
pyTFM

Release 1.1

Apr 14, 2022

Contents:

1	Installation	3
1.1	Dependencies	4
2	Introduction to Traction Force Microscopy and Monolayer Stress Microscopy	5
3	Tutorials	7
3.1	Analyzing Cell Colonies with Clickpoints	7
3.2	Using pyTFM in Python	23
4	Typical Measures for Force Generation and Stresses in Cells	29
4.1	Deformations in the Substrate	29
4.2	Strain Energy	29
4.3	Contractility	29
4.4	Average Normal and Shear Stress	30
4.5	Distribution of Stresses	31
4.6	Forces acting across Cell Boundaries	31
5	Using Config Files, Hidden Parameters and Plotting Behavior	33
5.1	Hidden Parameters	33
5.2	Using Config Files	33
5.3	Overview of Analysis Parameters	35
5.4	Overview of Plotting Parameters	36
6	Note	39

pyTFM is a python package that allows you to analyze force generation and stresses in cells, cell colonies and confluent cell layers growing on a 2 dimensional surface. This package implements the procedures of [Traction Force Microscopy](#) and [Monolayer Stress Microscopy](#). In addition to the standard measures for stress and force generation, it also includes the line tension, a measure for the force transfer exclusively across cell-cell boundaries. pyTFM includes an addon for the image annotation tool [clickpoints](#) allowing you to quickly analyze and visualize large datasets.

CHAPTER 1

Installation

It is recommended to use this package with the [Anaconda Distribution](#).

pyTFM can be installed in several ways. The most straight forward way is to use pip:

```
pip install pyTFM
```

Pip is included in the Anaconda Distribution and should be available as a command line program out of the box.

Alternative the very newest version is available at github: Download the package from [github](#), unzip the files, open a terminal and navigate into the “pyTFM-master”. Depending on how you unzip there you might need to go to “pyTFM-master/pyTFM-master”. In this folder you should see a file called “setup.py”. Next, install the package with pip:

```
pip install -e .
```

The -e setting will enable you to edit package files, and the “.” signifies that the “setup.py” file is located in the folder that you are currently in.

You can also download and install the package from github in one step using the command line program “git”. If you use the Anaconda Distribution, “git” can be installed with the command

```
conda install git
```

Now you can install pyTFM directly with the command

```
pip install git+https://github.com/fabrylab/pyTFM.git
```

This will download the script file into your Anaconda subdirectory, for example to “anaconda3/lib/python3.7/site-packages/pyTFM” This package includes an addon for the image display and annotation tool clickpoints. Clickpoints is installed automatically if you follow the steps outline below. However, if you want to access clickpoints via the “open with” program option for image files, you have to use the command:

```
clickpoints register
```

in the terminal.

Hint: Execute the command “clickpoints register” from the terminal to add clickpoints to the “open width” menu for image files. You can find detailed information for the usage of clickpoints [here](#).

1.1 Dependencies

The following packages will be installed automatically if necessary: numpy, cython, scipy, scikit-image, matplotlib, tqdm, [solidspy](#), clickpoints with a version higher than 1.9.0, [OpenPIV](#) version 0.20.8. Note that Clickpoints versions below 1.9.0 will fail to identify the pyTFM addon. Also note that OpenPIV is still in development, meaning that more recent versions of OpenPIV might not be compatible with pyTFM. Currently, if you are on windows, openpiv is installed from a compiled .whl file included in the pyTFM package. If you want to install openpiv on your own you are likely to need the [Microsoft Visual C++ build tools](#).

Introduction to Traction Force Microscopy and Monolayer Stress Microscopy

Traction Force Microscopy and Monolayer Stress Microscopy are methods to measure the force generation and internal forces of cells and cell colonies.

A typical experiment starts by seeding cells on a substrate containing small fluorescence labeled beads. The cells adhere to the substrate, start generating forces and consequently cause deformations in the substrate. These deformations can be measured by tracking the labeled beads. In practice the beads are imaged two times: Once when cells are attached to the substrate, and once when the cells are removed from the substrate. In the first image the substrate is strained by the cells, while the second image shows the completely relaxed substrate.

The deformations in the substrate can be used to calculate the traction forces that acted on the substrate surface. This is done by Fourier Transformed Traction Force Microscopy (TFM)¹. Any force that is applied from the cells to the substrate surface must be balanced by counteracting force inside of the cells. Forces inside of materials are described by stress.

A cell that generates two opposing forces at its ends, experiences a high stress between the points of force generation. The stress inside of cells and cell colonies is recovered by Monolayer Stress Microscopy (MSM)². In MSM the cells are modeled as a 2-dimensional sheet. Following the argument of force-balance, the traction forces calculated from TFM, with the opposite signs, are applied to the cell sheet. Then, the stresses in the cell sheet are calculated with 2-dimensional Finite Elements Methods

¹ **Traction fields, moments, and strain energy that cells exert on their surroundings** James P. Butler, Iva Marija Tolić-Norrelykke, Ben Fabry, and Jeffrey J. Fredberg *Am J Physiol Cell Physiol* 282: C595–C605, (2002)

² **Monolayer Stress Microscopy: Limitations, Artifacts, and Accuracy of Recovered Intercellular Stresses** Dhananjay T. Tambe, Ugo Croutelle, Xavier Trepate, Chan Young Park, Jae Hun Kim, Emil Millet, James P. Butler, Jeffrey J. Fredberg *PLOS ONE* 8(2): e55172 (2013)

Here you can find tutorials on how to use the pyTFM clickpoints addon, how to work on clickpoints databases without using the GUI, and how to use pyTFM on its own.

3.1 Analyzing Cell Colonies with Clickpoints

3.1.1 About this Tutorial

Using the pyTFM clickpoints addon requires a complete installation of clickpoints. If you have set up clickpoints correctly, you can open images by right clicking on the image files and select “open with clickpoints”.

You can find a small example data set, which is used during this tutorial, [here](#). The data is in the subfolder “clickpoints_tutorial”. This data set contains raw data for 2 types of cell colonies: In one group a critical cytoskeletal protein has been knocked out. We will compare these cell colonies to a set of wildtype colonies. The raw data, in the form of images, is contained in the subfolders “WT” and “KO”. All output files, databases and plots as you should produce them in the course of this tutorial are stored in the folders “KO_analyzed”, “WT_analyzed” and “plots”. Use these folders to check if your analysis was correct.

3.1.2 The Data

As you can see in [Fig. 3.1](#), there are 6 images for each colony type. This corresponds to two field of views for each wildtype and KO. For each field of view there are 3 images. One image (e.g. 03bf_before.tif) shows the colony and the boundaries between cells. In this case the image shows fluorescence stained cell membranes. The other two images show beads that are embedded in the substrate that the cells lie on. One image was recorded before the cells were removed (03before.tif) and the other was recorded after the cells were removed (03after.tif). The number in front of the filename (“03”, “10” and so on) indicates which field of view that image belongs to.

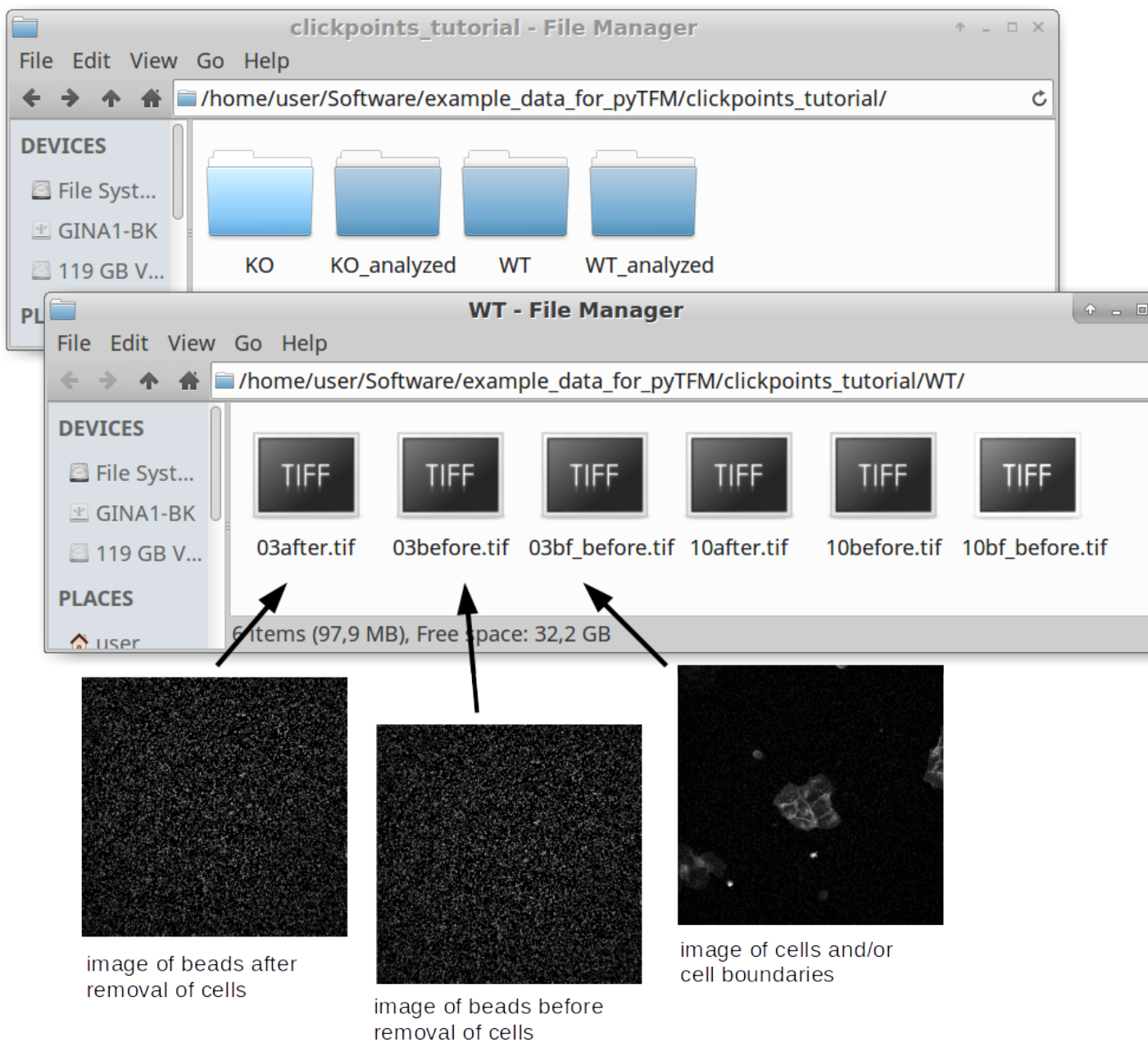


Fig. 3.1: Data structure of a Traction Force Microscopy experiment.

3.1.3 Opening Clickpoints and sorting Images

The first step to analyze the data is to create a clickpoints database, in which the images are identified correctly, concerning their type (whether it's an image of the cells or an image of the beads before or after cell removal) and concerning the field of view they belong to. **We are going to start with the wildtype data set.** To open a database simply right click on an image and select “open with” → “clickpoints”. The option to open with clickpoints might also be visible directly after you right clicked.

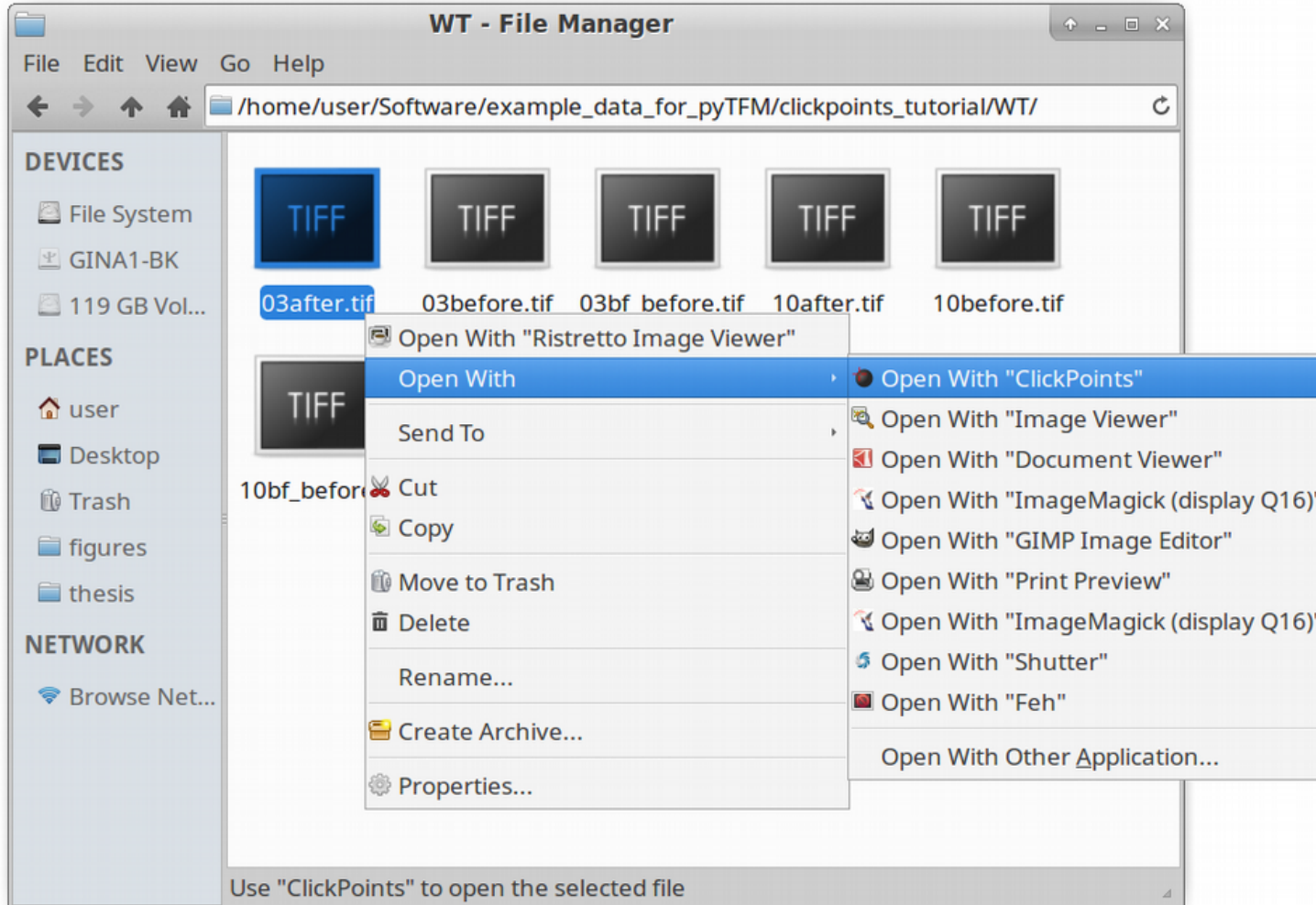


Fig. 3.2: Opening images with clickpoints.

Clickpoints sorts images in two dimensions: Frames and layers. The frames are displayed in the bar at the bottom. You can skip from frame to frame using the left and right arrows on your keyboard. Layers can be changed with the “Page Up” and “Page Down” keys. When you open the database, you will notice that there is only one layer and every image is sorted into a new frame. Our goal is to sort each field of view into one frame, with three layers per frame, each representing one type of image. In order to do this you need to open the pyTFM addon and open the “select image” menu. Follow the steps described in Fig. 3.3.

The “image selection” menu allows you to do three things: You can select where images are located and how they are classified. You can also set an output folder, where the database file and all analysis results will be saved and you can choose a name for the database. As mentioned above, the analysis requires three types of images. For each type you can select a folder (left hand side) and a regular expression that identifies the image type from the image filename

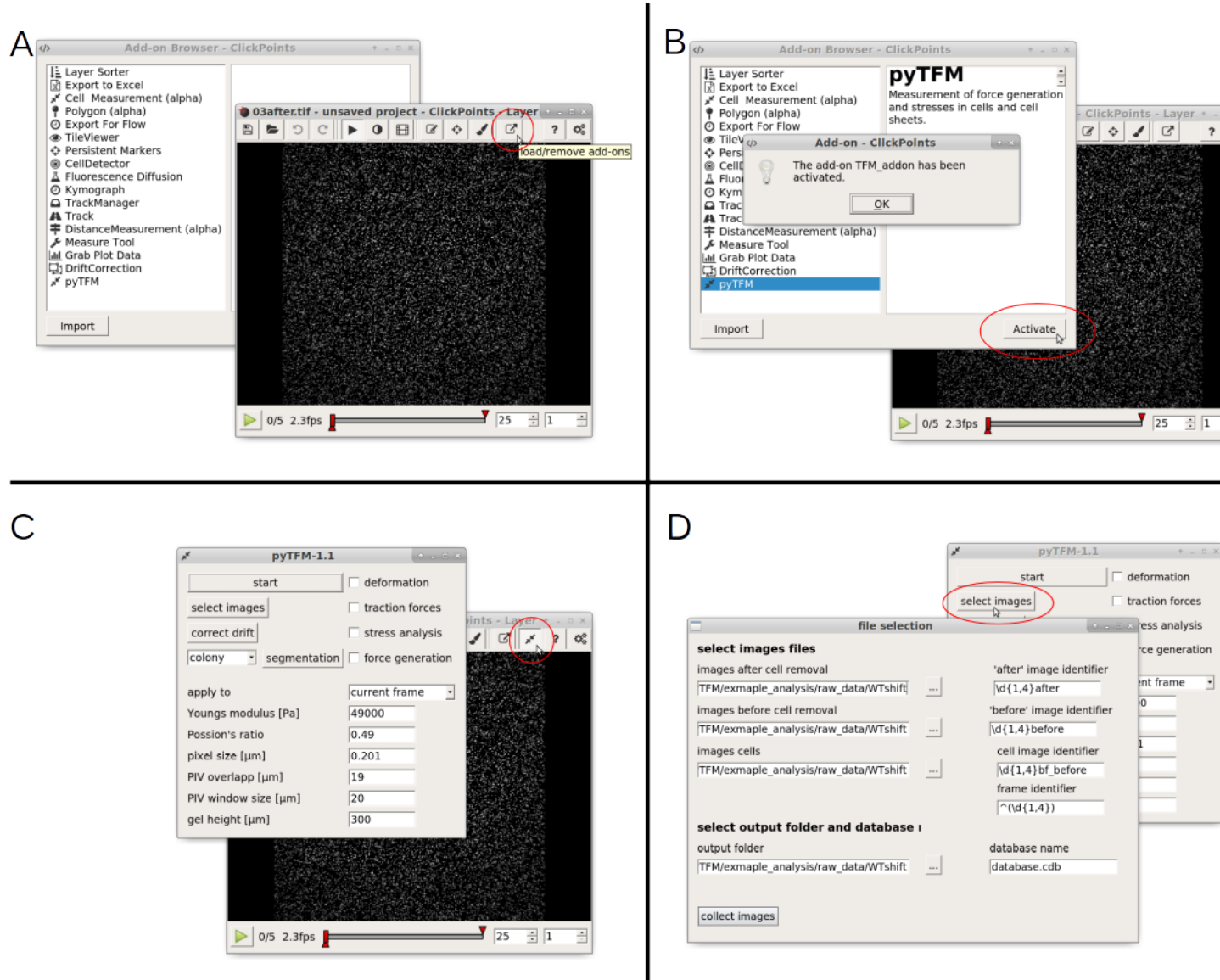


Fig. 3.3: A: Open the add-on-browser in clickpoints. A new window, with all available add-ons will open. B: Activate the pyTFM add-on by selecting pyTFM and clicking the “Activate” button. A window notifying you that the add-on has been loaded successfully will appear. After you press “OK” a new icon will appear in the clickpoints main window to the right of the add-on-browser button. C: Click on this button to open the pyTFM add-on. D: Finally, open the menu to select images by pressing the “select images” button.

(right hand side).

The default identifiers fit to the example data set, meaning that for now and in the future, if you are using the same naming scheme for your images, you **can leave the identifiers as they are**.

Note: Details on identifying images

The fields “‘after’ image identifier” and “‘before’ image identifier” are used to identify images of the substrate after and before relaxation. The field “cell image identifier” is used to identify images that show the cells or cell membranes. Finally, the “frame identifier” is used to identify the field of view each image belongs to. This must point to a unique part of the image filename, which is not limited to numbers. The part must be specifically enclosed by brackets “()”. Note that the extension (“.png”, “.tiff”, “.jpeg” ...) must not be included in the identifiers.

Regular expressions are the standard way to find patterns in texts. For example, it allows you to identify numbers of certain length, groups of characters or the beginning and end of a text. You find more information on regular expressions [here](#). Some useful expressions are listed in the table below:

search pattern	meaning
after	all files with “after” in the filename
^after	all files with “after” at the beginning of the filename
after\$	all files with “after” at the end of the filename
*	all files blank space also finds all files
^(\\d{1,4})	up to 4 numbers at beginning of the filename
(\\d{1,4})	up to 4 consecutive numbers anywhere in the filename
(\\d{1,4})\$	up to 4 numbers at end of the filename

Once you have entered identifiers for image types, frames, the output folder and the database name press the “collect images” button. You should see something like this:

Make sure your database didn’t contain any masks that you don’t want to delete. If you just opened the database from new images, you can press “Yes”. The path to the images that are sorted into the database, the type of the images (layer) and the field of view of the images (frame) are printed to the console. Make sure all images are sorted correctly. The program has generated a new clickpoints database file. Your currently opened clickpoints window updates automatically. You can close the “image selection” window now.

3.1.4 Correction of Stage Drift

Most often you will not manage to record exactly the same field of view when imaging the beads before and after cell removal. This will have an effect when you calculate the deformation field. In particular you will see your whole deformation field showing a bias to one direction. You can (and need to) correct this with the “correct drift” function (Fig. 3.5). This function will cut out the common field of view of the images of the beads before and after cell removal and shift one of the images with subpixel accuracy to match the other. The images of the cells will be cropped in the same way as the images of the beads.

First, go to the main pyTFM addon window and select “all frames” or “current frame” in the “apply to” option, depending on which frames you want to apply the drift correction to. Then press the “correct drift” button. You will see the drift in pixels in x and y direction printed to the console (Fig. 3.5, bottom).

Warning: This operation will permanently change you image files.

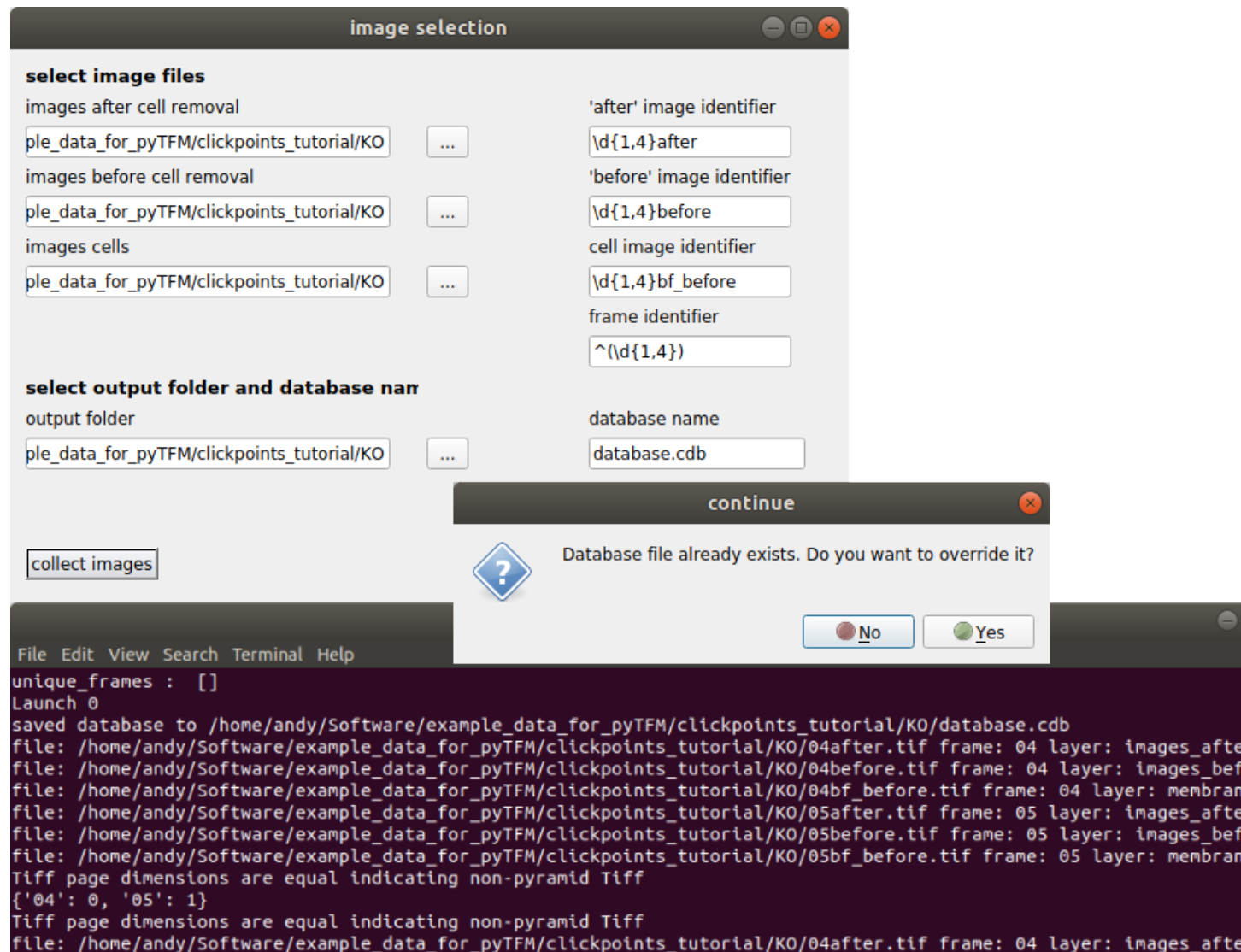


Fig. 3.4: Output of collect images.

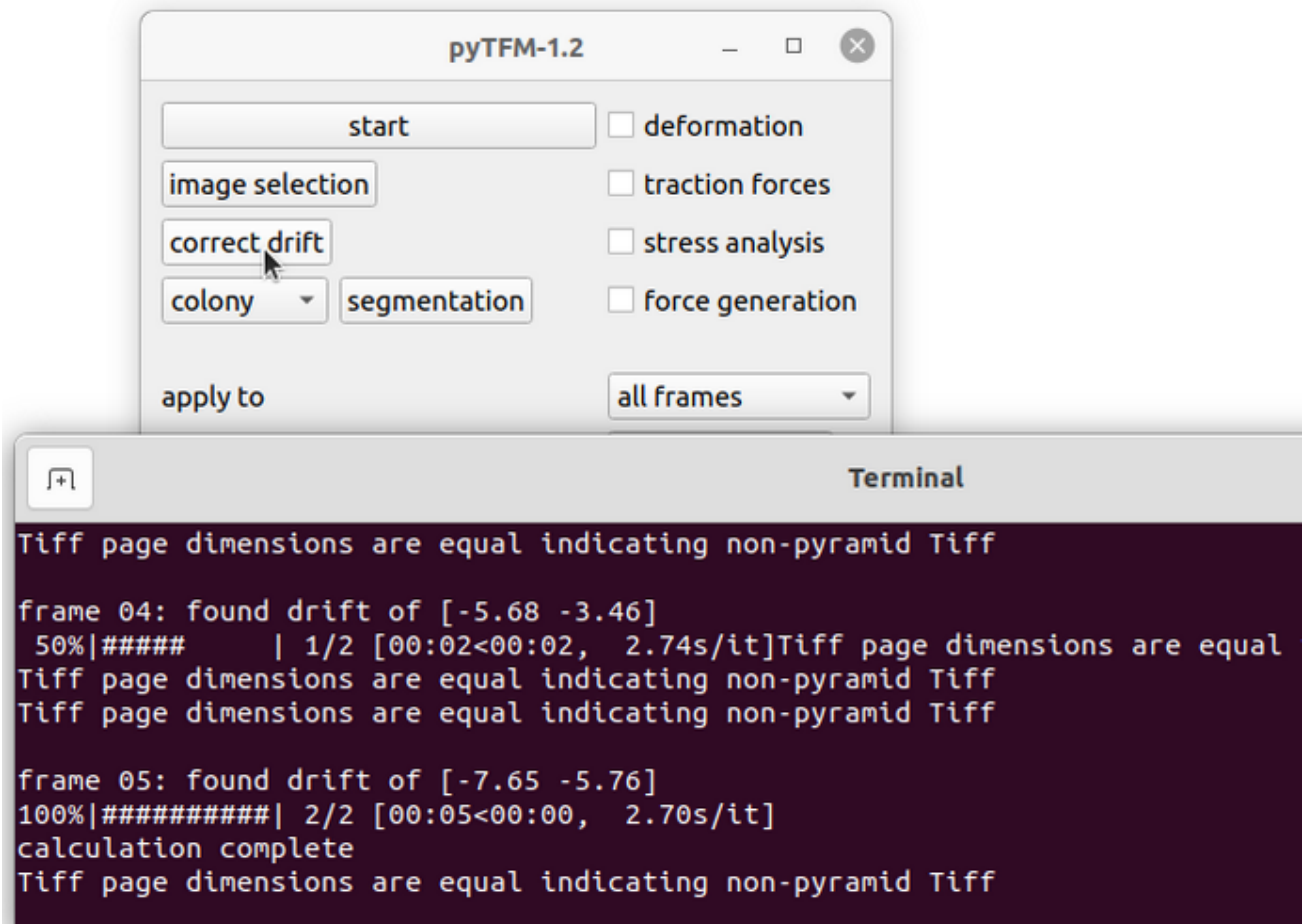


Fig. 3.5: Correction of stage drift.

3.1.5 Setting Parameters

Lets continue with calculating the deformation and traction field. Go to the pyTFM addon window (Fig. 3.6).

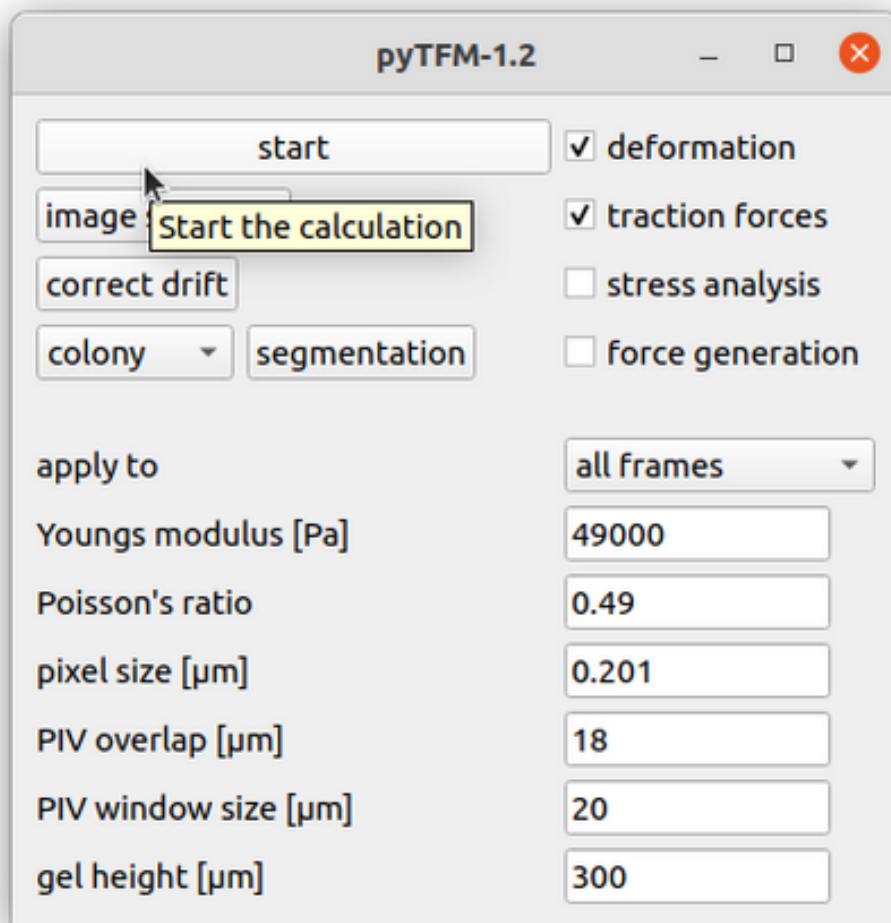


Fig. 3.6: Main addon window.

In this window you have to set the mechanical parameters of the substrate (“Youngs modulus” and “Poisson’s ratio”), the height of the substrate (“gel height”) and the pixel size (“pixel size”). Then you have to set two more parameters for the calculation of the deformation field. The deformation field is calculated with particle image velocimetry. This method essentially cuts out square shaped patches from the image of the beads before cell removal, places them on the image of beads after cell removal and checks how well they fit together. The vector from the original position of the patch to the position where the patch fits best in the image of beads after cell removal is the displacement vector. This is done for many positions to generate the complete displacement field.

You can control two things: the size of the patch that is cut out of the image of the beads after cell removal (with the parameter “PIV window size”) and the resolution of the resulting displacement field (with the parameter “PIV overlap”). A window size that is too large will blur the displacement field while a window size that is too small will introduce noise in the displacement field. As a rule of thumb the window size should be roughly 7 times the bead diameter - you should however try a few values and check which window size yields a smooth yet accurate deformation field.

Note: You can measure the beads diameter directly in clickpoints using another addon: The “Measure Tool”. It can be opened just like pyTFM from the addon browser

The “PIV overlap” mainly controls the resolution of the resulting displacement field and must be smaller than the “PIV window size” but at least half of the “PIV window size”. You need a high resolution for analyzing stress. In this step the area of cells should at least contain 1000 pixels. There will be a warning printed to the console and the output text file if this is not the case. However, if you are not calculating stresses, you can save a lot of calculation time by choosing a “PIV overlap” closer to half of the “PIV window size”. This is especially useful when you are for example trying out different window sizes or explore the influence of the gel height.

For this tutorial you can keep all parameters at their default value. If you are in a hurry you could set the “PIV window size” as low as 15 μm , and still obtain reasonable results.

3.1.6 Calculating Traction and Deformation Fields

Once you have set all parameters you can start the calculation: Use the tick boxes in the upper right to select which part of the analysis you want to perform. For now, we are gonna select only “deformation” and “traction forces”. Then use the “apply to” option to choose whether all frames should be analyzed or only the frame that you are currently viewing. Your window should now look like Fig. 3.6. Finally press “start” in the upper left to begin the analysis. With the default parameters this takes about 5 minutes per frame. “calculation complete” is printed to the console once all frames have been analyzed.

The traction and deformation fields are added to the database as new layers. Switch to these layers using the “Page Up” key on your keyboard. Traction and deformation for the first frame in the wildtype data should look like this:

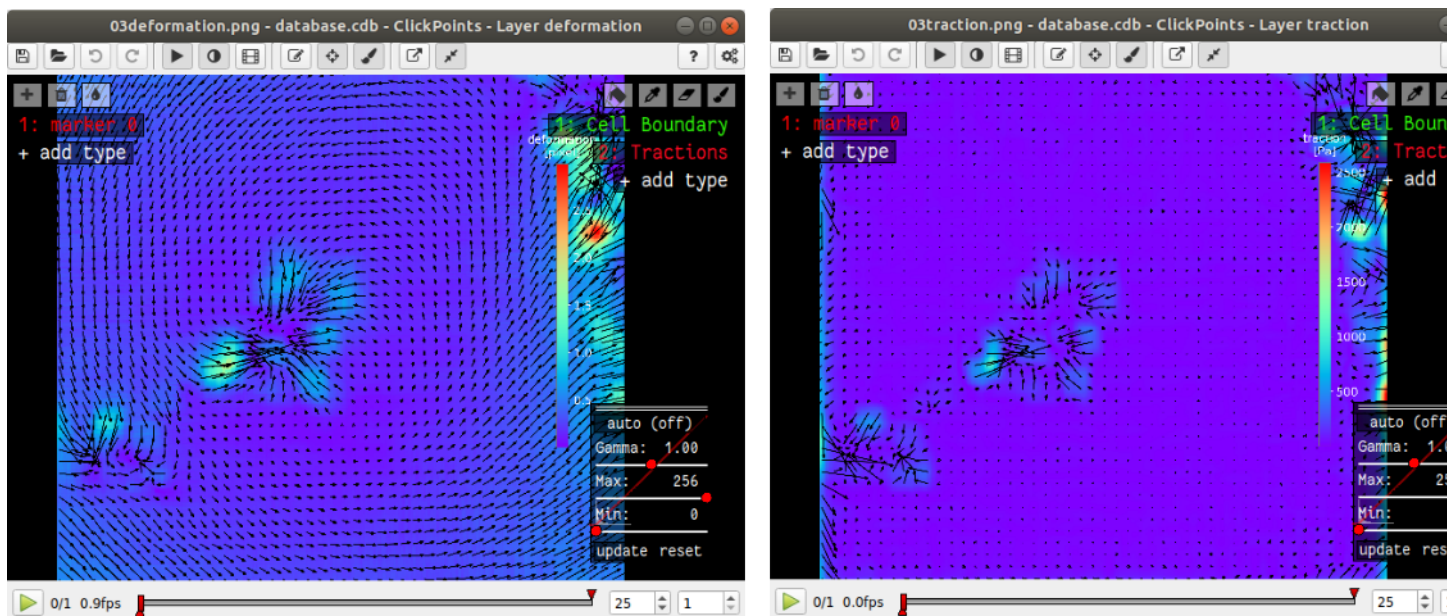


Fig. 3.7: Deformation and traction fields.

If you do not see the display tool and mask names (“Cell Boundary”, “Tractions”) on the right press F2.

3.1.7 Quantifying Force Generation

Force generation of the cell colony is quantified with the strain energy and the contractility. To avoid influence of nearby cells or noise, you have to select all tractions that are generated specifically by the cells that you are analyzing. You can do this by drawing a mask in clickpoints. In the top right of the clickpoints window you should see a set of tools to draw mask and two preset types of masks. If you don't see these tools, press F2.

Hint: Tips for masks in clickpoints. Select a mask and use the brush tool  to draw it. You can increase and decrease the size of the brush with the “+” and “-” keys. If you want to erase a part of a mask use the eraser tool .

Additionally you can fill holes in your mask with the bucket tool . Mask types cannot overlap, which means that one mask type is erased when you paint over it with another mask type. Sometimes you will have a hard time seeing things that are covered by a mask. Press “i” and “o” to decrease and increase the transparency of the mask.

The mask type used to calculate strain energy and contractility is called “Traction”. Select this mask and draw a circle around all tractions that you think originate from the cell colony. Typically, the area you encircle is large than the cell colony itself. It's not a big deal if your selection is a bit too big, but you should make sure not to include tractions that do not originate from the cell colony. You don't need to fill the area you have encircled; This is done automatically. However, if you see the “no mask found in frame ..” warning message in the console, you should first make sure that there is no gap in the circle that you drew. I drew the mask like this:

You could now press start again, and the program would generate a text file listing the contractility and strain energy for all frames. In order to be a bit more organized and get all results in one text file, we will first prepare to analyze stresses in the cell sheet at the same time.

3.1.8 Measuring Stresses, the Cell-Cell Force Transfer, counting Cells and measuring the Colony Area

Cellular stresses are calculated by modelling the cell colony as a 2 dimensional sheet and applying the traction forces that we have just calculated to it. Due to inaccuracies in the traction force calculation, namely that some forces are predicted to originate from outside of the cell sheet, the cell colony is modeled as an area slightly extending beyond the actual cell colony edge. We have shown that stresses are calculated with the highest accuracy if this area covers all cell-generated tractions but does not extend significantly further than that. When in doubt, it is better to choose the area larger rather than smaller.

pyTFM uses the area that you just marked with the mask “Traction” to model the cell colony. Stresses and cell-cell force transfer (quantified by the line tension) are however evaluated strictly on the inside of the cell colony. Thus you have to the edge of the cell colony to compute average stresses. To quantify the line tension, you have to also mark the internal cell-cell boundaries. Both is done with the Mask “Cell Boundary”; the program automatically distinguishes between internal cell-cell boundaries and the colony edge.

In the main window of clickpoints switch to the image showing the cell membrane using the the “Page Up” or “Page Down” key, select the mask “Cell Boundary” and mark all cell membranes.

Hint: Press F2 and use the controls (see below) in the bottom right to adjust the contrast of the image. This might help you to see the membrane staining better.

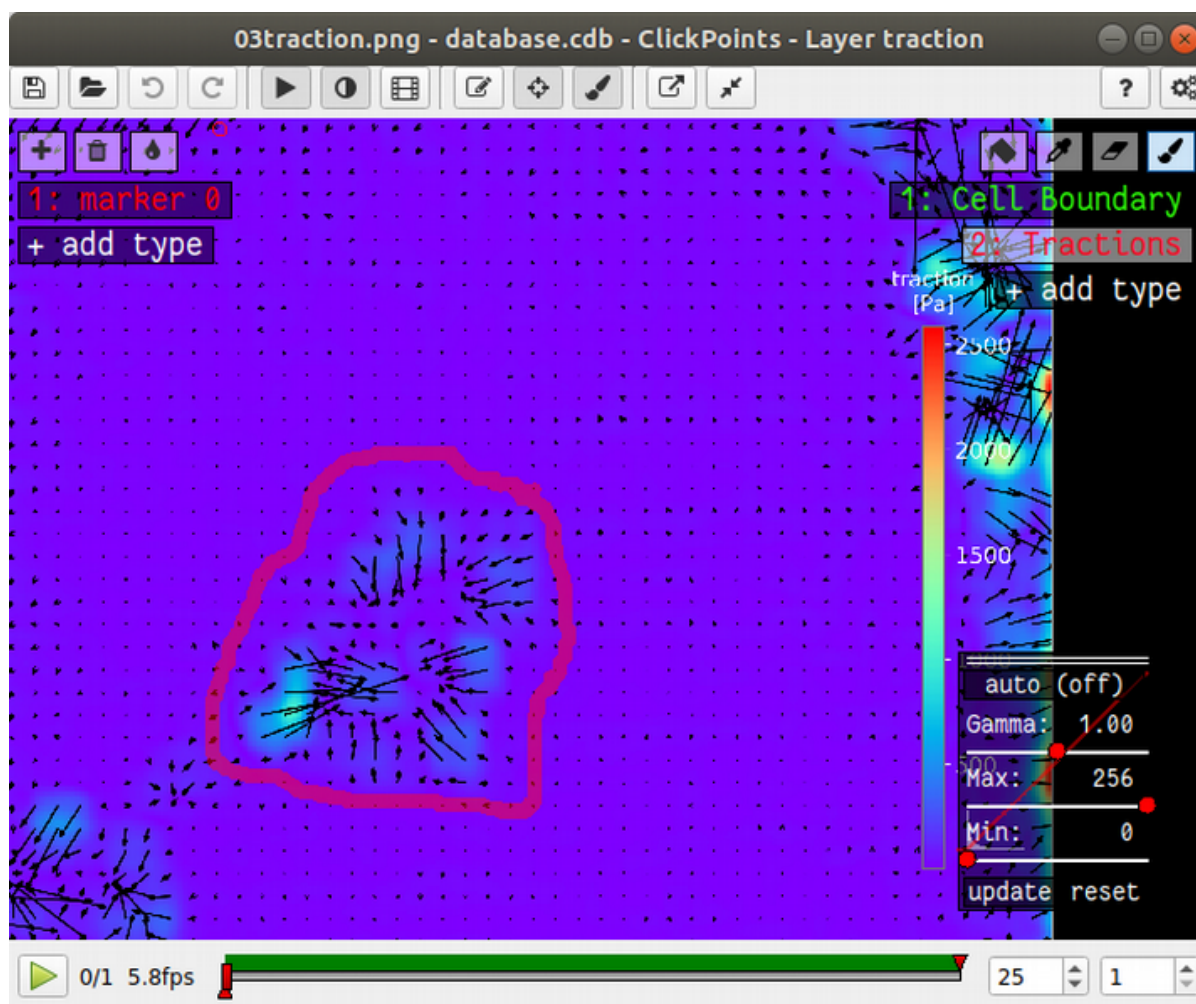
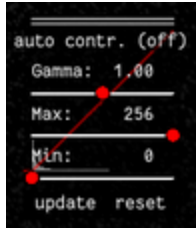


Fig. 3.8: Mask for quantification of force generation.



Use a thin brush and make sure that there are no unintentional gaps. Also mark the outer edge of the colony. I drew the mask like this:

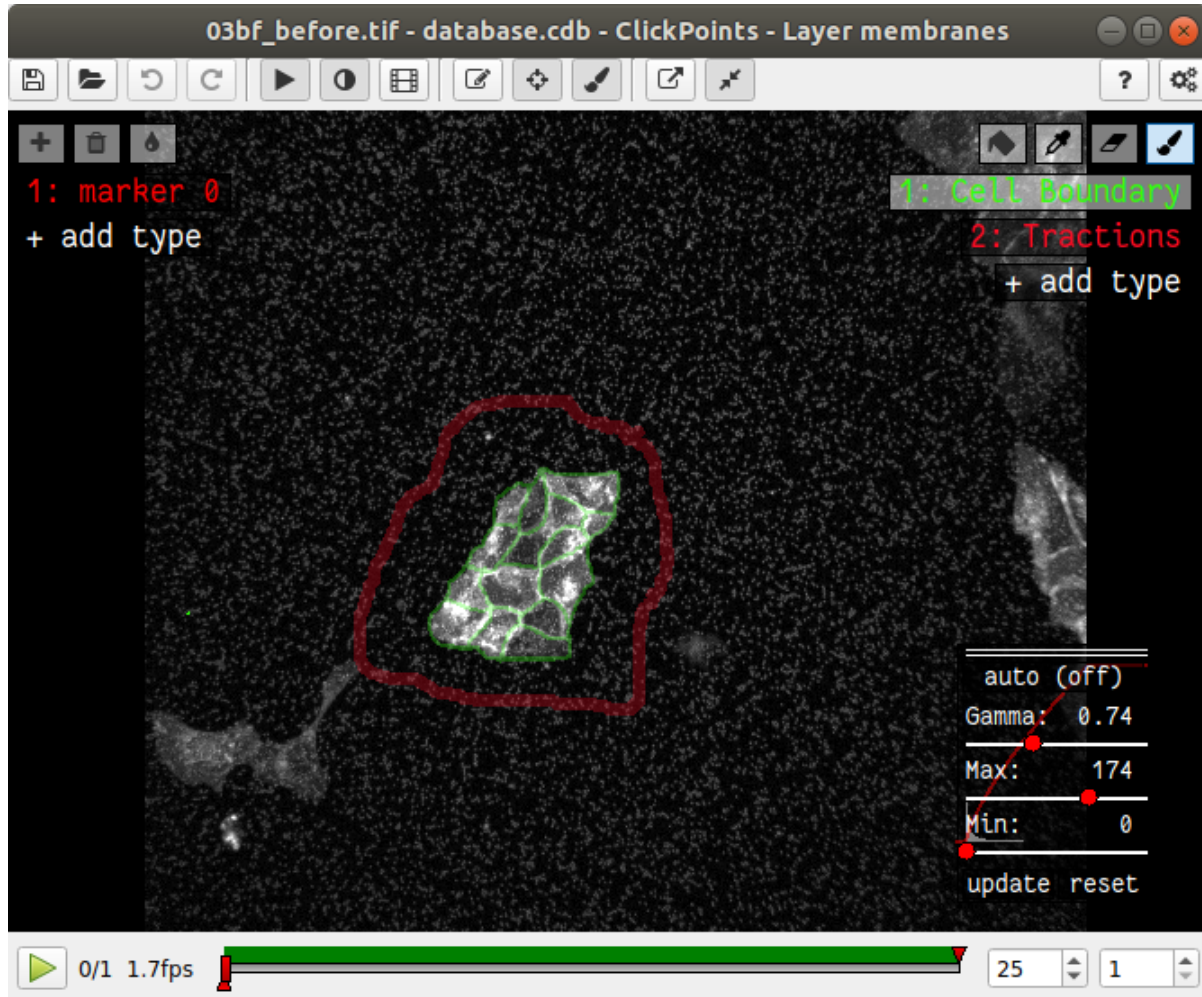
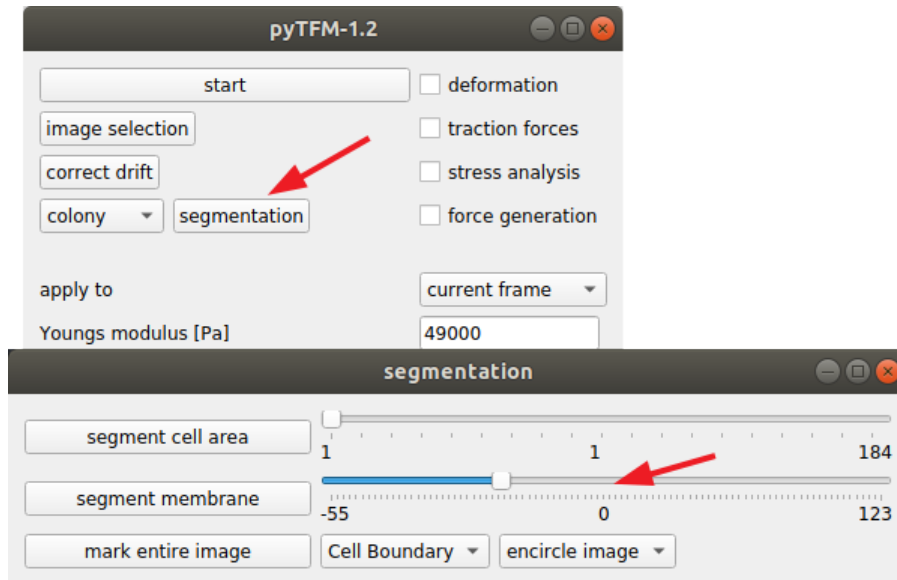


Fig. 3.9: Mask of cell membranes.

Hint: If there are a large number of cell boundaries, you can try the automatic segmentation tool of pyTFM. Press the segmentation button in the main window. The slider next to “segment membrane” allows you to control the segmentation sensitivity. The mask displayed in the clickpoints window will automatically be updated when you select a new value. You can apply this segmentation threshold to all frames by pressing the “segment membrane” button. Note however that this segmentation algorithm is rather primitive; the results may be poor, strongly depend on the quality of your images and will usually need some manual revision.



Once you have drawn all masks in all frames you are ready to start the calculation. Go to the pyTFM addon window, tick the check boxes for “stress analysis” and “force generation”, make sure you have set “apply to” to “all frames”, untick the “deformation” and traction forces” boxes and press start. The calculation should take up to 5 minutes.

After the calculation is complete two new plots will be added to the database. The first will show the mean normal stress in the cell colony and the second will show the line tension along all cell-cell borders. The outer edge of the cell colony is marked in grey. These lines are not used in the calculation.

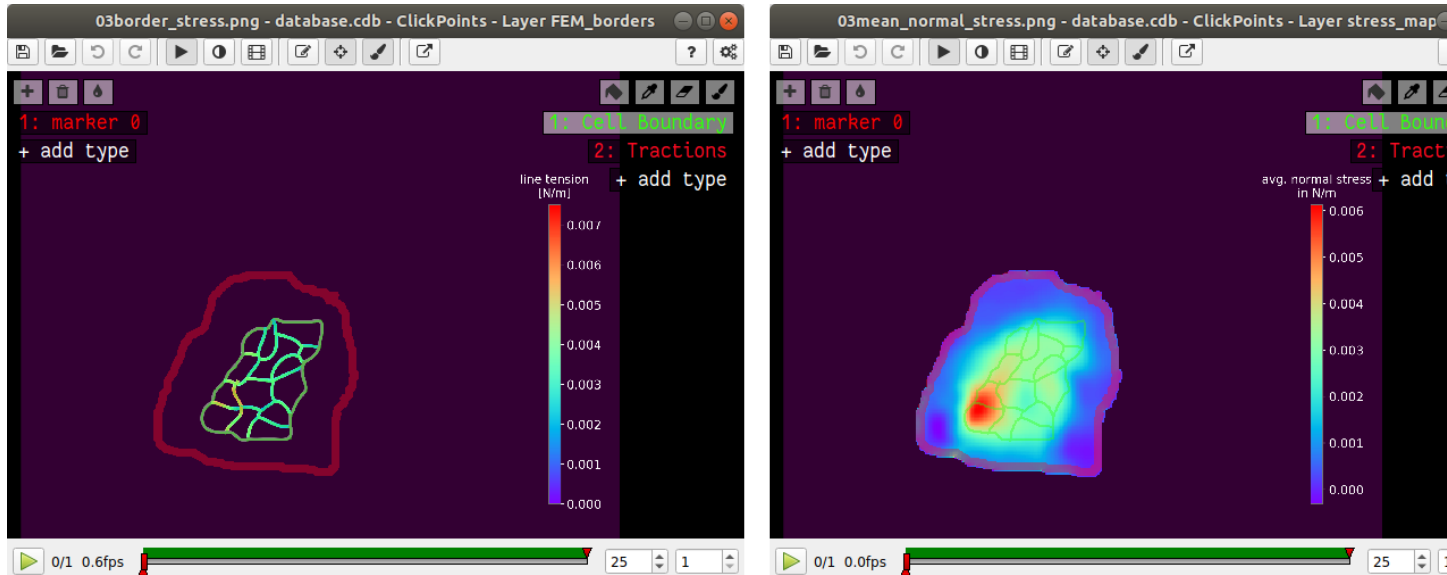


Fig. 3.10: Mean normal stress and line tension.

Note: A few notes on the calculation of stresses. The average stresses (average mean normal and average shear stress) and the coefficient of variation of these stresses is calculated by averaging over the true area of the cell colony, marked with the mask “membrane”. The mean normal stress should be high in areas where strong forces oppose each other. This can be seen in Fig. 3.10. Likewise, the line tension is high if strong forces oppose each other across the line. A high mean normal stress does not necessarily indicate a high line tension. It is better to look at the traction

forces, when checking if the values for the line tension make sense.

3.1.9 Understanding the Output File

Every time you press start the program creates a text file “out.txt” in the output folder. If such a file already exists, the text file is named out0.txt, out1.txt and so on. The output starts with a header containing important parameters of the calculation (Fig. 3.11). This is followed by a section containing all results. Each line has 4 to 6 tab-delimited columns, containing the frame, the id of the object in the frame (if you analyze multiple cells or cell colonies in this frame), the name of the quantity, the value of the quantity and optionally the unit of the quantity and also optionally a warning.

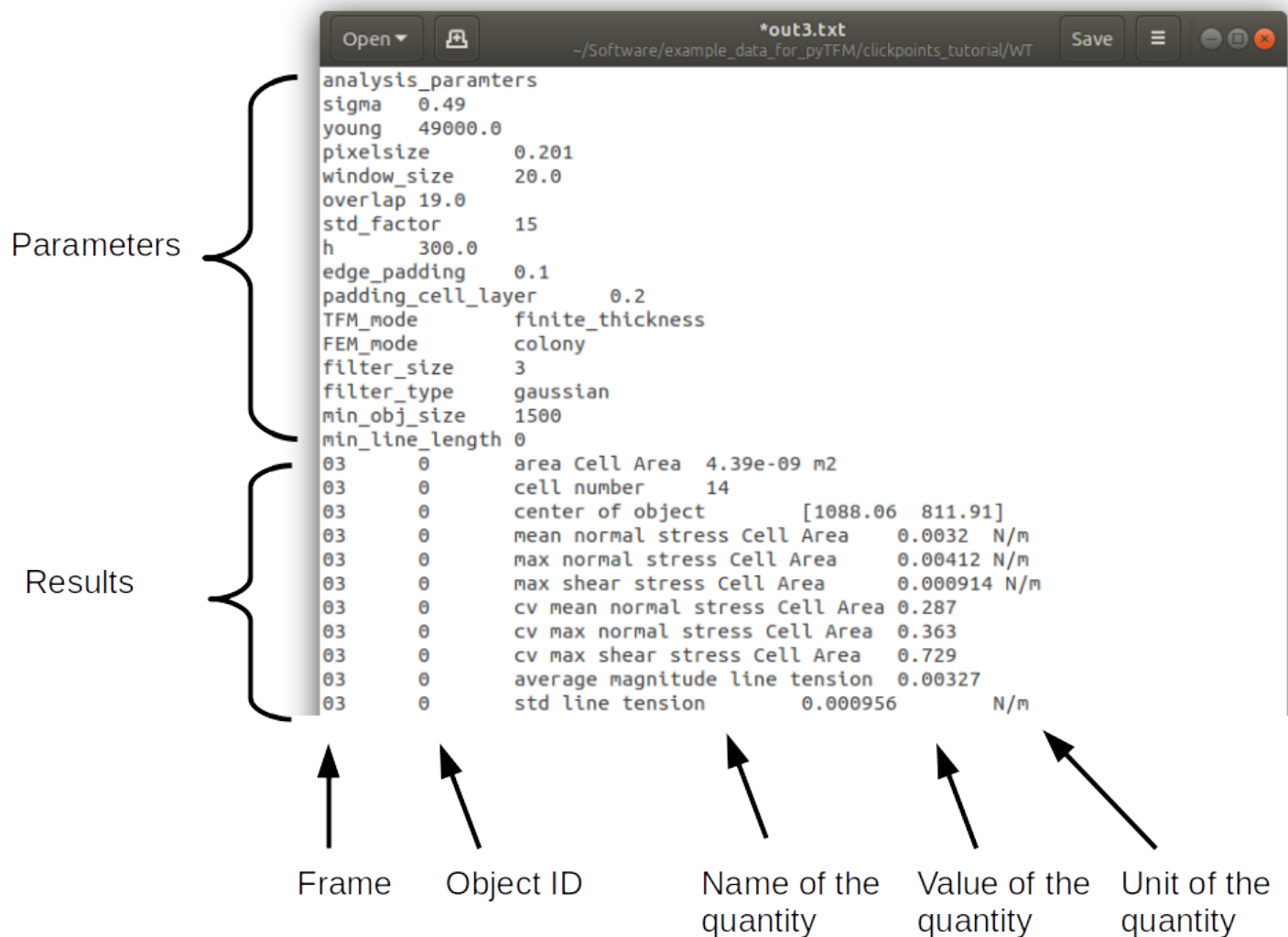


Fig. 3.11: The output file.

Warnings such as “mask was cut close to image edge” and “small FEM grid” should not be ignored.

3.1.10 Plotting the Results

Repeat the same analysis for the KO data set. Once you have the output text files for both data sets you could go ahead and use any tool of your choosing to read the files and plot the important quantities. Of course the best tool to do so is python: pyTFM provides its own python functions to read and plot data. The following code is also contained in the file “clickpoints_tutorial/data_analysis.py”. This file can be executed with python, however you must first edit the file paths for in- and output.

First lets import all functions that we need:

```
from pyTFM.data_analysis import *
```

Next, we read the output files from wildtype and KO data sets. This is done in two steps: First the text files are read into a dictionary where they are sorted for the frames, object ids and the type of the quantity. Then frames and objects are pooled to a dictionary where each key is the name of a quantity and the value is a list of the measured values. Note that our output text file for the last step should be called “out0.txt” if you followed the tutorial exactly.

```
# reading the Wildtype data set. Use your own output text file here
file_WT = r"/home/user/Software/example_data_for_pyTFM/clickpoints_tutorial/WT/out.txt
↪"
# reading the parameters and the results, sorted for frames and object ids
parameter_dict_WT, res_dict_WT = read_output_file(file_WT)
# pooling all frames and objects together.
n_frames_WT, values_dict_WT, frame_list_WT = prepare_values(res_dict_WT)
# reading the KO data set. Use your own output text file here
file_KO = r"/home/user/Software/example_data_for_pyTFM/clickpoints_tutorial/KO/out.txt
↪"
parameter_dict_KO, res_dict_KO = read_output_file(file_KO)
n_frames_KO, values_dict_KO, frame_list_KO = prepare_values(res_dict_KO)
```

We are going to use the dictionaries with pooled values (values_dict_WT and values_dict_KO) for plotting. First, let’s do some normalization: We can guess that a larger colony generates more forces. If we assume this relation is somewhat linear it is useful to normalize measures of the force generation with the surface area of the colony:

```
# normalizing the strain energy
values_dict_WT["strain energy per area"] = values_dict_WT["strain energy"]/values_
↪dict_WT["area Cell Area"]
values_dict_KO["strain energy per area"] = values_dict_KO["strain energy"]/values_
↪dict_WT["area Cell Area"]
# normalizing the contractility
values_dict_WT["contractility per area"] = values_dict_WT["contractility"]/values_
↪dict_WT["area Cell Area"]
values_dict_KO["contractility per area"] = values_dict_KO["contractility"]/values_
↪dict_WT["area Cell Area"]
```

Note that this only works if force generation and area were calculated for all colonies - this requires that you outline the colony with the mask “Cell Boundary”.

Now we can perform a t-test to identify any significant differences between KO and WT. We will do this for all quantity pairs at once and later display only the most important quantities. Unfortunately, due to the the fact that we analyzed only two colonies per data set you will find no significant differences in this case.

```
# t-test for all value pairs
t_test_dict = t_test(values_dict_WT, values_dict_KO)
```

Let’s produce some plots. First, we are going to compare some key quantities with box plots. The function “box_plots” expects two dictionaries with values, a list (“labels”) with two elements, which identifies these dictionary and a list

("types") of quantities that you want to plot. Additionally, you can provide a dictionary containing statistical test results and specify your own axis labels and axis limits:

```
lables = ["WT", "KO"] # designations for the two dictionaries that are provided to
↳ the box_plots functions
types = ["contractility per area", "strain energy per area"] # name of the quantities
↳ that are plotted
ylabels = ["contractility per colony area [N/m2]", "strain energy per colony area [J/
↳ m2"] # custom axes labels
# producing a two box plots comparing the strain energy and the contractility in WT
↳ and KO
fig_force = box_plots(values_dict_WT, values_dict_KO, lables, t_test_dict=t_test_dict,
↳ types=types,
    low_ylim=0, ylabels=ylabels, plot_legend=True)
```

We can do the same for the mean normal stress and line tension:

```
lables = ["WT", "KO"] # designations for the two dictionaries that are provided to
↳ the box_plots functions
types = ["mean normal stress Cell Area", "average magnitude line tension"] # name of
↳ the quantities that are plotted
ylabels = ["mean normal stress [N/m]", "line tension [N/m]"] #
fig_stress = box_plots(values_dict_WT, values_dict_KO, lables, t_test_dict=t_test_
↳ dict, types=types,
    low_ylim=0, ylabels=ylabels, plot_legend=True)
```

Another interesting way of studying force generation is to look at the relation between strain energy (a measure for total force generation) and contractility (measure for the coordinated force generation) This can be done as follows:

```
lables = ["WT", "KO"] # designations for the two dictionaries that are provided to
↳ the box_plots functions
# name of the measures that are plotted. Must be length 2 for this case.
types = ["contractility per area", "strain energy per area"]
# plotting value of types[0] vs value of types[1]
fig_force2 = compare_two_values(values_dict_WT, values_dict_KO, types, lables,
    xlabel="contractility per colony area [N/m2]", ylabel="strain energy per
↳ colony area [J/m2])
```

Finally, let's save the figures.

```
# define and output folder for your figures
folder_plots = r"/home/user/Software/example_data_for_pyTFM/clickpoints_tutorial/
↳ plots/"
# create the folder, if it doesn't already exist
createFolder(folder_plots)
# saving the three figures that were created beforehand
fig_force.savefig(os.path.join(folder_plots, "forces1.png")) # boxplot comparing
↳ measures for force generation
fig_stress.savefig(os.path.join(folder_plots, "fig_stress.png")) # boxplot comparing
↳ normal stress and line tension
fig_force2.savefig(os.path.join(folder_plots, "forces2.png")) # plot of strain energy
↳ vs contractility
```

3.2 Using pyTFM in Python

pyTFM makes it easy to perform Traction Force Microscopy and Monolayer Stress Microscopy in python. In this tutorial we will calculate strain energy, contractility, mean normal stress and average line tension of a cell colony. As always we need two images of fluorescent beads: One image before cell removal and one image after cell removal. Additionally, since we are not using the clickpoints addon, we need to provide 3 masks: A mask for the area on which force generation is evaluated (encircling all deformations and forces that originate from the cell colony), a mask for the Finite Elements Analysis (encircling all forces that originate from the cell colony) and a mask of the cell boundaries. Of course, the easiest way to generate these masks is to use clickpoints.

In the `example data set` in the subfolder “python_tutorial” you can find images for a single cell colony as well as suitable masks. I drew the masks in clickpoints and saved them as PNG files so that you can look at them with a standard image display tool. You can use any other format, as long as you can load them as a boolean (True and False) array to python.

The “python_tutorial” folder also contains the complete code of the tutorial in a single python script “tutorial.py”. This script prints all quantities and produces all figures, that we are going to see in this tutorial. It should work right away.

3.2.1 Calculating Deformation Fields

First, let’s import functions to calculate and plot the deformation field:

```
from pyTFM.TFM_functions import calculate_deformation
from pyTFM.plotting import show_quiver
```

Calculating the deformation field requires the images of the beads before and after cell removal. You can provide either paths to the files, or arrays (which must have the data type int32). The deformation field is calculated with Particle Image Velocimetry, using a cross correlation algorithm. You need to find appropriate values for the `window_size` and `overlap` that produce a smooth deformation field. For this data you can use:

```
# paths to the images
im_path1 = r"/home/user/Software/example_data_for_pyTFM/python_tutorial/04after.tif"
↪ # change to your location
im_path2 = r"/home/user/Software/example_data_for_pyTFM/python_tutorial/04before.tif"
# calculating the deformation
u, v, mask_val, mask_std = calculate_deformation(im_path1, im_path2, window_size = 100,
↪ overlap = 60)
# the unit of window size and overlap is pixels of the image of the beads
```

The overlap of 60 is a bit to small, especially for the FEM analysis later. If you want to be more accurate use, an overlap of 95. This will however increase the calculation time from a few seconds to roughly 5 minutes.

Let’s plot the deformation field:

```
# plotting the deformation field
fig1, ax = show_quiver(u, v, cbar_str="deformations\n[pixels]")
```

`show_quiver` accepts most of the plotting parameters listed in `OverviewofPlottingParameters`.

Hint: In some cases the images of the beads don’t share exactly the same field of view. You will notice this in the deformation field. pyTFM provides a function to find and extract the common field of view of both images with subpixel accuracy:

```
from pyTFM.frame_shift_correction import correct_stage_drift
from PIL import Image
import numpy as np

# load your images; dtype must be float32; images must be grayscale (you need a 2_
↳dimensional array)
image1 = np.asarray(Image.open(r"/home/user/Software/example_data_for_pyTFM/python_
↳tutorial/04after.tif"))
image2 = np.asarray(Image.open(r"/home/user/Software/example_data_for_pyTFM/python_
↳tutorial/04before.tif"))

# cutting out the common field of view of image1 with image2.
# This also normalizes the images and applies a subpixel
# accurate shift. You can provide an additional list of
# images that will be cut to the same field of view.
image1_cor, image2_cor, other_images, drift = correct_stage_drift(image1, image2,
↳additional_images=[])

# saving the output
image1_cor.save(r"/home/user/Software/example_data_for_pyTFM/python_tutorial/04after_
↳corr.tif")
image2_cor.save(r"/home/user/Software/example_data_for_pyTFM/python_tutorial/04before_
↳corr.tif")
```

The images in this tutorial are already corrected.

3.2.2 Calculating Traction Fields

Next, we are going to calculate the traction forces, that the cell colony generated. This is done with the “TFM_tractions” function:

```
from pyTFM.TFM_functions import TFM_tractions
import numpy as np
```

We have to set the elastic parameters of the substrate (Young’s modulus, Poisson’s ratio and the height of the substrate). You can also set the substrate height to “infinite”. We also need the pixel size of the image of the beads and the pixel size of the deformation field. The later can be calculate if you know the dimensions and pixel size of the image of the beads:

```
ps1 = 0.201 # pixel size of the image of the beads
im1_shape = (1991, 2033) # dimensions of the image of the beads
ps2 = ps1 * np.mean(np.array(im1_shape) / np.array(u.shape)) # pixel size of of the_
↳deformation field
young = 49000 # Young's modulus of the substrate in Pa
sigma = 0.49 # Poisson's ratio of the substrate
h = 300 # height of the substrate in µm, "infinite" is also accepted
```

Finally, the traction field can be calculated by:

```
tx, ty = TFM_tractions(u, v, pixelsize1=ps1, pixelsize2=ps2, h=h, young=young,
↳sigma=sigma)
```

We can plot it in the same way as we plotted the deformation field:

```
fig2, ax = show_quiver(tx, ty, cbar_str="tractions\n[Pa]")
```

3.2.3 Quantifying the Force Generation

In order to quantify the force generation of the cell colony, we have to select the area where deformations and tractions that are generated by the colony are located. This selection requires a mask, a boolean array, that has the value True in the area that we want to use and False elsewhere. I produced the appropriate mask in clickpoints and saved it as a grayscale image as “force_measurement.png”. After loading the mask, there are two more things we need to do: First, we need to fill all holes in the mask in order to produce a continuous area. Second, we need to resize the mask to the dimensions of the deformation and traction fields:

```
import matplotlib.pyplot as plt
from scipy.ndimage.morphology import binary_fill_holes
from pyTFM.grid_setup_solids_py import interpolation # a simple function to resize_
↳ the mask

# loading a mask that defines the area used for measuring the force generation
mask = plt.imread(r"/home/user/Software/example_data_for_pyTFM/python_tutorial/
↳ Tractions.png").astype(bool)
mask = binary_fill_holes(mask) # the mask should be a single patch without holes
# changing the masks dimensions to fit to the deformation and traction fields
mask = interpolation(mask, dims=u.shape)
```

This mask can now be used to calculate the contractility and the strain energy:

```
from pyTFM.TFM_functions import strain_energy_points, contractility

# strain energy:
# first we calculate a map of strain energy
energy_points = strain_energy_points(u, v, tx, ty, ps1, ps2) # J/pixel
# then we sum all energy points in the area defined by mask
strain_energy = np.sum(energy_points[mask]) # 2.14*10**-13 J

# contractility
contractile_force, proj_x, proj_y, center = contractility(tx, ty, ps2, mask) # 2.
↳ 0.3*10**-6 N
```

3.2.4 Measuring Stresses in Cell Colonies

Stresses are calculated with the Finite Elements Methods, modeling the colony as a 2 dimensional sheet and applying force opposite to the traction forces to it. The FEM algorithm largely uses the `solidspy` package. This package is also very instructive if you want to get into FEM in general.

We will use the previous area outlined in “Tractions.png” to model the cell colony. This area covers all cell generated tractions and is larger then the actual cell colony, due to inaccuracies in the calculation of tractions. All measures for stress are evaluated on the actual area of the cell colony. This area is generated from the cell boundaries.

```
# first mask: The area used for Finite Elements Methods
# it should encircle all forces generated by the cell colony
mask_FEM = plt.imread(r"/home/user/Software/example_data_for_pyTFM/python_tutorial/
↳ Tractions.png").astype(bool)
mask_FEM = binary_fill_holes(mask_FEM) # the mask should be a single patch without_
↳ holes
```

(continues on next page)

(continued from previous page)

```
# changing the masks dimensions to fit to the deformation and traction field:
mask_FEM = interpolation(mask_FEM, dims=tx.shape)

# second mask: The area of the cells. Average stresses and other values are
↳calculated only
# on the actual area of the cell, represented by this mask.
mask_cells = plt.imread(r"/home/user/Software/example_data_for_pyTFM/python_tutorial/
↳cell_borders.png").astype(bool)
mask_cells = binary_fill_holes(mask_cells)
mask_cells = interpolation(mask_cells, dims=tx.shape)
```

The traction forces in the FEM area are typically slightly unbalanced, leading to a net force and torque acting on the cell colony. We need to correct this:

```
from pyTFM.grid_setup_solids_py import prepare_forces

# converting tractions (forces per surface area) to actual forces
# and correcting imbalanced forces and torques
# tx->traction forces in x direction, ty->traction forces in y direction
# ps2->pixel size of the traction field, mask_FEM-> mask for FEM
fx, fy = prepare_forces(tx, ty, ps2, mask_FEM)
```

Now we are ready to perform a Finite Elements Analysis. This is split into two steps: First, the FEM grid is setup. The grid is build up of nodes. For each node the connectivity to other nodes (stored in “elements”), constraints on the displacements (stored in “nodes”) and forces acting (stored in “loads”) as well as elastic properties (Youngs’s modulus and Poisson’s ratio stored in mats) on the node are defined. Note that the Young’s modulus of the material has no influence whatsoever on the resulting stresses and that the Poisson’s ratio has only a small influence. Consequently, both Young’s modulus and Poisson’s ratio can be left at their default value (1 Pa and 0.5 respectively). The applied forces are simply obtained from the underlying tractions with a negative sign. Next, the FEM system is solved by calculating the deformations, followed by the strain and, based on the strain-stress relation of a linearly elastic 2-dimensional material.

```
from pyTFM.grid_setup_solids_py import grid_setup, FEM_simulation

# constructing the FEM grid
nodes, elements, loads, mats = grid_setup(mask_FEM, -fx, -fy, sigma=0.5)
# performing the FEM analysis
# verbose prints the progress of numerically solving the FEM system of equations.
UG_sol, stress_tensor = FEM_simulation(nodes, elements, loads, mats, mask_FEM,
↳verbose=True)
# UG_sol is a list of deformations for each node. We don't need it here.
```

The stress tensor completely defines the forces inside the cell colony. We can for example extract the average mean normal stress and the coefficient of variation of the mean normal stress (quantifying how much the stress varies in the colony) from the stress tensor. We will use the mask “mask_cells” which marks the actual area of the cell colony for these measurements.

```
# mean normal stress
ms_map = ((stress_tensor[:, :, 0, 0] + stress_tensor[:, :, 1, 1]) / 2) / (ps2 * 10**
↳6)
# average on the area of the cell colony.
ms = np.mean(ms_map[mask_cells]) # 0.0043 N/m

# coefficient of variation
cv = np.nanstd(ms_map[mask_cells]) / np.abs(np.nanmean(ms_map[mask_cells])) # 0.41 no
↳unit
```

3.2.5 Calculating the Line Tension

A particularly interesting feature are forces that are transmitted across cell-cell boundaries. This is quantified by the line tension. First, we need to load a mask marking all cell borders. Note that this is the same mask that we used to get the area of the cell colony, only this time we are not going to fill any holes or resize the mask.

```
# loading a mask of the cell borders
mask_borders = plt.imread(r"/home/user/Software/example_data_for_pyTFM/
python_tutorial/cell_borders.png").astype(bool)
```

The cell-cell borders are stored in an object “borders”, which among other things contains a spline interpolation of each border, assigns each border to a cell and contains a list of borders located at the edge of the cell colony:

```
from pyTFM.grid_setup_solids_py import find_borders

# identifying borders, counting cells, performing spline interpolation to smooth the
borders
borders = find_borders(mask_borders, tx.shape)
# we can for example get the number of cells from the "borders" object
n_cells = borders.n_cells # 8
```

We can use the cell-cell borders together with the stress tensor to calculate the line tension. The line tension is a force vector acting on a small slice of a cell border. We are going to calculate the average length of this vector (“avg_line_tension”) and the average normal component of the line tension (“avg_normal_line_tension”):

```
from pyTFM.stress_functions import lineTension

# calculating the line tension along the cell borders
lt, min_v, max_v = lineTension(borders.lines_splines, borders.line_lengths, stress_
tensor, pixel_length=ps2)
# lt is a nested dictionary. The first key is the id of a cell border.
# For each cell border the line tension vectors ("t_vecs"), the normal
# and shear component of the line tension ("t_shear") and the normal
# vectors of the cell border ("n_vecs") are calculated at a large number of points.

# average norm of the line tension. Only borders not at colony edge are used
lt_vecs = np.concatenate([lt[l_id]["t_vecs"] for l_id in lt.keys() if l_id not in
borders.edge_lines])
avg_line_tension = np.mean(np.linalg.norm(lt_vecs, axis=1)) # 0.00569 N/m

# average normal component of the line tension
lt_normal = np.concatenate([lt[l_id]["t_normal"] for l_id in lt.keys() if l_id not in
borders.edge_lines])
avg_normal_line_tension = np.mean(np.abs(lt_normal)) # 0.00566 N/m,
# here you can see that almost the line tensions act almost exclusively perpendicular
to the cell borders.
```

Finally let's produce a plot of the line tension:

```
from pyTFM.plotting import plot_continuous_boundary_stresses

# plotting the line tension
fig3, ax = plot_continuous_boundary_stresses([borders.inter_shape, borders.edge_lines,
lt, min_v, max_v], cbar_style="outside")
```

Typical Measures for Force Generation and Stresses in Cells

There is a number of different ways to measure force generation and stresses. Here you can find an overview of all quantities that can be calculated with this program.

4.1 Deformations in the Substrate

The simplest way is to sum up all deformations on the substrate surface that the cells are attached to. The deformations depend on the mechanical properties of the substrate, which means that this is not possible to compare results, when different substrates have been used.

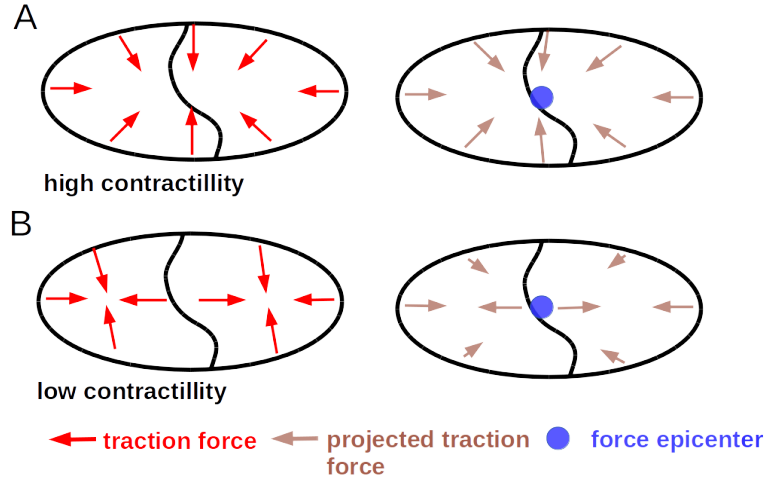
4.2 Strain Energy

The strain energy is the total work that cells have to commit to deform their substrate. It is defined as $\frac{1}{2} \int \vec{d} \times \vec{f}$, where $\vec{d}$ and $\vec{f}$ are the deformation and the traction force vectors. That means that a high strain energy only needs a local alignment of both vectors.

4.3 Contractility

The contractility is defined as the sum of the projection of all traction forces towards one point, called the force epicenter. As such, the contractility is high if all force are already orientated towards one central point. Locally opposing forces and forces not pointing towards the force epicenter do not contribute to the contractility. A cell or cell colony which is able to coordinate its force generation in such a way that the forces seem to originate from a single point can achieve a high contractility, while expending a comparably small amount of strain energy.

This is further illustrated in Fig. ???. Case A represents a cell colony with two cells with high coordination of force generation and Case B represents a cell with coordination. In case B each cell generates contractile forces on its own. Accordingly case B has a lower contractility if we assume equal strain energy in Case A and B.



To sum up: the strain energy is a measure for the total force generation, while the contractility is a measure for the coordinated force generation.

4.4 Average Normal and Shear Stress

Stress describes the forces that are transferred inside of a cell or cell sheet. For any given point in the cell sheet the stress is defined by a tensor with 4 components. Each component represents forces that would act on the edges of a square that was cut out of the cell sheet as shown in Fig. 4.1.

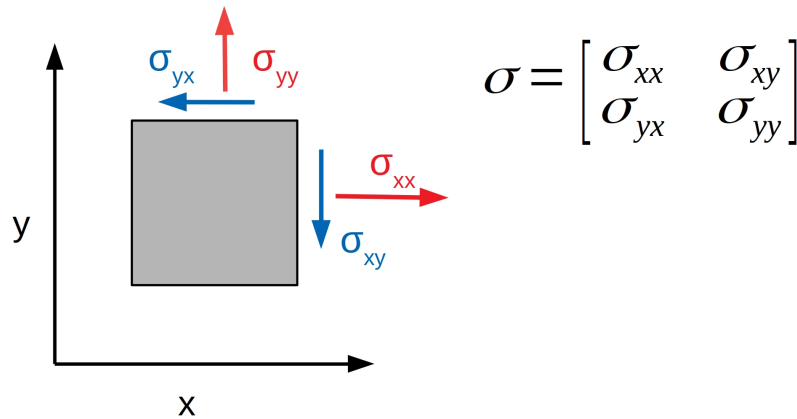


Fig. 4.1: σ_{xx} and σ_{yy} : Normal Stresses in x and y direction. σ_{xy} and σ_{yx} : Shear stresses.

We can distinguish between two types of stress: The shear stress, a force that acts parallel to the edges of this square, and normal stress, a force that acts perpendicular to the edges of this square. Due to geometric reasons both shear components of the stress tensor must be identical. This is not the case for the normal components. Since it is of no interest whether normal stress comes predominately from the x or y directions for our analysis, it is more useful to calculate the mean of both components. This leaves two stresses: the shear stress and the mean normal stress. These stresses can be averaged over the whole cell colony area.

Note: The mean normal stress can be either negative, indicating a compressive stress, or positive, indicating a tensile stress. A typical cell experiences tensile stress, as it pulls on the underlying substrate from its edges. The shear stress

can also have a positive or a negative sign. However, there is no useful interpretation in this case.

4.5 Distribution of Stresses

The distribution of stresses can be described by the Coefficient of Variation (CV), that is the standard deviation normalized with the mean, of mean normal or shear stress.

4.6 Forces acting across Cell Boundaries

As hinted above, the stress tensor can be used to calculate the force that acts across a boundary in the cell colony. This force is called the line tension, and has a straight forward interpretation: Imagine you were to actually cut the cell sheet along the boundary between two cells. If the cells continue to generate force the edges of this cut would drift apart or start overlapping as you have just cut the material holding both edges together. In order to hold both edges in place as they were before you cut them, you need to apply a force at the edges. This force, normalized by the length of the cut, is the line tension.

The line tension is a vector with x and y components. Similar to stresses it can be split in a shear component (the force acting parallel to the cut) and a normal component (the force acting perpendicular to the cut). Both contribute to the magnitude (length of the line tension vector) of the line tension.

Note: Similar to normal stresses, the normal component of the line tension can be negative or positive, indicating that the two sides of the edge along which the line tension was calculated, are pushed together or pulled apart. The shear component of the line tension lacks such an interpretation and the magnitude of the line tension can of course only be positive.

Using Config Files, Hidden Parameters and Plotting Behavior

5.1 Hidden Parameters

The most important parameters (Young’s modulus, Poisson’s ratio, gel height, window size and overlap for Particle Image Velocimetry) can be set in the clickpoints addons window. However, some parameters can only be set by using config files. Config files also allow you to set a wide range of plotting options, most notably: Color maps, the length of arrows, color bar positioning and dimensions and maximum and minimum displayed values for all frames. You can find a complete list of parameters in *Overview of Analysis Parameters* and *Overview of Plotting Parameters*.

5.2 Using Config Files

You can provide a config file by placing a file “config.yaml” in the same folder as your database (.cdb) file. YAML is a standard format for configuration files. In YAML files parameters are defined by a series of indented key words like this:

```
key1:
  key2:
    key3: value
```

You can find a config.yaml file that reproduces all default parameters in the [example data](#). The config file distinguishes two classes of parameters at the first level: The “analysis_parameters” and the “fig_parameters”. “analysis_parameters” set all parameters that are used by the program not related to plotting. They can only be numbers or strings. For example to change the minimum object size and change the initial value for the Poisson’s ration, your config file should contain this:

```
analysis_parameters: # setting parameters for the analysis
  sigma: 0.4 # changing the Poisson's ratio set at the script start up
  min_obj_size: 1500 # changing the minimal object size (cannot be changed in the_
↪addon window)
```

“fig_parameters” control the plotting. Most parameters can be specified as a single value that will apply to all plots. Alternatively you can specify a second key that defines a type of plot first and then a value. This value will only apply to the plot type you specified. All other plot types will use a default value. Possible plot types are:

“deformation” - Quiver plot of the deformation field
“traction” - Quiver plot of the traction forces
“FEM borders” - Plot of the line tension along cell-cell borders
“stress map” - Plot of the mean normal stress
“energy_points”. - Plot of the strain energy density

Let’s say you want to

- A): Change the color map to “jet” (all matplotlib color maps are accepted) for all plots.
- B): Set the maximum value of the colormap only in the deformation plots to 10 pixels.
- C): Generate a map of strain energy, when analysing cell colonies.

Then your yaml file should look like this:

```
fig_parameters:  # setting plotting parameters

  cmap: "jet" # changing the colormap for the deformation plot

  vmin: None # minimal value displayed in the colormap, none means an automatic_
↪value for each frame
      # this applies to all plot types if none is specified
  vmax:      # maximum value displayed in the colormap
  deformation: 10 # applies only to deformation plots. All other plots use a_
↪default value (None)

plots: # defining which plots are produced
  colony: # you need to specify whether this is for "colony" or "cell layer"
    - "deformation" # this needs a list of values (marked by "-")
    - "traction"
    - "FEM_borders"
    - "stress_map"
    - "energy_points"
```

Hint: It is best to mark strings with quotation marks (“”). You can use None, True or False without quotation marks to set None or boolean values.

5.3 Overview of Analysis Parameters

Parameter	Default Value	Type	Significance
Main Parameters			
sigma	0.49	int,float	Poisson's ration of the substrate.
young	49000	int,float	Young's modulus of the substrate in Pa.
pixelsize	0.201	int,float	Pixel size of the images of the beads in μm .
window_size	20	int,float	Size of the windows for PIV (Particle Image Velocimetry) in μm .
overlap	19	int,float	Size of the overlap for PIV in μm .
FEM_mode	"colony"	string	Analyzing colonies or cell layer. This changes the behavior, concerning which masks are used, which plots are generated and what area is used for stress measurements.
Hidden Parameters			
std_factor	15	int,float	Additional filter for the deformation field. Deformations greater then (and σ : mean and standard deviation of the norm of deformations) are replaced by the local mean deformation.
edge_padding	0.1	float	All masks are cut of close to the image edge, i.e. if they are closer then $\text{edge_padding} \times \text{axis_length}$. For FEM analysis, all pixels at this edge are fixed so that no displacement perpendicular to the axis is allowed.
padding_cell_layer	0.2	float	If you are analyzing cell layers, and additional region close to the image edge is ignored when analyzing stresses, to avoid boundary effects. The effectively ignored region for cell layers is $\text{edge_padding} + \text{padding_cell_layer}$.
sigma_cells	0.5	float	Poisson's ratio of the cell sheet in the MSM algortihm. This parameter should have negligible influence on the resulting stresses.
min_obj_size	1500	int	Minimum size of an object (cell or cell colony). All masks are added up and all encircled areas are filled to determine the object size.
cv_pad	0	int,float	File names. Include the ending (e.g. ".png")
TFM_mode	"finite_thickness"	string	Using a TFM algorithm assuming either finite substrate thickness ("finite_thickness") for infinite substrate thickness ("infinte_thickness"). Always use "finite_thickness".

5.4 Overview of Plotting Parameters

Parameter	Default Value	Type	Significance
file_names	specific	string	File names. Include the ending (e.g. “.png”)
cmap	“rainbow”	string	Color maps. All matplotlib color maps are accepted.
vmin	None	float, int, None	Minimal value of the color bar. None for automatic selection.
vmax	None	float, int, None	Maximal value of the color bar. None for automatic selection.
Color bar Parameters			
cbar_style	“clickpoints”	“clickpoints” or “outside”	Specifies whether the color bar is plotted inside or outside of the image. Plotting the color bar outside will lead to misaligned images in clickpoints.
cbar_axes_fraction	0.2	float < 1	Height of the color bar when using cbar_style “outside”. This number signifies the fraction of the length of the original image axis.
cbar_width	“2%”	string	Width of the color bar when using cbar_style “clickpoints”. Has to be a string signifying the percentage of of the original image axis.
cbar_height	“50%”	string	Height of the color bar when using cbar_style “clickpoints”. Has to be a string signifying the percentage of of the original image axis.
cbar_borderpad	6	int	Distance between the color bar and the right image edge.
cbar_str	specific	string	Title of the color bar. Use quotation marks (“”) in the config file.
cbar_title_pad	10	int	Distance between the color bar and the color bar title.
cbar_tick_label_size	15	int	Size of the color bar tick labels.
Arrows in Deformation and Traction Fields			
filter_factor	1	float,int > 0	Factor that defines how many arrows are filtered out for plotting (traction and deformation fields). A high filter_factor means less arrows are plotted.
scale_ratio	0.2	float (0,1]	Length of the arrows (deformation and traction fields). Arrows are scaled so that the longest arrow has the length scale_ratio * longest image axis.
width	0.002	float	Width of the arrow shaft (traction and deformation fields).
headlength	3	float,int	Length of the arrow heads (traction and deformation fields).
headwidth	3	float,int	Width of the arrow head (traction and deformation fields)

+++

Plotting the Line Tensions			
background_color	"#330033"	string, tuple	Color of the background. Can be any color format accepted by matplotlib. You can use "cmap_0" to use the color of zero in the colormap used for the plot.
plot_t_vecs	False	bool	Plotting the line tension vectors.
plot_n_arrows	False	bool	Plotting the normal vectors of the cell boundary lines.
linewidth	4	int, float	Width of the lines representing the cell boundary lines.
border_arrow_filter	1	int	Filter defining how many arrows are plotted along the cell boundary lines. Only every n-th arrow is plotted, where n is the border_arrow_filter.
boundary_resolution	6	int	Smoothness of the lines representing the cell boundary lines. A high boundary_resolution means less smooth plotting. Very high values will cost a considerable amount of computation time.
Choosing which Plots are generated			
plots - colony	"deformation", "traction", "FEM_borders", "stress map"		List of plots that are produced in "colony" or "cell layer" mode.
plots - cell layer	"deformation", "traction", "FEM_borders", "stress map", "energy points"		List of plots that are produced in "colony" or "cell layer" mode.

CHAPTER 6

Note

If you encounter any bugs or other problems please report them [here](#) or contact me at andreas.b.bauer@fau.de.