Supplementary information

JIPipe: Visual batch processing for *ImageJ*

Ruman Gerst[1,2,#], Zoltán Cseresnyés[1,#], Marc Thilo Figge[1,3,*]

[1] Applied Systems Biology, Leibniz Institute for Natural Product Research and Infection Biology – Hans Knöll Institute (HKI)

[2] Faculty of Biological Sciences, Friedrich-Schiller-University Jena, Germany

[3] Institute of Microbiology, Faculty of Biological Sciences, Friedrich-Schiller-University Jena, Germany

[#] These authors contributed equally

[*] Correspondence should be addressed to thilo.figge@leibniz-hki.de

Contents

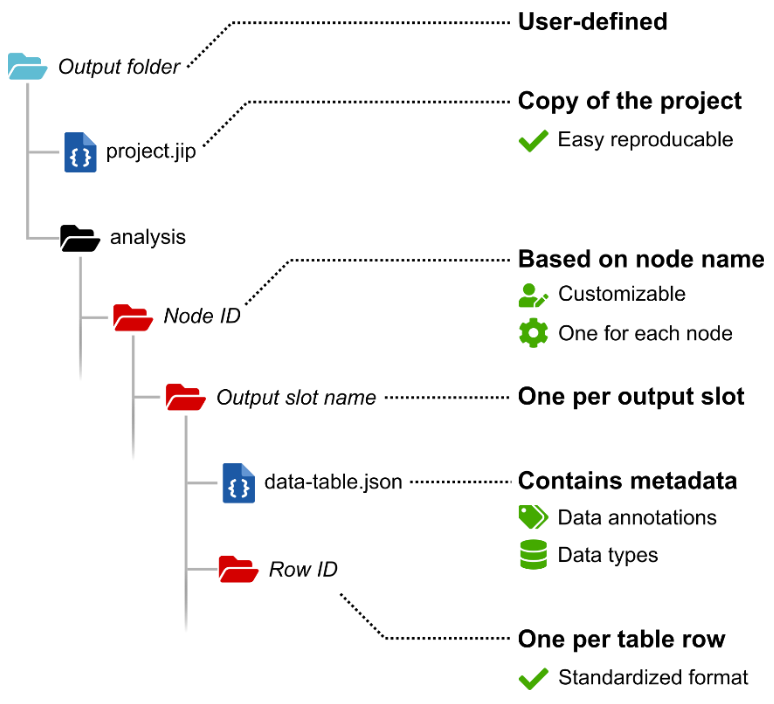
1 Symbiosis of *ImageJ* and *JIPipe*

1.1 Standardized output format

JIPipe writes results in a standardized format that allows results and annotations to be imported back into *JIPipe*. (see **Supplementary Figure 1.1**). The user only must provide the output folder. Here, *JIPipe* creates a sub-folder "analysis" that contains directories that correspond to the nodes in the graph. The name of these node folders is generated automatically and with de-duplication based on the user-customizable name of the node. This ensures that no data is overwritten, while users are still able to navigate through the results manually. Each node folder contains sub-directories that correspond to the output slots. *JIPipe* automatically ensures during the creation of these slots that they are compatible to filesystems and unique. Each of those slot folders contains metadata in a file "data-table.json". This file stores information about the data stored within the slot, annotations, expected data types, and true data types. Each row is also indexed with a unique identifier. Data is stored in sum-directories of the slot folder that correspond to this unique row ID. The format is defined by the stored data type and allows import back into *JIPipe*, as the metadata table provides all necessary info to direct *JIPipe* to the correct import routine. To improve the user experience, the metadata table is also present in CSV format that can be opened in standard software.

JIPipe makes use of this powerful result model by offering nodes that can import such results back into another analysis. The user only must provide the slot folder to allow *JIPipe* to import all data and metadata. This allows for postprocessing analyses that combine and structure results from multiple analyses.

To improve the usability of *JIPipe*, the standardized output format can be exported into a more commonly used where file names contain metadata. This feature is available within the cache browser, result viewer, and as dedicated node. This metadata-based export cannot be directly imported back into *JIPipe*.



Supplementary Figure 1.1 | Standardized result export format. *JIPipe* writes outputs in a standardized format. The user only must define the output directory (blue folder). *JIPipe* automatically generates a filesystem hierarchy based on the unique node Ids, output slot names, and row number in the output table (red folders).

1.2 Overview of *JIPipe* operations

JIPipe comes with a set of standard libraries that contain extensions for image analysis and other functions. It currently includes over 1000 nodes and over 120 data types. The function of the library components is briefly described here:

Filesystem library. A library of nodes that allow querying and manipulating file systems. It allows, for example, to search for files in a specified directory.

Annotation library. This library contains nodes that allow manipulation of data annotations.

Multi-parameter library. The core library provides the functions to execute a node on multiple parameter sets but lacks nodes to define such parameter sets. This library adds this functionality.

String library. A library that provides string data types (e.g., XML or JSON data).

***ImageJ* data type library.** This library integrates commonly used *ImageJ* functions, such as images, tables, and ROI management. Image data types are available in variants that restrict the bit depth or the dimensionality. Automated conversion is applied to ensure that the constraints are satisfied, for example, 8-bit grayscale images are converted into RGB images automatically. As each of these modes are available as separate data type, node developers and users can exactly control and review the inputs and outputs of a node, which improves usability and reduces the number of errors. This library also includes support for Bio-Formats[3].

***ImageJ1* algorithm library.** Commonly used commands from *ImageJ1* are integrated via this library. It provides functions to process images and ROI. The library also includes a macro node that can execute *ImageJ* macro code inside a *JIPipe* node.

***ImageJ2* algorithm library.** *ImageJ2* operations are automatically included via a translation layer.

CLIJ integration library. This library integrates functions from *CLIJ2*[1] into *JIPipe*. To improve performance, it provides a separate data type that encapsulates a GPU image and provides conversion from and to *ImageJ* images for ease of use. The functions were generated in an automated fashion via a Python script.

Table library. A library containing nodes that apply commonly used table operations (e.g., merging rows, or sorting). This library also adds data types that encapsulate only one table column for more advanced operations.

Forms library. Users can create interactive nodes that prompt the user to provide an input to the current data processing. There is an expandable set of such predefined input types available: Numeric inputs, check boxes, text fields, a choice of predefined values, and selecting a file system path. Multiple of these inputs can be connected into a form that is displayed for each processed data item. The standard forms library then writes user inputs into the annotations of the provided data. The library is modularized, which allows more complex form types, such as letting users draw or modify a mask interactively.

OMERO integration. *JIPipe* provides an integration into OMERO[2] that allows to query the database and download or upload images.

Python integration. *ImageJ* provides support for Python scripts via *Jython* (https://www.jython.org/) and a standard *Python* setup. The difference between *Jython* and Python is that *Jython* has access to all Java data types, including ones from *ImageJ* and *JIPipe* – while it is currently not possible to integrate C-based packages, such as *Numpy* or *Tensorflow*. To allow the integration of such powerful tools, *JIPipe* provides an environment system to integrate any existing Python environment. To increase usability of this approach, *JIPipe* also comes with one-click installers to setup new *Python* environments via *Conda*.

Python scripts communicate with *JIPipe* via a file-based API. *JIPipe* automatically includes a *Python* library into Python scripts to make use of any node-specific functionality, such as accessing inputs and writing outputs.

R integration. *JIPipe* utilizes an environment system similar to *Python* environments to integrate *R* scripts. Similar to *Python*, users will find nodes to integrate custom *R* scripts into the pipeline. Again, *JIPipe* provides a file-based API to communicate data and metadata with the R script.

Cellpose integration. We included the ability to run *Cellpose* into *JIPipe*. As *Cellpose* is a *Python*-based tool, we make use of the *Python* integration functionality. *JIPipe* supports segmentation with *Cellpose* on the provided pretrained cytoplasm/nuclei models or a custom model. We also included the ability to train new models – either from scratch or by retraining a custom or pretrained model.

Utility library. Miscellaneous functions, like interaction with *JIPipe* outputs, manual data conversion, and data table sorting.

1.3 Extension API

All non-core functionality is split into dedicated Java libraries that are usually distributed with the *core* library but can be left out for specialized distributions of *JIPipe* that focus on other usages like non-image-analysis workflows. If only the *core* library is loaded, *JIPipe* will contain no usable data types and nodes. Image analysis functions are provided in dedicated libraries as extensions. Extensions for *JIPipe* are *SciJava* plugins that provide the necessary metadata for *JIPipe*, for example the name, authors, and dependencies, and a `register()` function that executes all necessary steps to add additional functionalities. Following functions can be added via this function:

Data types. Java developers can add new data types into *JIPipe*. Data is organized into tables and annotated with additional string columns. To allow for automated reading and saving, data types must provide functions to export itself into a folder and be imported from an exported directory. Additionally, data types must be able to be displayed in the GUI, via a `display()` function and an optional preview method. Each data type has a unique identifier string that allows safe serialization of user-customizable slot configurations.

Node types. Node types are Java classes that contain the workload function. Slots are either added via Java annotations or created in the object constructor. Like data types, they have a unique identifier.

Data type conversions. *JIPipe* automatically applies trivial conversions (e.g., from a child class to one of its parent classes). Other conversions (e.g., converting a plot to an image) must be handled by a dedicated converter object that can be registered into *JIPipe*. The converter creates an edge within the conversion graph, which allows higher-order conversions with multiple steps.

Data type display operation. Each data object comes with a default function to display the data in the GUI (e.g., displaying an image in *ImageJ*). Additional display operations can be registered via an extension.

Data type import operation. An import operation imports data from a *JIPipe* result folder and displays it to the user. An example is the import and display of ROI.

Parameter type. Developers can register custom parameter types. Each parameter type requires a unique identifier, required for serialization of user-defined parameters, and a user interface.

Expression function. The set of functions available in expression parameters can be further expanded.

Table column operation. Independent of expression functions, there exist functions that apply one-to-one or integrating operations on table columns. This set can be expanded. Such operations are automatically available inside expressions.

Menu extensions. Developers can create custom menu entries for various uses. They can choose between multiple locations (e.g., “Project” menu or “Tools” menu).

An alternative to Java extensions is extensions provided as JSON files. They can be created via a user-friendly GUI from existing pipelines or sets of nodes and allows non-developers to create custom node types, akin to *ImageJ2* scripts or macros. Such extensions are loaded via a “JSON Extension Loader” Java extension that automatically scans the *ImageJ* plugins directory for valid extensions.

2 Hallmarks of *JIPipe* by representative applications

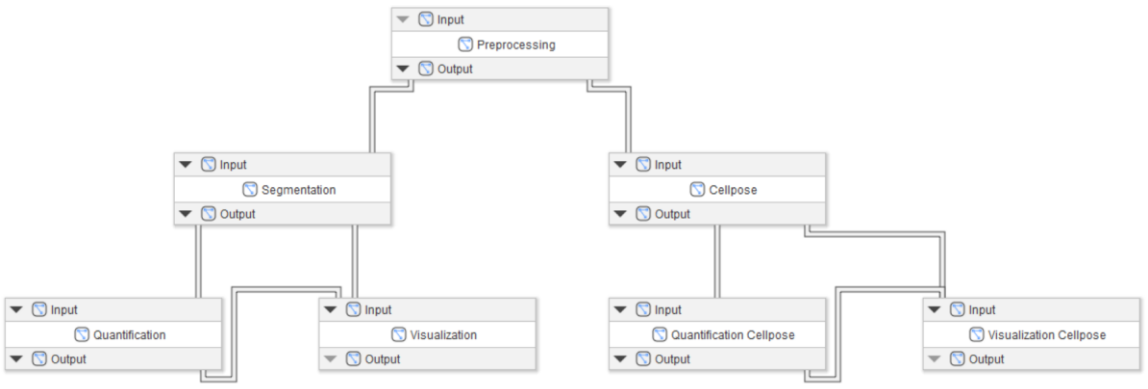
Here we describe the details of the *JIPipe* application that were utilized to illustrate the wide applicability of *JIPipe* in the Results. The examples are the following:

a) Bacterial growth inside fluid droplets
b) Nanoparticle delivery in liver
c) Host-pathogen interactions
d) Nematode viability test
e) Kidney status check via glomeruli counting

2.1 Bacterial growth measured in fluid droplets

Picoliter droplets are miniature bioreactors used in microfluidic experiments to test various growth conditions on bacteria[3–5]. Due to the extremely large number of droplet images, the native batch-processing and parallelization features made *JIPipe* an ideal candidate to quantify the bacterial growth in potentially millions of droplets[3] (**Supplementary Figure 2.1**). The workflow was successful in finding the targeted droplet, identify its inner zone and detect bacterial growth (**Figure 3 Row 1**). In this case, microfluidic droplets of approximately 100 micrometer diameter were filled with a solution containing *E. coli* bacteria and the bacterial growth was observed via brightfield transmitted light microscopy[3]. The following *JIPipe* workflow determines the droplets that show bacterial growth. When compared with a set of 1500 images with manual annotations (growth vs. no growth), the *JIPipe* workflow produced 100% agreement with the ground truth.

The processing workflow starts with scanning the input file folder(s) and annotating the internal *JIPipe* table with the folder names and the image identifiers. It is sufficient to drop only the top folder into the flow (node “Folder list”) because the subsequent nodes will automatically extract the rest of the information, e.-g. the subfolders (here we use the Recursive list option in the Parameters setting to handle multiple layers of subfolders automatically). In the “List files” node we search for files that are of the CZI type by introducing a filter for the absolute path. After adding the image names to the annotation table, we provide another filtering opportunity by the “Filter paths” node, which can be useful when limiting the analysis to a subset of images during testing the analysis workflow.



Supplementary Figure 2.1 | Compartment graph of the classical and Cellpose-based image analysis approach to identify microfluidic droplets that show bacterial growth inside. For the node arrangement within individual compartments, see the supplied JIP project file "Droplets.jip" and the detailed nodes map "compartments-figure-droplets.png" in Supplementary Materials.

In detail:

(i) Preprocessing:
- Read the images into memory and annotate the data table as described in **Supplementary Information 2.1**
- Pass the results to the Segmentation and Cellpose compartments

(ii) Segmentation:
- Hessian segmentation(parameters 2, "Largest", 2)
- Gaussian blur (2 px)
- Auto threshold (Triangle algorithm)
- Morphological hole filling
- Distance transform watershed
- Morphological opening (10 px)
- Morphological erosion (7 px)
- Create ROI and split multiple droplets to create the outer line
- Morphological erosion (5 px)
- Create ROI and split multiple droplets to create the inner line
- Pass the results to the Quantification compartment

(iii) Quantification:
- Filter ROI by statistics (Area above 1000)
- Calculate variance (1 px)
- Auto threshold (Otsu algorithm)
- Extract ROI measurements (Area, Area fraction, Integrated density)
- Create Growth column (percentage Area above 0.5 is classified as growth = 1)
- Pass the results to the Visualization compartment

(iv) Visualization:
- Convert growth areas into ROIs
- Merge raw image with growth area ROIs
- Convert inner line of droplets into ROIs
- Merge raw image with inner line ROIs

(v) Cellpose:

    - Access the TL images from Preprocessing
    - Run Cellpose segmentation
        - Object diameter 120 px
        - Use pretrained Cellpose model “Cytoplasm”
        - Thresholds: 2 (probability), 0.8 (flow)
    - Filter ROIs by Roundness > 0.7
    - Morphological opening (10 px)
    - Morphological erosion (1 px for outer line, 7 px for inner line of droplets)
    - Turn masks to ROIs and split them

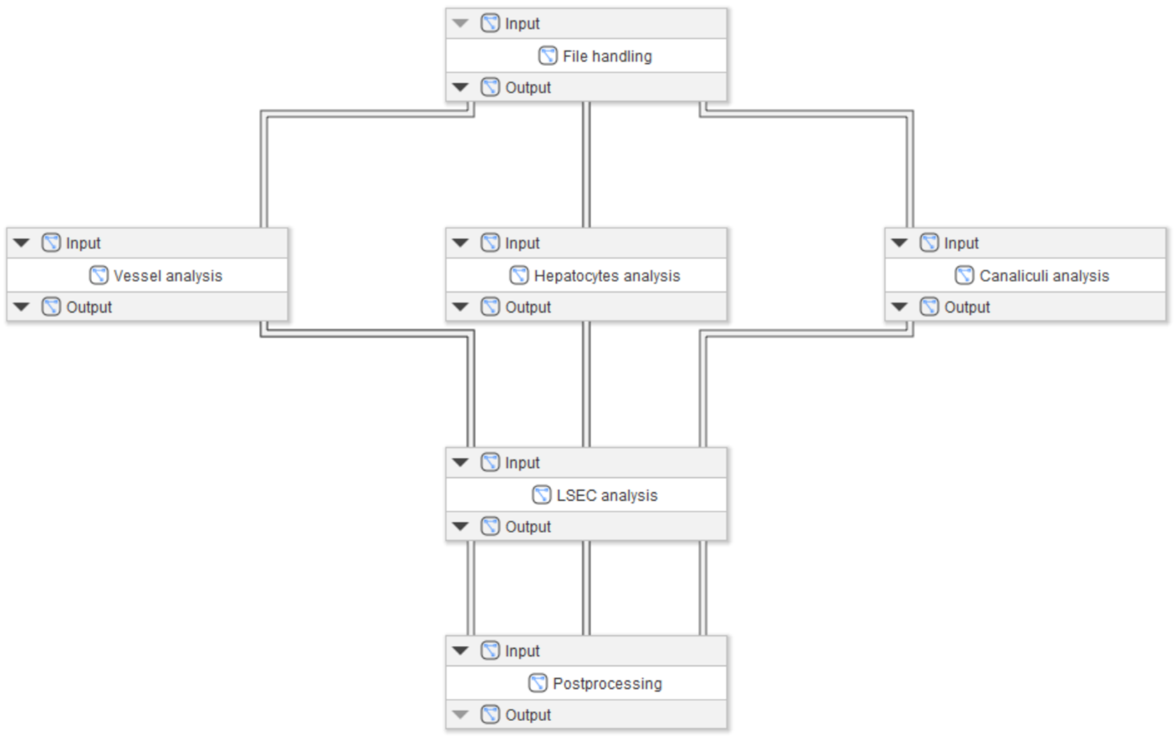
(vi) Quantification Cellpose: see Quantification

(vii) Visualization Cellpose: see Visualization

2.2 Nanoparticle delivery analysis in liver

JIPipe’s native batch processing ability and built-in time series algorithms were of particular advantage in a project using nanoparticles (NPs) to counteract liver fibrosis caused by non-alcoholic fatty liver disease. NPs are utilized to deliver precisely targeted agents to the liver tissue[6]. The analysis of such microscopy data requires the identification of various liver components including hepatocytes, liver sinusoidal endothelial cells (LSECs), sinusoids, and canaliculi. The segmentation was carried out without the help of specific labeling, and the extraction of time series information about the uptake and extrusion of the NP-delivered agents. The extensive set of morphological filters available in *JIPipe* were invaluable in identifying the various components of the liver without specific labelling, based solely upon the autofluorescence signal (**Supplementary Figure 2.2)**. The resulting high-fidelity segmentation of the LSECs, canaliculi and sinusoids (**Figure 3 D-2**) indicate the precision and utility of the *JIPipe* processing framework.

The live-animal microscopy experiments were described in Muljajew et. al., 2021[7]. Micelle nanocarriers were injected into the circulatory system of the mouse vie the tail veins. Two-photon microscopy was utilized to image the cargo delivered by the micelles to the hepatocytes, sinusoids, canaliculi and liver-sinusoidal endothelial cells the time-series images were analyzed by the JIP protocol “LiverAnalysis.jip” (see Supplementary Materials)



Supplementary Figure 2.2 | Compartment graph of the analysis protocol for liver drug delivery assay, designed to quantify the spatio-temporal distribution of nanoparticle-delivered cargo to various parts of the murine liver. For the node arrangement within individual compartments, see the supplied JIP project file " LiverAnalysis.jip" and the detailed nodes map "compartments-figure-liver.png" in Supplementary Materials.

The workflow was based on principles similar to those shown in the previous chapter. The processing consisted of six compartments: i) file handling, ii) blood vessel analysis, iii) hepatocyte analysis, iv) canaliculi analysis, v) the analysis of liver sinusoidal endothelial cells (LSECs), and vi) postprocessing.

In detail:

(i) Images were read into memory and the data table was annotated as described in **Supplementary Information 2.1**

(ii) Vessel analysis:
- Median blur (radius=5 pixels)
- Illumination correction (20 px)
- Auto threshold (Yen algorithm)
- Despeckle
- Morphological erosion (7 px)
- Particle finder (circularity range 0.0-0.4)
- Multi-node algorithm to arrange the time series based on slice numbers
- Calculate the intensity time series for the segmented vessel region
- Calculate the number of segmented objects per time point to check the segmentation
- Pass the results to the LSEC analysis compartment

(iii) Hepatocyte analysis:
- Median blur (radius=5 pixels)
- Illumination correction (20 px)
- Auto threshold (Li algorithm)
- Despeckle

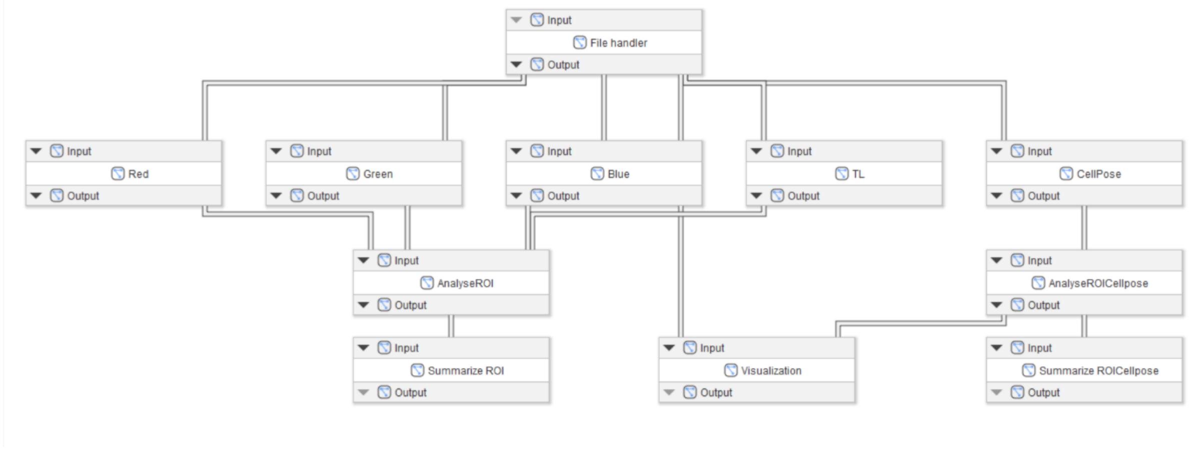
- Morphological erosion (1 px)
- Morphological skeletonize
- Particle finder (size range 10^2-10^6)
- Multi-node algorithm to arrange the time series based on slice numbers
- Calculate the intensity time series for the segmented vessel region
- Calculate the number of segmented objects per time point to check the segmentation

(iv) Canaliculi analysis
- Median blur (radius=5 pixels)
- Illumination correction (20 px)
- Auto threshold (Li algorithm)
- Despeckle
- Morphological erosion (1 px)
- Skeletonize
- Particle finder (no filtering)
- Multi-node algorithm to arrange the time series based on slice numbers
- Calculate the intensity time series for the segmented vessel region
- Calculate the number of segmented objects per time point to check the segmentation

(v) LSEC analysis:
- Take inputs from vessel analysis and file handler (fluorescence image)
- Illumination correction of fluorescence image (20 px)
- Create mask from segmented vessel image
- Morphological dilation (7 px)
- Mask fluorescence image with segmented vessel image
- Auto threshold (RenyiEntropy algorithm)
- Particle finder (no filtering)
- Multi-node algorithm to arrange the time series based on slice numbers
- Calculate the intensity time series for the segmented vessel region
- Calculate the number of segmented objects per time point to check the segmentation

2.3 Confrontation assays

The interaction between alveolar macrophages and fungal spores was examined as described in earlier research[6,8,9]. The macrophages and fungal spores were identified by label-free segmentation algorithms, whereas counterstained fungi were identified by fluorescence labeling[8–10]. The workflow was parallelized to segment labeled and unlabeled cells and spores separately. In addition, classical and deep learning—based approaches of the macrophage segmentation algorithms were also organized into separate parallel compartment groups (**Supplementary Figure 2.3**). The essential nodes consisted of Hessian filtering to identify unlabeled macrophages and fungal spores, background correction with appropriate parameters, thresholding steps, and fine-tuned morphological operators. Here the "Define multiple parameters" node was of high importance by allowing to test many parameters in one run. This special node allows the definition of one or more parameters that will be chosen to fit a processing node, with each parameter allowed to be given any number of values to be tested, and then connected to the corresponding processing node. For example, when testing various thresholding methods, a “Define multiple parameters” node was set up to contain the parameter "Method" with values set to the seventeen methods provided by *ImageJ*. The node was then plugged into an "Auto threshold 2D " node to provide an overview of the effectiveness of all seventeen methods on the test images in just one process. The outcome of the analysis consists of phagocytic measures and of segmented images of all participants (host cells, phagocytosed, adherent, and free pathogens, phagocytosing, and passive macrophages), see **Figure 3 D-5**. For the classification of the segmented objects, ROI-analysis nodes were developed; these enable the quantification of ROI overlap, e.g., between host cells and fungi to identify phagocytosed spores. A set of the ROI comparison nodes were arranged into a separate compartment "Analyze ROI", followed by a set of nodes to calculate the various phagocytosis measures arranged in the compartment "Summarize ROIs". For the deep learning—based segmentation of the macrophages, the recently published Cellpose[9] method was fully integrated into *JIPipe*. The Cellpose-related nodes include "Cellpose" (to apply the Cellpose model either in its original form, or after transfer learning, or following training from scratch); "Cellpose training" (for transfer learning and training a model from the beginning); "Import Cellpose model" and "Import Cellpose size model" to read in an already trained model for predefined size or for trained object size, respectively. Using Cellpose via these *JIPipe* nodes vastly simplifies the workflow building process, which is of great advantage for those with little experience in applying Deep Learning methods in image analysis.



Supplementary Figure 2.3 | Compartment graph of the confrontation assay analysis protocol, designed to quantify host-pathogen interactions. For the node arrangement within individual compartments, see the supplied JIP project file " ConfrontationAssay.jip" and the detailed nodes map "compartments-figure-confrontation.png" in Supplementary Materials.

In the example provided here, we limited the analysis to the "LabeledHosts_LabeledPathogens" dataset, which contained images where both the immune cells (hosts) and the fungal spores (pathogens) were imaged not only in transmitted light modality, but also with fluorescence microscopy using specific labeling of the assay components. The images are then read into memory with the "Import image" node, and the channels are separated before passing the data into the output node. As shown in **Supplementary Figure 2.3**, the output node is connected to the subsequent five segmentation compartments: i) antibody-labeled hosts ("Red"), ii) FITC-labeled pathogens ("Green"), iii) calcofluor white (CFW)-labeled pathogens ("Blue"), iv) transmitted light images ("TL"), and v) the deep-learning based segmentation workflow ("CellPose").

In detail:

(i) Images of the labeled host cells are processed as follows:
- Gaussian blur (radius=3 pixels)
- Internal gradient (25 px)
- Contrast enhancement
- Background subtraction (Rolling Ball, 50 px)
- Auto threshold (Triangle algorithm)
- Morphological closing (2 px)
- Morphological hole filling

- Watershed transformation
- Morphological erosion (5 px)
- Particle finder to identify macrophages by size (3000-30000) and circularity (0.1-1.0)

"Define multiple parameters" nodes were used originally to test a range of rolling ball radii, and a series of automated thresholding algorithms, respectively.

(ii) Images of the FITC-labeled fungi are processed as follows:
- Remove outliers (radius=20 pixels, threshold=5)
- Background subtraction (Rolling Ball, 22 px)
- Auto threshold (Triangle algorithm)
- Watershed transformation
- Particle finder to identify fungal spores by size (100-3000) and circularity (0.4-1.0)

Two "Define multiple parameters" nodes were used originally to test a range of rolling ball radii, and a series of automated thresholding algorithms, respectively.

(iii) Images of the CFW-labeled fungal cells are processed as follows:
- Remove outliers (radius=20 pixels, threshold=5)
- Contrast enhancement
- Background subtraction (Rolling Ball, 22 px)
- Auto threshold (Li algorithm)
- Watershed transformation
- Particle finder to identify fungal spores by size (100-3000) and circularity (0.4-1.0)

Two "Define multiple parameters" nodes were used originally to test a range of rolling ball radii, and a series of automated thresholding algorithms, respectively.

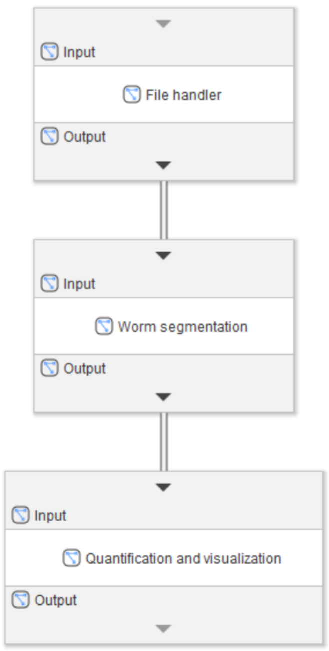
(iv) Images of the TL images of hosts and pathogens are processed as follows:
- Laplacian sharpening with a 3x3 kernel
- Hessian filtering using the smallest eigenvalues with smoothing of 3 pixels
- Gaussian blur (radius=5 px)
- Auto threshold (Huang algorithm)
- Annotating with maximum and minimum threshold values
- Morphological closing (2 px)
- Morphological hole filling
- Remove outliers (radius=10 pixels, threshold=20)
- Remove outliers (radius=20 pixels, threshold=20)
- Watershed transformation
- Morphological erosion (2 px)
- Particle finder to identify macrophages by size (3000-30000) and circularity (0.1-1.0)

(v) The TL and fluorescence images of hosts and pathogens were also segmented using the default trained networks of the Cellpose environment[15]. Here no pre- or post-processing steps were applied. Rather, the outcome from the Cellpose node provided the ROI lists of the hosts and pathogens, and the lists were passed on to the output node, from where they were directed to the "AnalyseROICellpose" compartment (see below).

The segmented images are used to generate lists of regions of interest (ROIs) that describe the locations of the host cells, as well as the green-labeled and blue-labeled pathogens. These ROIs are further examined in the "AnalyseROI" and "AnalyseROICellpose" compartments (the latter one applied for the Cellpose-based analysis) via testing the overlap between pairs of the three object groups to identify associated fungal spores (i.e., fungi that are interacting with a host cell based on their overlapping ROIs), adherent fungi (associated pathogens that are CFW-positive) and phagocytosed fungi (associated fungi that are not adherent, i.e. CFW-negative). In addition, phagocytosing host cells are identified as objects that contain at least one phagocytosed pathogen. In the last step, the "SummarizeROI" or "SummarizeROICellpose" compartments calculate the four phagocytic measures[12], using the "Modify tables" node that contain the calculations in a Python script.

2.4 Track analysis of unlabeled nematodes

When beneficial soil fungi are consumed by nematodes (earth worms), a way to protect the soil quality is to provide the fungi with symbiotic bacteria that produce agents that are toxic for the worms but not for the fungi, thus protecting the soil-enhancing fungi from the nematodes[11]. *JIPipe* was used to segment and track the nematodes, and to calculate a viability ratio (the total footprint area covered by a worm divided by the area of the worm averaged over time) that characterized the efficiency of various symbiotic bacteria to protect the fungi (**Supplementary Figure 2.4**). *JIPipe* was extended for this project with a node to find connected components that allowed the analysis of time series experiments. The outcome produced by the project included the merged outlines of every worm (**Figure 3 D-3)**, the binarized nematodes and the outline of one animal at a selected number of time points, as well as the footprint of a single worm superimposed onto the original images (**Figure 3 E-3)**. The postprocessing steps included the calculation of the worm areas and footprints (i.e., the collection of pixels that were touched by the worm during the course of the time series), and measuring the ratio between the footprint and the individual worm area, which measures the motility of the animal; the higher the ratio, the more motile the worm. The time-series images were analyzed by the JIP protocol "Nematodes.jip" (see Supplementary Materials).



Supplementary Figure 2.4 | Compartment graph of the kinetic analysis workflow designed to characterize nematodes according to their motility. For the node arrangement within individual compartments, see the supplied JIP project file "Nematodes.jip" and the detailed nodes map "compartments-figure-nematodes.png" in Supplementary Materials.

The workflow was based on principles similar to those shown in the previous chapters. The processing consisted of three compartments: i) file handling, ii) worm segmentation, iii) quantification and visualization.

In detail:

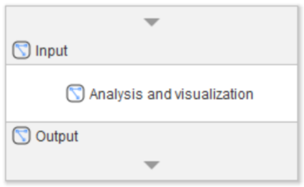
(i) Images were read into memory and the data table was annotated as described in **Supplementary Information 2.1**

(ii) Worm segmentation:
  - Splitting stacks to reduce data size for easier testing (optional)
  - Gaussian blur (3 px)
  - Auto threshold (Triangle algorithm)
  - Morphological closing (7 px, diamond)
  - Morphological hole filling
  - Particle finder (minimum size 4000 px)
  - Pass the results to the Quantification and visualization compartment

(iii) Quantification and visualization:
  - Time tracking individual worms based on the “#Component” annotation
  - Create and measure worm area with logical OR
  - Create total area per worm using the “Total” annotation
  - Calculate total area over individual worm area
  - Recreate time series using the “Slice” annotation in ascending order
  - Calculate average, standard deviation and count of area ratios

2.5 Kidney status check via glomeruli counting

Kidney diseases, e.g. nephrotoxic nephritis lead to a diminished function of the kidney tissue, indicated by the reduced number of glomeruli[12]. Light sheet microscopy can be utilized to image whole kidneys in 3D. These images were generated by staining the glomeruli, the functional units of the kidney. Due to the high dimensionality of the data and the number of glomeruli that can range up to 16000, manual counting is highly time-consuming and impractical. Therefore, our group already developed fully automated solutions in Python and C++[13]. The disadvantage of these tools is that they require programming to be adapted and improved. Here, we exemplify how *JIPipe* can be used to apply an equivalent analysis, but without the need for programming (**Supplementary Figure 2.5**). For this example, we reduced the size of the image stack from 700 to 20, which even non-workstation computers can process without computing and memory capacity problems. The outcome of the *JIPipe* analysis included the identification of individual glomeruli (**Figure 3 E-4**) and the outline of the entire kidney tissue. We provide the *JIPipe* protocol file, as well as the input data in the Supplementary Materials. Here we also demonstrate the use of a single-compartment configuration of a *JIPipe* workflow, combing all processing and visualization steps into a single space.



Supplementary Figure 2.5 | Compartment graph of the glomeruli analysis workflow designed kidney light sheet microscopy images. For the node arrangement within individual compartments, see the supplied JIP project file "kidney_example_pipeline.jip" and the detailed nodes map “compartments-figure-glomeruli.png” in Supplementary Materials.

The processing workflow is organized into three logical steps: i) file handling: these are nodes for reading and organizing the input images, which are then passed on to the processing nodes

ii) glomeruli segmentation nodes; iii) tissue segmentation and quantification nodes; iv) quantification of glomeruli; v) visualization nodes: the segmented ROIs are quantified and plots are generated. These plots include glomerular number bar diagrams and a histogram of the glomerular volumes per kidney. The final plots of the tissue and glomeruli outlines can also be directly accessed via bookmarks. To execute or visit the bookmarked nodes, go to *Project → Project overview* and find the bookmarks on the right side in the "Bookmarks" tab. Click on any bookmark and choose either "Run" (to execute the pipeline up to that node, inclusive), or "Go to bookmark" to visit the node directly.

In detail:

(i) File handling:
- o Images are provided as list of folders, containing the slices of an image stack. *JIPipe* converts these user-provided folders into a managed path data structure.
- o Folders are annotated with their name that will be used in the pipeline to distinguish images from each other
- o An "Import image stack" node is used to load the slices contained inside each folder into a 3D image. Annotations are preserved.
- o The imported images are passed to the segmentation nodes.

(ii) Glomeruli segmentation:
- o Input images are received from the output of the file handling nodes
- o White Top Hat (radius = 5, disk shape) is applied
- o Auto threshold (Otsu method)
- o Morphological opening (radius = 2, disk shape)
- o "Find Particles 2D" (default settings)
- o "Split into connected components" is applied to the set of 2D ROIs generated by the particle finder. This node applies a 3D connected components algorithm and groups 2D ROIs of the same component into dedicated ROI lists. Each output ROI list is annotated with an identifier and corresponds to one glomerulus.
- o The glomeruli are passed to the quantification and visualization nodes

(iii) Tissue segmentation and quantification:
- o Input images are received from the output of the file handling nodes
- o Median blur (radius = 1) is applied
- o Auto threshold (Default method)
- o Morphological closing (radius = 20, disk shape)
- o Morphological hole filling
- o Find particles (default settings)

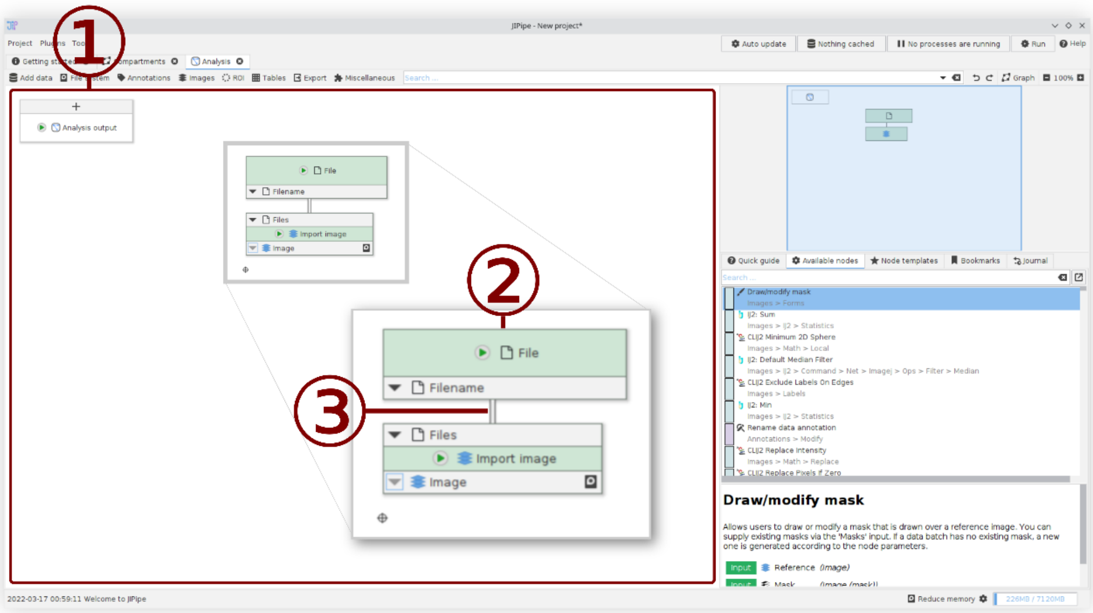
(iv) Quantification of glomeruli:
- o "Extract ROI statistics" (Extracted measurements = "Area") creates a table with one row per 2D ROI containing its area
- o "Integrate table columns" (Input column = Area, Function = Sum, Output column = Volume) calculates the volume in px^3 for each glomerulus. Its output is a table with one row
- o "Add annotations as columns" (Annotation name filter: value == "#Component") adds the glomerulus identifier into each table
- o "Merge table rows" (Data batches/Grouping method = "Custom", Data batches/Custom grouping columns = "#Dataset") merges all quantified results of the same kidney into one table
- o "Filter table" (Volume >= 28 AND Volume <= 2300) removes glomeruli outside the expected volume range

(v) Visualization

- The distribution of glomerular volumes is plotted via “Plot tables” (Plot type = Histogram, Value = Volume) plots the “Volume” column as histogram
- The tissue is visualized via a “Convert ROI to RGB” node that consumes the extracted tissue ROI and the raw input image enhanced via a “Histogram-based contrast enhancer”. A “Change ROI properties” node modifies the ROI to be drawn in green
- A combined visualization is generated as follows:
- The glomerular ROIs are modified via “Change ROI properties” to be drawn as yellow lines
- “Merge ROI lists” combines the glomerular ROIs and the tissue ROI into a single list
- A “Convert ROI to RGB” node overlays the glomeruli and tissue ROIs on top of the tissue

3 *JIPipe* user interface and data model

Here we focus on key features of the *JIPipe* user interface and explain how our software implements a scalable data model. The full organization of our software can be retrieved from the Supplementary Material as well as from the *JIPipe* website (http://www.JIPipe.org/). Familiarizing with the user interface is assisted by numerous training videos that can be accessed at the website as well. The central component of the *JIPipe* GUI is a graph that contains all functional units in form of nodes (see **Supplementary Figure 3.1**). Each of these nodes has one or multiple input and output slots that represent the data entered and produced by this functional unit (see **Supplementary Figure 3.2**). To create a pipeline, these slots are connected via edges to indicate a transfer of data from one node’s output to another node’s input. Outputs can be connected to multiple inputs, e.g., for creating branches to apply different methods or to generate visualizations of intermediate steps.



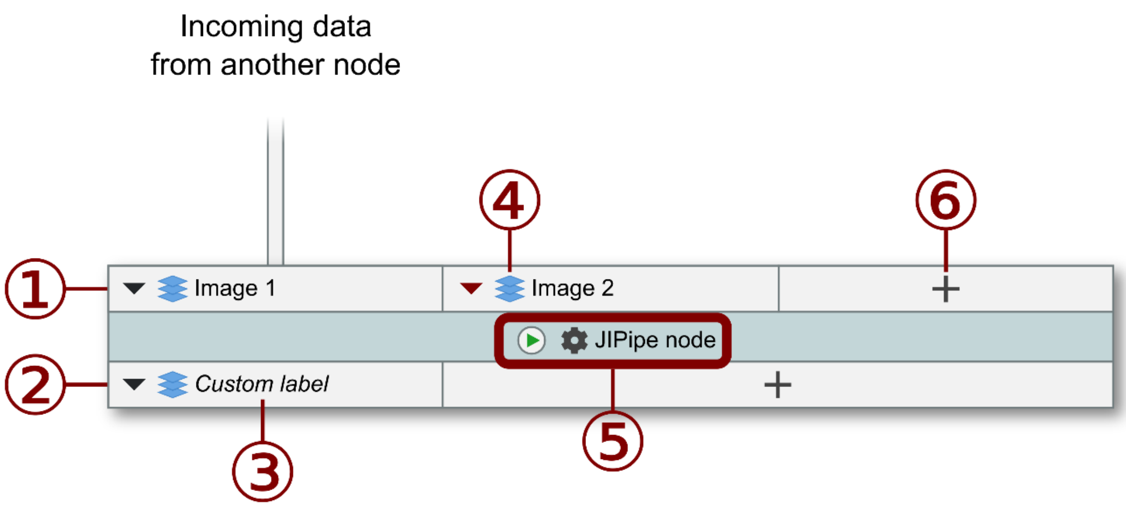
Draw/modify mask

Allows users to draw or modify a mask that is drawn over a reference image. You can supply existing masks via the 'Masks' input. If a data batch has no existing mask, a new one is generated according to the node parameters.

Input Reference (Image)
Input Mask (Image (mask))

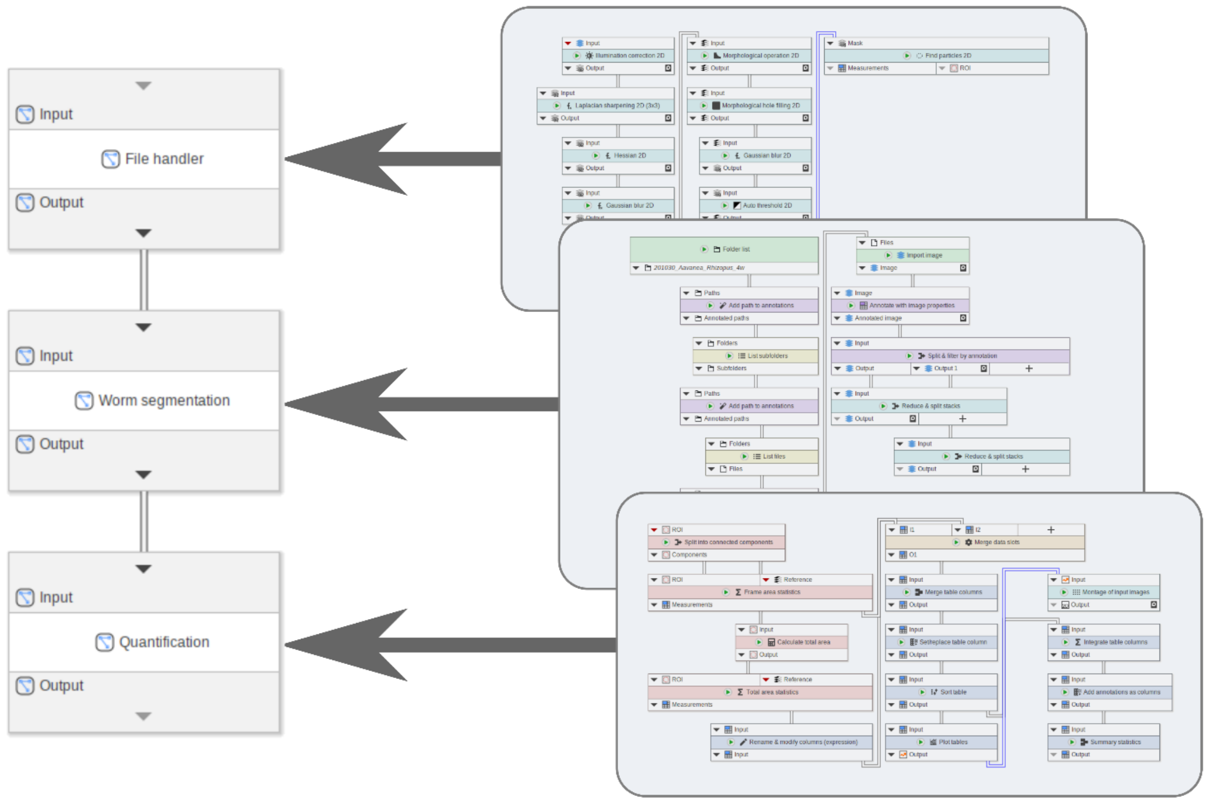
2022-03-17 00:59:11 Welcome to JIPipe

Supplementary Figure 3.1 | *JIPipe* graph editor UI. ① The central graph area where operational nodes can be placed by the user. ② Graphical representation of a node in *JIPipe*. Nodes have one or multiple input and output slots (grey areas within each node). ③ Output slots can be connected to inputs via edges (gray line).



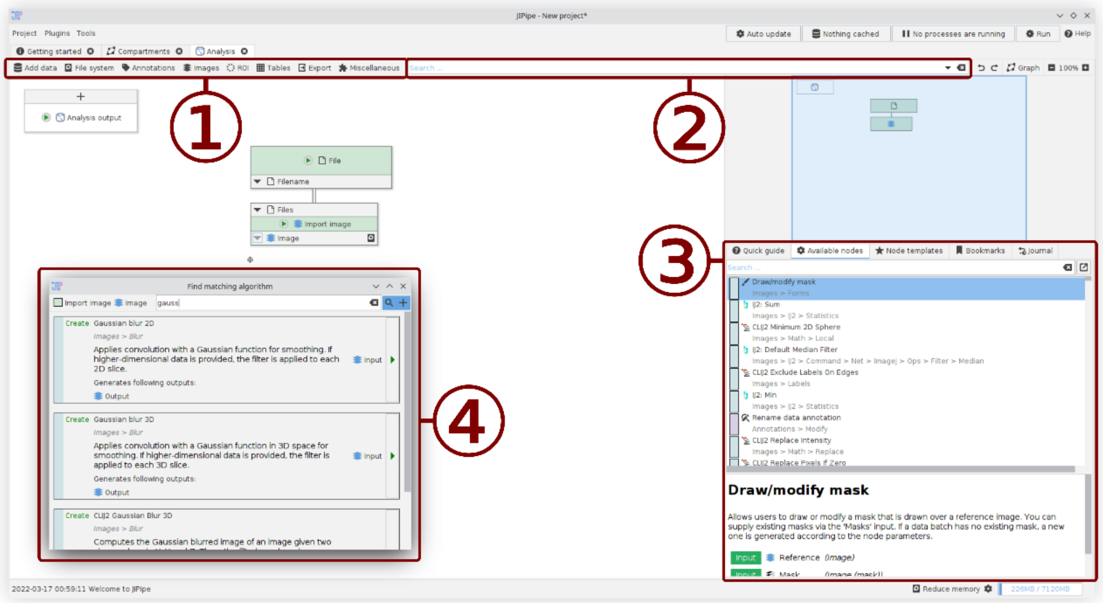
Supplementary Figure 3.2 | Graphical representation of a node with two inputs and one output slot. Only one input is connected (grey line). ① Inputs of the node are in the top row. ② The bottom row contains the node's outputs. ③ Users can customize the label names. These are displayed in an italic style. ④ Each slot displays its supported data type as icon. ⑤ The middle row contains the customizable node name, its icon representation, and a button to run the node and its predecessors. ⑥ Various nodes allow the creation of custom slots by clicking the "+" button.

Users are able to freely compartmentalize their pipelines (see **Supplementary Figure 3.3**), for example into preprocessing, segmentation, and postprocessing (see **Fig. 2a**). Within compartments, users can add additional nodes via a menu and arrange them freely.



Supplementary Figure 3.3 | Compartmentalization of pipelines. Users can use a compartment graph to organize a pipeline. Compartments are created and connected via a dedicated compartment graph (left). Each node in this structure contains a space where functional nodes can be placed (right).

JIPipe provides over 1000 nodes that include tools for data management and generation; mapping the file and folder structure of the data; annotation tools to keep track of file origins and experimental conditions; image processing nodes; ROI management; table processing and filtering; nodes to export the results in various formats and structures; and a group of additional miscellaneous nodes to add utilities to the system. Multiple ways are provided to search for specific nodes (see **Supplementary Figure 3.4**), e.g., by name, functionality, and compatibility to the preceding node. According to the symbiotic principle outlined earlier, these nodes can also be directly accessed from *ImageJ*.



Create Gaussian blur 2D
Images > Blur
Applies convolution with a Gaussian function for smoothing. If higher-dimensional data is provided, the filter is applied to each 2D slice.
Generates following outputs:
Output
Input

Create Gaussian blur 3D
Images > Blur
Applies convolution with a Gaussian function in 3D space for smoothing. If higher-dimensional data is provided, the filter is applied to each 3D slice.
Generates following outputs:
Output
Input

Create CLIJ2 Gaussian Blur 3D
Images > Blur
Computes the Gaussian blurred image of an image given two

Draw/modify mask

Allows users to draw or modify a mask that is drawn over a reference image. You can supply existing masks via the 'Masks' input. If a data batch has no existing mask, a new one is generated according to the node parameters.

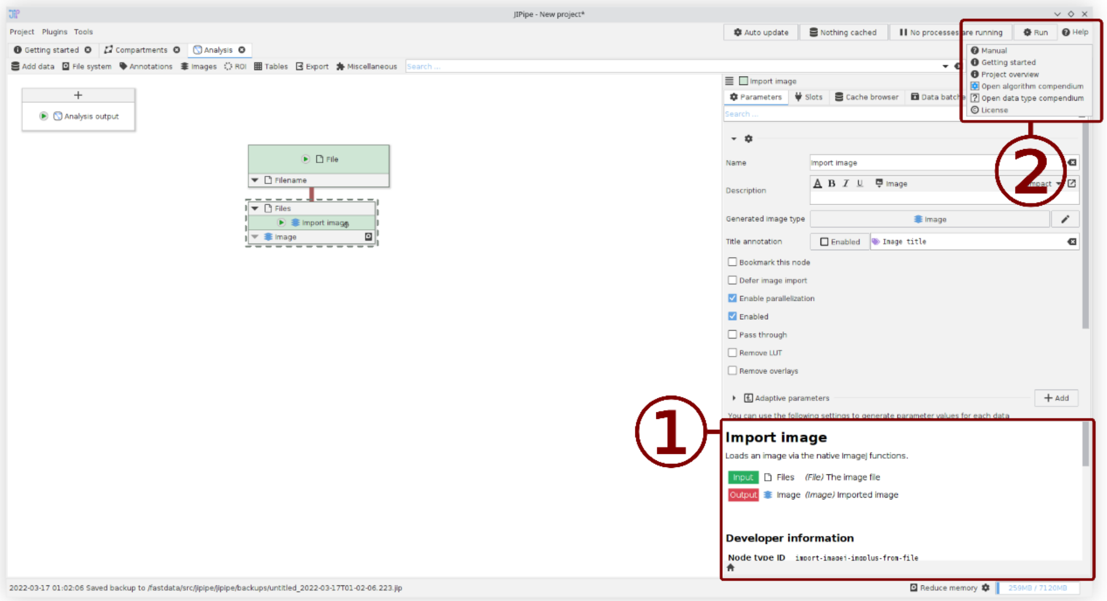
Input Reference *(Image)*
Input Mask *(Image (mask))*

2022-03-17 00:59:11 Welcome to JIPipe

Supplementary Figure 3.4 | GUI functions to add nodes into a pipeline. ① Nodes are organized into a menu. ② New and existing nodes can be searched via a search bar. ③ Users who are familiar with other visual programming languages find a toolbox where nodes can be dragged into the graph. ④ Each input and output provide an "Algorithm finder" feature that lists all compatible sources or targets based on their data type.

Nodes can be intuitively connected into a pipeline by creating edges between them via using the mouse. Alternatively, an algorithm finder can be used to locate nodes that match the data. Unique to *JIPipe*, all nodes are self-documenting, meaning that users can infer the functionality of the nodes and their slots without referencing a manual (see **Supplementary Figure 3.2**). The nodes are fully customizable by the user, thus simplifying the execution of multi-parameter sets.

A comprehensive context-sensitive documentation of all nodes and their parameters can be accessed any time. Alternatively, *JIPipe* includes complete documentations for nodes and data types that can be exported to HTML, PDF, or text files (see **Supplementary Figure 3.5**). To provide users of more complex nodes with a starting point, we implemented minimal examples that also reveal syntax details. Additionally, *JIPipe* nodes are capable of automated parameter validation to warn users about possibly invalid inputs.



Import image

Loads an image via the native ImageJ functions.

Input Files *(File)* The image file

Output Image *(Image)* Imported image

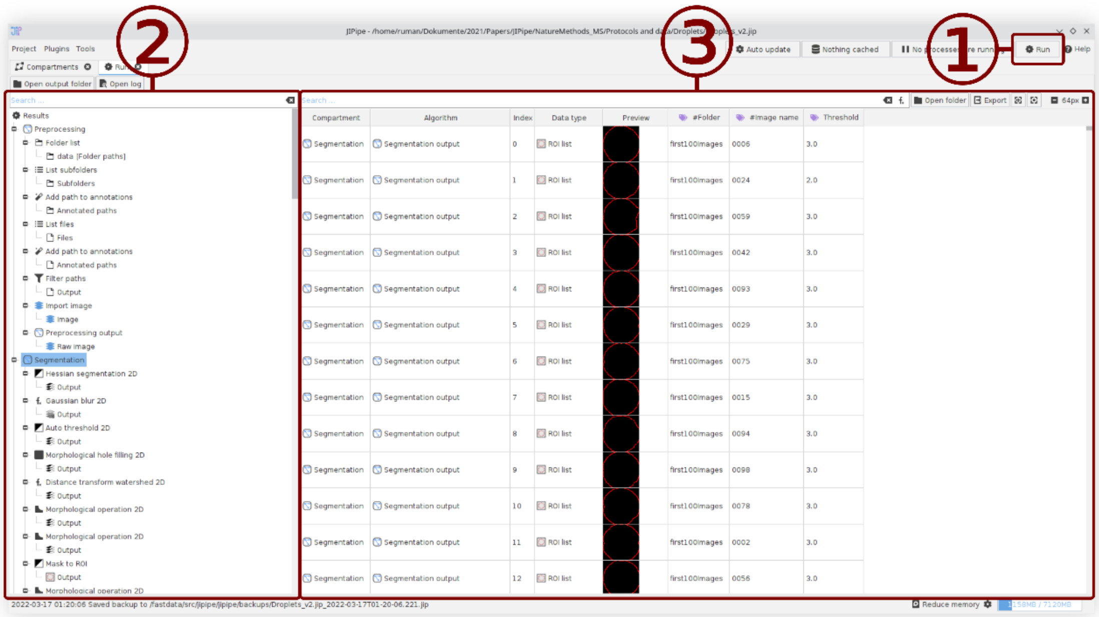
Developer information

Node type ID import-imagej-imgplus-from-file

2022-03-17 01:02:06 Saved backup to /fastdata/src/jipipe/jipipe/backups/untitled_2022-03-17T01-02-06.223.jip

Supplementary Figure 3.5. | Integrated documentation. ① The node parameter editor provides access to a brief description of the selected node. Parameter documentations are context sensitive and displayed after hovering the parameter control. ② The "Help" menu allows access to full documentations for nodes and data types.

A pipeline is executed by clicking the "Run" button, which will automatically validate the project, and offer options for optimization and multi-threading. The user only has to set an output directory and confirm the settings. *JIPipe* will automatically execute the project and store all results in a standardized format, together with the parameters, as a full project file (see **Supplementary Figure 1.1**. Afterwards, the results are displayed in a separate interface that allows to review and export the data (see **Supplementary Figure 3.6**). Due to the standardized output format, *JIPipe* can open existing results in this viewer, even if the analysis was not applied on the same machine. To work on data interactively, *JIPipe* allows the execution of a selected node where results are cached inside the *random-access memory* (RAM) to be accessed later from within the GUI and displayed in the appropriate tool, e.g., images are displayed by *ImageJ* or other tools, or re-used by another *JIPipe* run. To improve the usability of caching, *JIPipe* comes with cache-aware viewers for common *ImageJ* data that automatically update themselves to the newest cached versions, display metadata, and allow to browse through all results efficiently.



JIPipe - /home/ruman/Dokumente/2021/Papers/JIPipe/NatureMethods_MS/Protocols and data/Droplets/Droplets_v2.jip

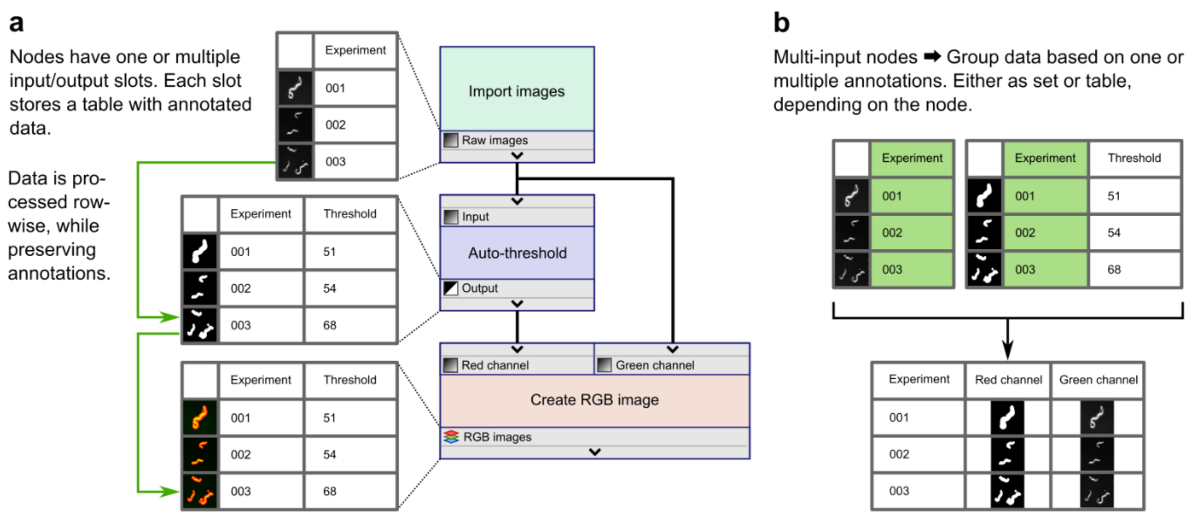
```
Results
⊟ Preprocessing
    ⊟ Folder list
        └ data [Folder paths]
    ⊟ List subfolders
        └ Subfolders
    ⊟ Add path to annotations
        └ Annotated paths
    ⊟ List files
        └ Files
    ⊟ Add path to annotations
        └ Annotated paths
    ⊟ Filter paths
        └ Output
    ⊟ Import image
        └ Image
    ⊟ Preprocessing output
        └ Raw image
⊟ Segmentation
    ⊟ Hessian segmentation 2D
        └ Output
    ⊟ Gaussian blur 2D
        └ Output
    ⊟ Auto threshold 2D
        └ Output
    ⊟ Morphological hole filling 2D
        └ Output
    ⊟ Distance transform watershed 2D
        └ Output
    ⊟ Morphological operation 2D
        └ Output
    ⊟ Morphological operation 2D
        └ Output
    ⊟ Mask to ROI
        └ Output
    ⊟ Morphological operation 2D
```

Compartment	Algorithm	Index	Data type	Preview	#Folder	#Image name	Threshold
Segmentation	Segmentation output	0	ROI list		first100Images	0006	3.0
Segmentation	Segmentation output	1	ROI list		first100Images	0024	2.0
Segmentation	Segmentation output	2	ROI list		first100Images	0059	3.0
Segmentation	Segmentation output	3	ROI list		first100Images	0042	3.0
Segmentation	Segmentation output	4	ROI list		first100Images	0093	3.0
Segmentation	Segmentation output	5	ROI list		first100Images	0029	3.0
Segmentation	Segmentation output	6	ROI list		first100Images	0075	3.0
Segmentation	Segmentation output	7	ROI list		first100Images	0015	3.0
Segmentation	Segmentation output	8	ROI list		first100Images	0094	3.0
Segmentation	Segmentation output	9	ROI list		first100Images	0098	3.0
Segmentation	Segmentation output	10	ROI list		first100Images	0078	3.0
Segmentation	Segmentation output	11	ROI list		first100Images	0002	3.0
Segmentation	Segmentation output	12	ROI list		first100Images	0056	3.0

```
2022-03-17 01:20:06 Saved backup to /fastdata/src/jipipe/jipipe/backups/Droplets_v2.jip_2022-03-17T01-20-06.221.jip
```

Supplementary Figure 3.6 | Result analysis GUI. ① The interface is opened after a successful pipeline run. Alternatively, *JIPipe* can open existing directories. ② Results are organized by their compartment, node, and output slot. On selecting an entry, the corresponding data is displayed. ③ Data is displayed as table, containing information about the compartment, node, slot, index within the data table. Text and data annotations are displayed as well. The main data item is previewed.

JIPipe organizes data into tables that are associated to each slot (see **Supplementary Figure 3.7a**). Each table has one column containing binary data of a type defined by the slot, and an arbitrary number of metadata columns containing strings or other data. There are various nodes available that generate or modify the set of metadata. For example, it can be used to track biological conditions, dataset identifiers, or image properties. The flexibility of this approach allows the easy management of research data and assists users in finding and reproducing data and analysis details according to the FAIR principles[3]. Generally, nodes iterate over the rows of the table and generate one result per row; this strategy also provides the opportunity to parallelize computationally expensive workloads. Another benefit of this design is zero-cost up- and downscaling: Users only need to modify the set of input files or folders to change the scale of the analysis without the need for updating the pipeline structure. During the processing, metadata is conserved. These annotations are helpful for the postprocessing and review steps but are also actively used by various algorithms that iterate through multiple inputs or merge data (see **Supplementary Figure 3.7b**).


a
Nodes have one or multiple input/output slots. Each slot stores a table with annotated data.
Data is processed row-wise, while preserving annotations.
Experiment
001
002
003
Threshold
51
54
68
Import images
Raw images
Input
Auto-threshold
Output
Red channel
Green channel
Create RGB image
RGB images
b
Multi-input nodes ➡ Group data based on one or multiple annotations. Either as set or table, depending on the node.
Experiment
Threshold
Red channel
Green channel

Supplementary Figure 3.7 | *JIPipe* data model. (a) Each node in a pipeline (right) has multiple inputs and output slots (gray box). Each slot contains a table of binary data (left), annotated with additional string columns (e.g., "Experiment", "Threshold"). A connection between two slots (black lines) leads to the data being passed to the input and processed row-wise (green arrows). The annotations are preserved. (b) A node with multiple inputs (Figure 4a, "Create RGB image") groups data (left) by testing for the equivalence of annotation sets (green highlight). The resulting grouped table (right) is processed row-wise.

4 Online training and documentation resources

We already provide a substantial amount of online documentation that simplify the process of adapting *JIPipe* into a bioimage analysis workflow, develop plugins and extensions, and to connect our software to third-party tools.

4.1 User guide and tutorials

To provide resources for new users of *JIPipe*, we created both step-by-step tutorials in text and video form, as well as documentations that guide users through the features of the user interface. As users of *ImageJ* might be new to the concept of visual programming and are possibly unaware of the benefits gained by utilizing our software, we created a video abstract that explains these aspects within three minutes (see https://www.youtube.com/watch?v=Zyl52bluWYI). All tutorials are listed on https://www.jipipe.org/tutorials/ and already include the following items:

- A step-by-step tutorial guiding through a basic image analysis task with batch processing
  - Text (25 steps): https://www.jipipe.org/tutorials/analysis/
  - Video (9:25 minutes): https://www.jipipe.org/tutorials/analysis_video/)
- A comprehensive tutorial that compares the analysis workflow between *ImageJ* and *JIPipe*
  - Video (22:36 minutes): https://www.jipipe.org/tutorials/jipipe-for-imagej-users/
- A short overview of the *JIPipe* user interface
  - Video (4:35 minutes): https://www.jipipe.org/tutorials/guide-user-interface/
- A brief explanation of *JIPipe*'s data caching feature
  - Video (4:16 minutes): https://www.jipipe.org/tutorials/guide-data-caches/
- An explanation of the graph editor features
  - Video (3:48 minutes): https://www.jipipe.org/tutorials/guide-graph-editor/
- A tutorial that explains the setup of a batch analysis and the backgrounds of *JIPipe*'s data management
  - Video (7:47 minutes): https://www.jipipe.org/tutorials/guide-batch-processing/
- A guide through the design of a custom node via the *JIPipe* GUI:
  - Text (10 steps): https://www.jipipe.org/tutorials/extension/

Information that is not covered by our tutorials is made available in the text documentation (see https://www.jipipe.org/documentation/) that covers the following topics:

- The basic concepts behind *JIPipe*
    - The basic concepts of visual programming with focus on users familiar to *ImageJ*: https://www.jipipe.org/documentation/basic-concepts/visual-programming/
    - An overview of the batch processing functionality with illustrations to explain the concepts behind it: https://www.jipipe.org/documentation/basic-concepts/batch-processing/
- Information about important GUI and *JIPipe* features related to designing pipelines
    - An overview of the graph editor user interface: https://www.jipipe.org/documentation/create-pipelines/pipeline-editor/
    - A detailed explanation of *JIPipe*'s expression system with illustrations, examples, and a list of operators and their precedence: https://www.jipipe.org/documentation/create-pipelines/expressions/
    - An explanation of the purpose of graph compartments: https://www.jipipe.org/documentation/create-pipelines/compartments/
    - A guide through the node grouping functionality: https://www.jipipe.org/documentation/create-pipelines/groups/
    - An explanation on the usage of loop nodes: https://www.jipipe.org/documentation/create-pipelines/loops/
- Guides relating to running pipelines and reviewing results
    - A brief guide on how to run a pipeline: https://www.jipipe.org/documentation/run-pipelines/run/
    - A guide through the result viewing component: https://www.jipipe.org/documentation/run-pipelines/result-analysis/
    - An overview of *JIPipe*'s data storage format: https://www.jipipe.org/documentation/run-pipelines/connect-external-software/
    - Information on how users can cache data: https://www.jipipe.org/documentation/run-pipelines/quick-run/ and https://www.jipipe.org/documentation/run-pipelines/cache/
- Information about the *ImageJ* integration and how to run *JIPipe* nodes inside *ImageJ*: https://www.jipipe.org/documentation/imagej-integration/
- An overview of *JIPipe*'s plugin list GUI: https://www.jipipe.org/documentation/plugins/
- An overview of all functionalities included in the standard *JIPipe* distribution
    - The *ImageJ* integration library: https://www.jipipe.org/documentation/standard-library/imagej-integration/
    - A guide on how to utilize macro nodes: https://www.jipipe.org/documentation/standard-library/macro-node/
    - Important remarks regarding the file system nodes: https://www.jipipe.org/documentation/standard-library/filesystem/
    - A guide through the multi-parameter feature supported by many nodes: https://www.jipipe.org/documentation/standard-library/multi-parameter/
    - Remarks about the usage of data annotations: https://www.jipipe.org/documentation/standard-library/annotations/
    - A guide through the plotting features included in *JIPipe*: https://www.jipipe.org/documentation/standard-library/plots-tables/

    - An overview of the integrated Jython and Python wrappers: https://www.jipipe.org/documentation/standard-library/jython/, https://www.jipipe.org/documentation/standard-library/python/, https://www.jipipe.org/documentation/standard-library/python/api/
    - Information about the R integration: https://www.jipipe.org/documentation/standard-library/r-integration/
    - An overview of the Cellpose nodes and information about how they are utilized: https://www.jipipe.org/documentation/standard-library/cellpose/
- Information about the creation custom *JIPipe* extensions via a graphical interface: https://www.jipipe.org/documentation/create-json-extensions/
- The usage of *JIPipe* within a command line interface: https://www.jipipe.org/documentation/cli/

4.2 Java API documentation

To aid with the continued development of *JIPipe* and to facilitate the creation of new extensions, we published documentation about *JIPipe*'s Java API. This includes the automatically generated JavaDocs that contain all classes, methods, and packages (see https://www.jipipe.org/apidocs/index.html), but also detailed guides on how to setup an extension project, create nodes, data types, parameters, and other features:

- The setup of a Java Maven project that provides features for *JIPipe*: https://www.jipipe.org/documentation-java-api/create-extension/
- An overview of the node type classes, including an example node implementation: https://www.jipipe.org/documentation-java-api/algorithm/
    - Documentation on the development of iterative multi-input nodes: https://www.jipipe.org/documentation-java-api/algorithm/iterating-algorithms/
    - An alternative multi-input node type that merges multiple data items: https://www.jipipe.org/documentation-java-api/algorithm/merging-algorithms/
    - Remarks regarding the modification of node input and outputs: https://www.jipipe.org/documentation-java-api/algorithm/slot-configuration/
    - A basic guide to defining node parameters: https://www.jipipe.org/documentation-java-api/algorithm/parameters/
    - Guidelines for creating nodes that support parallelized workloads: https://www.jipipe.org/documentation-java-api/algorithm/parallelization/
    - Definition of node types that do not have a one-to-one relationship with a Java class: https://www.jipipe.org/documentation-java-api/algorithm/custom-info/
    - A guide on creating interactive buttons in parameter lists: https://www.jipipe.org/documentation-java-api/algorithm/context-actions/
- An overview on how the Java API is used to create a new data type: https://www.jipipe.org/documentation-java-api/data-type/
    - Explanations on how to create user-selectable data importers: https://www.jipipe.org/documentation-java-api/data-type/result-ui/
    - Documentation on implementing result previews: https://www.jipipe.org/documentation-java-api/data-type/result-preview/
- A guide through the creation of a new parameter type: https://www.jipipe.org/documentation-java-api/parameter-type/
- Interfacing with *JIPipe* through its Java API to run nodes, pipelines, and projects: https://www.jipipe.org/documentation-java-api/usage-in-java/

4.3 Data and JSON API documentation

To store data and projects, *JIPipe* utilizes JSON files that follow a standardized format. This includes the format for projects (see https://www.jipipe.org/documentation-json-api/project/) and for non-Java extensions (see https://www.jipipe.org/documentation-json-api/json-extension/). A similar standardization is applied in the storage of output files to allow automated data reading and writing operations. Its highest-order implementation is the standardized format for whole-pipeline and is described in https://www.jipipe.org/documentation-data-api/pipeline-output/. The format makes use of the "data table" standard (see https://www.jipipe.org/documentation-data-api/data-table/) that makes the automated reading and writing of data and metadata possible, due to the presence of a standardized metadata file ("data-table.json", see https://www.jipipe.org/documentation-json-api/data-table/). Within the a data table directory, data items are stored in various directories that contain hierarchies of files and folders that follow a standard defined by the data type definition in Java (see https://www.jipipe.org/documentation-data-api/row-folder/). A list of available data types, associated standards and properties is given in https://www.jipipe.org/documentation-data-api/data-types/.

5 Methods

5.1 *JIPipe* dependencies

JIPipe is written in Java version 8 and utilizes libraries provided by *SciJava* (https://scijava.org/) . The full list of required libraries is shown in **Supplementary Table 1**. *JIPipe* is open source and licensed under BSD-2-Clause license.

Dependency	Version	Author
Bio-Formats	6.5.1	Linkert et al.[14]
CLIJ2	2.0.0.14	Haase et al.[1]
Feature_Detection	2.0.2	Fiji.sc
FeatureJ	2.0.0	Erik Meijering[15]
Flexmark	0.62.2	Vladimir Schneider
Guava	26.0-jre	Google Inc.
ImageJ	2.1.0	Rueden et al.[16]
ImageScience	3.0.0	Erik Meijering
ImgLib2	2.0.0-beta-46	Pietzsch et al.[17]
Jackson	2.11.0	FasterXML
Javaluator	3.0.3	Jean-Marc Astesana
JFreeChart	1.5.0	JFree.org
JFreeSVG	3.4	JFree.org
JGraphT	1.4.0	Barak Naveh
JUnit	5.7.0	JUnit Team
Jython	2.7.2	Jython Project
MorphoLibJ	1.4.1	Legland et al.[18]
MPICBG	1.3.0	Stephan Saalfeld
MSLinks	1.0.5	Dmitrii Shamrikov
MTrackJ	1.5.4	Erik Meijering
Multi-Template-Matching	-	Thomas, Gehrig[19]
OMERO	5.5.8	Allan et al.[2]
RandomJ	2.0.0	Erik Meijering
Reflections	0.9.12	ronmamo
Scijava	29.2.1	Rueden et al.[20]
SLF4J	1.7.9	QOS.ch
SwingX	1.6.1	SwingLabs

Trove4J	3.0.3	Rob Eden
Apache Commons Exec	1.3	The Apache Software Foundation
Apache Commons Compress	1.9	The Apache Software Foundation
JNA	4.5.2	Timothy Wall, Matthias Bläsing

Supplementary Table 5.1 | List of libraries used by *JIPipe*.

5.2 *JIPipe* system requirements

JIPipe was designed to run on any operating system supported by ImageJ and tested on Linux (Ubuntu 22.04), Windows (Windows 10), and macOS (macOS Sierra 10.12.6). We must note that due to our limited access to test MacOS, we can't ensure that all features will work as expected on Apple computers. We are open to contributions from the community. We recommend running our software on hardware with at least 8 GB of system memory and on a 64-bit operating system. To execute all example pipelines provided with this manuscript, at least 16 GB of memory are required. To utilize GPU processing functionality provided by CLIJ2, a graphics card supporting OpenCL 1.2 or higher and capacity to store the analyzed images must be available.

6 References

1. Haase, R. *et al.* CLIJ: GPU-accelerated image processing for everyone. *Nat. Methods* **17**, 5–6 (2020).
2. Allan, C. *et al.* OMERO: flexible, model-driven data management for experimental biology. *Nat. Methods* **9**, 245–253 (2012).
3. Svensson, C. M. *et al.* Coding of Experimental Conditions in Microfluidic Droplet Assays Using Colored Beads and Machine Learning Supported Image Analysis. *Small* **15**, 1802384 (2019).
4. Mahler, L. *et al.* Enhanced and homogeneous oxygen availability during incubation of microfluidic droplets. *RSC Adv.* **5**, 101871–101878 (2015).
5. Zang, E. *et al.* Real-time image processing for label-free enrichment of Actinobacteria cultivated in picolitre droplets. *Lab. Chip* **13**, 3707–3713 (2013).
6. Cseresnyes, Z., Kraibooj, K. & Figge, M. T. Hessian-based quantitative image analysis of host-pathogen confrontation assays. *Cytometry A* **93**, 346–356 (2018).
7. Muljajew, I. *et al.* Stealth Effect of Short Polyoxazolines in Graft Copolymers: Minor Changes of Backbone End Group Determine Liver Cell-Type Specificity. *ACS Nano* **15**, 12298–12313 (2021).
8. Hassan, M. I. A. *et al.* The geographical region of origin determines the phagocytic vulnerability of Lichtheimia strains. *Environ. Microbiol.* **21**, 4563–4581 (2019).

9. Cseresnyes, Z., Hassan, M. I. A., Dahse, H.-M., Voigt, K. & Figge, M. T. Quantitative Impact of Cell Membrane Fluorescence Labeling on Phagocytosis Measurements in Confrontation Assays. *Front. Microbiol.* **11**, 1193 (2020).

10. Stringer, C., Wang, T., Michaelos, M. & Pachitariu, M. Cellpose: a generalist algorithm for cellular segmentation. *Nat. Methods* **18**, 100–106 (2021).

11. Büttner, H. *et al.* Bacterial endosymbionts protect beneficial soil fungus from nematode attack. *Proc. Natl. Acad. Sci. U. S. A.* **118**, 2110669118 (2021).

12. Klingberg, A. *et al.* Fully Automated Evaluation of Total Glomerular Number and Capillary Tuft Size in Nephritic Kidneys Using Lightsheet Microscopy. *J. Am. Soc. Nephrol. JASN* **28**, 452–459 (2017).

13. Gerst, R., Medyukhina, A. & Figge, M. T. MISA++: A standardized interface for automated bioimage analysis. *SoftwareX* **11**, (2020).

14. Linkert, M. *et al.* Metadata matters: access to image data in the real world. *J. Cell Biol.* **189**, 777–782 (2010).

15. Meijering, E. FeatureJ. https://imagescience.org/meijering/software/featurej/.

16. Rueden, C. T. *et al.* ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics* **18**, 529 (2017).

17. Pietzsch, T., Preibisch, S., Tomančák, P. & Saalfeld, S. ImgLib2—generic image processing in Java. *Bioinformatics* **28**, 3009–3011 (2012).

18. Legland, D., Arganda-Carreras, I. & Andrey, P. MorphoLibJ: Integrated library and plugins for mathematical morphology with ImageJ. *Bioinformatics* **32**, 3532–3534 (2016).

19. Thomas, L. S. V. & Gehrig, J. Multi-template matching: a versatile tool for object-localization in microscopy images. *BMC Bioinformatics* **21**, 44 (2020).

20. Rueden, C., Schindelin, J., Hiner, M. & Eliceiri, K. SciJava Common [Software]. (2016).