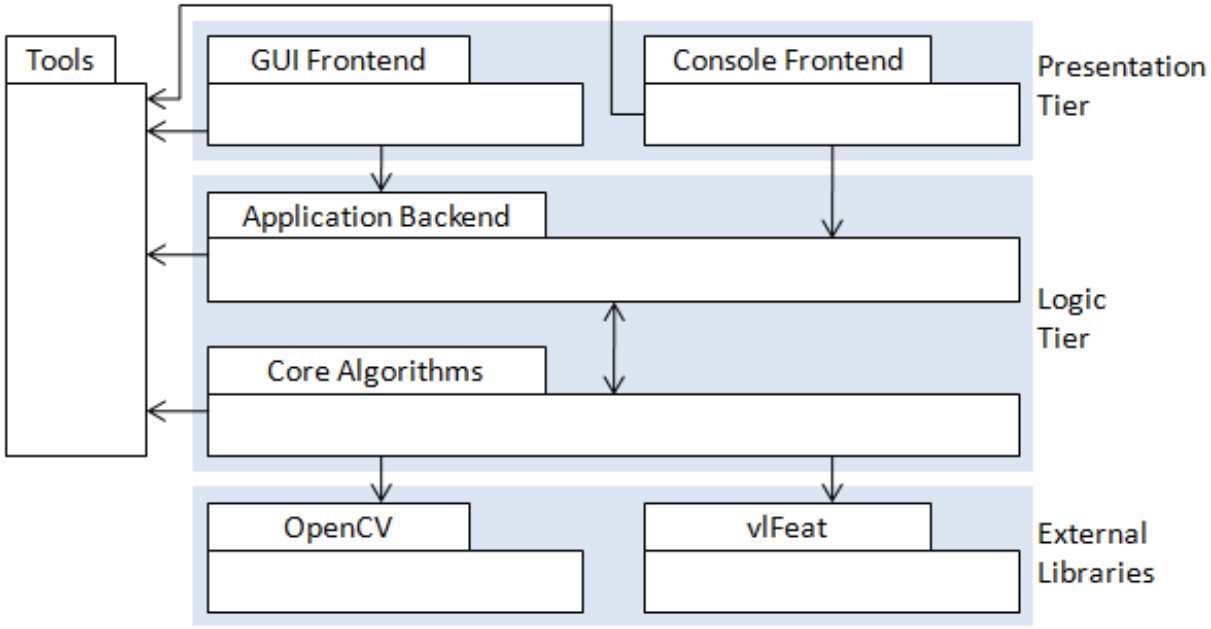

Fig. 9 Overview of the program architecture. As in UML's *package diagrams*, each box represents a *package* and the arrows describe the interactions between them. The program is divided into two *tiers* and the *Tools* package. According to the rules of the used *multi-tier architecture*, the logic tier does not directly access the presentation tier, which contains either the *GUI* or the *Console Frontend* package. Furthermore, various external libraries are used.

Fig. 9 gives an overview of the software design. The *Presentation Tier* contains the *GUI* and the *Console Frontend*. Since the *Logic Tier* below does not directly access any part of the presentation tier, the two modules in there are interchangeable. This has the advantage that the console and the GUI version of the program can be compiled without changing a single line of code. Since the *Application Backend* and the *Core Algorithms* packages access each other, they are not divided into different tiers.

Apart from the two tiers, the software also contains a *Tools* package providing functionality used by all parts of the software, mainly for error handling and settings management, and accesses external libraries. This currently includes *vlFeat* [38] for the MSER implementation, *OpenCV* [39] for Gaussian Blurring and the CLAHE implementation provided by K. Zuiderveld [29]. The whole code also heavily relies on *QT*, which is a C++ library currently developed by Nokia [40]. It is platform independent and does not only provide tools to create powerful GUIs but also a lot of low level functionality e.g. for data-storage, input/output and multi-threading.

Both the segmentation algorithm and the tracking algorithm use multi-threading to reduce the required computation time. As mentioned before, this works very well for the segmentation algorithm as every picture can be processed independently. In order to distribute the required work as efficiently as possible over the available threads, the implementation is based on the *thread pool design pattern*. This means that all images to process are put in a queue (the *job-queue*) and every thread requests one image, processes it and then requests the next image, until the queue is empty. The tracking algorithm uses multiple threads only for finding possible successor blobs for the current cells (i.e.

for algorithm 1) using the *openTracks* list as job-queue (see 2.1.4). This improves performance especially, if many positions have been selected which results in a large number of blobs per time point.

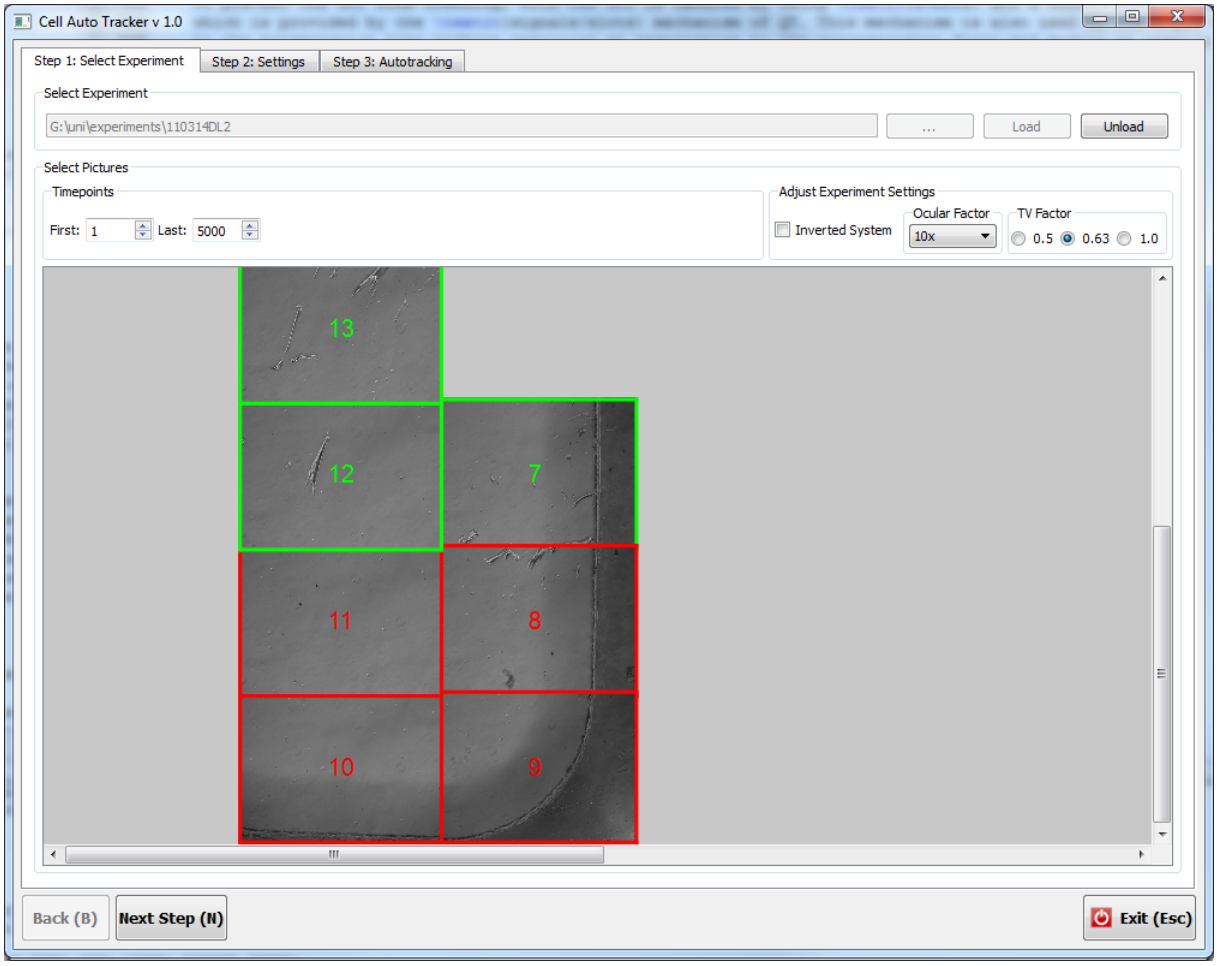

Fig. 10 AutoTracking GUI - experiment selection. As in TTT, the positions are laid out according to their actual sizes and coordinates in the experiment and can be selected or deselected by clicking on them. Furthermore, the time points for tracking and various experiment specific options can be set here.

The corresponding code is part of the Application Backend package. In both cases, access to the job-queue and other data structures was made thread-safe by using *locks*. This is necessary, since write access to the same variable by several threads can lead to *race-conditions*. This means that the result of a calculation depends on the order in which instructions are executed, which can be controlled by using locks. For example, the outcome even of a simple instruction like $i = i + 1$ is undefined if several threads execute it using the same variable i. The communication of the *worker threads*, which run in the background to prevent the GUI from freezing, with the frontend is handled by using *events* and a thread-safe *event-queue*, which is provided by the *signal/slots* mechanism of QT. The same technique is also used by the logic tier to send notifications to the presentation tier without needing a direct association to the frontend at compile time, as required by the multi-tier architecture.

Fig. 10 shows the GUI of the auto tracking program after loading an experiment and selecting the positions 8, 9, 10 and 11. The positions are displayed according to their

actual place in the experiment and can be conveniently selected or deselected by clicking on them with the mouse. In the example, time points 1 to 5000 were specified for tracking. To be as intuitively as possible to use, the GUI has been divided into 3 consecutive steps where this part corresponds to the first step.

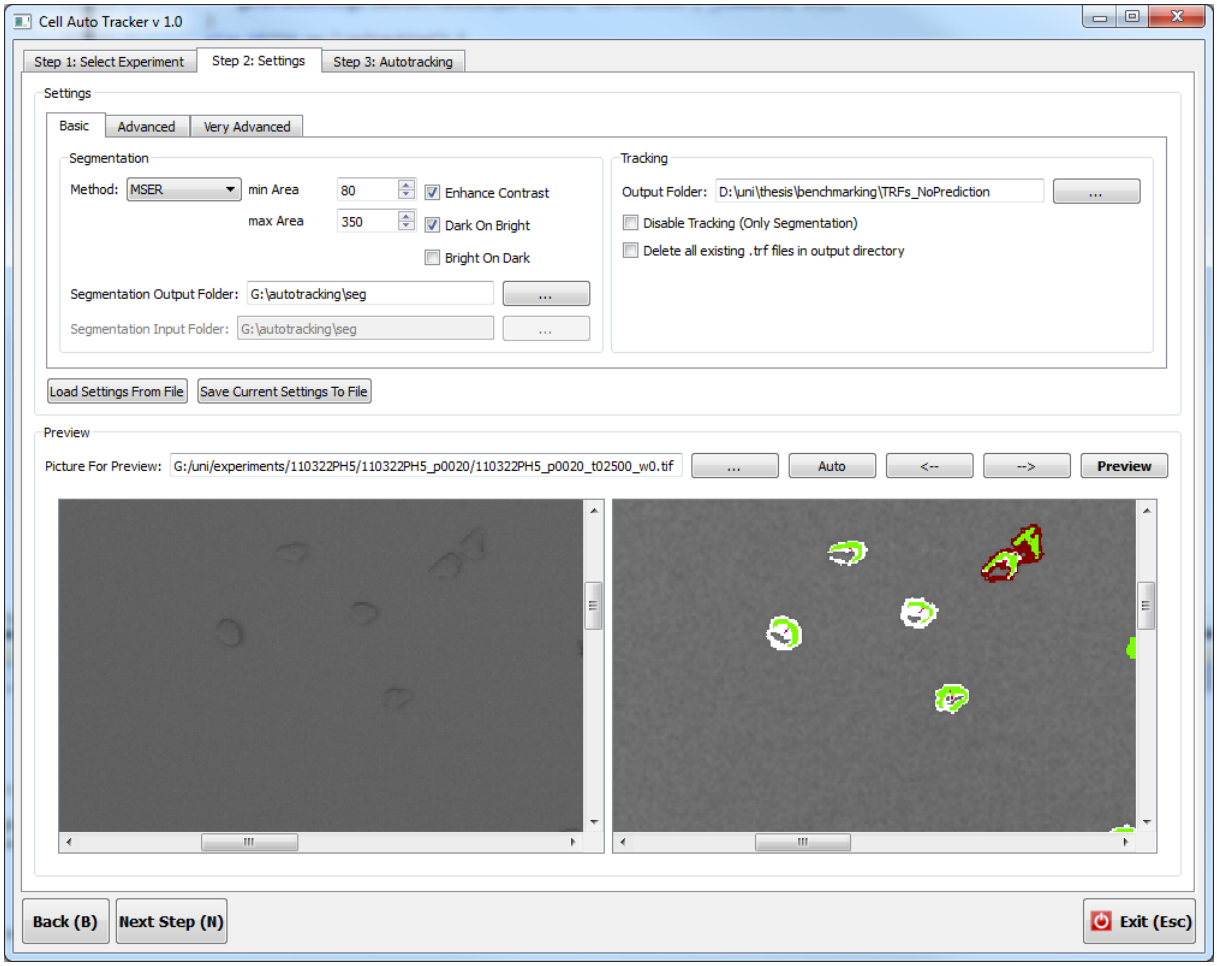

Fig. 11 AutoTracking GUI - segmentation and tracking settings. Here the necessary parameters for segmentation and tracking can be adjusted. In the lower half, the segmentation results with the current settings can be previewed. The same color scheme as in Fig. 8 is used.

Fig. 11 shows the part of the GUI corresponding to the second step. Here, all relevant settings have to specified. To make this easier, the parameters have been divided into the 3 groups *basic*, *advanced* and *very advanced*. Basic settings have to be changed very often for different experiments, but are also very easy to understand without requiring knowledge of the used algorithms. Currently, this group includes the segmentation method, the minimum and maximum cell areas in pixels, if contrast should be enhanced, if the segmentation algorithm should look for *dark on bright* spots, *bright on dark* spots or both to detect cells, and general settings. As segmentation methods, currently only *MSER* and *load from files* to use previously computed results are supported. Here, the current settings can also be exported to a text file or imported from a previously exported file.

The lower half of the image shows the preview feature, which can be used to quickly see how segmentation works with the current settings. It can automatically propose an image for preview, or the user can select any image file from the loaded experiment. There are

also buttons to conveniently go to the picture at the previous or at the next time point of the same position. The left part always shows the original image and the right part shows the image after preprocessing with the segmentation results. The same color scheme as in Fig. 8 is used. In the preview, the user can scroll and zoom conveniently using the mouse as in *Google maps*. Scrolling and zoom are always synchronous in the left and the right part.

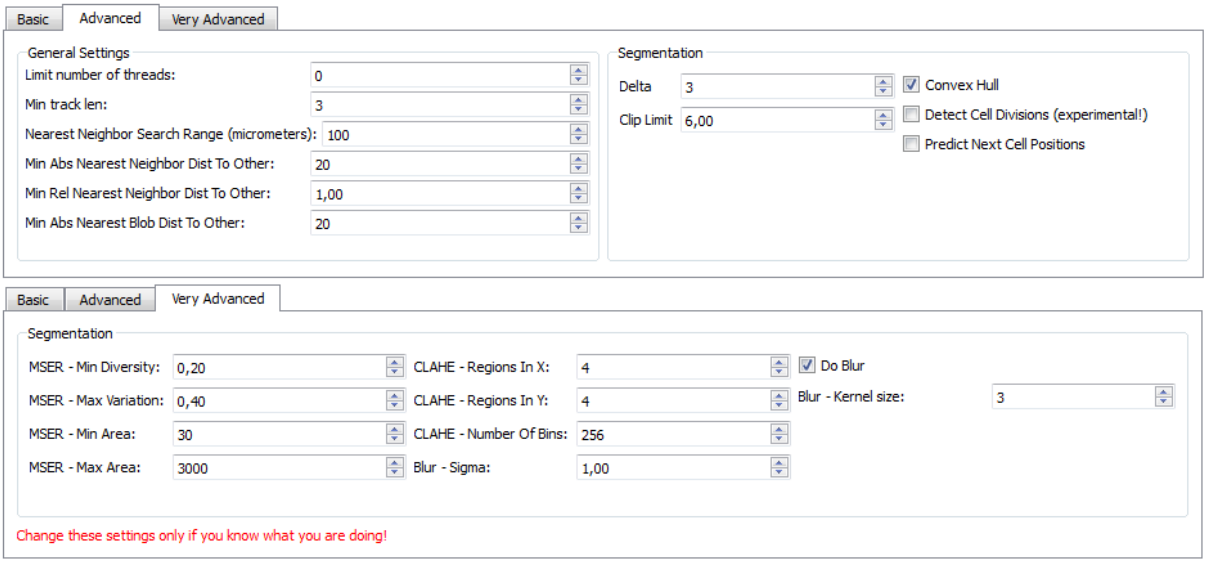

Fig. 12 AutoTracking GUI - *advanced* and *very advanced* settings. They have to be changed less often than the *basic* settings, and understanding them requires more knowledge about the underlying algorithms.

Fig. 12 shows the *advanced* and the *very advanced* settings pages. In the *General Settings* group of the *advanced* settings page, the user can specify an optional limit for the number of worker-threads to use, the minimum track length (all cells with less track points will be filtered out) and the parameters for the tracking algorithm (*searchRange*, *blobThreshold* and *trackThreshold*). If no maximum number of worker-threads was specified, the program attempts to automatically detect the number of available physical CPU cores to determine the optimal number of threads. In the *Segmentation* group, the *delta* parameter of MSER (see 2.1.3) and the *ClipLimit* used by CLAHE (see 2.1.2) can be specified. *Convex Hull* specifies if the convex hull should be used to calculate cell areas (see 2.1.3). Here it can also be set, if it should be attempted to detect cell divisions and if the next position should be predicted for each cell (see 2.1.4).

The settings in the *Very Advanced* group mostly refer to details of the algorithms used for segmentation (see 2.1.3) and have carefully chosen default values. Changing them rarely improves segmentation results and is more likely to cause the affected algorithms not to work properly anymore or other unexpected results (hence the warning at the bottom). The *MinArea* and *MaxArea* parameters for MSER are not used, since this would cause the segmentation algorithm to return no information about blobs with invalid sizes. However, this is very helpful for finding optimal parameters by using the preview feature (compare Fig. 11) and it is also planned to use this information for tracking (see 4.2).

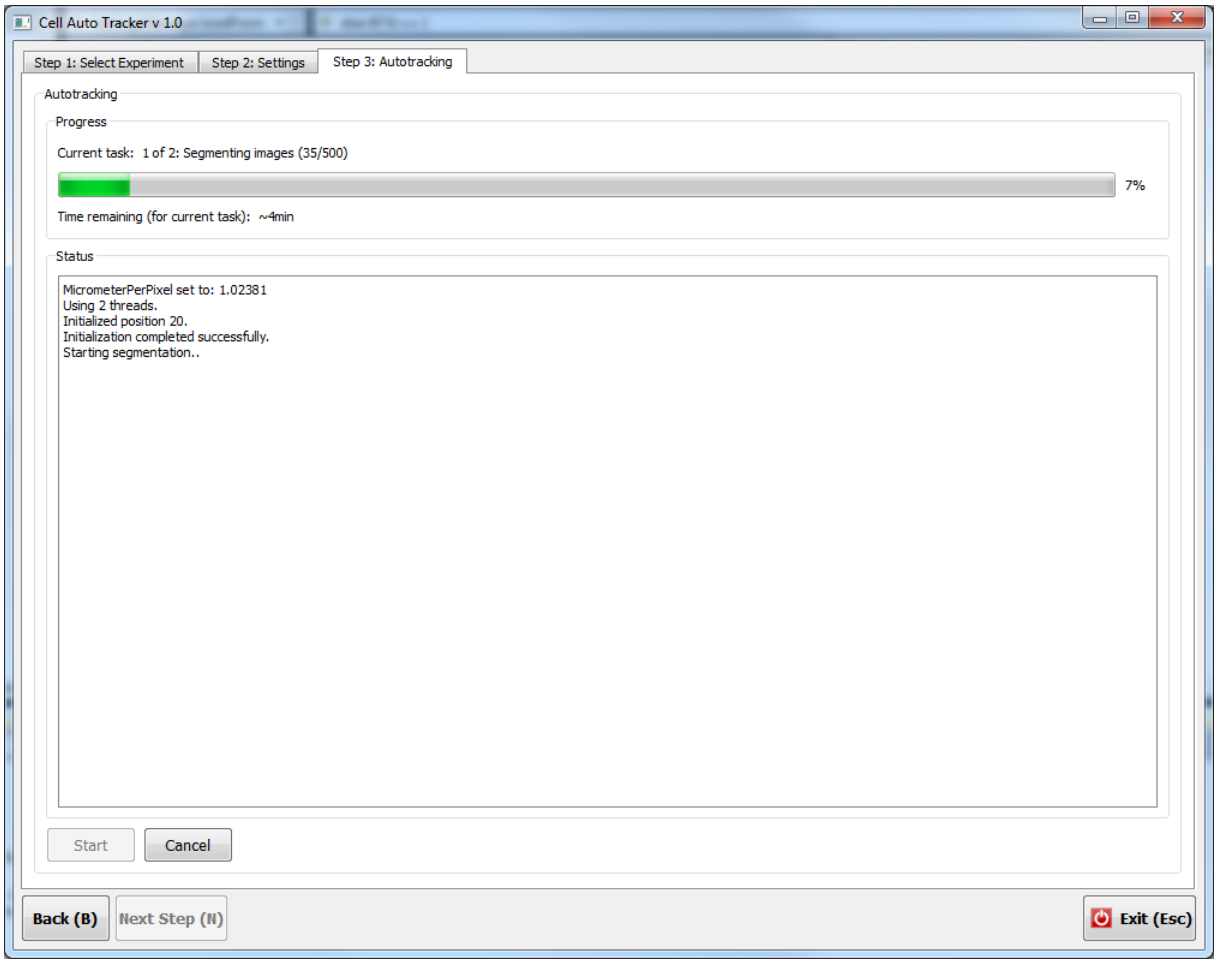

Fig. 13 AutoTracking GUI - progress display. After clicking on the *Start* button, the program starts image segmentation and/or tracking. The overall progress, the estimated remaining time and important status messages or errors are shown here.

Fig. 13 shows the part of the GUI corresponding to the last step. After the user has clicked on the *Start* button, the program begins with image processing and/or automated cell tracking, depending on the specified settings. Based on the time required so far, it is estimated how much remaining time the current task will need to complete. The operation can be canceled at any time.

2.2.2. TTT Interface

In Fig. 14 the *Autotracking - Main Window* of the interface to analyze the tracking results is shown. After specifying the folder with the tracking results, the position overview shows how many tree fragments start in each position. In the example, only position 20 was tracked. On the right, all fragments for the currently selected position are displayed. For each fragment, the starting time point, the unique fragment number, the number of tracks and the number of track points are shown. The rightmost column indicates if the fragment has already been used to create or has been integrated into a regular TTT lineage tree by displaying the file name of the corresponding .ttt file. These fragments can also be hidden from the list by activating the corresponding option below.

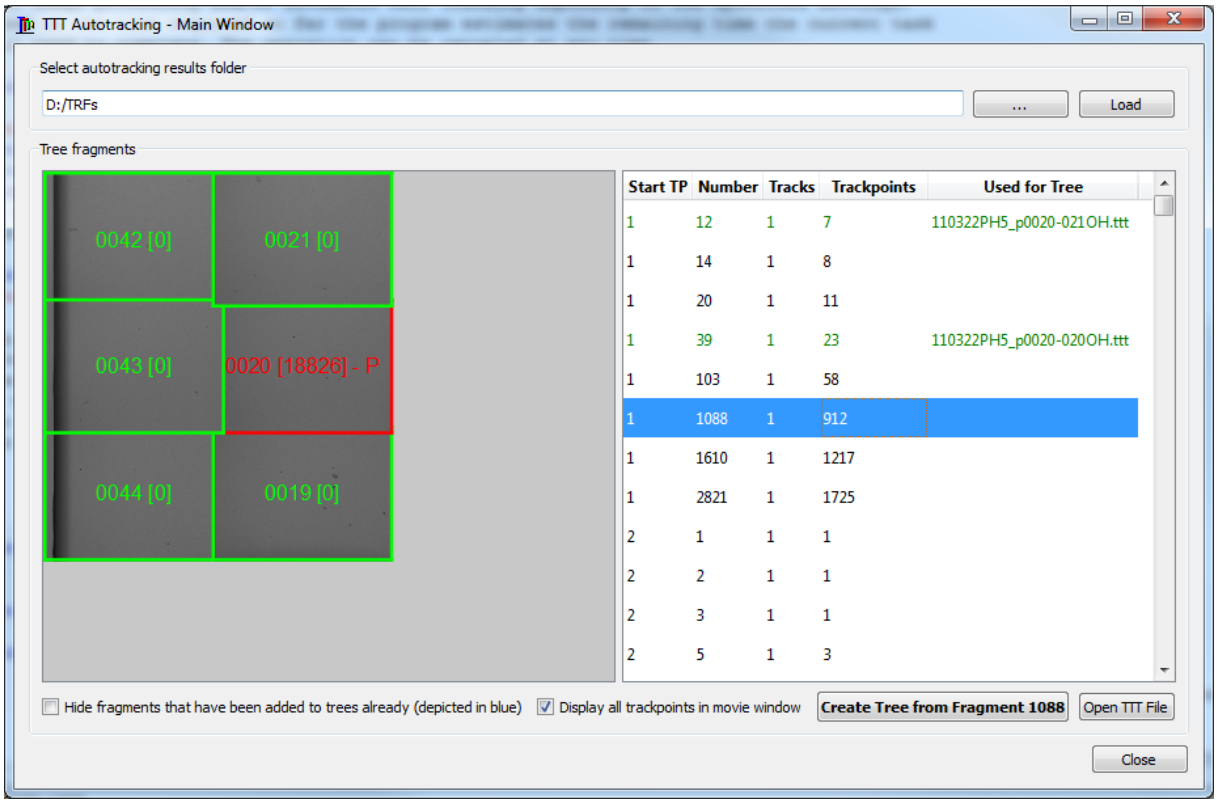

Fig. 14 TTT Interface - *Main Window*. For each position the number of available automatically generated tree fragments is displayed. After selecting a position, all fragments starting in it are shown in the list on the right together with statistical information and if the fragment has already been integrated into a regular tree.

In the regular *Movie Window* of TTT, all track points for the current time point are shown as circles by default (compare Fig. 15). The used color represents the confidence level of the corresponding track point, where green is used for high confidence levels close to 1 and red for low confidence levels close to 0. To be able to quickly get a list of all track points for a given time point, all track points are put into a *hash table* when loading the tree fragments, which uses time points as keys and lists of track points as values. This way, the user can scroll through the experiment fluently while all cells are being displayed.

In the *Autotracking - Main Window* the user can select a fragment and click on the button below or just double-click on it in the list to create a regular TTT tree from it. The tree is then shown in the *Autotracking - Tree Window* (Fig. 16). On the right, the fragments used for the current tree are displayed and the lower part of the window can be used to connect tree fragments manually. After selecting a cell from the tree, the image with the last track point of the cell (or its mother cell if it has no track points yet) is automatically displayed and view is centered on its position. Using the mouse wheel, the user can conveniently go to the previous or next time point, which works fluently since the corresponding images are automatically preloaded in the background. The program also determines which fragments could be the correct successors of the current cell and displays the corresponding cell circles in red. The fragments are also shown in the tree view next to the current tree. The user can then select the next fragment by clicking on the corresponding cell circle. This way, fragments can be connected requiring only minimal user interaction. Cell circles corresponding to fragments that have already been

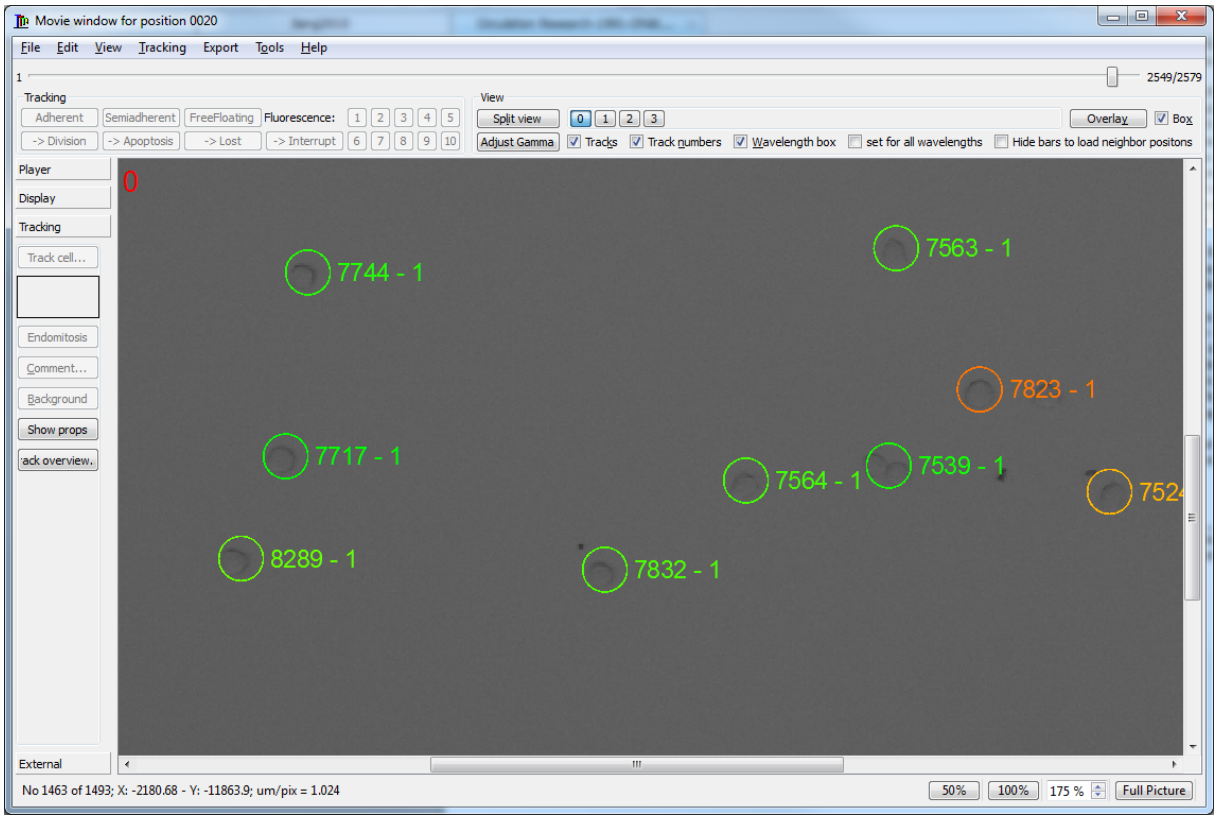

Fig. 15 TTT Interface - track points are displayed in the regular movie window. The color used for the circles corresponds to the confidence level of the displayed track points, where green is used for high and red for low confidence.

used for a tree are displayed in dark green. The buttons next to the tree can be used to close or to save the current tree. When saving the tree, a .log file is created for all used tree fragment files, which contains the file name of the current tree.

The user can also click on the *Track Manually* button to track the selected cell manually in TTT. In that case, the cell circles of possible successor tree fragments are also displayed in the window used for manual tracking and can be selected with one click.

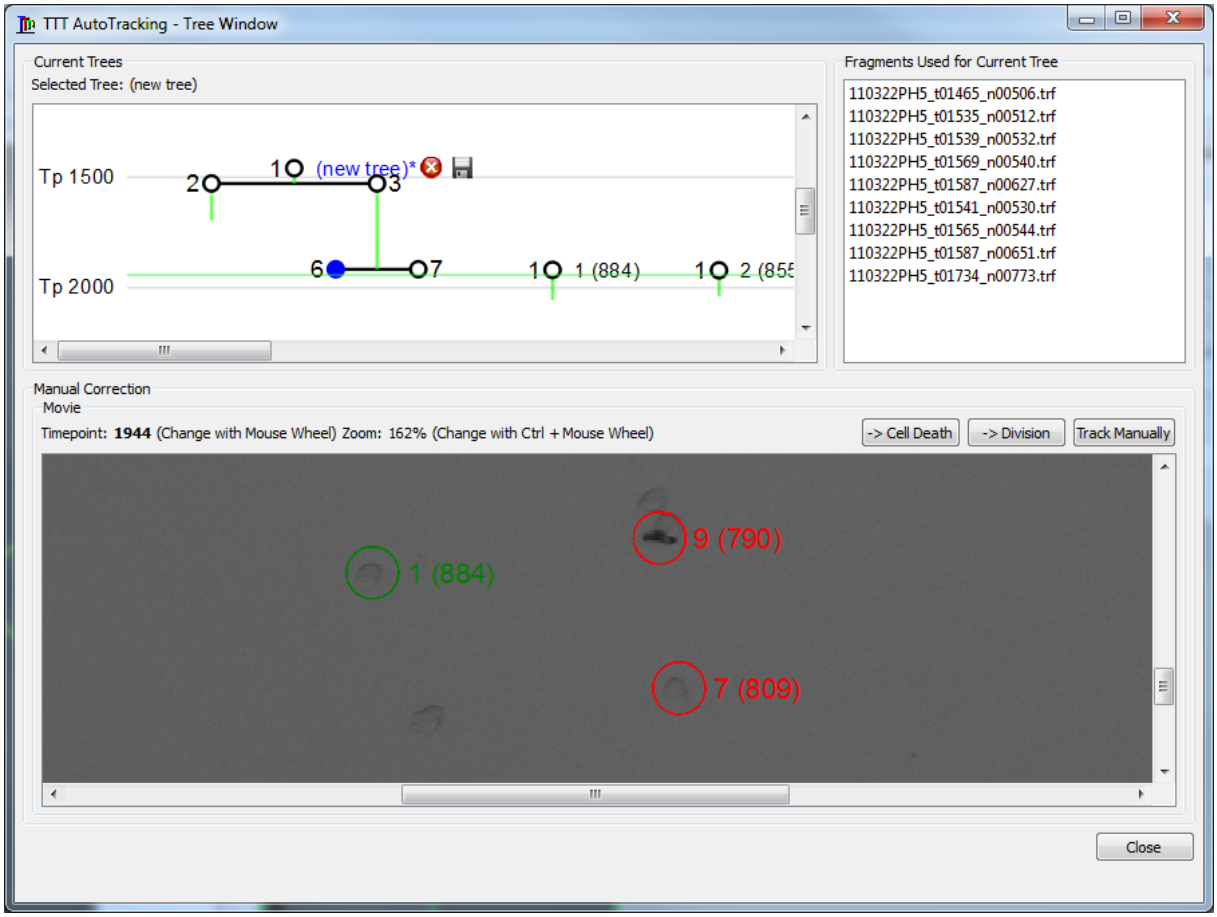

Fig. 16 TTT Interface - *Tree Window*. In this window the generated tree fragments can be used to create regular TTT lineage trees as described in the text.

3. Results

3.1. Benchmarking

3.1.1. Comparison with Manual Tracking

We first want to study, how quality and size of the generated tree fragments are affected by the values specified for the two thresholds *BlobThreshold* and *TrackThreshold* (see section 2.1.4) by comparing automated cell tracking results with data generated by manual cell tracking. Since there are exactly two thresholds, it actually makes sense to use 3D-plots for that purpose. This way it can be directly visualized, how each combination of thresholds affects the quality and size of the generated tree fragments. Since the most interesting kind of error is, when two cells are accidentally mistaken for each other, an algorithm has been developed that attempts to detect such errors by comparing auto tracking with manual tracking results.

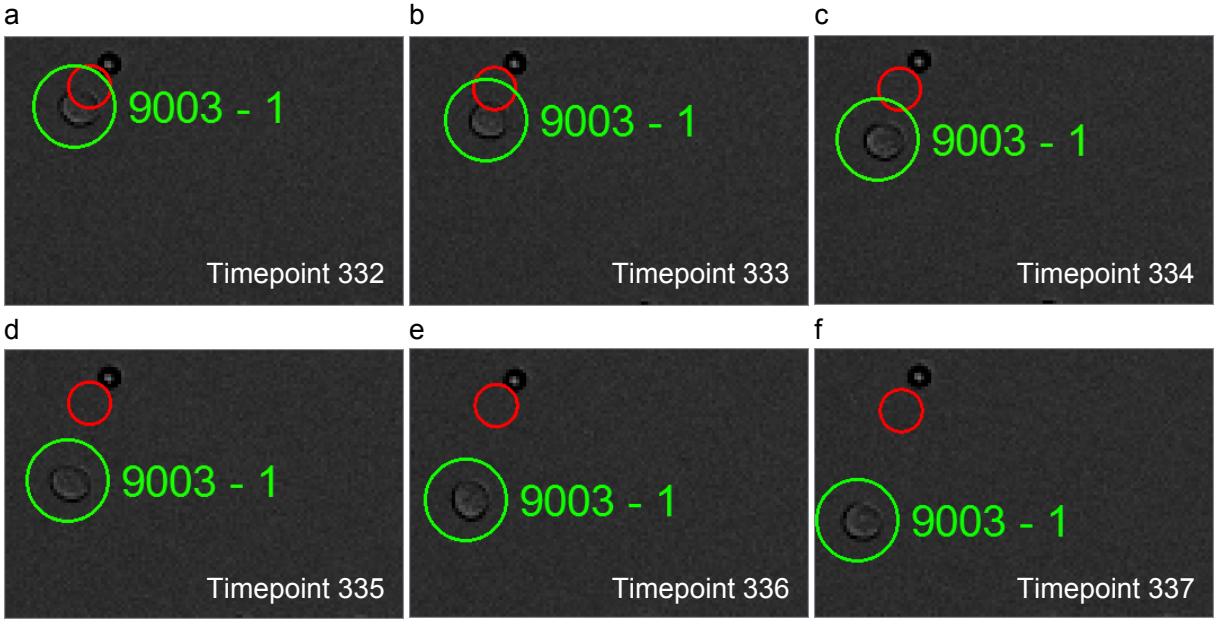

Fig. 17 Example for highly inaccurate manual tracking results in experiment *100104PH43*. a-f) Images of consecutive time points show a moving cell, the corresponding track points of cell 2 of tree *100104PH4_p001-004PH* generated by manual tracking (red circles) and of cell 1 of fragment 9003 generated by automated tracking (green circles). The high accuracy of automatically generated track points will make further analysis using tools like AMT easier and more reliable.

The algorithm works by iterating over all cells of every manually tracked tree and identifying corresponding tree fragments generated by auto tracking. It is then tested, if the distance between the track points generated by manual tracking to the ones generated by auto tracking exceeds a certain threshold. This is what happens mostly if cells were assigned wrongly during auto tracking, since the selected cell and the actual successor cell will usually move into different directions. However, if the two cells stay in close proximity or if the automatically generated trajectories are interrupted immediately after the wrong assignment, errors can also remain undetected by this approach.

For testing, first the experiment *100104PH43* had been chosen, since a lot of manually tracked trees are available for it. However, tests quickly revealed that almost all detected errors were actually caused by inaccurately manually set track points, as demonstrated in Fig. 17. This problem has also been observed by others before [13]. It can be seen clearly that the automatically generated track points are much more accurate than the ones that were set manually. The improved quality of available tracking data will make further computational analysis, like fluorescence quantification with AMT, much easier and more reliable.

For benchmarking, a set of high quality trees was then generated for one position of experiment *110322PH5* by using automated cell tracking combined with manual tracking to connect the generated fragments without gaps. It contains about 20 trees and 20,000 track points, which is slightly more than the set of manually tracked trees of experiment *100104PH43* contains per position.

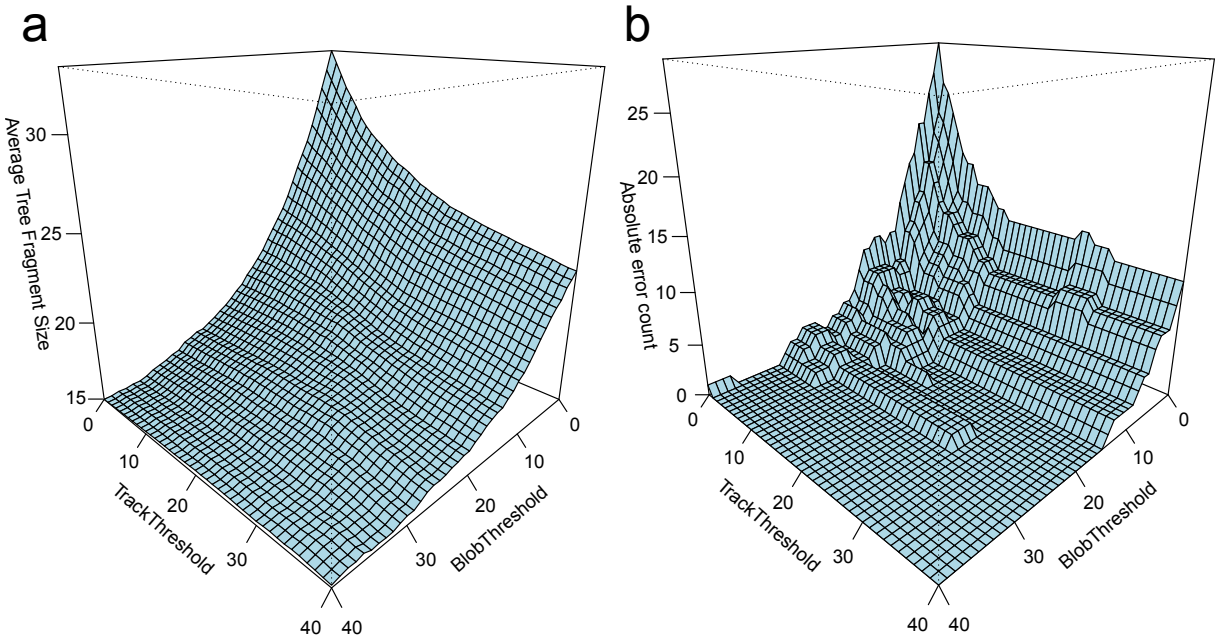

Fig. 18 The 3D plots shows how the chosen thresholds affect the average tree size, which is indicated by the number of track points in a tree, and the number of detected errors. For each threshold combination, a) the average tree size of all generated fragments and b) the absolute number of detected errors are shown.

Fig. 18 shows how the average size (i.e. the average number of track points) and the absolute number of found errors is influenced by the thresholds. To create these plots, position 20 of experiment *110322PH5* was tracked automatically for all 1681 possible combinations of values between 0 and 40 for the two thresholds. The minimum track length was set to 1 to prevent the results from being influenced by filtering out very short tree fragments.

Since processing the binary files with segmentation results is very fast in C++, tracking the complete position with about 3000 pictures took only a few seconds and all combinations could be tested within a few hours using a normal computer. For that purpose, a *Perl* script was written that executes the console version of the auto tracking program for each combination of the thresholds. The results were then analyzed by using C++ and

the plots were created using *R*. An assignment error was assumed, if the distance between a manually and an automatically set track point exceeded 15 µm.

As expected, both the average tree size and the number of found errors decrease if the thresholds are increased. If both are set to 0 (i.e. disabled), the average tree fragment size is ~ 33.1 track points and the number of errors is 29, which corresponds to an error rate of only $\sim 0.14\%$ relative to the number of track points in the manually created set of trees. The number of errors reaches 0 and the average tree size ~ 19.9 track points per tree fragment for the first time if $TrackThreshold$ is set to 14 and $BlobThreshold$ to 15.

Surprisingly, the error count also increased again in a few cases for higher thresholds. Manual inspection of one such error revealed that it was caused by two cells that moved into each other's direction until they eventually clumped together for some time points. If $BlobThreshold$ exceeded a certain value, one of the two cell trajectories was interrupted, before the cells touched each other, but the other one was not. The latter was then assigned to the two clumping cells, which were detected as one blob, causing an assignment error. On the other hand, if $BlobThreshold$ was lower, the first cell trajectory was not interrupted due to that, but tracking of both cells was stopped later because of the $TrackThreshold$ and the error was avoided. After all, it can be said that the aim of avoiding errors has been fulfilled. Nevertheless, most of the generated tree fragments are very small, but it is probably possible to improve that without causing more errors (see section 4 for a discussion of possible algorithmic improvements).

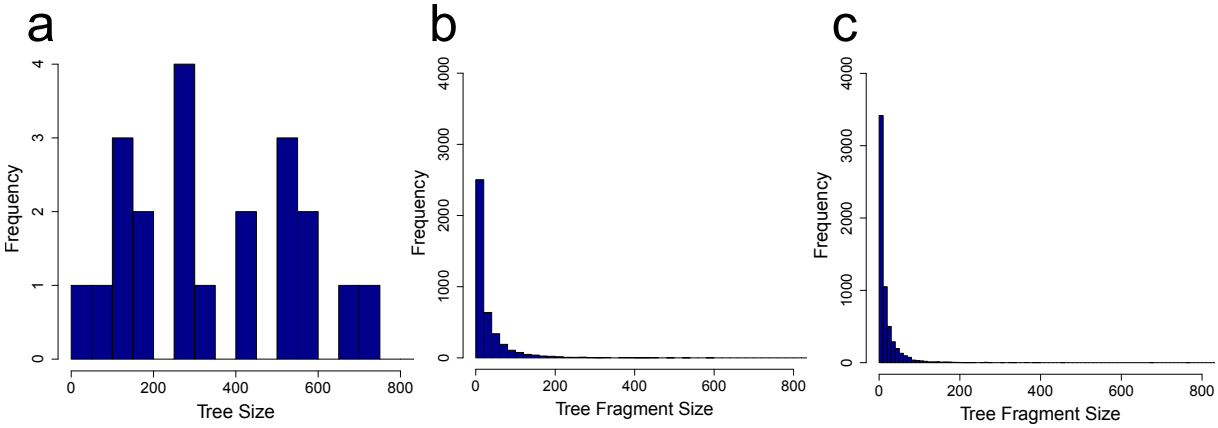

Fig. 19 Histograms of tree size distributions for (a) manual tracking, (b) automated tracking with $BlobThreshold = 0 \wedge TrackThreshold = 0$ and (c) $BlobThreshold = 15 \wedge TrackThreshold = 14$.

Fig. 19 shows the histograms of the sizes of manually and automatically tracked trees using either 0 for both thresholds or the combination of 15 for $BlobThreshold$ and 14 for $TrackThreshold$. It can be seen that the sizes of the manually tracked trees by far exceed those of the automatically generated ones and that higher values for the two thresholds result in the generation of more, smaller tree fragments.

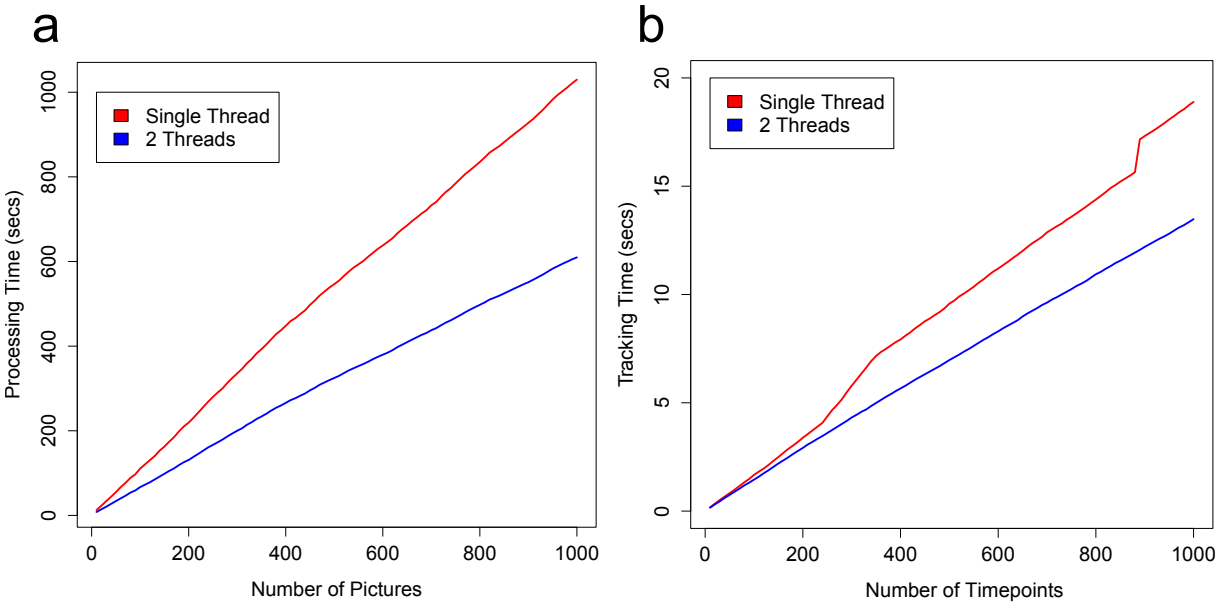

Fig. 20 Required runtime (seconds) for segmentation (a) and tracking (b) of 1 to 1000 images using either one thread only (red) or two threads (blue). Only one position was used for tracking, thus the number of time points in (b) is equal to the number of pictures. For testing, an *Intel Core2 Duo* CPU with 2.66 GHZ was used.

3.1.2. Runtime

The required runtime for segmentation (Fig. 20 a) and tracking (Fig. 20 b) using one or two threads is compared. It can be seen clearly that the image processing part is responsible for most of the required computation time. The plot also shows that segmentation scales very well when using multi-threading. It has been observed that in some cases, however, the used MSER implementation requires much more time for processing an image if several threads are running simultaneously. For that reason, segmentation speed improves notably but is not doubled if two threads are used instead of one. Section 4.1 discusses alternative MSER implementations.

As expected, the tracking algorithm does not scale equally well to a higher number of threads, since only parts of it make use of multi-threading. If two threads are used, however, the increase of required runtime for tracking is much more stable. This is probably caused by the fact that the number of cells in the used images increases and, therefore, the number of tracked cells and blobs that have to be considered per time point also becomes higher. As mentioned before, this increases the effects of multi-threading (see section 2.2.1). For the same reason, the effects of multi-threading would also be much higher if more than one position had been used for tracking.

3.2. Analysis of Cell Movement

To analyze cell movement, time points 1 to 3000 of position 20 of experiment *110322PH5*, which contained HSCs and early multipotent progenitors (MPPs), were used for auto-

mated cell tracking. A minimum track length of 15 was specified and both tracking thresholds were set to 15, which resulted in the generation of 1991 tree fragments.

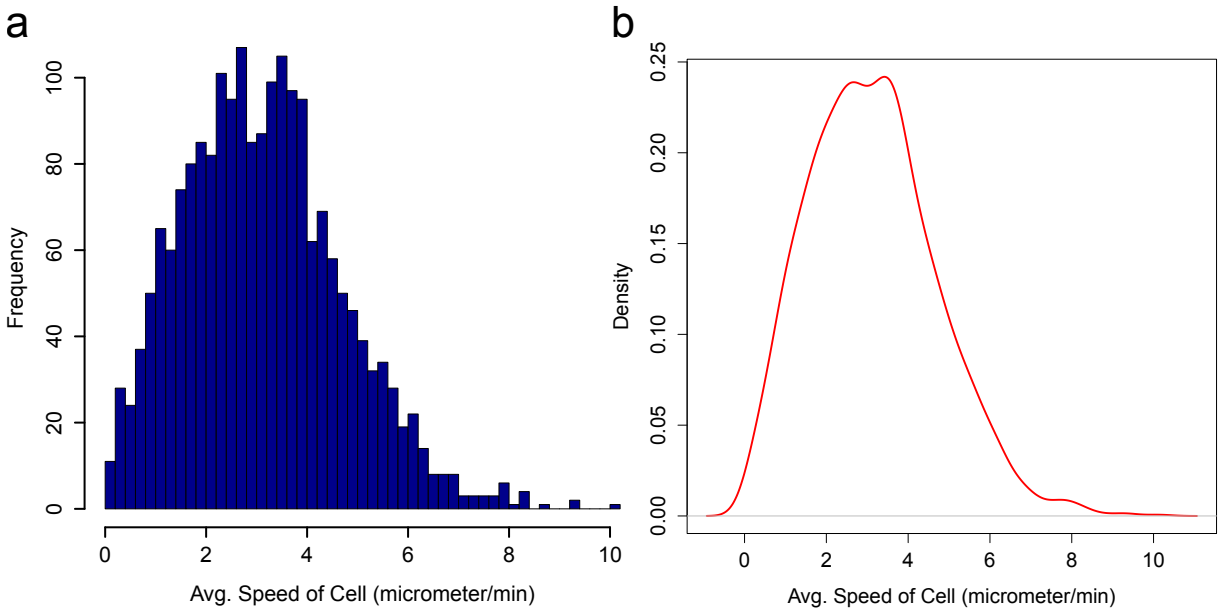

Fig. 21 a) Histogram and b) estimated density function of average cell speed in µm / minute.

The speed of the cells was analyzed first. For that purpose, a histogram of the average cell speed of the trees in µm per minute was generated (Fig. 21 a). It was calculated by measuring the distances of successive track points of all tree fragments and the seconds between the acquisition times of the corresponding images. In addition to the histogram, *kernel density estimation* [41] was used to estimate the probability density function of the random variable underlying the dataset (Fig. 21 b). Although the data was not generated by a random experiment, this makes sense since the resulting plot is very similar to a histogram, but is usually much smoother and does not depend as much on parameters like the *number of bins* for histograms. It can be seen that most cells are moving very slowly with a few exceptions. During manual inspection of the movie, some very fast moving variants of debris were observed, which were also tracked and probably caused these exceptions. The two peaks could be caused by MPPs that differentiated into MEPs and GMPs, since it has been observed that the former move less than the latter [42].

For further analysis, the average speed along the (Fig. 22 a) x- and (Fig. 22 b) y axis of all cells existing at a given time point were plotted. This plot confirms that most cells are moving only very slowly in both directions. Furthermore, it can be seen that in average there is no movement along the x-axis, but a lot downwards along the y-axis. In both directions, average cell speed seem to fluctuate, which could be caused by the movement of the experiment stage. The movement along the y-axis seems to become much stronger after time point 800, but this is probably caused by the very low number of cells before about time point 1000.

To observe cell movement at a higher temporal resolution, the average cell speed was plotted again along both axis (Fig. 23), but this time only for time points in the interval $[2000, 2100]$. The fluctuation of average cell speed can be seen clearly in this plot.

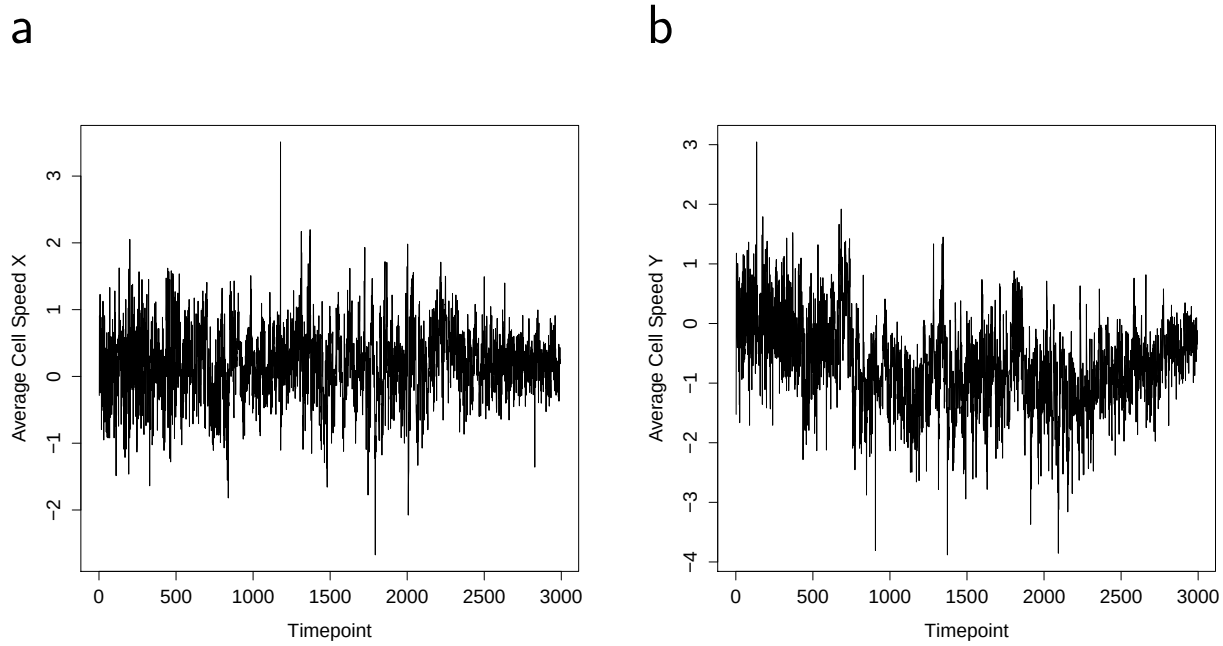

Fig. 22 The average speed in µm / minute of all cells at each time point along the a) x- and the b) y-axis is shown.

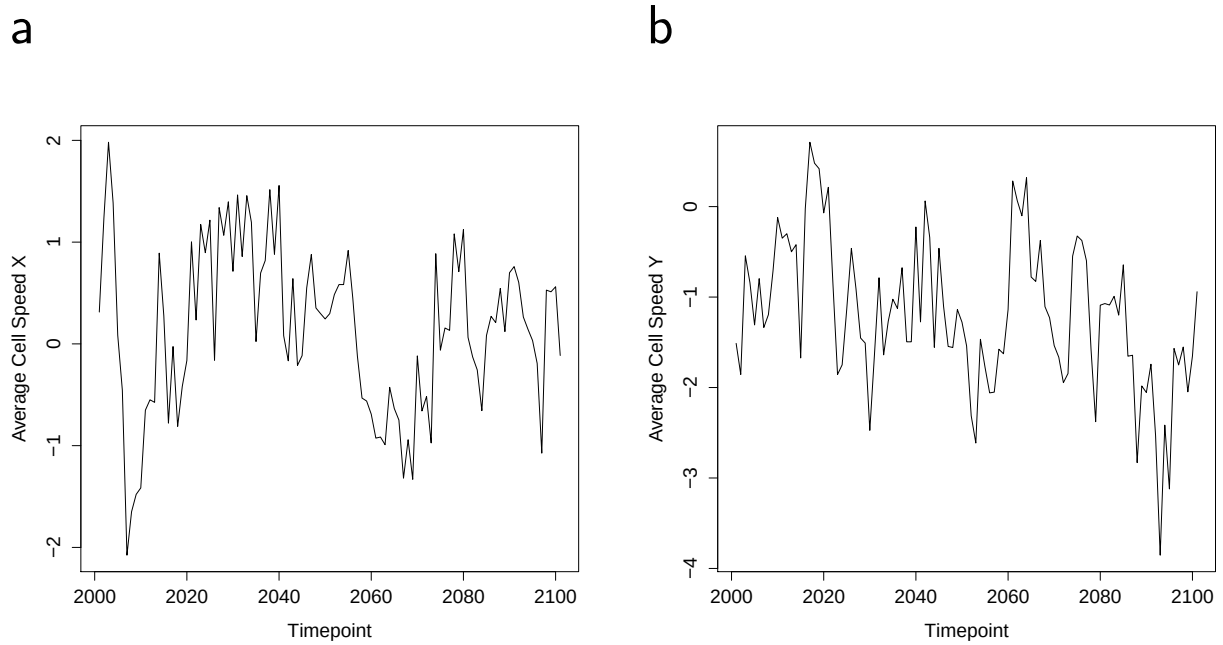

Fig. 23 The average speed in µm / minute of all cells at each time point along the a) x- and the b) y-axis is shown, but only for time points in the interval $[2000, 2100]$.

Finally, in Fig. 24 the average absolute cell speed is shown over time along both axis. Despite the notable differences of the average movement along the axes, this plot shows that the absolute cell speed is quite similar. The low values at early time points are probably also caused by the low number of cells. The decrease after about time point 2300 could also be a result of differentiation into MEPs and GMPs.

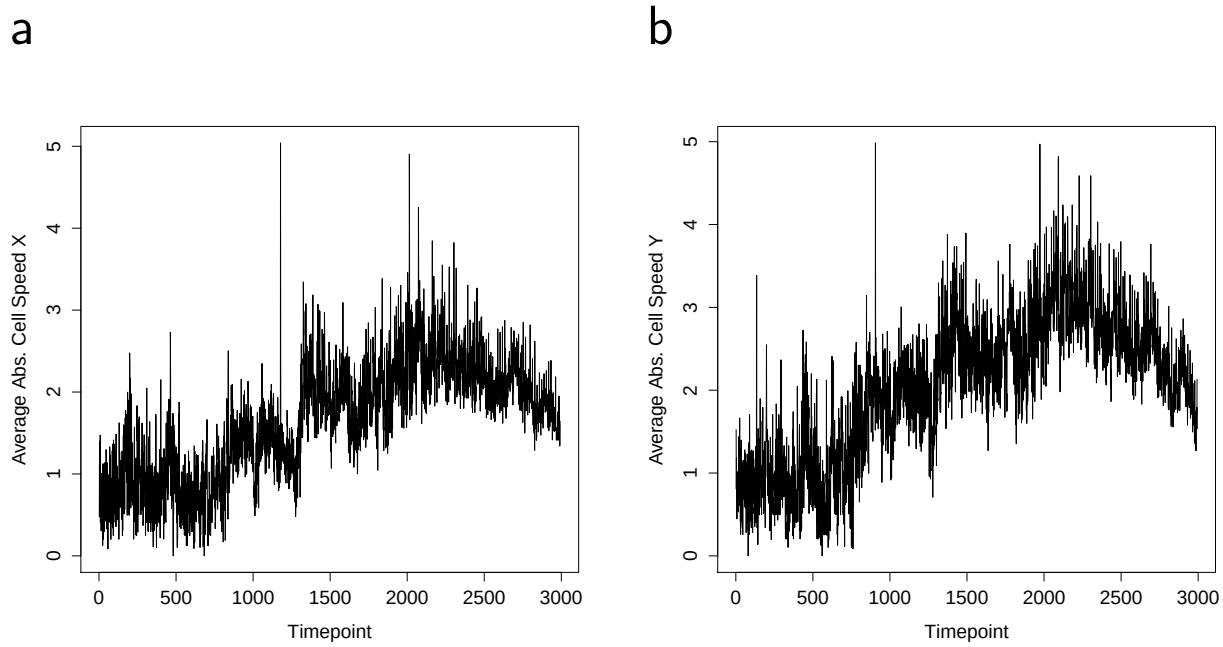

Fig. 24 The average absolute cell speed in µm / minute in a) x and b) y direction is shown over time.

In Fig. 25 the histograms and estimated density functions of the displacement distributions of all cells along the x- and the y-axis are shown. As expected, the distribution of displacement along the y-axis appears to be shifted to the left, which means that cells are actually moving downwards along the y-axis.

To test if the different displacements along the axes observed in Fig. 25 is significant, a *paired-sample t-test* was performed resulting in a significant *p-value* of $8.48 \cdot 10^{-82}$. The strong movement along the y-axis could also be caused by the movement of the experiment stage.

In Fig. 26 the displacement of all cells is shown for their first 100 track points. For this plot, only fragments with at least 100 track points were used, which applied to 143 of the 1991. Again, this plot indicates that cells are moving downwards along the y-axis but are not changing their average positions along the x-axis (Compare Fig. 26 a and b), whereas the absolute speed is very similar along both axes (Compare Fig. 26 c and d).

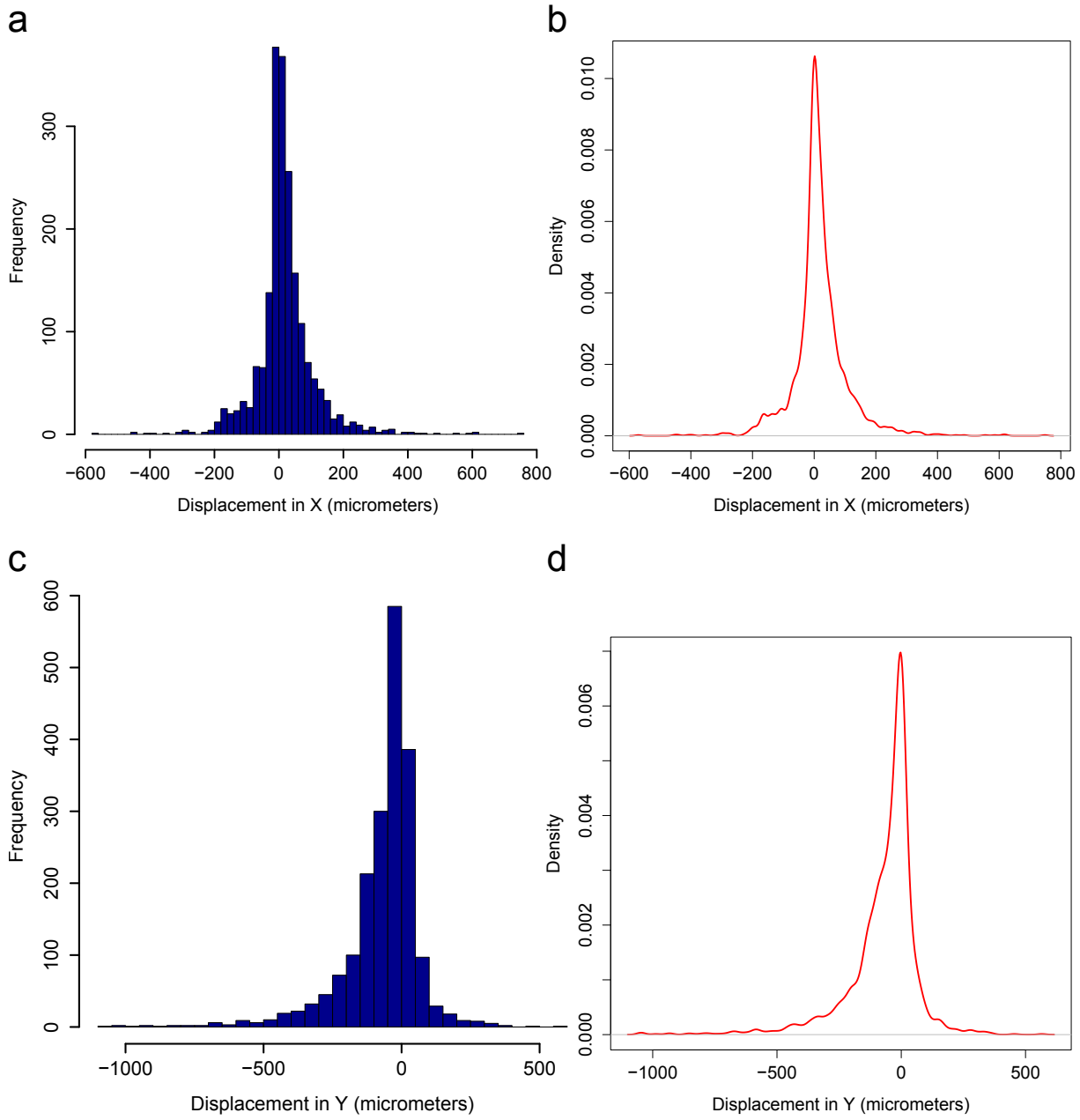

Fig. 25 The displacement distributions of all cells along the a,b) x- and the c,d) y-axis are shown. This plot confirms that the vast majority of cells is only moving along the y-axis.

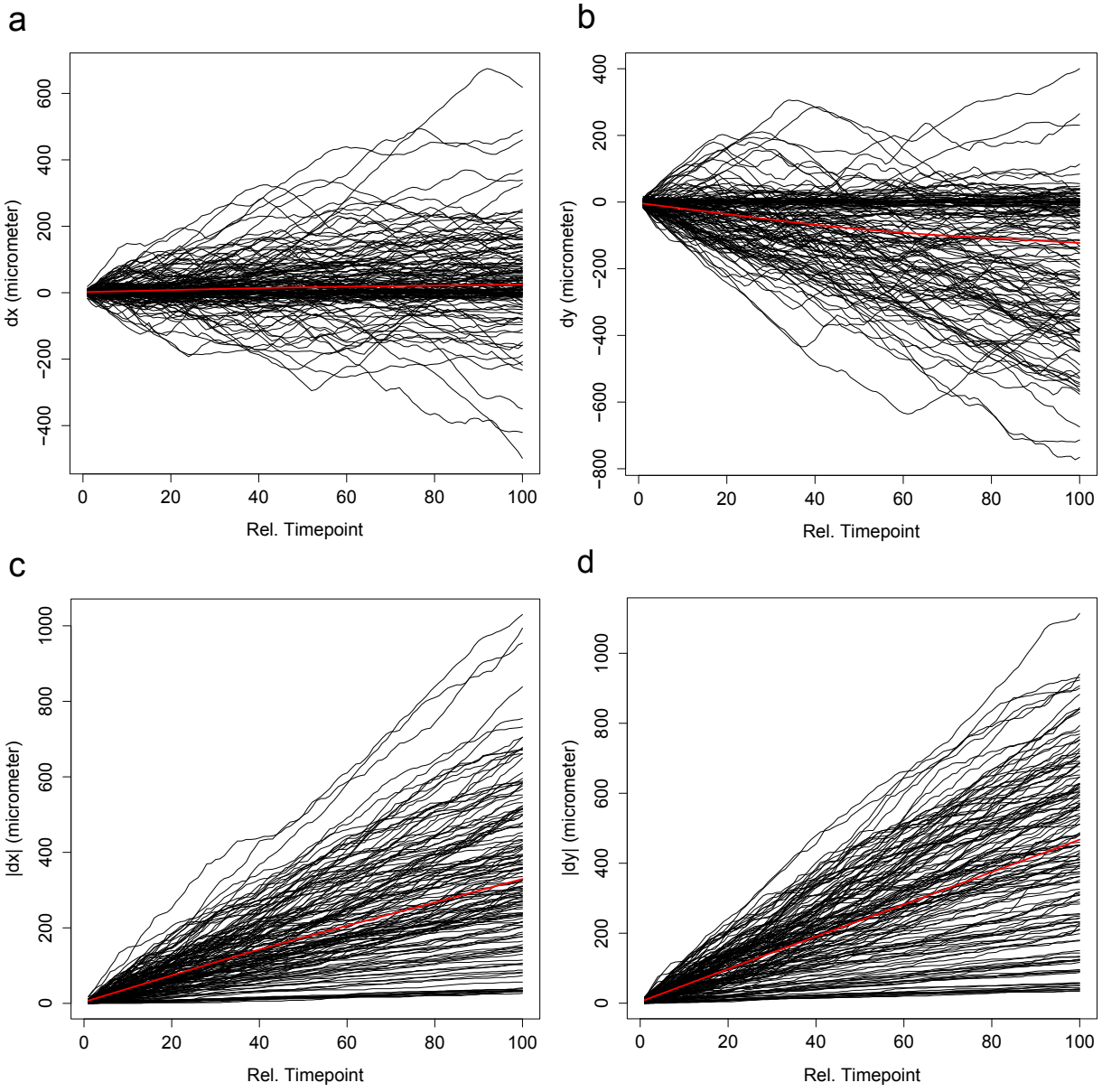

Fig. 26 Displacement depending on time points relative to the first track point of each cell. Only fragments with at least 100 track points were used. A *LOESS* (locally weighted scatterplot smoothing) curve was fitted to the data points (red line). a,b) Displacement in x and y directions. c,d) Absolute displacement in x and y directions.

4. Summary and Outlook

4.1. Improving Segmentation

It has been shown that the used approach for segmentation works very well in most cases. Despite that, there are of course still possibilities for improvement.

Otsu's algorithm, a segmentation method based on thresholding, has already been mentioned before. It is much simpler than MSER and mostly generates much worse results, when used for segmenting cell images. However, if there are only a few cells that are not or almost not moving and if the contrast of cells to background is high for all of them, it can still be sufficient. Since it is much simpler, it also runs much faster and therefore it could make sense to offer segmentation using Otsu's method and related algorithms as an option. Though, it could be difficult to decide beforehand, whether or not Otsu's algorithm is sufficient for a given experiment.

The MSER implementation used in this work has been chosen, because it is widely used and has proven to be able to extract blobs from cell images efficiently and reliably. Nevertheless, since MSER is responsible for most of the required computation time, it makes sense to look for more efficient implementations. An approach that combines detection and tracking of MSERs to reduce runtime has been proposed in 2006 [43]. Combining segmentation and tracking could, however, reduce the positive effects of multi-threading on the required runtime, since it would no longer be possible to process all images of the experiment independently. In 2008, a new method to compute MSERs has been presented that actually runs in $\mathcal{O}(n)$ and has been optimized to use CPU caches more efficiently resulting in notably reduced runtime requirements [44]. OpenCV provides a linear time implementation of the MSER algorithm that has, however, generated unexpected results during tests that were most likely caused by severe unresolved bugs [45].

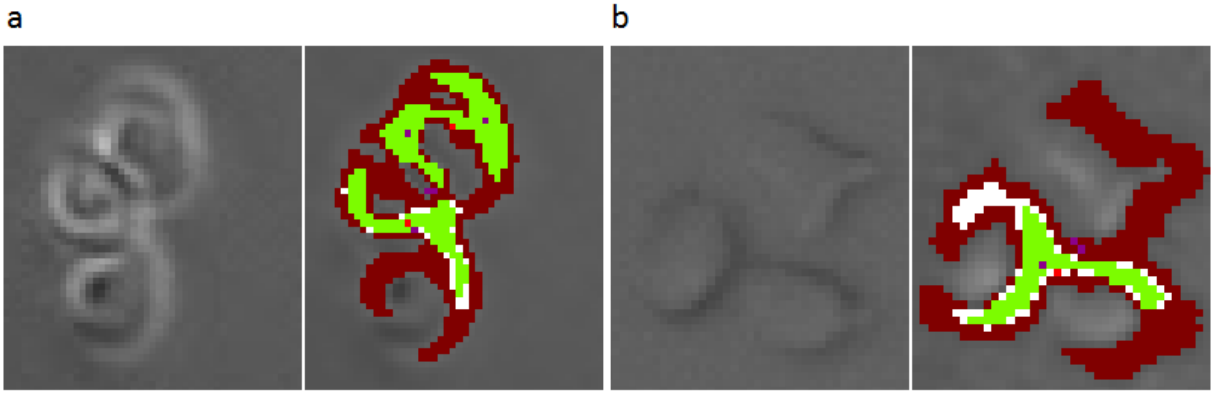

Fig. 27 Segmentation problems caused by clumping cells. a) Of the three cells only the upper one was recognized correctly as distinct cell, whereas the other two were detected as one cell. b) Three cells are detected as one cell. By combining the used *dark on bright* with *bright on dark* search, the cells in this example can be detected correctly, but finding a universal solution to this problem is very difficult.

It has been shown that the segmentation approach based on MSER used in this works can reliably separate cells that are very close together. However, if cells actually stick

together with no space in between them, separation often fails as shown in Fig. 27. In the left example, it would be possible to correctly detect the lower cell by adjusting some parameters, but not without causing *over-segmentation* of other cells (i.e. several cells are detected when there is actually only one). In the right example, it is possible to separate the cells by combining the used *dark on bright* with *bright on dark* search. However, developing a universal solution to this problem that does not only work for one experiment, is very challenging. In both cases, it is probably not possible to automatically separate the cells using the Watershedding algorithm, since the ideas behind this approach and MSERs are very similar. Solving this problem would be especially helpful to be able to better detect cell divisions, since the daughter cells often stick together for about 30 minutes after division.

Therefore, it could make sense to test other segmentation approaches, especially to see how they can handle the problem of clumping cells. Methods based on *active contours*, for example, have been used successfully in this context [46, 47, 48]. The idea of this approach is to position a *contour* in the image in such a way that the associated *energy* is minimized, which typically depends on the gradient in the image next to the contour and on the contour's shape. The higher the gradient is and the smoother the shape, the lower is the associated energy. This way the outlines of smoothly formed objects can be detected. Other successfully used methods that are not described any further here include *wavelet* [49] and *levelset* [50, 51] techniques.

It is already planned to make it possible to use *ilastik* for segmentation, a tool that has been mentioned before. Ilastik is based on *Random Forest Classifiers*, which is a technique that is very different compared to the other approaches that were discussed previously. It has been shown that it can successfully segment cell images, especially if cells differ from background rather by texture than intensity, and it is also quite user-friendly. Therefore, ilastik and MSER could perfectly complement each other as segmentation methods, depending on the properties of a given experiment. A GUI is provided where the user can load an example picture and mark areas in the image that correspond to arbitrary user-defined classes such as background, cell boundary and cell interior. The algorithm then uses this information to segment similar images by deciding for each pixel to which class it belongs.

4.2. Improving the Tracking Algorithm

Despite its simplicity, the tracking algorithm has generated usable results, but there are some aspects that can and need to be improved. For example, cell divisions and other cell fates like apoptosis cannot be detected reliably yet. To improve cell fate prediction, typical features of cells that are about to divide or to undergo apoptosis could be quantified. For example, many blood cells are perfectly round before division and the daughter cells also remain perfectly circular and clump together for about 30 minutes. While undergoing apoptosis, cells often become smaller and completely stop moving.

One widely used technique that can be very helpful for visual tracking problems is provided by the *Kalman filter* [52]. Originally developed mainly for the purpose of handling errors and noise in data measured by sensors, it can also be used to reliably predict the position

of tracked objects based on their last positions. It has already been used successfully for tracking cells in time-lapse microscopy data [53].

Apart from that, it has been shown that tracking results can be improved notably by measuring similarities of cells in successive frames and using that information for assignment. For example, features like position, cell area, intensity and compactness can be stored in a vector for each cell. To measure the similarity of two cells in successive frames, the associated feature vectors can then be compared. Based on statistical properties of previously observed differences between the feature vectors associated with a cell in different frames, this approach can be extended to estimate the likelihood that a blob in the current frame belongs to a given cell [23]. Other approaches focusing on similarities of cells that have been used in this context are based on *image registration* [54], *active contours* [55, 56, 57] and *Markov models* [36]. However, it is important to note that cells can look very similar and change their appearance suddenly from one frame to another. Therefore, relying too much on similarities of cells could also introduce hard to spot errors in some cases.

Other tracking approaches that might be interesting are based on the *mean-shift* algorithm [58]. It can be used, for example, to find local extrema in an image by considering intensities as density function values and iteratively shifting the position of a *kernel* to the average of data points computed in its neighborhood [59]. Mean-shift tracking can be extended [60] and combined with other approaches like the Kalman filter to generate better results [53].

Finally, in many tests it happened a few times that parts of cells were not visible in single frames causing the segmentation algorithm to classify them as being too small. Since blobs are currently neglected by the tracking algorithm if their size is invalid, this mostly leads to the premature ending of cell trajectories or even wrong assignments. Therefore, it is already planned to modify the tracking algorithm in such a way that blobs which are too small can still be accepted in single frames.

Although the presented software can still be improved on the algorithmic level, it will help to eliminate the bottleneck of manual cell tracking by making it possible to generate tree fragments of very high quality with just a few clicks. The higher accuracy of automatically generated track points will make it much easier not only to automatically quantify fluorescence intensities but also to, for example, predict cell fates by investigating their morphological properties [61, 42]. Since tracking data can be generated rapidly now, it will also be possible to work on new biological questions of high relevance such as the influence of cell contact on cell fate decisions, since this requires most cells of the experiment to be tracked.

List of Figures

References

[1] M.A. Rieger and T. Schroeder. Hämatopoetische stammzellen. *Biospektrum*, 13(3):254–256, 2007.

[2] G. Thews, E. Mutschler, and P. Vaupel. *Anatomie, Physiologie, Pathophysiologie des Menschen*. Wissenschaftliche Verlagsgesellschaft, 1999.

[3] H.M. Eilken, S.I. Nishikawa, and T. Schroeder. Continuous single-cell imaging of blood generation from haemogenic endothelium. *Nature*, 457(7231):896–900, 2009.

[4] T. Schroeder. Imaging stem-cell-driven regeneration in mammals. *Nature Reviews*, 453(7193):345, 2008.

[5] G. Karlebach and R. Shamir. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology*, 9(10):770–780, 2008.

[6] J. Krumsiek, C. Marr, T. Schroeder, and F.J. Theis. Hierarchical differentiation of myeloid progenitors is encoded in the transcription factor network. *PLoS ONE*, 6(8), 2011.

[7] D. Iber, J. Clarkson, M.D. Yudkin, and I.D. Campbell. The mechanism of cell differentiation in bacillus subtilis. *Nature*, 441(7091):371–374, 2006.

[8] B. Neumann, T. Walter, J.K. Hériché, et al. Phenotypic profiling of the human genome by time-lapse microscopy reveals cell division genes. *Nature*, 464(7289):721–727, 2010.

[9] J. Huth, M. Buchholz, J.M. Kraus, K. Mølhave, C. Gradinaru, G. Wichert, T.M. Gress, H. Neumann, and H.A. Kestler. Timelapseanalyzer: Multi-target analysis for live-cell imaging and time-lapse microscopy. *Computer Methods and Programs in Biomedicine*, 2011.

[10] I.F. Sbalzarini and P. Koumoutsakos. Feature point tracking and trajectory analysis for video imaging in cell biology. *Journal of Structural Biology*, 151(2):182–195, 2005.

[11] M.R. Lamprecht, D.M. Sabatini, and A.E. Carpenter. Cellprofiler^{TM}: free, versatile software for automated biological image analysis. *Biotechniques*, 42(1):71, 2007.

[12] H. Shen, G. Nelson, S. Kennedy, D. Nelson, J. Johnson, D. Spiller, M.R.H. White, and D.B. Kell. Automatic tracking of biological cells and compartments using particle filters and active contours. *Chemometrics and intelligent laboratory systems*, 82(1-2):276–282, 2006.

[13] M. Arnold, A. Nissen, and F. Scharf. A fully automatic system for cell segmentation and tracking of hematopoietic stem cells on time-lapse microscopy data with integrated analysis toolbox, 2010.

[14] M.A. Rieger, P.S. Hoppe, B.M. Smejkal, A.C. Eitelhuber, and T. Schroeder. Hematopoietic cytokines can instruct lineage choice. *Science*, 325(5937):217, 2009.

[15] M.R. Costa, O. Bucholz, T. Schroeder, and M. Götz. Late origin of glia-restricted progenitors in the developing mouse cerebral cortex. *Cerebral Cortex*, 19(suppl 1):i135, 2009.

[16] B. Dykstra, J. Ramunas, D. Kent, L. McCaffrey, E. Szumsky, L. Kelly, K. Farn, A. Blaylock, C. Eaves, and E. Jervis. High-resolution video monitoring of hematopoietic stem cells cultured in single-cell arrays identifies new features of self-renewal. *Proc National Acad Sciences USA*, 103(21):8185–8190, 2006.

[17] Q. Shen, Y. Wang, J.T. Dimos, C.A. Fasano, T.N. Phoenix, I.R. Lemischka, N.B. Ivanova, S. Stifani, E.E. Morrisey, and S. Temple. The timing of cortical neurogenesis is encoded within lineages of individual progenitor cells. *Nature Neuroscience*, 9(6):743–751, 2006.

[18] R. Ravin, D.J. Hoeppner, D.M. Munno, L. Carmel, J. Sullivan, D.L. Levitt, J.L. Miller, C. Athaide, D.M. Panchision, and R.D.G. McKay. Potency and fate specification in cns stem cell populations in vitro. *Cell Stem Cell*, 3(6):670–680, 2008.

[19] D.G. Lowe. Object recognition from local scale-invariant features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.*, volume 2, pages 1150–1157. Ieee, 1999.

[20] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision and Image Understanding (CVIU)*, 110(3):346–359, 2006.

[21] Z. Kalal, K. Mikolajczyk, and J. Matas. Face-tld: Tracking-learning-detection applied to faces. In *2010 17th IEEE International Conference on Image Processing (ICIP)*, pages 3789–3792. IEEE, 2010.

[22] R.M. Jiang, D. Crookes, N. Luo, and M.W. Davidson. Live-cell tracking using sift features in dic microscopic videos. *IEEE Transactions on Biomedical Engineering*, 57(9):2219–2228, 2010.

[23] O. Al-Kofahi, R.J. Radke, S.K. Goderie, Q. Shen, S. Temple, and B. Roysam. Automated cell lineage construction. *Cell Cycle*, 5(3):327–335, 2006.

[24] Y. Shinyama. Fifth avenue. `http://commons.wikimedia.org/wiki/File:Fifth_Avenue.jpg`, 2002. [Online; accessed 31-August-2011].

[25] C. Sommer, C. Straehle, U. Koethe, and F.A. Hamprecht. "ilastik: Interactive learning and segmentation toolkit". In *8th IEEE International Symposium on Biomedical Imaging (ISBI 2011)*, 2011.

[26] M. Schwarzfischer. Single-cell analysis of multipotent hematopoietic progenitor cells. Master's thesis, Ludwig-Maximilians-Universität Technische Universität München, 2009.

[27] J. Krumsiek. Computational modeling of regulatory networks in hematopoietic differentiation. Master's thesis, Ludwig-Maximilians-Universität Technische Universität München, 2009.

[28] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language*. University Video Communications, 1996.

[29] K. Zuiderveld. Contrast limited adaptive histogram equalization. In *Graphics gems IV*, pages 474–485. Academic Press Professional, Inc., 1994.

[30] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, 2004.

[31] N. Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11:285–296, 1975.

[32] X. Chen, X. Zhou, and S.T.C. Wong. Automated segmentation, classification, and tracking of cancer cell nuclei in time-lapse microscopy. *IEEE Transactions on Biomedical Engineering*, 53(4):762–766, 2006.

[33] M. Donoser, C. Arth, and H. Bischof. Detecting, tracking and recognizing license plates. *Computer Vision–ACCV 2007*, pages 447–456, 2007.

[34] H. Ramoser, V. Laurain, H. Bischof, and R. Ecker. Leukocyte segmentation and classification in blood-smear images. In *Engineering in Medicine and Biology 27th Annual Conference*, pages 3371–3374. IEEE, 2005.

[35] L. Vincent and P. Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 583–598, 1991.

[36] X. Zhou, F. Li, J. Yan, and S.T.C. Wong. A novel cell segmentation method and cell phase identification using markov model. *IEEE Transactions on Information Technology in Biomedicine*, 13(2):152–157, 2009.

[37] O. Debeir, D. Milojevic, T. Leloup, P. Van Ham, R. Kiss, and C. Decaestecker. Mitotic tree construction by computer in vitro cell tracking: a tool for proliferation and motility features extraction. In *Proceedings of EUROCON 2005: Computer as a Tool*, volume 2, pages 951–954. IEEE, 2005.

[38] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. `http://www.vlfeat.org/`, 2008.

[39] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[40] Nokia. Qt - cross-platform application and ui framework. `http://qt.nokia.com/`, 2011.

[41] E. Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[42] F. Buggenthin. Computational prediction of hematopoietic cell fates using single cell time lapse imaging. Master's thesis, Ludwig-Maximilians-Universität Technische Universität München, 2011.

[43] M. Donoser and H. Bischof. Efficient Maximally Stable Extremal Region (MSER) Tracking. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1 (CVPR'06)*, volume 1, pages 553–560. IEEE, 2006.

[44] D. Nistér and H. Stewénius. Linear time maximally stable extremal regions. *Computer Vision–ECCV 2008*, pages 183–196, 2008.

[45] I. Lysenkov. Mser detector is missing keypoints on binary images. `https://code.ros.org/trac/opencv/ticket/756`, 2010. [Online; accessed 31-August-2011].

[46] F. Leymarie and M.D. Levine. Tracking deformable objects in the plane using an active contour model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 617–634, 1993.

[47] C. Zimmer, E. Labruyère, V. Meas-Yedid, N. Guillén, and J.C. Olivo-Marin. Segmentation and tracking of migrating cells in videomicroscopy with parametric active contours: a tool for cell-based drug testing. *IEEE Transactions on Medical Imaging*, 21(10):1212–1221, 2002.

[48] O. Debeir, I. Camby, R. Kiss, P. Van Ham, and C. Decaestecker. A model-based approach for automated in vitro cell tracking and chemotaxis analyses. *Cytometry Part A*, 60(1):29–40, 2004.

[49] D. Padfield, J. Rittscher, and B. Roysam. Spatio-temporal cell segmentation and tracking for automated screening. In *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008.*, pages 376–379. IEEE, 2008.

[50] T.F. Chan and L.A. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2):266–277, 2001.

[51] J. Degerman, T. Thorlin, J. Faijerson, K. Althoff, P.S. Eriksson, RVD Put, and T. Gustavsson. An automatic system for in vitro cell migration studies. *Journal of microscopy*, 233(1):178–191, 2009.

[52] R.E. Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[53] X. Yang, H. Li, and X. Zhou. Nuclei segmentation using marker-controlled watershed, tracking using mean-shift, and kalman filter in time-lapse microscopy. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(11):2405–2414, 2006.

[54] A.J. Hand, T. Sun, D.C. Barber, D.R. Hose, and S. MacNeil. Automated tracking of migrating cells in phase-contrast video microscopy sequences using image registration. *Journal of microscopy*, 234(1):62–79, 2009.

[55] J. Degerman, J. Faijerson, K. Althoff, T. Thorlin, JH Rodriguez, and T. Gustavsson. A comparative study between level set and watershed image segmentation for tracking stem cells in time-lapse microscopy. *Microscopic Image Analysis with Applications in Biology (MIAAB 06)*, 2006.

[56] K. Li, E.D. Miller, L.E. Weiss, P.G. Campbell, and T. Kanade. Online tracking of migrating and proliferating cells imaged with phase-contrast microscopy. *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*, 2006.

[57] K. Li, E.D. Miller, M. Chen, T. Kanade, L.E. Weiss, and P.G. Campbell. Cell population tracking and lineage construction with spatiotemporal context. *Medical Image Analysis*, 12(5):546–566, 2008.

[58] Y. Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):790–799, 1995.

[59] O. Debeir, P. Van Ham, R. Kiss, and C. Decaestecker. Tracking of migrating cells under phase-contrast video microscopy with combined mean-shift processes. *IEEE Transactions on Medical Imaging*, 24(6):697–711, 2005.

[60] S. Avidan. Ensemble tracking. *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

[61] A.R. Cohen, F.L.A.F. Gomes, B. Roysam, and M. Cayouette. Computational prediction of neural progenitor cell fates. *Nature Methods*, 7(3):213–218, 2010.

A. Segmentation Results File Format

```
INTRODUCTION
============
- FileVersion:      01
- FileExtension:    .seg
- Date:             08/01/2011

- Binary file for segmentation results for one picture
- All values are stored in Little-Endian format
- Floating  point numbers are saved in the IEEE 754 format with 32 bit precision
- All coordinates are image local pixel indexes

FORMAT SPECIFICATION
====================
- 4 bytes: signature (magic number) to identify a valid segmentation data file:
           0x73 0x65 0x67 0x64 ("segd" in ASCII)
- 4 bytes, unsigned integer: Fileversion
- 4 bytes, unsigned integer: Number of blobs
- for each blob:
    - 4 bytes, unsigned integer: blob identifier, unique for this picture
    - 2 bytes, unsigned integer: centroid coordinate X
    - 2 bytes, unsigned integer: centroid coordinate Y
    - 2 bytes, unsigned integer: number of pixels in this blob
    - 4 bytes, float: area of convex hull or 0 if not set
    - 4 bytes, unsigned integer: blob status
    - 2 bytes, unsigned integer: children of this blob (blobs that are
      inside this blob)
    - 2 bytes, unsigned integer: children of this blob with
      status = NORMAL
    - for each pixel of blob (ordered in ascending order by Y, X):
        - 2 bytes, unsigned integer: coordinate X
        - 2 bytes, unsigned integer: coordinate Y

REMARKS
=======
- Blob status flags: 0 (Normal) or other combination of BLOB_STATUS flags
    enum BLOB_STATUS_FLAGS {
        NORMAL      = 0x00,
        CUTOFF      = 0x01,     // Is cut off, i.e. not completely in picture
        TOO_SMALL   = 0x02,     // Is too small
        TOO_BIG     = 0x04      // Is too big
    };
```

B. Tracking Results File Format

```
INTRODUCTION
============
- FileVersion:      02
- FileExtension:    .trf
- Date:             08/01/2011

- Binary file for auto tracking results containing one tree fragment
- All values are stored in Little-Endian format
- Floating  point numbers are saved in the IEEE 754 format with 32 bit precision
- All coordinates are saved in experiment global micrometer values

CHANGES TO VERSION 01
=====================
- name of tree for which fragment was used is no longer saved in .trf file
  but in a separate file with extension '.log'

FORMAT SPECIFICATION
====================
- 4 bytes: signature (magic number) to identify a valid autotracking tree
           file: 0x74 0x72 0x66 0x72 ("trfr" in ASCII)
- 4 bytes, unsigned integer: Fileversion
- 4 bytes, unsigned integer: Number of tracks in this tree
- for each track of the tree (sorted in ascending order by tracknumber):
    - 8 bytes, unsigned integer: track number
    - 4 bytes, unsigned integer: starting time point
    - 4 bytes, unsigned integer: stopping time point
    - 4 bytes, unsigned integer: number of track points in this track
    - for each track point of current track (sorted in ascending
      order by time point):
        - 4 bytes, unsigned integer: time point of this track point
        - 2 bytes, unsigned integer: position number of this track point
        - 4 bytes, float: X coordinate
        - 4 bytes, float: Y coordinate
        - 4 bytes, float: confidence level between 0 and 1


.log FORMAT
===========
- Latin1 encoded string with name of associated ttt file (without path)
```