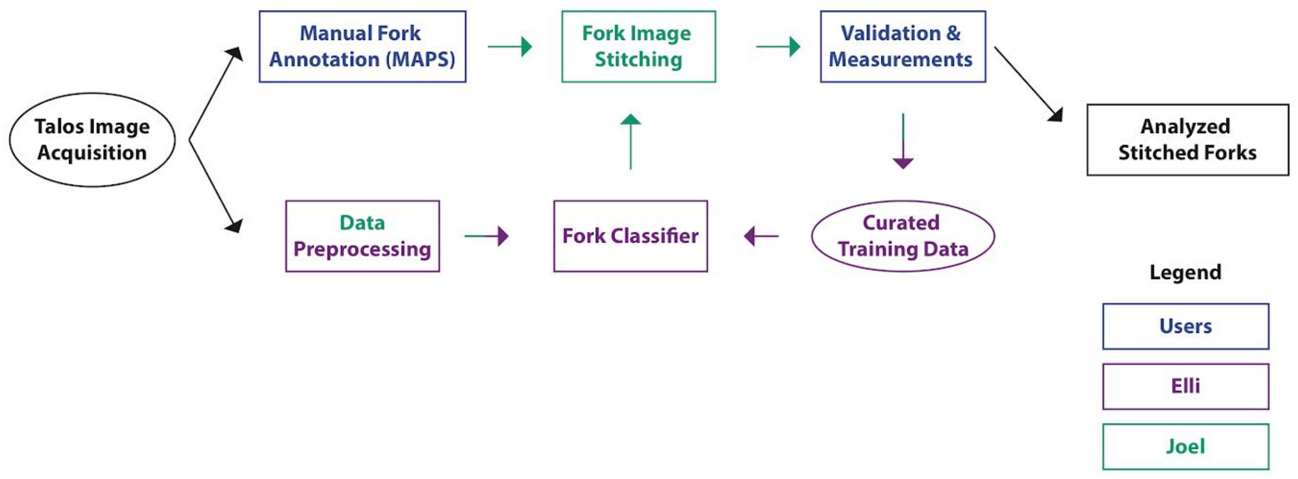
Talos Fork Detection Workflow

General Strategy



Known Issues (20/06/2019)

Logging: The logging window of the user interface does not show any logs from the multiprocessing part of the program (the actual stitching). Everything is logged to the command line and to the log file though, so the users can check the log entries there.

Multiprocessing: Currently the program only runs in multiprocessing when using the stitcher. Otherwise, it produces a jnius.JavaException: JVM exception occured: Could not initialize class net.imagej.LegacyImageMap. Therefore, even with 1 process, the stitching function is run as multiprocessing.

User Interface does not work on Mac: See here for ideas about how to work around this issue: https://github.com/imagej/pyimagej/issues/39

Dependency installation: The program installs Fiji and dependencies into the .m2 folder in the home directory. This should not be larger than 250 Mb, so hopefully won't interfere with the VMs. This is hardcoded in jgo and thus can't be adapted.

Stitching only works for 3x3 with 10% overlap: The stitching is hard coded for 3x3 and subsets (e.g. 2x3, 3x2 etc.) and 10% overlap. If one wants to generalize this, the positions of the images depending on image size, number of images to be stitched and overlap would need to be calculated. The necessary information is parsed from the MAPS xml file, but currently not given to the annotations.

Multiple annotations on the same image: If a user or the classifier makes multiple annotations on the same image, the stitcher processes them as separate images, thus creating an image per annotation (instead of putting multiple annotations on the same stitched image). This is based on the logic of the program. To get around this, one would need to check in the stitch_annotated_tiles function whether any tiles occur multiple times (and somehow deal with other batches as well).

Interface & Errors: There are different errors that only appear on the console and don't show a pop-up warning to the user. Adding more checks would be helpful for that.

MAPS for manual annotation

Initial situation

Th Lopes lab uses a TrackEM plugin to visualize the images from the Talos. TrackEM has multiple issues:

- Slow to process the experiment: Takes 2-3 days after imaging until it's ready
- Hard to maintain. Seems to have many hard-coded parameters that influence how things need to be required and that the lab doesn't know how to change.
- Crashes or fails often
- Creates big data overhead, approximately doubles the amount of raw data

Potential alternative: Switch to using MAPS for the visualization and the manual detection of replication forks. Use an automated stitching tool to stitch the images around the forks, which can then be looked at for quality control & confirmation that they are forks. The confirmed forks are then analyzed by the Lopes lab.

Developments & Learnings

8/5/19: Replacing TrackEM with MAPS is generally agreed upon, under the condition that there is an automated tool to extract the annotated forks from MAPS and stitch them. Joel will develop this tool.

8/5/19: The MAPS XML file is huge, badly documented and annoying to parse. See attachment MAPS XML Scheme for rough overview.

10/5/19: The MAPS XML File uses 2 coordinate systems: A global coordinate system for the annotations (in meters) and a relative coordinate system for the tiles (= images) within each layer (= square). These coordinate systems are rotated relative to each other at an arbitrary angle (often 90.02 degrees). All this information is contained in the metadata. Therefore, it is feasible to extract all necessary information about the manual annotations from MAPS. A first prototype of the MAPS XML parser gets the relevant information out: https://github.com/jluethi/ForkStitcher

15/5/19: The MAPS XML parser reliably gets the information about the annotations from different MAPS experiments.

17/5/19: The parser can now also calculate the position of all tiles, find out what tiles an annotation is on and calculate the position of an annotation within a given image. Therefore, it fulfills all the critical requirements to use a manual annotation workflow via MAPS

21/5/19: The MAPS XML parser is now fully documented (via Sphinx)

3/6/19: Fixed a bug that lead to some of the annotation positions being wrong (if multiple annotations were made on the same tile)

5/6/19: Made the parser more general, such that it can deal with nested LayerGroups. This is important because the images could be saved directly in a LayerGroup or in a LayerGroup within a LayerGroup. Also tested it on MAPS 3.7 (old experiments opened in the 3.7 viewer, saved and then processed) and in MAPS 3.8 (with a SEM experiment from Johannes. Could not actually calculate the locations, because things seem to be saved different for the SEM data. But it still parses the XML file fine and extracts all the information).

Stitching

Initial situation

The stitching needs to be moved from TrackEM to a python-based tool that gets inputs from MAPS annotations and runs reliable.

Developments & Learnings

9/5/19: Stitching directly in Python is not easily feasible. The best existing framework for this is OpenCV, see documentation here. But this framework is not built to handle images with camera translation (where the camera doesn't rotate, but it or the scene moves, see details here). When trying to apply it anyway according to the tutorial in the first link, it fails to find a stitching match and does not procude an image. On the other hand, the Fiji-based plugin from Preibisch does perform well on the EM data. Thus, we will focus on integrating this in the workflow.

13/5/19: I built a first stitching prototype using pyimagej as a framework to access the Preibisch stitching tool via Python that takes input from the XML parser. It stitches reasonably well, except for the lense distortion issues (see topic below). Current downsides of this implementation: It uses the ImageJ1 Plugin, which is somewhat unstable for multiple reasons. First, it saves a file to disk directly with a temporary name. This can lead to race conditions and has lead to the program crash when run overnight. Second, it depends on a local Fiji installation that does not break or get updated in a way that changes the plugin functionality. Thus, I have started to work on a version that uses the Java API on a lower level, see this github issue and this ImageJ forum conversation.

16/5/19: Changed the way the file is saved such that the python script directly saves it in the correct location and with the correct filename. This eliminates the issue with potential crashes due to the temporary file

20/5/19: We decided to save the stitched image as 8 bit, because there is little information in the extra dynamic range and it halves the required disk space for the stitched output. Also, the image should already contain the pixel size in its metadata (exif data). This is way harder in python than expected (most tools that can do this are either no longer supported, do not have TIFF/PNG support or are so hard to install that I can't manage like py3exiv2). Luckily, it is possible via pyimagej following the instructions here. Therefore, I decided to switch all the processing of the stitched image to pyimagej and away from other python solutions like opencv.

21/5/19: Working with people in the ImageJ forum to work out how to use the lower level APIs directly via pyimagej, see thread here.

22/5/19: I am having issues with memory management in pyimagej. It runs out of memory after ~45 images being processed when the images are displayed by the Grid stitching. When images were directly saved, this was not an issue (but there were other issues with the saved files having to be found, renamed and moved and sometimes not being found and thus crashing the program). Currently trying to solve this by reinitializing or clearing the memory of the imagej gateway in certain intervals. May also be less of an issue once lower level APIs can be used. See this forum post for details.

24/5/19: Stitching runs fine on intermediary datasets (~80 forks), but runs into memory issues on larger datasets (e.g. 500 annotations). I made the stitcher more modular and able to save csvs of annotations, divide it into batches, load batches and run them. First I thought the lower level APIs may not have those issues, as images aren't displayed but just assigned to a Java variable. This is not the case, even the lower level API calls lead to constantly increasing memory usage (e.g. using 140 GB of RAM when processing 6 images in parallel, even though it started only needing around 20GB). Then I thought I could work around the memory usage issues by relying on separate processes for each batch using pythons multiprocessing library. But unfortunately, pyimagej does not play well with the multiprocessing library, see my github issue here.

Another issue using multiprocessing is that processes can seemingly fail silently (or failure warning gets lost in the print mess from ImageJ) and do not escalate Exceptions as expected. Maybe also JAVA VM related.

24/5/19: Current workaround: The run_stitching_batches script can be run multiple times. The annotations are divided into multiple csv files and run_stitching_batches processes them. If everything works, the csv file is replaced with a renamed one. If something fails, it is left in the folder and will be tried again on a repetition of the run. This rerunning is currently not automated in python, as there are some issues with the pyimagej library and it is hopefully only a temporary solution.

27/5/19: Brought the stitcher fully into the lower level ImageJ APIs, which adds stability. Also brought the Stitcher into

28/5/19: Figured out how to manually close images through the ImageJ API, so that it runs much more stably now. Currently having issues with pyimagej when trying to parallelize using multiprocessing, see here.

29/5/19: Multiprocessing issue solved. Stitcher runs fast & stable now.

31/5/19: Started work on an interface for the stitcher. I'm using tkinter for this. Generally works cross-plattform. Currently pyimagej interfers with it when running on a Mac (pyimagej issue, see here), but it runs fine on Windows. As our production will run on Windows VMs, this is ignored for the moment.

4/6/19: Logging is an issue, pyimagej interferes with the logging of the stitcher. See details in this issue.

5/6/19: Interface now uses threading, such that it does not freeze while an experiment is processed.

6/6/19: Added an arrow overlay to the Stitcher, so that forks are more easily detected. The arrow can be turned on and off by showing/hiding the image overlay.

7/6/19: Added the functionality to do local contrast enhancement on the stitched images (see below for details on contrast enhancement).

12/6/19: Logging now works. It's a bit complicated due to the multiprocessing and using multiple modules, so it there is a static function in the MapsXmlParser that takes care of it now

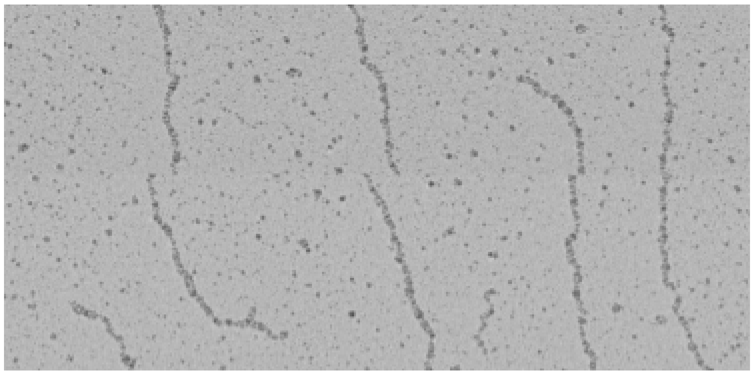
Lense Distortion Correction

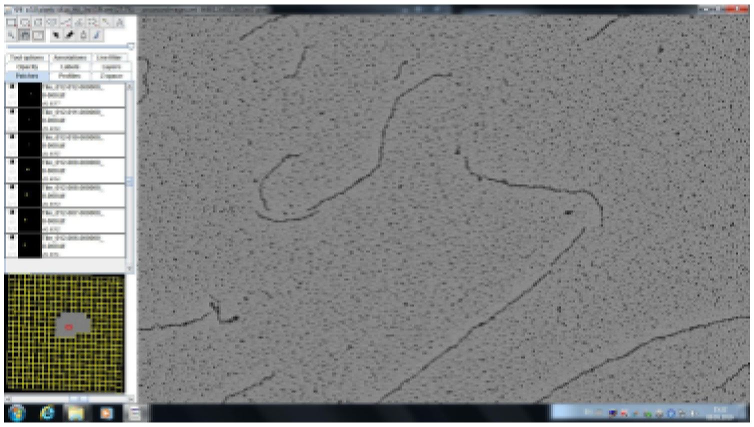
Initial situation

Stitching quality is limited by the lense distortion of the Talos. Stitching without correcting for this leads to artefacts like these. These artefacts should in most cases not interfere with the analysis of the fork, but would make the image unsuitable for showing in a publication.

Developments & Learnings

9/5/19: The artefacts observed with ImageJ stitching without lense distortion correction is similar to those observed in TrackEM. But Massimo & Jonas recommend that I invest time to fix this, as it would interfere with the usefulness of those images (sometimes their analysis,

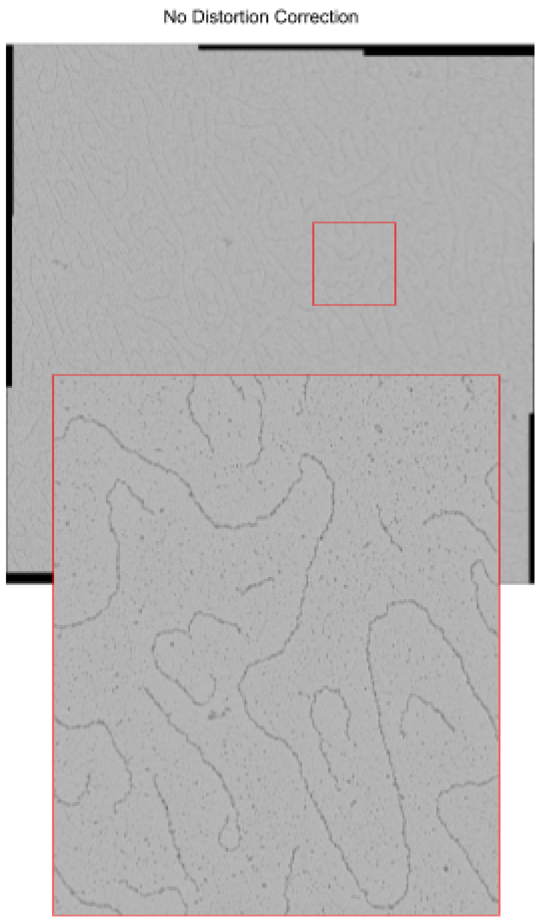

Stitching artefact from Python based Fork Stitcher, 20/5/19

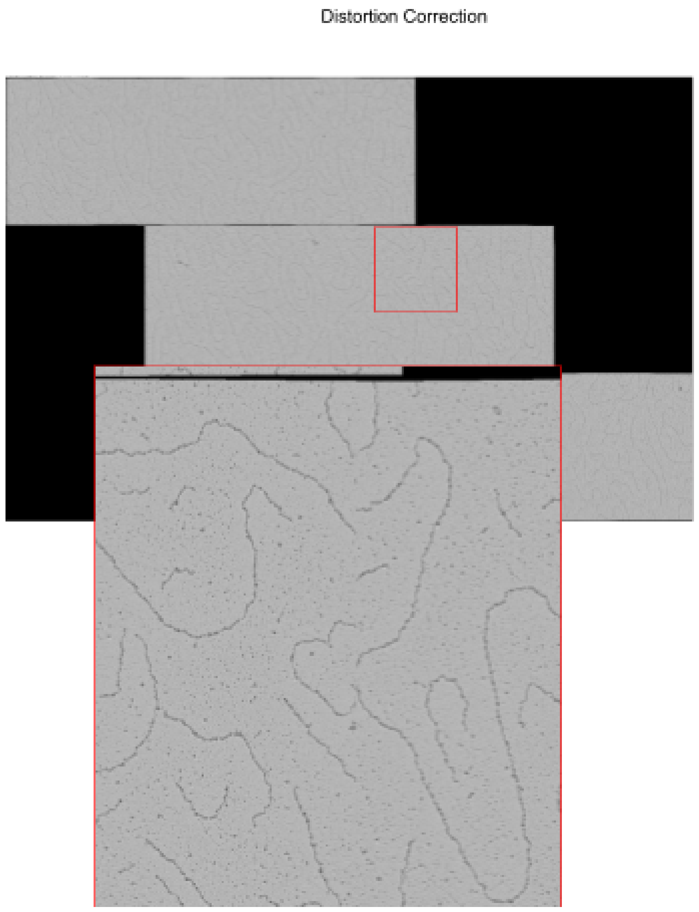

Stitching artefacts in TrackEM (provided by Sebastian, 9/5/19)

mostly that they could not be shown in a publication.

14/5/19: Acquired lense distortion calibration data from a special grid with small squares & straight lines at different focus levels.

16/5/19: Calculated the lense distortion with the ImageJ1 Plugin from Verena Kaynig because it allows for arbitrary images to be used in the calculation, as long as they have are 3x3 grids with 50% overlap. Alternatively, there is this python framework that could be worth looking into. Applying the correction to the image itselfs improves the stitching. Applying it to Talos data from early April makes the stitching worse. Thus, lense distortion correction from old calibration measurements is not feasible.





20/5/19: Acquired lense distortion calibration data directly from the DNA sample that was imaged in the same image acquisition (added three 3x3 grids with 50% overlap to the targets for acquisition).

21/5/19: Lense distortion can be calculated directly on Talos images of DNA if they are imaged with 50% overlap, 3x3 images (easy to add to an acquisiton).

22/5/19: In some of the new acquisitions, there seem to be fewer artefacts due to lense distortion. This makes this mostly an edge case issue. Therefore, it is not pursued with priority at the moment (decision from the meeting with Massimo on 22/5/19)

Quality Control & Fork Confirmation

Initial situation

Developments & Learnings

22/5/19: Created a short instruction for MAPS annotation and scoring of potential forks by users: https://docs.google.com/document/d/13fF9OVmjsIdmIIdtLgEyChKGl66SW4J-PkAGSYVtL0o/edit#

Preprocessing of Images for the Classifier

Initial situation

Local contrast enhancement is performed manually on images that need to be added to the training set. An automated method is required so that also new project data can be processed that way and go into the classifier after contrast enhancement.

Developments & Learnings

7/6/19: I built a tool that performs local contrast enhancement on images. One design criteria was that the contrast enhancement would be the same as what was manually applied to training images in the past. Even though we always talked about this contrast enhancement being CLAHE, the algorithm used was not CLAHE, but the NormalizeLocalContrast Plugin from Fiji. This performs a local contrast enhancement by measuring single pixel statistics for blocks of pixels, not by using histograms. I built a method in the Stitcher called local_contrast_enhancement that can perform CLAHE and NormalizeLocalContrast and save those images. This uses lower level ImageJ APIs directly, so that they can be called via pyimagej.

Links

Code repository: https://github.com/jluethi/ForkStitcher

Attachments

MAPS XML Scheme

MAPS XML Scheme:

PerseusProject:

- History
- LastAlignedOn
- LayerGroups
    - LayerGroup
    - LayerGroup
    - LayerGroup: LayersData\highmag\28000 (contains most of the XML file)
        - Layers: This are the layers of high mag data. There are 18 of them for the 18 squares
            - Contains MetaData Location => path to Metadata
            - Contains pixelSize
            - Contains Stage Position of the Square
    - LayerGroup: Text Layer 6 forks
        - => Contains all the annotations and their position
- QueuedPyramidUpdates
- QueuedTileOptomizations
- QueuedTileRegistrations
- applicationVersion
- currentHolder
- currentHolderDefinition
- description
- displayName
- fileSystemLocation
- guid
- projectVersion

- currentLayer
- lastFovRectangle
- QueuedJobDescriptors

The Layer Groups contain the AnnotationLayers, which contain the info about the annotation position (StagePosition) and their name (RealDisplayName). Can be in any LayerGroup.

The LayerGroup for the highmag contains most of the XML file and information about all the layers containing actual acquisitions.

MetaData Folder contains info about the stitching, another XML file and the different zoom levels of the image_pyramid. Each square has 1 metadata folder

MetaData XML File

StitchingDataStore

- registrations
    - _registrationsForSerialization
        - TileRegistration (many of them)
            - TileRegistration (each of them contains many)

- tileSet
    - HasUniformTileSizes
    - TileImageFolder
    - TileSize
    - _columns
    - _overlapX
    - _overlapY
    - _rows
    - _tileCollection: This contains some positional information