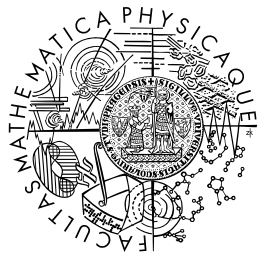


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Soňa Molnárová

Interactive environment for flow-cytometry data analysis

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. Miroslav Kratochvíl
Study programme:	Computer Science
Study branch:	Programming and Software Development

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First of all, I want to thank my supervisor RNDr. Miroslav Kratochvíl for his patience, invested time, and valuable advice. Many thanks also belong to my family and friends for their endless support and understanding.

Title: Interactive environment for flow-cytometry data analysis

Author: Soňa Molnárová

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: Flow cytometry is a method for measuring chemical characteristics of single cells in a solution, with applications in biological and clinical research. Recent advances in the technology of flow cytometers make it exceedingly easy to produce larger datasets that describe more interesting phenomena, which creates new challenges for data processing and analysis software. This thesis describes and implements a proof-of-concept software that simplifies the data processing by implementing an interactive analysis pipeline editor designed to efficiently handle huge datasets. The functionality of the software is demonstrated by implementing a recent data analysis algorithm, and by comparing the resource efficiency to a typical R analysis tool. Future use of the software by biologists and medics is facilitated by providing a simple interface for including new algorithms, thus arbitrarily extending the functionality.

Keywords: bioinformatics, clustering, visualisations, flow cytometry

Contents

Introduction

Cytometry is a general term, used in biological research, standing for many different methods that measure the properties of single cells. Branches of cytometry are differentiated by the types of devices used for such measurements (flow-cytometry [25], mass cytometry [31], imaging cytometry [9], or single-cell RNA sequencing [14]). Flow-cytometry has a wide range of applications in many biological and medical fields, such as diagnosing and monitoring of leukemia, cross-matching organs for transplantation, or development of vaccines.

Data obtained from a flow or mass cytometer contain an entry for each cell, each with several measured numerical parameters of the cell. The parameters usually correspond to physical or chemical properties of the cell (size, transparency, presence of chemicals on the surface). Technically, the parameters can be interpreted as dimensions, and the single cell measurements can be interpreted as points in a multidimensional feature space.

The most common method of processing data is to manually group cells into clusters — discrete populations based on similar properties. This method is called gating; it is based on sequential visual inspection and marking of the cells in simple 2-dimensional projections of the feature space. The whole process is extremely time-consuming, especially if there are many parameters (i.e. consequently, many possible projections to use for viewing and categorizing the cells), or if the software that provides the gating interface is not sufficiently interactive.

The recent development in flow-cytometry methods and technologies brings increasing amount of data. Software problems associated with the analysis of large datasets include the following challenges:

- Larger datasets are often downsampled to (indirectly) improve software performance, which discards many fine details in the data.
- Current software is usually able to process only lower millions of cells that fit into the main memory of the computer.
- Manual analysis of higher-dimensional datasets (40-dimensional datasets are common) is laborous because of the quadratic number of 2D projections

to examine during the gating. Often, not all combinations of parameters are analyzed and some clusters may be overlooked.

- There are many automated algorithms (e.g. unsupervised clustering) that facilitate the analysis, but their use from various programming or scripting environments (Python, R) is hardly suitable for biological or medical user audience.

The thesis addresses these problems by providing software with:

1. a highly interactive and fast responsive graphical user interface,
2. efficient visualizations of the two-dimensional plots that can be extended to any visualization that may work on current GPUs,
3. ability to process sufficiently large datasets without downsampling provided by effient memory management,
4. simple possibility for expansion by including new algorithms, using a well-defined programming interface for 'Tools'

The resulting software offers a workflow for flow-cytometry data analysis. The aim of the thesis is not to provide a product applicable for biology, but only a proof-of-concept that demonstrates that the proposed principles work as intended, and can be easily extended for biological applications. The result therefore provides a simple GUI that allows the user to manipulate the cytometry data, and construct the FlowSOM algorithm [33] that is currently used for clustering many realistic datasets. The visualization is provided by the colored scatterplots usual in cytometry.

The most significant improvement is the ability to process large data: The software can handle many gigabytes of the data at once, thanks to the memory management facilities of current operating systems. That allow the users to obtain a complete picture of untruncated datasets. All computation in the program is done asynchronously, which allows the user to edit the workflow without waiting for the computations to finish. Finally, the possibility to include custom tools using the simple programming interface provides sufficient expansion space to support future applications in many areas of cytometry.

Layout of the thesis The first chapter of the thesis gives an overview of what cytometry is, how the datasets are obtained, and what algorithms are currently used for processing flow-cytometry data. The second chapter explains the design and implementation of the software. The last chapter demostrates the results of the software implementation, and its functionality by processing an example dataset.

Chapter 1

Background

In this chapter, we provide an overview of the origin of the cytometry data. We describe what cytometry is, how the samples are measured and analyzed, and what are the usual problems with their analysis.

1.1 Cytometry

In biological research, cytometry is a general term standing for many different methods that measure the properties of single cells. Characteristics of cells may be, for instance, count of the cells, its size, shape, structure, DNA content, or existence of proteins. Cytometry methods have a wide range of applications in biological fields, such as oncology [8, 5, 27], marine biology [24, 20], cell biology [7, 6], clinical applications [32, 17], and biomedicine [10].

In cytometry, there are various types of devices which can produce such measurements. Two currently used kinds of cytometers are flow and mass cytometers. Flow cytometers are subdivided into normal [25] and spectral [23] flow cytometers. In mass cytometry, there exists a mass cytometer called CyTOF [4]. An expansion of mass cytometry is Imaging mass cytometry [3].

Multidimensional data In cytometry, characteristics of cells (parameters) are in the form of multidimensional data. Each parameter stands for one dimension in multidimensional data space (see figure 1.1 for the detailed description of the representation of flow cytometry data). This approach allows observing relationships between each parameter thanks to various visualization techniques.

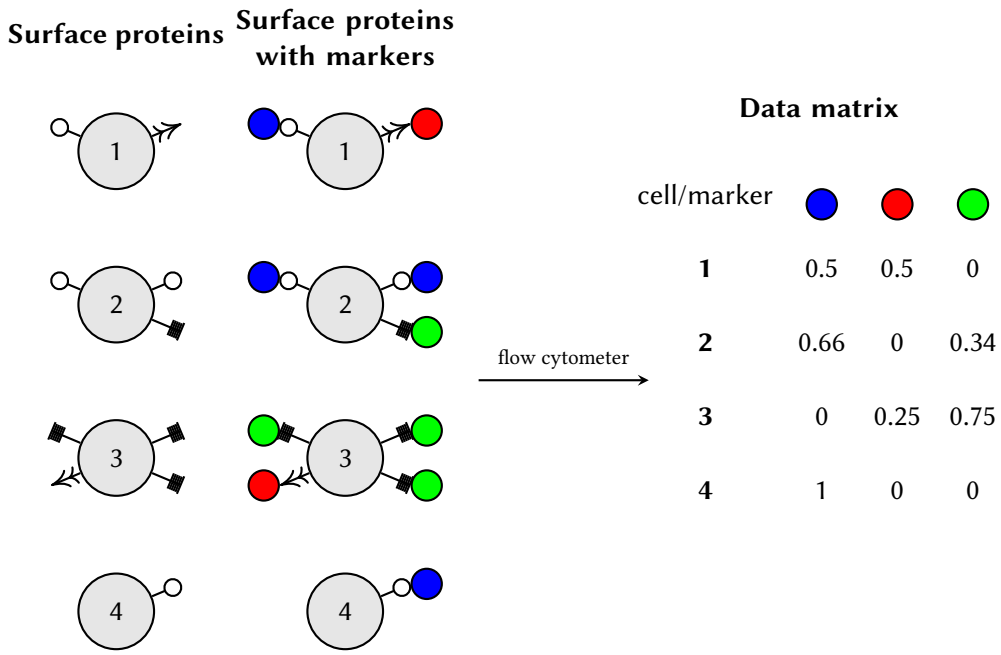


Figure 1.1: **Overview of the method of obtaining the flow cytometry data.** Cells (in this figure, the grey circles numbered 1–4) have various specific molecules on the surface (in this figure, three types of surface proteins are present). The surface proteins are marked by chemicals that can bind only to certain specific cell molecules, and have a specific distinguishing "color" (i.e., they emit specific light spectra upon excitation). The chemicals are excited by laser, and the amount of the emitted light in different parts of the spectrum is measured and stored in the data matrix. Cells can be distinguished by combinations of the substances (each bonding to the different cell molecule). Measurements in the example table are idealized, neglecting physical properties of the process (such as measurement noise and spectral overlaps).

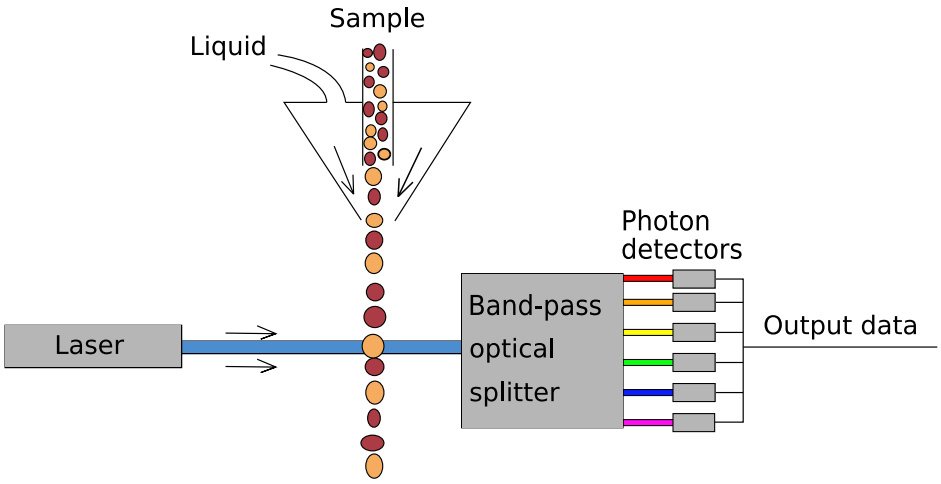


Figure 1.2: **Simplified architecture of a traditional flow cytometer.** The marked cells (in the figure are called "Sample") (marking process is described in figure 1.1) are aligned by hydrodynamic forces (in the figure are called "Liquid") so they can be analyzed one by one by a laser beam. This laser beam excites the markers on each cell, and afterward, the markers emit light. The emitted light is divided into the parts of the visible light spectrum (usually 5–10 colors). The division of the light is done by "Band-pass optical splitter", which is formed by an optics system — mirrors and filters which reflect or let through only specific frequencies of the light. The parts of the spectrum are measured by separated "Photon detectors" (each detector for one part of the spectrum). Each detector returns one column of the result multidimensional data matrix.

1.1.1 Flow cytometry

Flow cytometry [25] detects and measures characteristics of a single cell in populations of cells by a flow cytometer (see figure 1.2 for the detailed visualization of the simplified architecture of flow cytometer).

Cells are analyzed one by one by a laser beam. When the laser beam hits the cell, it emits light. The emitted light is divided by an optics system into different parts of the visible light spectrum. The parts of the spectrum are measured, and results are stored in a multidimensional data matrix. The detailed description of the process is in the figure 1.2.

Recent flow cytometers can measure up to 20 single-cell characteristics for millions of individual cells per sample. Characteristics of the cells measured by the flow cytometer can be examined for quantity of proteins, types of white blood cells, DNA and RNA content, or intracellular pH.

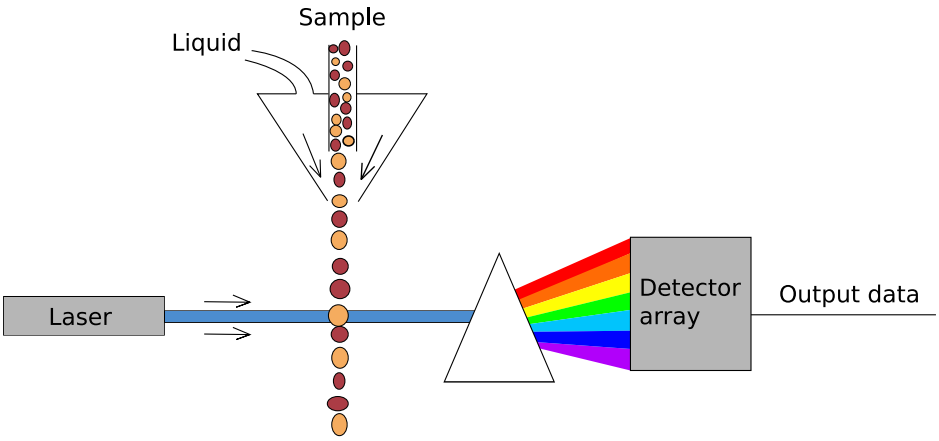


Figure 1.3: **Simplified architecture of a spectral flow cytometer.** The marked cells emit light under the laser beam as in the flow cytometer (described in the section 1.1.1). The difference is in the measuring of the emitted light. In a normal flow cytometer, only discrete parts of the spectra are measured, while in a spectral flow cytometer, the emitted light is analyzed by the whole visible spectrum. The whole visible spectrum is obtained by a polychromatic dispersion element (the white triangle in the figure). The spectrum is detected and measured by "Detector array", which produces the result data.

1.1.2 Spectral flow cytometry

Spectral flow cytometer [23] (for the detailed description of the architecture see figure 1.3) is based on similar foundations as a standard flow cytometer (described in the section 1.1.1). The difference is in the processing of the light emitted from the observed cell. While the standard flow cytometer uses a system of optical filters and mirrors, the spectral flow cytometer utilizes a polychromatic dispersion element. This element spreads the emitted light into an detector array that allows us to analyze each cell by the full spectrum of the emitted light.

1.1.3 Mass cytometry

Mass cytometry [31] measures characteristics of a single cell by a device called CyTOF (Cytometry by Time-Of-Flight) [4] (for the detailed description of the architecture see figure 1.4).

Mass cytometry can obtain more single-cell parameters than from flow cytometry. Flow cytometry has the number of parameters limited by the visible light spectrum. Emitted light spectra often overlap, which makes it hard to distinguish one from another.

Mass cytometry has limited applications because the examined sample is

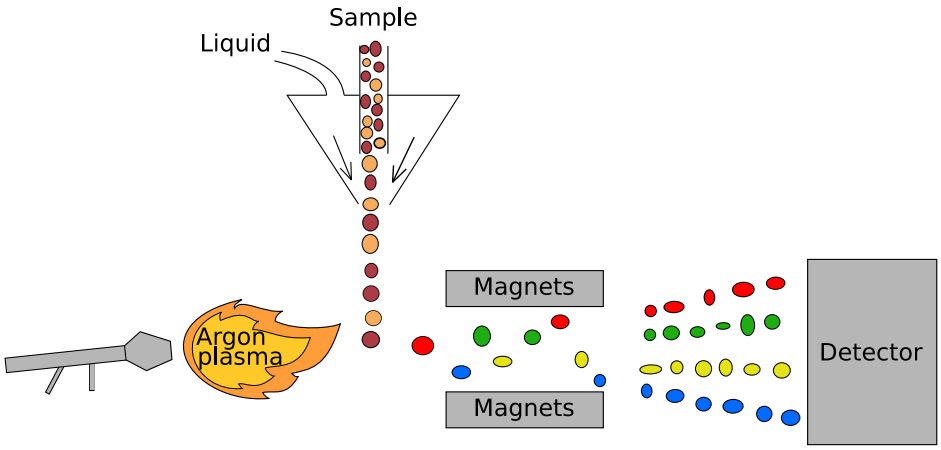


Figure 1.4: **Overview of the function of a mass cytometer.** (Source of plasma may differ.) The functionality principle is similar to the flow cytometer (described in the section 1.1.1), but the process is modified for heavy metal ion markers. The cells are burned to ions (electrically charged atoms) by argon plasma. These ions fly through a magnetic field, which aligns the ions and groups them into a distinct stream for each ion mass. The grouped ions fly into a "Detector", which measures the number of ions in each group and produces columns of the result matrix data.

always destroyed during the measurement. Also, the measurement process is longer than in flow cytometry. And finally, chemicals used for marking cells are nowadays more expensive than in flow cytometry.

1.2 Analysis and interpretation of cytometry data

The motivation of analyzing cytometry data is, for example, the detection of leukemia. The amount of normal and plasma B cells is counted. If the ratio of the normal cells and plasma B cells is too high, there is a high probability that the patient has some form of leukemia. To count the cells, they must be first classified. The classification of cells is a problematic part of the analysis. Various techniques exist (manual or automatic), which facilitate the analysis process and give a good insight into cell populations.

Recent advances in the development of cytometry technologies provide a constantly increasing amount of measured single-cell characteristics. Therefore, multidimensional cytometry data are larger because they contain more cells and more dimensions (characteristics). This fact makes the manual analysis of data

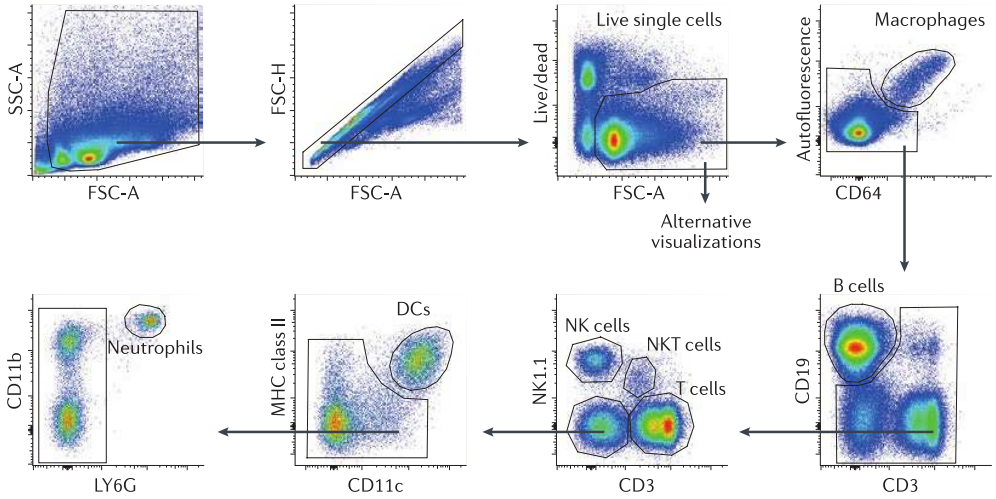


Figure 1.5: Example of manual gating process — marking similar populations in different subdimensions. (Image originally authored by Saeys, Van Gassen, and Lambrecht [28])

(manual gating) more complicated because more characteristics have to be inspected sequentially in two-dimensional space. Hence, the reliable and accurate automatic analyzing algorithms are created and used (e.g., clustering or dimensionality reduction).

A critical part of cytometry data analysis is the visualization of measured data. The interpretation of results of automatic analysis algorithms is often hard for biologists, and thus, many visualization algorithms are needed. There are many approaches from two-dimensional plots to trees of cell clusters.

1.2.1 Manual analysis by gating

Manual analysis by gating (i.e., manual gating) is a method where cells have to be separated into clusters manually. Current manual gating approach utilizes mainly projections of the measured data into two dimensions, where the cells naturally form distinguishable clusters. These can be picked out manually by drawing a 'gate' that classifies them into two subsets. Then, the clusters are visualized by visualization methods. The 'gate' is drawn by shapes consisting of lines that specify the desired area (a manual gating process is described in the figure 1.5).

Two main problems associated with manual gating are:

- Variance of drawing 'gate' around cluster is quite big.
- Combinations of two dimensions is $\binom{n}{2}$. Therefore the analysis of more

dimensional datasets (e.g., forty-dimensional) is very time-consuming. Hence, not all combinations of subdimensions are analyzed and some clusters can be overlooked.

Recently, automatic gating algorithms are preferred thanks to their significantly higher reproducibility. Also, data-driven approach detects populations of cells that might be overlooked or discarded by a human. Manual gating analysis is extremely time-consuming on a large amount of data, whereas automatic gating algorithms are more efficient and, therefore, frequently used.

1.2.2 Clustering

Clustering is a data analysis method that analyzes similarities of given objects (in this case, characteristics of cells) and categorizes them into different groups called clusters. Clusters often correspond to cell subpopulations.

Clustering has been used as an automated alternative to manual gating (described in the section 1.2.1). This method is more objective than manual gating and considers all cells in the dataset; hence no cells are overlooked or discarded. Real results of the clustering method may often differ from expected biologically relevant interpretation, and human intervention is needed. An example of interactive software is idendro [1]. The method of automated clustering combined with human intervention is called semi-supervised clustering.

Clustering method solves problems of manual gating (mentioned in the section 1.2.1):

- The problem of choosing correct and unbiased 'gate' is not relevant here because the clusters are not classified manually.
- Datasets with many dimensions are not a problem anymore because the classification of clusters is done in the original multidimensional space. Therefore, the dimensions do not have to be analyzed by two-dimensional subdimensions.

An example clustering algorithm is FlowSOM [33]. For clustering, FlowSOM uses a combination of self-organizing maps [13] (SOM) and a hierarchical clustering [12]. Cells are first processed by SOM that outputs dataset categorized into groups. These groups are sent as input for hierarchical clustering, which joins the groups into clusters (this process is called metaclustering [2]). The hierarchical clustering is much faster on those groups because the number of groups is lower than the number of objects in the whole dataset. Moreover, clusters can be suitably visualized, for example, by a minimal spanning tree built on aggregated

[1] `https://github.com/tsieger/idendro`

clusters from SOM (a similar approach is used in SPADE algorithm [34]). The main advantage of the FlowSOM algorithm is that it is an effective algorithm thanks to the pre-processing of the dataset by SOM, and at the same time, results highly correspond to professional manual gating analysis.

1.2.3 Dimensionality reduction

Dimensionality reduction is a data analysis method that embeds original multidimensional data into low-dimensional space (for the purpose of visualization data for people, a two-dimensional space is the only choice). The new coordinate system captures all cells and their mutual relations. The relations are simplified because not all details from high-dimensional space can be preserved in the lower dimensional projection. On the other hand, in clustering (described in the previous section 1.2.2), all cells are grouped into clusters in the original high-dimensional space and all relations are preserved.

Similarly to clustering, this method might also reveal hidden populations of cells and gives better insight to the dataset, but as in clustering, fully automatic dimensionality reduction may be inaccurate and human intervention is needed.

Since dimensionality reduction methods visualize each cell, their drawing is much more time consuming on large datasets than in clustering, where only clusters are visualized.

Examples of dimensionality reduction algorithms are PCA [26] (which is primitive but less accurate), t-SNE [19], UMAP [21], EmbedSOM [15], or PHATE [22].

t-SNE t-SNE (T-distributed Stochastic Neighbor Embedding) [19] is a machine learning algorithm that embeds multidimensional data into low-dimensional space (in this case, two-dimensional space) and preserves cell similarities. It is a nonlinear dimensionality reduction technique that for each multiparameter cell embeds a point in two-dimensional space. Similar cells are close to each other and form clusters. Thus this technique is efficient only on a few tens of thousands of cells, while the FlowSOM algorithm allows processing efficiently millions of cells.

EmbedSOM EmbedSOM [15] is a nonlinear dimensionality reduction algorithm that uses the results of built SOM as a basis for further analysis. The main feature of EmbedSOM is the performance of embedding points into two-dimensional space, which is linear with the number of data points. However, the number of dimensions of datasets slows down the computation. Hence this technique is suitable for physically limited dimensions of flow and mass cytom-

etry data with many measured cells, while the t-SNE algorithm is suited to high-dimensional data with the low number of measured cells.

1.2.4 Visualizations of the cytometry data

Visualizations are one of the most important parts of the data analysis because only the raw dataset obtained from measurements is not readable for people. It can also be the bottleneck of the pipeline processing, so the right technique must be considered and chosen for each case individually. Therefore, many new and more efficient methods have been developed as an alternative to traditional two-dimensional scatter plots. Many of those new techniques use either dimensionality reduction (described in the 1.2.3) algorithms or clustering (described in the 1.2.2).

Scatter plot A traditional way of cytometry data visualization is a two-dimensional scatter plot that embeds each cell into the two-dimensional space. The purpose of scatterplots is to show cells in clusters by meaningful selection of axes. The meaningful axes can be two dimensions chosen from multidimensional space (a visualized correlation between two parameters is in the figure 1.7) or results of the dimensionality reduction algorithms (for example, the results of the EmbedSOM [15] (visualized embedded cells are in the figure 1.6) or t-SNE [19] algorithms).

Heatmap Heatmap is a visualization technique that uses a matrix of values from the dataset. Each value of the matrix has a color based on the level of observed characteristics (for the heatmap visualization see figure 1.8).

Dendrogram A dendrogram is a visualization technique with a tree structure that shows the hierarchical structure of clusters obtained from the clustering algorithm. Dendrogram visualization is usually used with heatmaps or scatterplots (for the usage of the dendrogram see figure 1.9).

Minimal spanning tree Minimal spanning tree (MST) is used for visualizing the results of the various clustering algorithms (e.g., FlowSOM [33] (for an example of the MST see figure 1.10)) visualizing clusters of the cells).

1.3 Current cytometry data analysis problems

Recently available software is not prepared for the advanced development of technologies obtaining cytometry data. The development brings the increasing

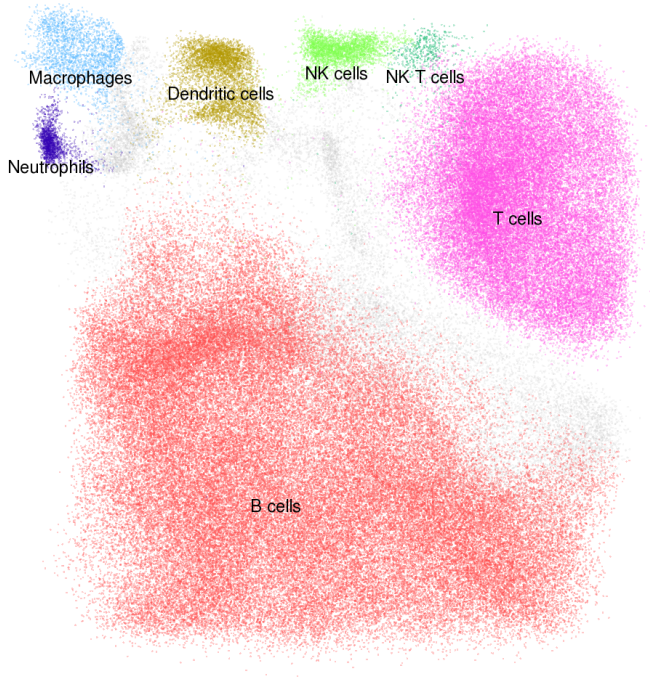


Figure 1.6: **Scatter plot of the results of EmbedSOM and FlowSOM algorithms**. All characteristics (dimensions) of cells are visualized together in two-dimensional space. The cells are colored by FlowSOM clustering algorithm. The dataset is the same as in the figure 1.5, where the characteristics often overlap. Thanks to the EmbedSOM, the cells do not overlap and, additionaly, they are grouped in correct clusters. Source: `https://gitlab.com/exaexa/ShinySOM/-/blob/master/TUTORIAL.md`

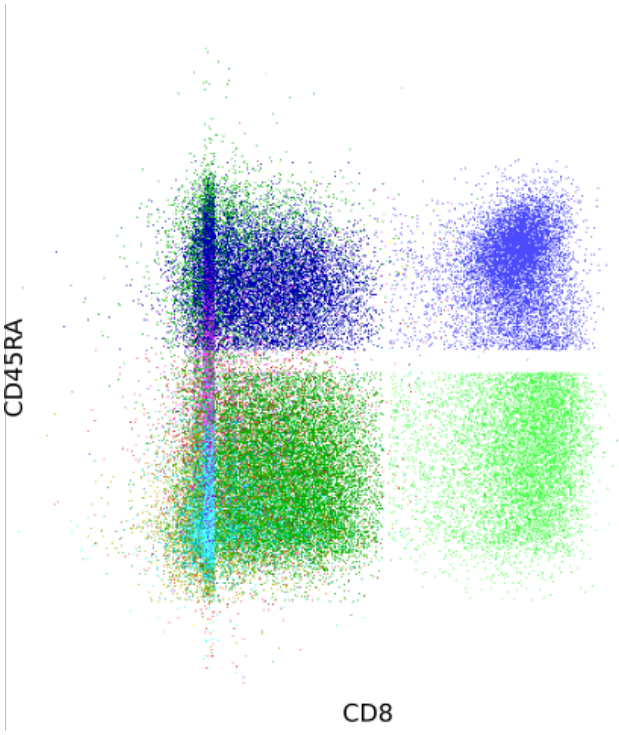


Figure 1.7: Corellations between two cell parameters visualized as colored clusters (clusters obtained from the “label” column — manual clustering).

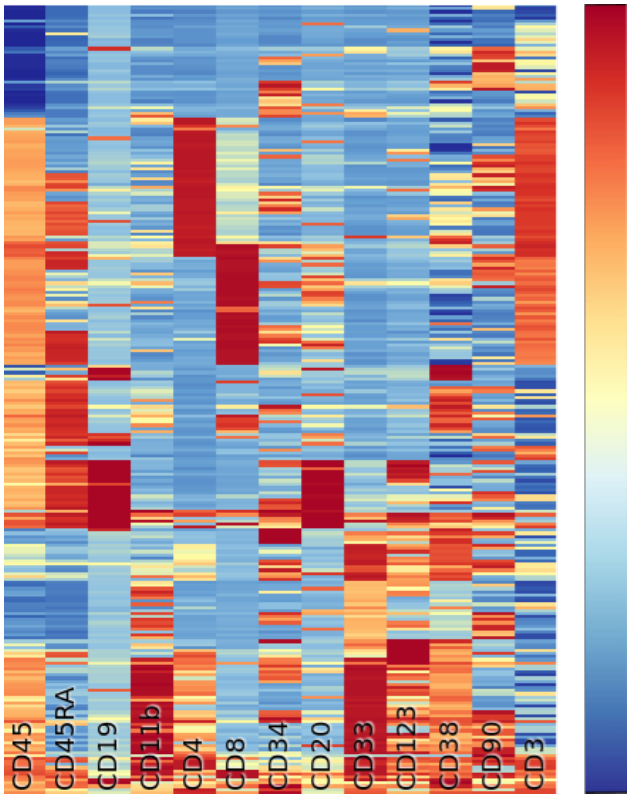


Figure 1.8: Heatmap showing the measured parameters (parameters names are in the bottom part of the picture) for each cell (one cell in each row). (Screenshot taken from the ShinySOM software [16])

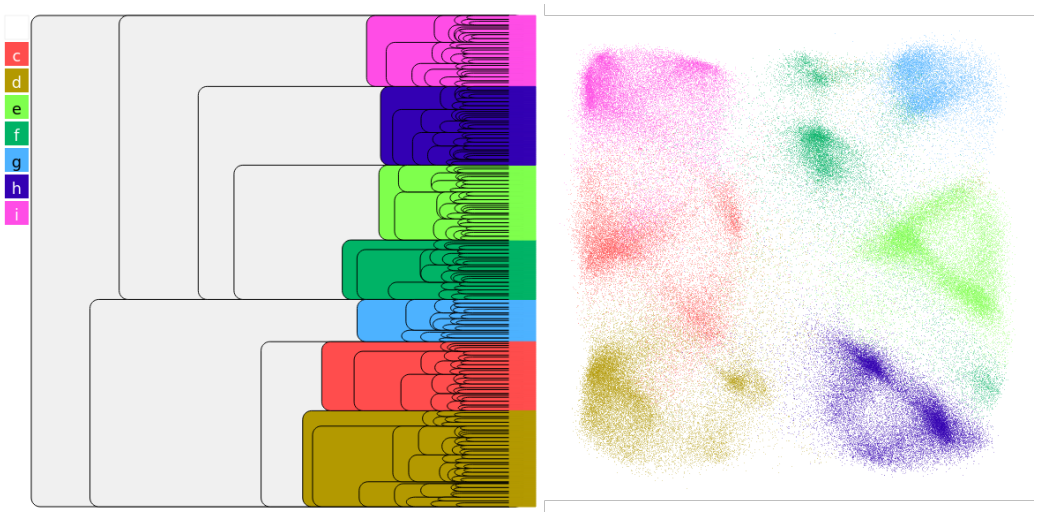


Figure 1.9: The dendrogram (on the left) shows the hierarchy of the clusters, and the scatterplot (on the right) visualizes the results of the EmbedSOM, both being built over the same built SOM. The colored clusters serves to highlight specific clusters in the dendrogram and the corresponding cell populations in the scatterplot. (Screenshot taken from the ShinySOM software [16])

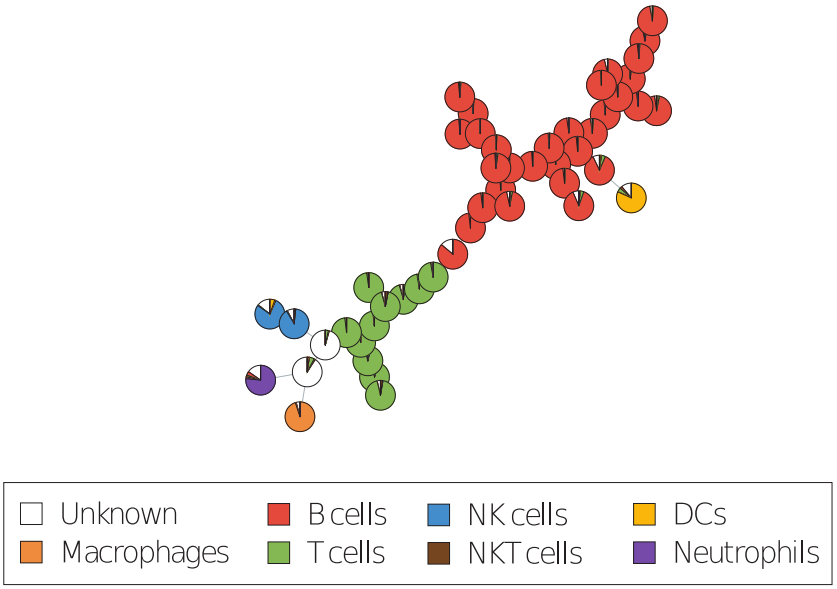


Figure 1.10: MST (minimal spanning tree) of the resulting clusters from the FlowSOM algorithm. Each node in the tree is a pie chart indicating the percentage representation of the cell types falling in the cluster. (Image originally authored by Saeys, Van Gassen, and Lambrecht [28])

number of parameters of cells, associated processing, or new visualization techniques, which may be useful but are not supported. Some software provides such features, but the user needs to know programming principles or at least scripting, which is not applicable in clinical usage. Current software is not prepared for processing more than millions of cells, which is a problem with their growing amount. Also, outside extensions of software by new algorithms are not provided.

Storage The most significant problem with the increasing size of the data files is that they do not fit into the main memory. Hence, working with them requires more sophisticated approaches such as file mapped into the memory [11] or cache efficient algorithms [30]. Otherwise, accessing files directly from the hard drive is extremely slow and insufficient for the requirements of flow cytometry analysis software.

Computation speed The whole computational process is getting more difficult and time-consuming with expanding datasets. The meta results of the algorithms may not fit into the memory, and therefore direct disk access is needed. One of the possible solutions is the pipelining method with temporary files stored on a hard drive while mapped into the memory. The computational process is divided into several steps, where each step is dependant on the result of the previous step. A similar approach of pipelining is also employed in 'make' command on Unix-like operating systems, which sequentially compiles the given program.

Visualization flexibility The performance of the visualization process is decreased with the increasing size of the datasets (more demanding visualization requests — more cells sent to the rendering system for visualization). Hence a performant rendering system providing features such as optimized shaders and effective matrix operations must be used.

Chapter 2

Flow cytometry workflow tool

As a solution to problems in question 1.3, we present a new flow cytometry tool that is capable of handling several millions of cells sufficiently while offering a user-friendly interface. This chapter focuses on the design and implementation of the software.

2.1 Design goals

The primary objective is to solve all three problems described in the previous section 1.3.

Efficient utilization of available RAM Cytometry datasets can be large files that do not fit into the main memory while reading/writing the usual way. Therefore, even computers with sufficient performance to process such large files can not process them. We solved this problem by storing all large data on a hard drive and using memory-mapping of the operating system [11] for efficient usage of the RAM while loading/storing data on disk.

This approach has three advantages:

1. Practically all modern operating systems support memory-mapping.
2. Mmapped memory actually does not need RAM (operating system can swap RAM to the disk at any time), and hence the application will not run out of memory.
3. The application and most of the algorithms access the mmapped data on the disk the same as data stored in the main memory.

Asynchronous computation A single-threaded application becomes unresponsive during the computation required for processing large datasets.

Hence, running complex computations on a different thread in the background keeps the application still able to perform other tasks such as interaction with a user interface.

Hardware-accelerated support for visualisations The algorithms produce lots of data for visualization, so an efficient rendering system is required. We use OpenGL that is suitable for this purpose because it has hardware support for matrix operations and shaders.

Simple and extensible GUI The GUI should be easily programmable, so the future extensions of the tool are simple. Dear ImGui is easily connectable to the OpenGL that is used for visualizations in the application. Dear ImGui is an immediate-mode GUI. The immediate-mode style of the GUI was designed to facilitate the connection of the user interface software to any rendering system. Its connection to OpenGL (or any other rendering pipeline) is fairly easy because the whole state of the UI is stored in a separate global context that is reachable within the whole application. Surprisingly, no rigorous description of the advantages and disadvantages of the immediate-mode GUI exists (for more details, see discussion [1]).

2.2 Workflow tool architecture

2.2.1 Main abstractions

Abstractions are necessary and help logically divide the executive parts of the program. We simplify the software design problem by several systematic constructions from which the whole workflow is built.

The software is designed as a workflow in a form of DAG (directed acyclic graph) (the example diagram is in the figure 2.1).

Definition 1 (Workflow graph)**.** *Workflow graph is a directed acyclic multigraph; vertices are replaced by independent single-purpose computational tools, tools have vectors of inputs and outputs; edges are defined as (out, in), out is an output of tool and in is an input of tool.*

Tools are also means of communication between the user and the computational part via the user interface. Each type of tool has assigned its functional purpose, and their mutual connection leads to the desired results.

[1] `https://github.com/ocornut/imgui/wiki#About-the-IMGUI-paradigm`

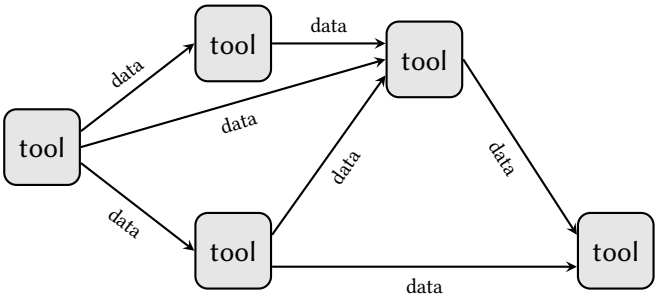


Figure 2.1: Workflow in the form of DAG.

2.2.2 Main modules and interfaces

The workflow logic is divided into three executive modules, each with separated functionality. Three ideas of the modules are: user interaction with the program, data flow control, and computational part. Also, these modules must communicate with each other and exchange information through defined interfaces (the diagram of the modules and interfaces is in the figure 2.2).

User interaction module The user interaction module must provide communication between the user and computer via some tool to manage data flow and set algorithm parameters. The interface must provide its inputs and outputs, check if algorithm parameters were changed, and the computational algorithm operating on inputs and outputs of the module.

Data flow module The data flow module must be able to connect tools and take care of the data flow between them. This module must be able to notify if something in the workflow graph has changed and, if so, get current workflow graph, inform tools that their outputs were recomputed or that input has changed, and provide all tools which need recomputation.

Computing module The computing module must execute algorithms on the given data and store results in the data cache. The module must notify about the recently recomputed tool and update computational states.

2.3 Class structure overview

2.3.1 Tool

`Tool` implements a generic interface for a module that communicates with the user (the description of the user interaction module is in the section 2.2.2). It is

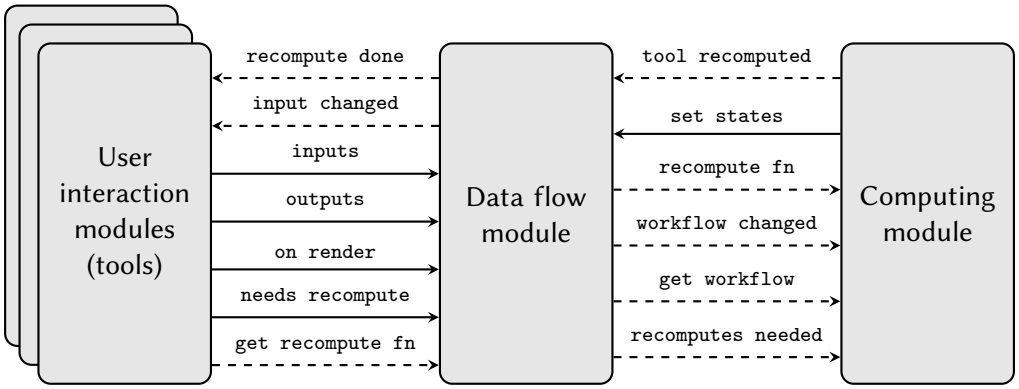


Figure 2.2: Software interface diagram, where the functions are called every frame (normal line) or only if requested (dashed line).

an abstract class that is a base class for all newly created tools. *Tool* forces its child classes to override its virtual methods and therefore meet the criteria of the interface. The instances of *Tool* acquire the necessary parameters needed as input into algorithms. Also, each child class must have a fixed size of inputs and outputs. For types of implemented child classes see section 2.7.

2.3.2 Glue

Glue (Global Linkage User Environment) implements the requirements of the data flow module (the description of the data flow module is in the section 2.2.2). In the whole program, there is only one instance of this class. It connects tools into logical sequences that create a workflow graph, and takes care of the flow of data between them. Moreover, it displays the current computational state of the algorithms by coloring corresponding connection lines between tools with colors of the respective computational state.

2.3.3 Worker

Worker implements the computing module (the description of the computing module is in the section 2.2.2). There is also only one instance of this class in the program. The instance communicates with the instance of *Glue* and computes requested tools algorithms. *Worker* contains one thread executed exclusively on computational algorithms and runs in the background. The main task of the *Worker* is to execute algorithms on the computational thread. Other tasks are: obtain tools that need to recompute their outputs and updating states of the computations of the tools and sending them to the *Glue*.

2.3.4 Data and work description formats

Parameters and *Data* use polymorphism for the abstraction of the output data of the tools and input algorithm parameters. The polymorphism provides an easy addition of new child classes.

Parameters The algorithms executed by the instance of *Worker* need input arguments for working correctly. The unique pointer pointing to *Parameters* is one of the arguments. *Parameters* contains information selected by the user in *Tool* through the user interface. Each *Tool* has its own *Parameters* child class with specified attributes.

Data The data cache in the *Worker* serves as the middle layer between algorithms — the computed outputs are stored here, and inputs are taken from here. The *Data* holds the elements of the cache. For each new data type, the new *Data* child class must be created with the specified metadata. For types of implemented child classes, see section 2.6.

2.4 Implementation

2.4.1 GUI

The goal is to have a window with GUI widgets that can fastly draw complex visualizations with OpenGL [2]. For handling input and output events and creating windows, there are two equal options — GLFW [3] and SDL [4] — for this application, SDL is used. Magnum Engine [5], with its abstraction over OpenGL calls, is suitable for connection to SDL. Also, the Ogre engine [6] could be used since it also provides an abstraction over OpenGL. But Magnum Engine has more transparent documentation and more comfortable usage. Dear ImGui [7] is a C++ library that offers easy, clear, and portable implementation of the desired widgets. Because the Dear ImGui outputs vertex buffers that can be rendered by any rendering engine, it is connected to Magnum engine.

[2] https://www.opengl.org/
[3] https://www.glfw.org/
[4] https://www.libsdl.org/
[5] https://magnum.graphics/
[6] https://www.ogre3d.org/
[7] https://github.com/ocornut/imgui

2.4.2 Update cycle

Magnum Engine contains the main loop of the application. `drawEvent` function, defined in *Cytometry*, is called every frame. In *Cytometry*, choosing and instantiating new *Tool* is handled. Additionally, `update` methods of *Glue* and *Worker* are called from the `drawEvent` method.

Glue update The main part of the *Glue* tasks (handling all connections of the tools and their drawing) is happening in the `update` method (the flowchart of the `update` method is in the figure 2.4a and the algorithm is described in the algorithm 1). Main ideas are divided into three parts:

- **Draw tools** *Glue* draws tool window and its inputs and outputs (obtained from the tool). The specific content of each tool type (where parameters are updated) is drawn by *Tool* (line 10).
- **Connect tools** If two tools were connected (output with input, on line 5), then the new edge is added to the workflow graph. Similarly, an edge is removed from the graph when the user disconnects the input. Workflow graph edges are visualized by drawing lines between each input/output tool pair (line 14), colored by the computational state color.
- **Remove tools** If the user removes a tool (line 12), all edges in the workflow graph connected with this tool are removed. Therefore tools, where outputs of the removed tool were their inputs, must be recomputed.

Worker update The principal task of the *Worker* is to compute outputs of the tools. Acquiring computational tasks and their execution is regulated by `update` method (the flowchart of the `update` method is in the figure 2.4b and the algorithm is described in the algorithm 2). Main parts are divided into two parts:

- **Handle finished work** If the worker thread running in the background has finished its work (checked on line 3), computed outputs are stored in the data cache and distributed to the corresponding tools (mediate through *Glue*) (line 5). If the worker thread is not finished yet, *Worker* waits until the thread has recomputed its work.
- **Launch new work** If the worker thread does not have assigned any work, the work is retrieved from the *Glue* (line 12) and executed sequentially (line 17). Also, the input data are sent to the second of the recently connected tools (output/input tool pair) (line 9) because the *Glue* does not have access to the data cache of the *Worker*.

Algorithm 1 Glue update cycle with sections described in the figure 2.3

```
1:  function UPDATE
2:      for all active tools do
3:          draw tool window
4:          draw tool input buttons
5:          check if tools were connected
6:          if exists warning message then
7:              draw warning message
8:          end if
9:          draw collapsing header
10:         update tool settings
11:         draw tool output buttons
12:         remove all closed tools
13:     end for
14:     draw connection lines
15: end function
```

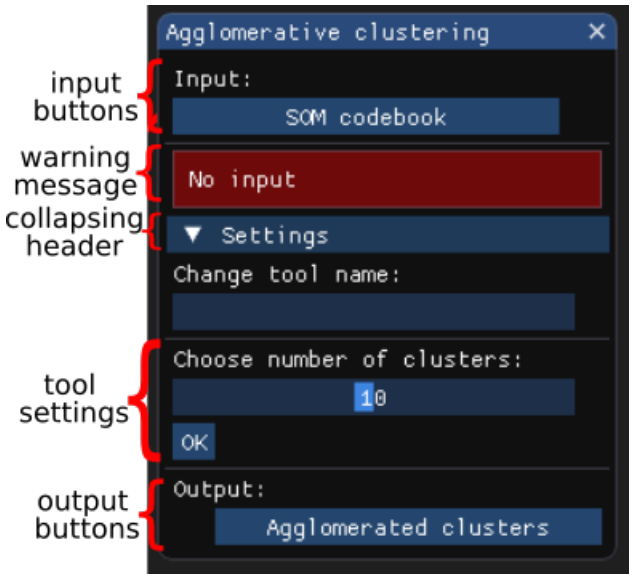


Figure 2.3: An example tool with described sections mentioned in the algorithm 1

The computational states of the currently processed tools are sent to the *Glue* every frame (line 20).

Algorithm 2 Worker update cycle

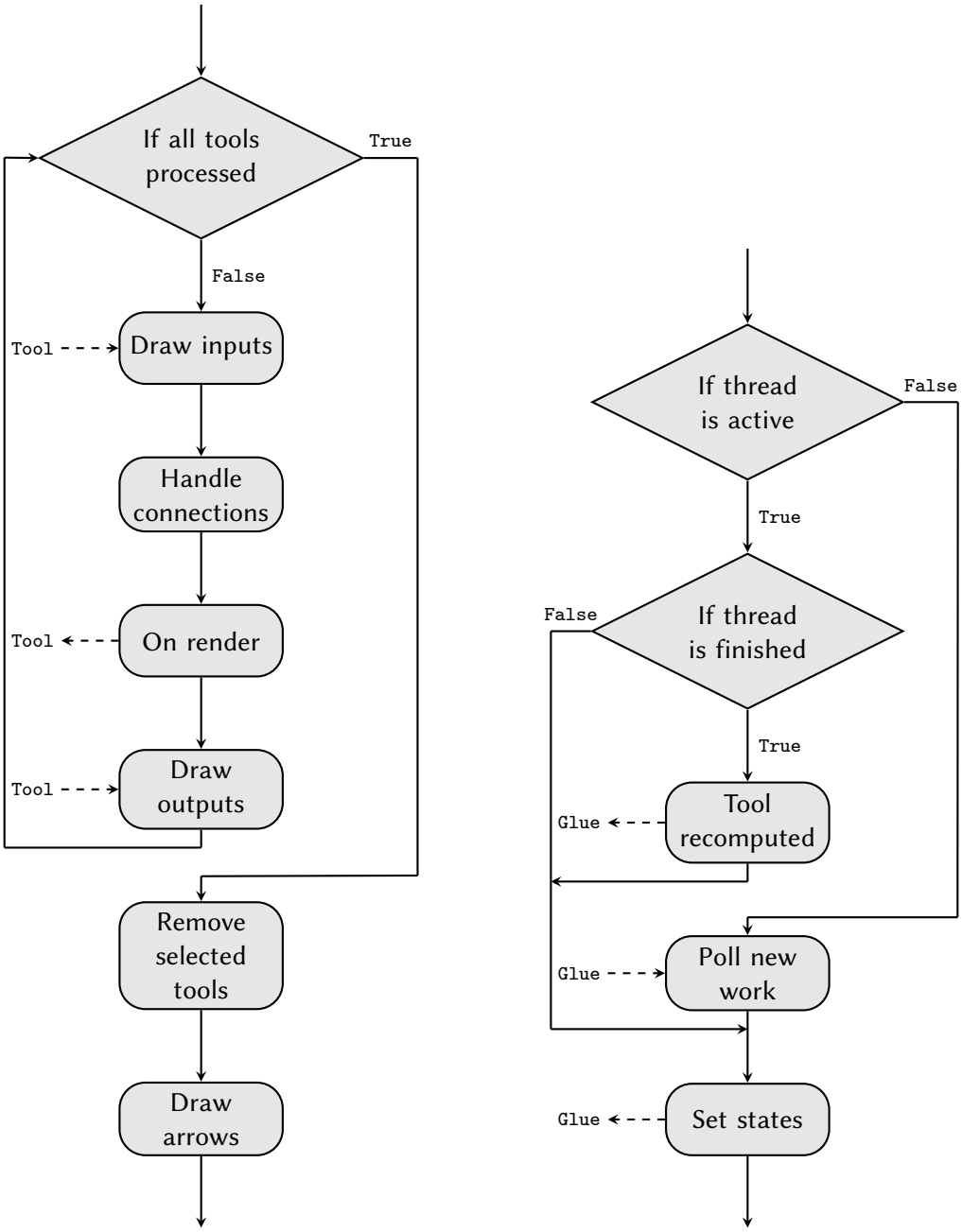
```
1:  function UPDATE
2:      if thread is active then
3:          if thread is finished then
4:              recomputed tool ← finished state
5:              notify glue about recomputed tool
6:          end if
7:      else
8:          if new connected tools pair then
9:              send input data to the connected tool
10:         end if
11:         if workflow graph has changed then
12:             get tools for recomputation
13:             recomputation tools ← waiting states
14:         end if
15:         if exists tool for recomputation then
16:             recomputing tool ← working state
17:             start tool recomputation in worker thread
18:         end if
19:     end if
20:     glue ← states of tools
21: end function
```

2.4.3 Data flow

The *Worker* is the owner of the temporary pipeline results (outputs of the tools) and stores them in the data cache (the diagram of the data cache usage is in the figure 2.5). The *Glue* only controls the flow of the data, but does not know anything about the actual data. The *Tool* has only insight into the actual data to take what it needs to change the parameters.

The *Worker* continuously updates the data cache:

- deletes data of the removed tools,
- adds new data obtained from computational functions and
- updates existing data through computational functions.



(a) *Glue* (b) *Worker*

Figure 2.4: Diagrams of the `Update` methods.

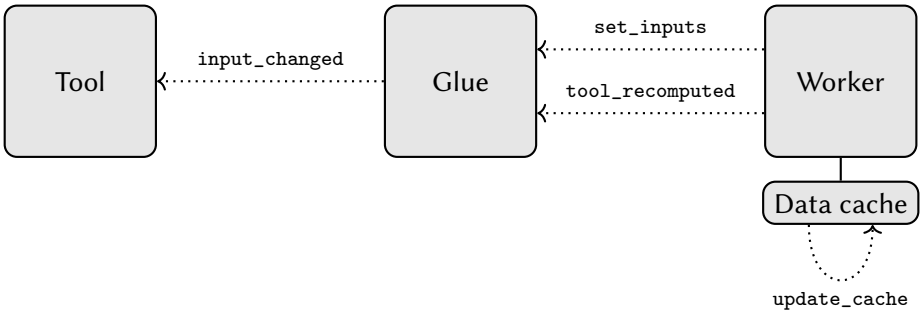


Figure 2.5: Data flow diagram.

2.4.4 Visualization

Data visualization is in the form of two-dimensional plots called scatterplots (the description of the scatterplot is in the section 1.2.4). The scatterplot visualizes the correlation between the two selected columns. Two-dimensional coordinates are obtained from the matrix. The coordinates are stored into the texture as RGB color values. This texture is subsequently sent to the OpenGL shader and rendered. By filling texture in advance, the performance is increased, because drawing pre-computed texture is much faster, then drawing each point alone through the shaders.

Three types of coloring are available: color based on expression matrix, labels, and density.

Expression coloring The expression coloring adds new information into the two-dimensional plot — relation with the third column (the third characteristic of the cell). The relation adds the third dimension, interpreted as the level of the relevance of the third parameter. The more the cell meets the criteria of the parameter, the closer to the screen the cell is (the depth is visualized by color) (an example of cells colored by expression is in the figure 2.6, used color palette is derived from the color palette RdYlBu from RColorBrewer).

The color of the cells visualized in the scatterplot is based on the third column of the matrix. The value in the same row as the current drawn cell is used for indexing the expression color palette, and the corresponding color is returned.

Labels coloring The label column categorizes cells into clusters (each cell has assigned cluster number). The cluster numbers index the cluster color palette and return the corresponding color (an example of cells colored by labels is in the figure 3.3).

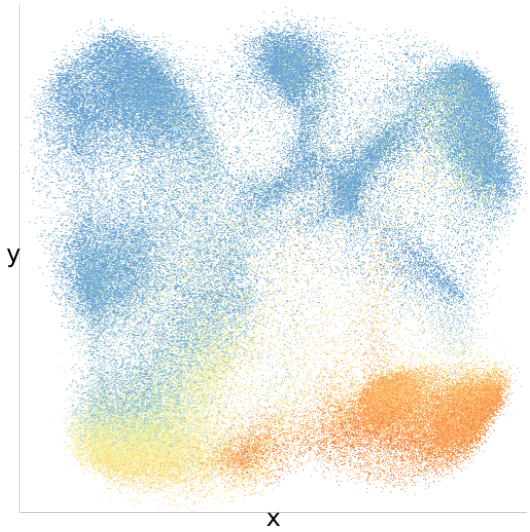


Figure 2.6: **The scatterplot of the cells embedded into the two-dimensional space (by EmbedSOM algorithm).** The color of the cells is based on the parameter CD4 of the expression matrix. The orange cells are the cells with CD4 parameter (memory cells).

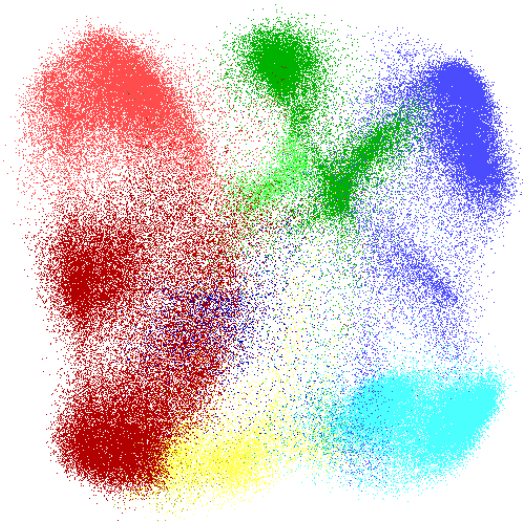


Figure 2.7: **The scatter plot of the results of EmbedSOM and FlowSOM algorithms**. All characteristics (dimensions) of cells are visualized together in two-dimensional space. The cells are colored by FlowSOM clustering algorithm.

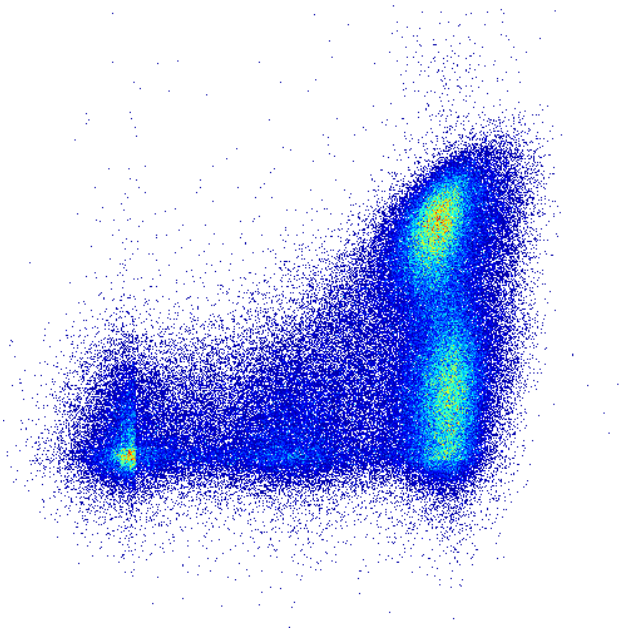


Figure 2.8: **The scatterplot of the cells showing a correlation between the CD45 and CD45RA markers.** The color of the cells is based on the density. Dark blue means sparse density and red color indicates the dense cells.

Density coloring The density coloring visualizes the density of the cells embedded into the same two-dimensional coordinate. The density color is based on the density color palette at the given coordinate (an example of cells colored by density is in the figure 2.8).

2.5 External storage formats

2.5.1 FCS format

FCS (Flow Cytometry Standard) file standard [29] is a file specification describing the structure of the flow cytometry data. FCS is the uniting file format widely used in the world. In the application, it is assumed that all input cytometry data files are in this file format because, in parsing, its main structure is used to create new *FloatMtxData* instance utilized in the workflow. The software has its own FCS file parser implemented in `FCSParser.cpp`.

2.5.2 Workflow description format

Workflow descriptions are serialized into JSON (JavaScript Object Notation). JSON is a standardized[8] text file format used for interchanging data. Its structure consists of the collections of key/value pairs and ordered lists of values. This structure provides an abstraction over the programming languages because each language have different representation of lists or such collections (for instance array/vector or dict/map). There are various C++ JSON libraries, where the verified library (JSON for Modern C++ [9]) is used in the application.

2.6 Data class types

2.6.1 MappedData

The resource management of the temporary files in the pipelining process must be handled. Therefore, the lifetime of the files is bound to the shared pointer pointing to *Data* stored in the *Worker* data cache. Storing a file in the *Data* is provided by a *MappedData* child class containing the *TmpFile* attribute (the diagram of classes is in the figure 2.9). The *TmpFile* attribute creates a temporary file during its construction and removes the file on its destruction, and therefore RAII principle is fulfilled.

The *MappedData* contains RAII struct *View*, which manages the lifetime of the mapped file into the memory. The file is mapped/unmapped during the construction/destruction of the instance of the *View*. In fact, one mapped file may be accessed from more *View* instances. Hence the number of the references to the mapped file is counted. The file is unmapped from memory only if the last instance is destroyed.

2.6.2 FloatMtxData

FloatMtxData contains raw binary data of the 4-byte float numbers converted into endianity of used architecture. The raw binary data contain the two-dimensional matrix with named columns. Because the *FloatMtxData* works with temporary files, it inherits from the class *MappedData* that provides the required interface (the inheritance class diagram is in the figure 2.9 and the *MappedData* is described in the section 2.6.1).

[8] `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

[9] `https://github.com/nlohmann/json`

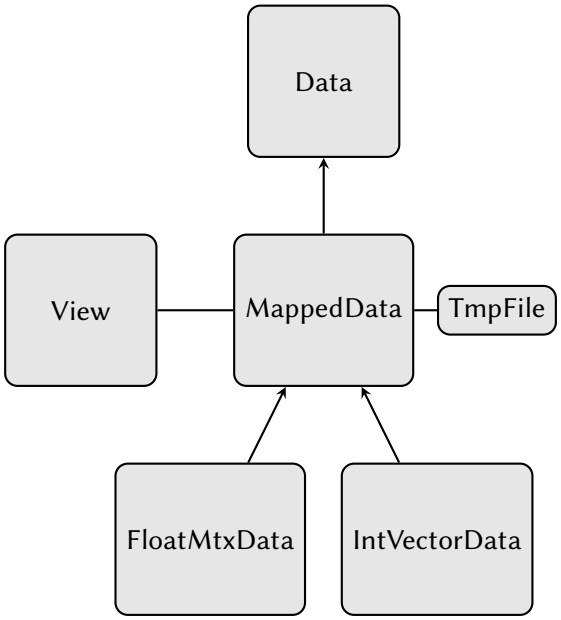


Figure 2.9: Class inheritance diagram of *Data*.

2.6.3 IntVectorData

IntVectorData contains raw binary data that constains one-dimensional vector (one-column matrix). This vector holds unsigned integer numbers. Because the *IntVectorData* stores the vector in the temporary file on the disk, it inherits from the class *MappedData* (the inheritance class diagram is in the figure 2.9 and the *MappedData* is described in the section 2.6.1).

2.7 Tool class types

Data input tools Data input tools have zero inputs and at least one output (the table of the input tools is in the table 2.1).

Matrix manipulation tools Matrix manipulation tools have at least one input and at least one output (the table of the matrix manipulations tools is in the table 2.2).

Computational tools Computational tools have at least one input and at least one output (the table of the computational tools is in the table 2.3). Each tool in this category transforms input data to new output data.

Visualization tools Visualization tools have at least one input and zero outputs (the table of the visualization tools is in the table 2.4).

Tool name	Input types	Output types	Description
OpenFile	—	*FloatMtx*	Loads one chosen FCS file from memory and creates a matrix.
LoadAggregate	—	*FloatMtx*	Loads more FCS file and aggregate them into one matrix.

Table 2.1: Overview of all implemented data input tools.

Tool name	Input types	Output types	Description
Rename	*FloatMtx*	*FloatMtx*	Renames columns of the input matrix and outputs renamed matrix.
SelectColumns	*FloatMtx*	*FloatMtx*	Selects columns from input matrix and creates matrix with chosen columns.
SeparateLabel	*FloatMtx*	*IntVector* *FloatMtx*	Separates "label" column from input matrix.

Table 2.2: Overview of all implemented matrix manipulation tools.

- *Scatterplot* The *Scatterplot* visualizes the two chosen columns of the input matrix and plots them to a 2D graph, where the type of the coloring can be chosen. Coloring by the cluster assigns a color to the cells by the number of the cluster it belongs to (obtained from the *IntVectorData* input). Coloring by expression colors cells by the one chosen column from the second input *FloatMtx* matrix. Density coloring visualizes the amount of the cells in one pixel.

Statistics tools Statistics tools have at least one input, but output is not defined (the table of the statistics tools is in the table 2.5).

Tool name	Input types	Output types	Description
Aggl Clustering	*FloatMtx*	*IntVector*	Agglomerates input clusters into output chosen number of clusters.
Aggregate Clusters	*IntVector* *IntVector*	*IntVector*	Maps input cell clusters into agglomerated clusters.
EmbedSom	*FloatMtx* *FloatMtx* *FloatMtx*	*FloatMtx*	Embeds given data into two-dimensional space.
Mapping	*FloatMtx* *FloatMtx*	*IntVector*	Assigns cluster to each cell.
Som	*FloatMtx*	*FloatMtx* *FloatMtx*	Builds SOM on input data.

Table 2.3: Overview of all implemented computational tools.

Tool name	Input types	Output types	Description
Scatterplot	*FloatMtx* *FloatMtx* *IntVector*	—	Visualizes 2D plot wiht two chosen parameters and with chosen coloring.

Table 2.4: Overview of all implemented visualization tools.

Tool name	Input types	Output types	Description
CellCount	*IntVector*	—	Counts cells in each cluster and colors them by the palette used in *Scatterplot* cluster coloring.

Table 2.5: Overview of all implemented statistics tools.

Chapter 3

Results and discussion

The result of this thesis is the software for flow-cytometry data analysis called by the name convention as Flower. In this chapter, we demonstrate that Flower is more efficient than other software, and furthermore, it can perform easier cytometry analysis tasks.

The software implements the mentioned design goals (the design goals are described in the section 2.1). Thanks to the OpenGL used for rendering, the software is prepared for the advanced visualization techniques. Additionally, the software can be extended by the tools, thanks to the provided API (the tutorial for addition of a new tool is in the appendix C). Also, the FlowSOM algorithm with a two-level clustering is implemented (the FlowSOM agorithm is described in the section 1.2.2 and the workflow of the FlowSOM is in the figure 3.4). The following sections describe the RAM usage and the basic functionality of the software.

3.1 Reduced RAM usage

The efficient utilization of the available RAM brings the possibility of processing large datasets without the limitation of the available main memory. The significant improvement is visible on the 5.5GB dataset (dowloaded from Flow repository [1] and selected only FCS files with Unstim suffix (authored by Aghaeepour et al. [1])) (the plot of the memory usage is in the figure 3.1). The 9MB dataset is downloaded from Flow repository [2] and selected only `Levine_13dim.fcs` file (authored by Levine et al. [18]).

The task was to aggregate the dataset into the one *`FloatMtxData`* matrix and visualize the two chosen columns. This task was not possible (tested with

[1] `https://flowrepository.org/id/FR-FCM-ZY3Q`
[2] `https://flowrepository.org/id/FR-FCM-ZZPH`

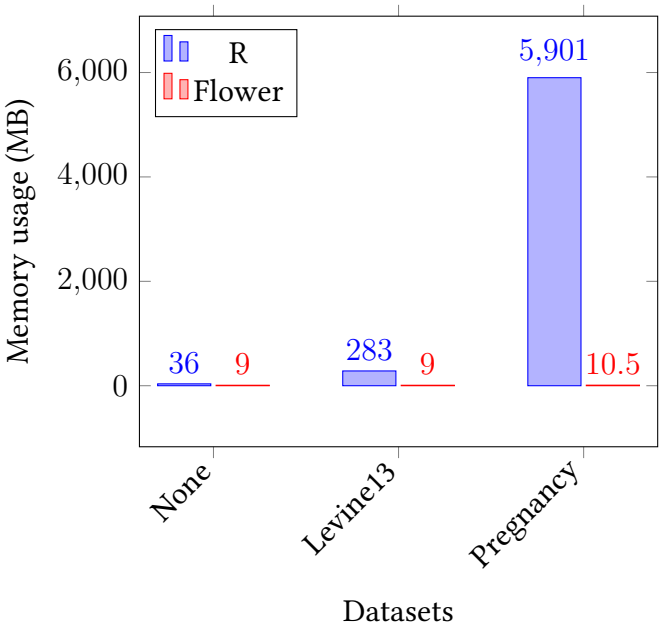


Figure 3.1: **Memory usage while processing datasets.** "Flower" (result of this thesis) is compared with R. The memory usage was measured in three different situations, depending on the loaded dataset: *None* is the state of the application immediately after start with no dataset loaded. *Levine13* is the state when loading a small 'Levine 13' 9MB dataset. *Pregnancy* is the state after loading an aggregated dataset 'Pregnancy' of total size 5.5GB. Notably, on the last dataset R consumed the specified amount of memory and exited with an error of being unable to allocate additional memory.

8GB RAM) by the R program because the process needed too much of the main memory. Even though the memory was swapped, the memory eventually ran out of space, since the memory swapping was limited, and the process exited with an error.

On the other hand, the task was possible in the Flower program. The memory did not run out of space thanks to the mapping files into the memory. The dataset was aggregated into the matrix and visualized in less than 5 minutes (tested with the same 8GB RAM).

3.2 Basic functionality

There are two ways of adding a new tool: directly select a tool or import the JSON file. The tool can be selected directly from the add tool menu (Tools menu item in the top left corner), where all implemented tools are listed. The JSON file de-

Listing 1 JSON file format of the imported/exported workflow.

```
{
    "connections": [
        {
            "from": toolID,
            "output": outputID,
            "to": toolID,
            "input": inputID
        },
        ...
    ],
    "tools": {
        "toolID": {
            "name": "toolName",
            "position": [
                x coord,
                y coord
            ],
            "tool": "toolName in register",
            "data": {
                specific data of the tool, may be null
            }
        },
        ...
    }
}
```

scribes the state of the workflow — instantiated tools, their selected parameters, and connections between them (the workflow format of JSON file is described in the listing 1). The workflow can also be exported in the same format as the imported file.

3.2.1 Workflow editing

The first example workflow displays the usage of the two basic tools — open file and visualize (an example workflow snapshot is in the figure 3.2). The visualization tool has one extra input — clusters — used for the coloring by cluster. Therefore the "label" column (i.e., cluster numbers) must be extracted from the FCS file and sent as separated input.

The second example extends the basic workflow with computational tools (an example workflow snapshot is in the figure 3.3 and corresponding workflow diagram 3.4). The computational tools embed all characteristics of the cells into two-dimensional space, instead of plotting only two chosen characteristics as

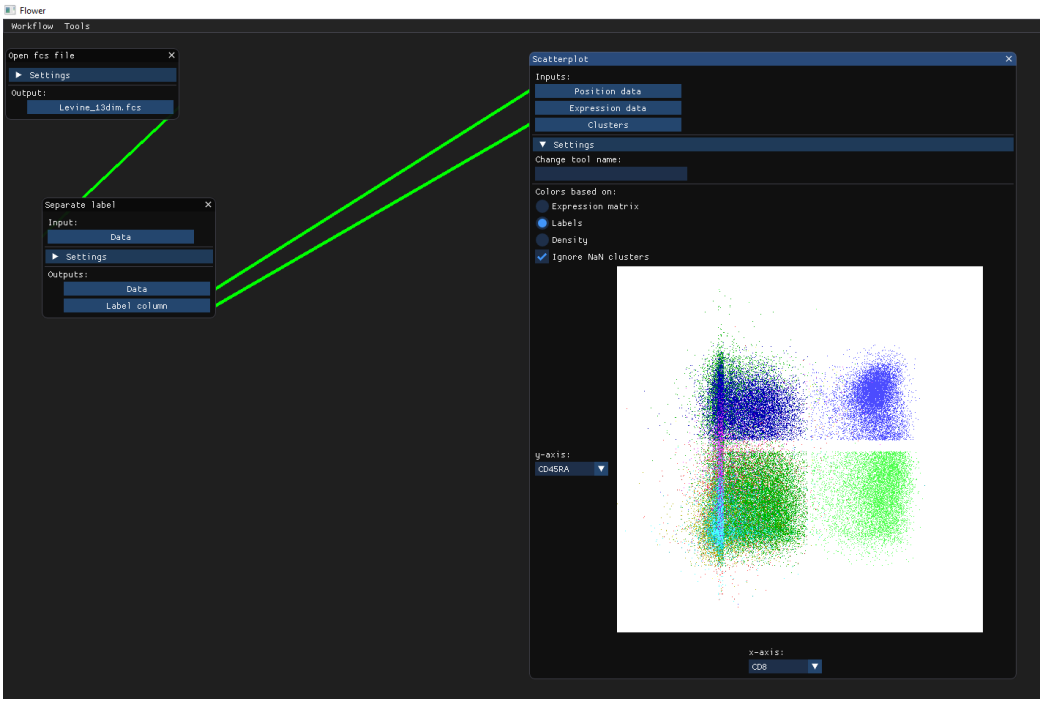


Figure 3.2: **The example workflow snapshot of the** `Levine_13dim.fcs` **dataset stored in the** `levine_basic.json` **file.** The *`OpenFile`* tool (on the left) opens the given FCS file and converts it into *`FloatMtxData`*. The *`SeparateLabel`* tool (in the middle) extracts the "label" column from given *`FloatMtxData`* if the column exists. The *`Scatterplot`* tool (on the right) visualizes two selected columns from the input *`FloatMtxData`* and colors them by input *`IntVectorData`* clusters.

in the first example. The clusters are computed from the input FCS file. The visualization tool do not use the "label" column because the "label" column is not usually present.

3.2.2 Example: identifying a cell cluster

The available tools can be used for the identification of various properties and types of cells. For demonstration, we show how to identify a cluster of activated helper T cells based solely on exploring the quickly available visualizations. The example workflow is available in file `levine_embed_expr.json` file, and demonstrates the embedding of the `levine_13dim.fcs` dataset into the two-dimensional space. The user can select various matrix columns (markers) to display, and successively identify the correct cell subsets by finding a cell population that matches all required properties. The process is shown in figure 3.5.

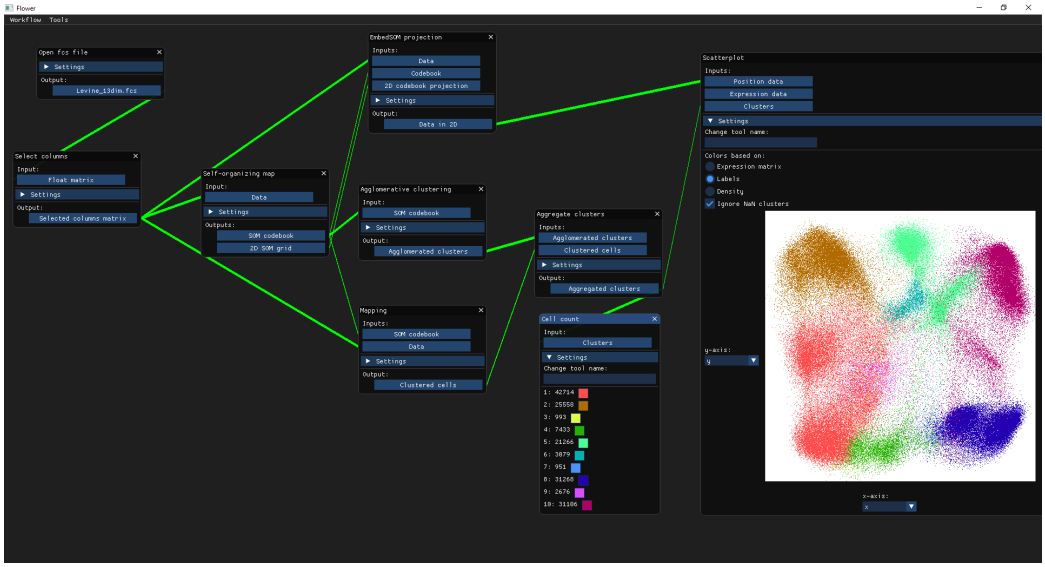


Figure 3.3: **The example of the workflow snapshot of the `Levine_13dim.fcs` dataset stored in the `levine_embed.json` file.** Input cells are embedded into two-dimensional space from the original multidimensional space (each dimension is one characteristic of the cell). The cluster colors are obtained from the clustering algorithm FlowSOM (FlowSOM algorithm is described in the section 1.2.2). Cells are embedded into the two-dimensional space by the dimensionality reduction algorithm EmbedSOM (EmbedSOM algorithm is described in the section 1.2.3). For statistics, the tool *`CellCount`* is used. It shows number of cells in each cluster with the same color as used in scatterplot. A more detailed description of the used tools is in the figure 3.4.

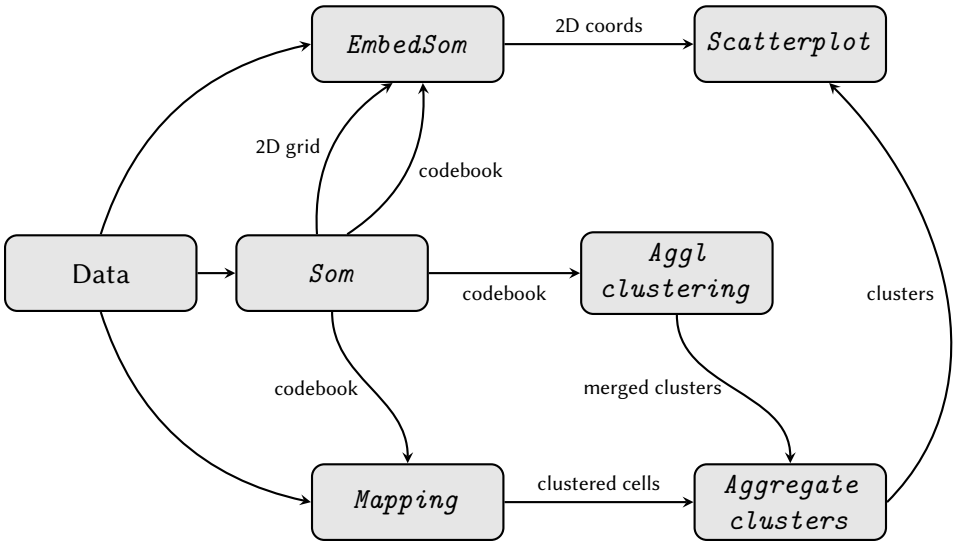


Figure 3.4: **A workflow diagram of the visualization by FlowSOM and EmbedSOM algorithms.** The Data box (on the left) can be any of the Data input tools (list of the input tools is in the section 2.7) combined with the Matrix manipulation tools (section 2.7). The Scatterplot box (on the top right) is a Visualization tool (section 2.7). All other boxes are the Computational tools (section 2.7). The input data (FCS files) are processed by the Computational tools and visualized by the Visualization tool.

For further analysis, the count of the cells in a cell cluster (population) can be obtained from the tool *CellCount* (the usage is shown in the figure 3.3).

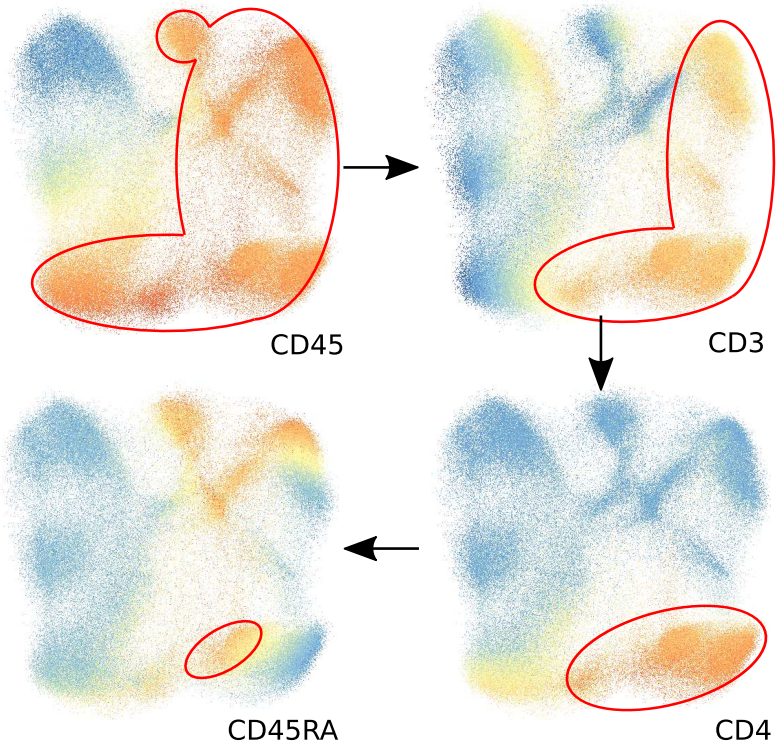


Figure 3.5: **Process of identifying the cluster of activated helper T cells from the embedding.** In the embedding of the `levine_13dim.fcs` dataset, the target cell population can be visually identified by the presence of the relevant surface markers, as an intersection of cell groups where markers CD45 (lymphocyte marker), CD3 (T cell marker), CD4 (helper T cell marker) and CD45RA (one of the cell activation markers) are present. In Flower, this is achieved by the visualization tool, where the user can easily select the markers and select the population of interest e.g. in the clusters from FlowSOM. In this example, cell populations are visualized by expression coloring (described in section 2.4.4).

Conclusion

In this thesis, we implemented a proof-of-concept software called 'Flower' that solves all goals (stated in the section 2.1) with results as follows:

1. Mapping files into the memory enables the processing of large datasets without run out of memory. The memory usage does not increase rapidly, even with gigabytes of data.

2. OpenGL provides efficient hardware-accelerated visualizations of colored two-dimensional plots.

3. The separate thread for executing algorithms in the background makes GUI responsive even during long computations. However, there are cases when the computational thread must be finished before the new task is executed (e.g., importing new workflow or closing the application).

4. The FlowSOM algorithm is implemented in the software, and hence the basic functionality and usefulness are already provided. The new tool classes can be easily added to the workflow, and hence the software is prepared for future extensions.

Future work Because our results have confirmed the expectations on the software, it resulted in a solid basis for future expansions that will make the software accessible and useful for researchers and clinicians. In the future, we want to expand Flower by new tools and new visualization techniques. Logical separation of the computational part from the GUI gives the posssibility of moving computations into remote computers. This would allow transparent usage of huge compute clusters even for parallel operations (for example, Galaxy [3] uses this approach but not so interactively).

[3] `https://www.immportgalaxy.org/user/login?use_panels=True&redirect=%2F`

Bibliography

[1] Nima Aghaeepour et al. “An immune clock of human pregnancy”. In: *Science immunology* 2.15 (2017).

[2] Rich Caruana et al. “Meta clustering”. In: *Sixth International Conference on Data Mining (ICDM’06)*. IEEE. 2006, pp. 107–118.

[3] Qing Chang et al. “Imaging mass cytometry”. In: *Cytometry part A* 91.2 (2017), pp. 160–169.

[4] Regina K Cheung and Paul J Utz. “CyTOF—the next generation of cell detection”. In: *Nature Reviews Rheumatology* 7.9 (2011), pp. 502–503.

[5] Elaine Coustan-Smith et al. “Prognostic importance of measuring early clearance of leukemic cells by flow cytometry in childhood acute lymphoblastic leukemia”. In: *Blood, The Journal of the American Society of Hematology* 100.1 (2002), pp. 52–58.

[6] Zbigniew Darzynkiewicz et al. “Cytometry in cell necrobiology: analysis of apoptosis and accidental cell death (necrosis)”. In: *Cytometry: The Journal of the International Society for Analytical Cytology* 27.1 (1997), pp. 1–20.

[7] Hazel M Davey and Douglas B Kell. “Flow cytometry and cell sorting of heterogeneous microbial populations: the importance of single-cell analyses.” In: *Microbiol. Mol. Biol. Rev.* 60.4 (1996), pp. 641–696.

[8] Lynn G Dressler and Sue A Bartow. “DNA flow cytometry in solid tumors: practical aspects and clinical applications.” In: *Seminars in diagnostic pathology*. Vol. 6. 1. 1989, pp. 55–82.

[9] William Galbraith et al. “Imaging cytometry by multiparameter fluorescence”. In: *Cytometry: The Journal of the International Society for Analytical Cytology* 12.7 (1991), pp. 579–596.

[10] B Greve et al. “Flow cytometry in transfusion medicine: development, strategies and applications”. In: *Transfusion Medicine and Hemotherapy* 31.3 (2004), pp. 152–161.

[11] Andrew B Hastings, Alok N Choudhary, and Harriet G Coverston. *Method and system for collective file access using an mmap (memory-mapped file).* US Patent 7,647,471. 2010.

[12] Stephen C Johnson. "Hierarchical clustering schemes". In: *Psychometrika* 32.3 (1967), pp. 241–254.

[13] Teuvo Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[14] Aleksandra A Kolodziejczyk et al. "The technology and biology of single-cell RNA sequencing". In: *Molecular cell* 58.4 (2015), pp. 610–620.

[15] Miroslav Kratochvíl, Abhishek Koladiya, and Jiří Vondrášek. "Generalized EmbedSOM on quadtree-structured self-organizing maps". In: *F1000Research* 8.2120 (2020), p. 2120.

[16] Miroslav Kratochvíl et al. "ShinySOM: graphical SOM-based analysis of single-cell cytometry data". In: *Bioinformatics* (2020).

[17] Peter O Krutzik et al. "Analysis of protein phosphorylation and cellular signaling events by flow cytometry: techniques and clinical applications". In: *Clinical immunology* 110.3 (2004), pp. 206–221.

[18] Jacob H Levine et al. "Data-driven phenotypic dissection of AML reveals progenitor-like cells that correlate with prognosis". In: *Cell* 162.1 (2015), pp. 184–197.

[19] Laurens van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE". In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[20] Dominique Marie et al. "Enumeration of marine viruses in culture and natural samples by flow cytometry". In: *Appl. Environ. Microbiol.* 65.1 (1999), pp. 45–52.

[21] Leland McInnes, John Healy, and James Melville. "Umap: Uniform manifold approximation and projection for dimension reduction". In: *arXiv preprint arXiv:1802.03426* (2018).

[22] Kevin R Moon et al. "PHATE: a dimensionality reduction method for visualizing trajectory structures in high-dimensional biological data". In: *bioRxiv* (2017), p. 120378.

[23] John P Nolan and Danilo Condello. "Spectral flow cytometry". In: *Current protocols in cytometry* 63.1 (2013), pp. 1–27.

[24] RJ Olson, D Vaulot, and SW Chisholm. "Marine phytoplankton distributions measured using shipboard flow cytometry". In: *Deep Sea Research Part A. Oceanographic Research Papers* 32.10 (1985), pp. 1273–1280.

[25] Michael G Ormerod and Patrick R Imrie. “Flow cytometry”. In: *Animal Cell Culture*. Springer, 1990, pp. 543–558.

[26] Matthew Partridge and Rafael Calvo. “Fast dimensionality reduction and simple PCA”. In: *Intelligent data analysis* 2.3 (1997), pp. 292–298.

[27] F Reggeti and D Bienzle. “Flow cytometry in veterinary oncology”. In: *Veterinary pathology* 48.1 (2011), pp. 223–235.

[28] Yvan Saeys, Sofie Van Gassen, and Bart N Lambrecht. “Computational flow cytometry: helping to make sense of high-dimensional immunology data”. In: *Nature Reviews Immunology* 16.7 (2016), p. 449.

[29] Larry Seamer. “Flow cytometry standard (FCS) data file format”. In: *In Living Color*. Springer, 2000, pp. 57–61.

[30] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. “Towards a theory of cache-efficient algorithms”. In: *Journal of the ACM (JACM)* 49.6 (2002), pp. 828–858.

[31] Matthew H Spitzer and Garry P Nolan. “Mass cytometry: single cells, many features”. In: *Cell* 165.4 (2016), pp. 780–791.

[32] Attila Tárnok and Andreas OH Gerstner. “Clinical applications of laser scanning cytometry”. In: *Cytometry: The Journal of the International Society for Analytical Cytology* 50.3 (2002), pp. 133–143.

[33] Sofie Van Gassen et al. “FlowSOM: Using self-organizing maps for visualization and interpretation of cytometry data”. In: *Cytometry Part A* 87.7 (2015), pp. 636–645.

[34] Mohammed J Zaki. “SPADE: An efficient algorithm for mining frequent sequences”. In: *Machine learning* 42.1-2 (2001), pp. 31–60.

Appendix A

How to build Flower software

A.1 Prerequisites

The same prerequisites for Linux and Windows are Cmake [1] (version $\geq$ 3.17.0) and C++ compiler complying with C++17 standard. The make command is a prerequisite for Linux. For Windows, the build tutorial works with Visual studio 2019 (but any other building software can be chosen).

A.2 Installation of external dependencies

The only external dependency is SDL2 library [2].

A.2.1 Linux

Install `libsdl2-dev` (on Debian-based systems) or `SDL2-devel` (on Red Hat-based systems), or similar (depending on the Linux distribution).

A.2.2 Windows

Recommended installation of the SDL2 library is through `Vcpkg` [3] package manager (it installs libraries inside the Vcpkg directory and not into the system).

1. Install Vcpkg following the instructions [3] (if not installed already).

2. Run Powershell as Administrator.

[1] `https://cmake.org/`
[2] `https://www.libsdl.org/`
[3] `https://github.com/microsoft/vcpkg`

3. Change directory to the directory of installed Vcpkg.

   ```
   cd vcpkg
   ```

4. Install SDL2 package.

   ```
   .\vcpkg install sdl2:x64-windows
   ```

A.3 Build instructions

A.3.1 Linux

1. Change to the project directory.

   ```
   cd Flower
   ```

2. Clone all external libraries.

   ```
   git clone https://github.com/mosra/corrade.git
   git clone https://github.com/mosra/magnum.git
   git clone https://github.com/mosra/magnum-integration.git
   git clone https://github.com/ocornut/imgui.git
   ```

3. Create build subdirectory and change to build directory.

   ```
   mkdir build
   cd build
   ```

4. Run Cmake.

   ```
   cmake ..
   ```

5. Make files are ready, build the application.

   ```
   make
   ```

6. The application is in the ./bin/.

   ```
   cd bin
   ./cytometry
   ```

7. The application is ready, follow the How to use instructions in appendix B.

A.3.2 Windows

1. Change to the project directory.

   ```
   cd Flower
   ```

2. Clone all external libraries.

   ```
   git clone https://github.com/mosra/corrade.git
   git clone https://github.com/mosra/magnum.git
   git clone https://github.com/mosra/magnum-integration.git
   git clone https://github.com/ocornut/imgui.git
   ```

3. Create builld subdirectory and change to build directory.

   ```
   mkdir build
   cd build
   ```

4. Run Cmake, with specified Visual Studio version, architecture and with path to the vcpkg.cmake file in the directory of installed Vcpkg.

   ```
   cmake ..  -G "Visual Studio 16 2019" -A x64
   -DCMAKE_TOOLCHAIN_FILE=
   "[vcpkg root]\scripts\buildsystems\vcpkg.cmake"
   ```

5. Open “Flower\build\Cytometry.sln” and set “cytometry” as Startup Project in the Visual Studio.

6. Build the solution in the Visual studio.

7. Now, the solution is ready! Follow the How to use instructions in appendix B.

Appendix B

How to use Flower software

B.1 Technical documentation

The technical documentation is generated by `Doxygen` [1]. The documentation is stored in the `Flower/src/doc/html/index.html` file. It covers only an overview of the used classes with basic commentaries of the methods and attributes. The main reason why the Doxygen was used is because of the future development of the tool.

B.2 Data preparation

This software works with FCS files — cytometry datasets. These datasets are obtained from the Flow repository [2]. All of the result pictures that are used in the thesis use the `Levine_13dim.fcs` dataset [3]. Download this dataset for following example usage. The file can be stored anywhere, because its path is chosen in the application.

B.3 Functionality

Add a new tool Choose the tool from the top left menu and category (the descriptive screenshot is in the figure B.1).

"Settings" bar in the tools Each tool has "Settings" sub menu, where the tool can be renamed, and other actions can be done (unique for each tool type) (the

[1] `https://www.doxygen.nl/index.html`
[2] `https://flowrepository.org/`
[3] `https://flowrepository.org/experiments/817/download_ziped_files`

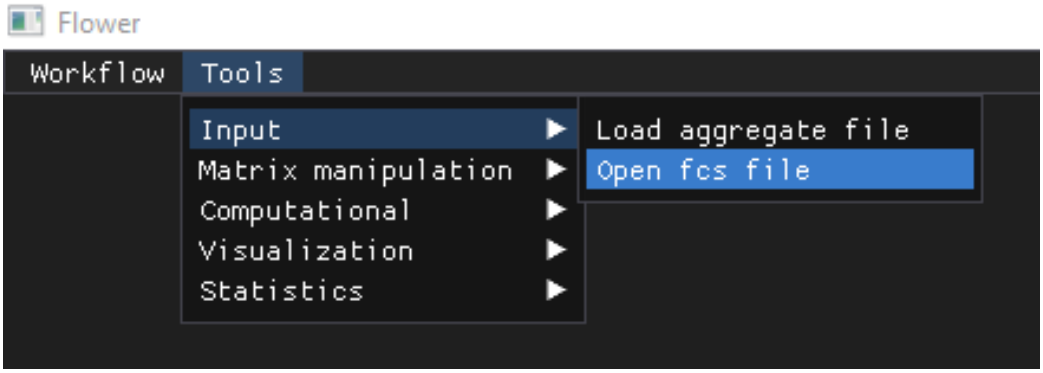


Figure B.1: Add a new tool.

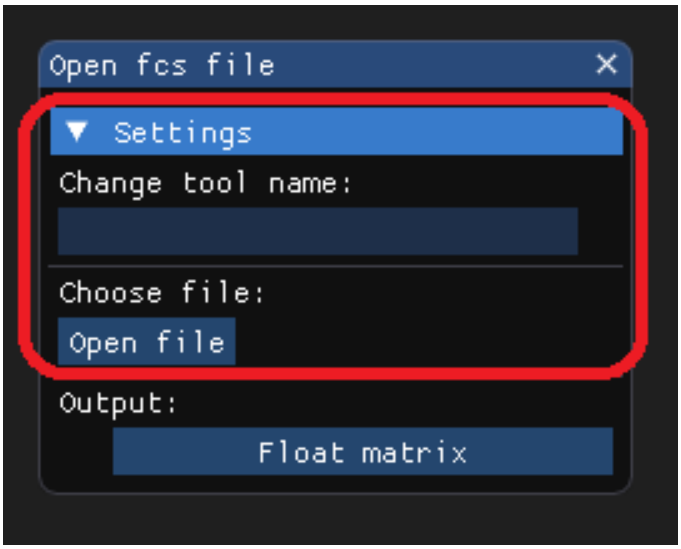


Figure B.2: Tool settings.

descriptive screenshot is in the figure B.2). The settings sub menu is opened by clicking on the “Settings” bar.

Connect tools Tools can be connected by input and output buttons. Input buttons are in the top of the tool and output buttons are in the bottom of the tool. The tools can be connected by buttons only if the buttons exist (i.e., if the tool has no input, it has no input buttons and the same for output). The output button must be pressed first and then the input button (the descriptive screenshot is in the figure B.3).

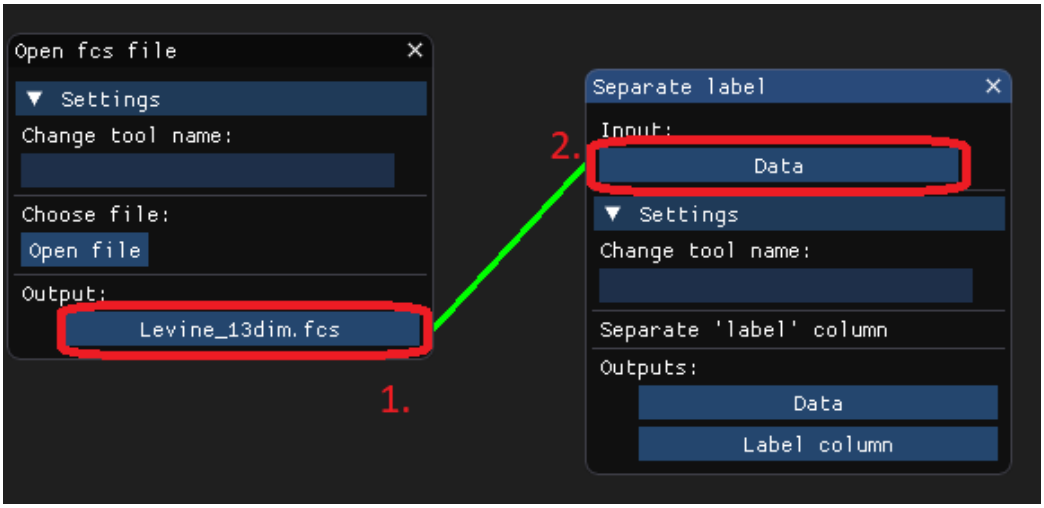


Figure B.3: Connection of the tools.

Disconnect tools The tool buttons can be disconnected by clicking the `input` button of the connected pair (the descriptive screenshot is in the figure B.4).

Export the workflow The tools, their settings and their connections can be exported as a JSON file. The "Export" is chosen in the top left menu (under the Workflow menu item; the descriptive screenshot is in the figure B.1). The destination file path and file name are chosen (the descriptive screenshot is in the figure B.5 and figure B.6).

Import the workflow The workflow can be imported as JSON file (under the Workflow menu item; the descriptive screenshot is in the figure B.7). There are two workflows prepared in the Flower/src/json/. The example workflows work with `Levine_13dim.fcs` dataset. The path to the `Levine_13dim.fcs` must be modified in the JSON files (the descriptive screenshot is in the figure B.8), so that they match the file layout on the used computer.

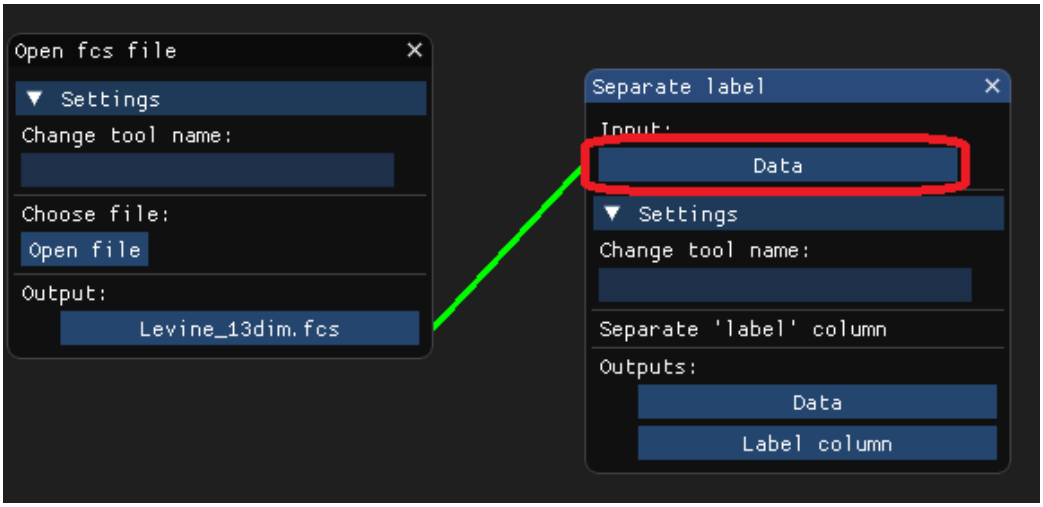


Figure B.4: Disconnection of the tools.

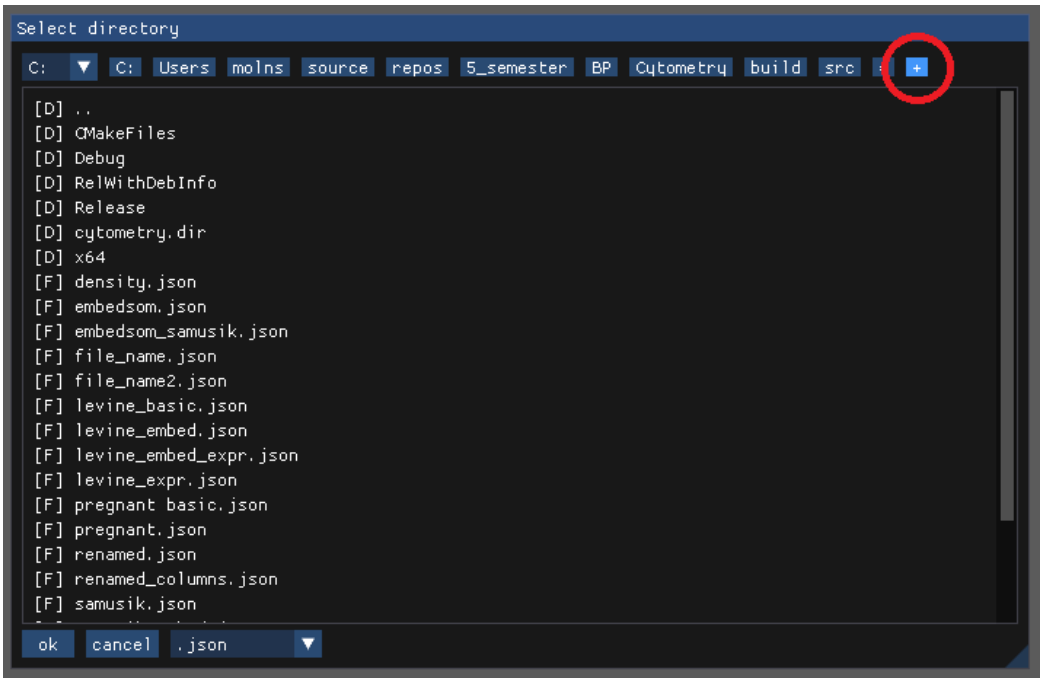


Figure B.5: **Select directory dialog window.** New directory can be created by "+" button in the red circle (top right). By clicking "ok", the current directory will be selected and Write file name dialog will open (the descriptive screenshot is in the figure B.6)

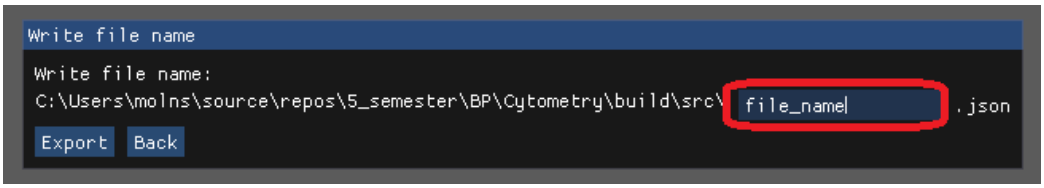


Figure B.6: `Write file name dialog window`. The file name can be changed in the red box. By clicking "Export", the workflow will be exported into the selected file path with written file name. By clicking "Back", it will return to the Select directory file dialog (the descriptive screenshot is in the figure B.5).

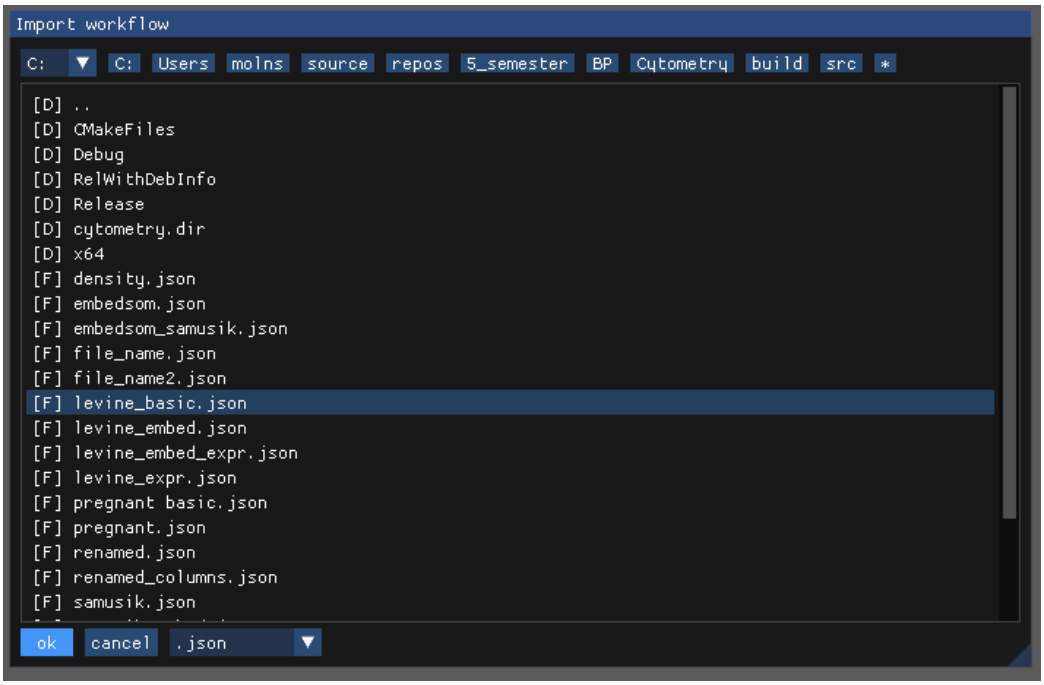


Figure B.7: Import file dialog works as normal open file dialog. Choose JSON file and click "ok" or exit by "Cancel".

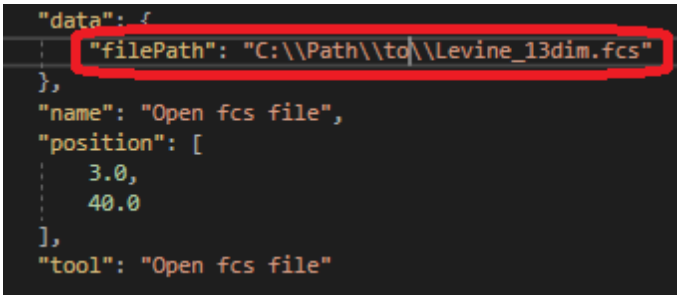


Figure B.8: Change the path to the Levine_13dim.fcs in the JSON files.

Appendix C

Creating new tools

The tutorial explains how to create and add a new tool class (later called *NewToolClass*) into a program. A new .cpp file must be created (NewTool.cpp) in the src/tools directory. Its name must be added into CMakeLists.txt (tools/NewTool.cpp) in the src directory. The template of the new tool is in the listing 2. The template can be copied to the .cpp file and extended as needed. The tutorial follows the structure of the template.

Includes

1. The current implementation of the software provides two data types: *FloatMtxData* and *IntVectorData*. They are included if they are needed.
2. The *Tool* must always be included because it is the base class of the *NewToolClass*. Also, the *ToolRegistry* needs to be always included, so the tool can be instantiated.

Class declarations

1. The *NewToolClass* must publicly inherit from the *Tool*.
2. The *NewToolClass* constructor must contain definitions of the inputs and outputs (as described in the listing). The same tool name as used in the REGISTER_TOOL must be sent to the *Tool* constructor.
3. All virtual methods of the *Tool* must be overridden and implemented.

Parameter struct

1. The struct *NewToolParams* must always be implemented and inherited from the *Parameters*. The parameter attributes are the settings obtained from the `on_render` function.

Computational function

1. The computational function is a static function defined before the definition of the `get_recompute` function. The computational function may use more static functions defined in the .cpp file.

2. The new computational function computes outputs from given inputs and input parameters defined in the struct *NewToolParams*.

3. The `get_recompute` function returns a pointer to the previously defined computational function and the already defined struct *NewToolParams*.

Registration

1. The tool must be registered by the REGISTER_TOOL macro. The first argument is the category of the tool.

2. Currently, the supported categories are: "input", "matrix", "comp", "visualize", and "stat". Each category represents the functionality of the tool described in the section 2.7.

3. The second argument is the name of the new tool, also sent as input in the constructor of the base class *Tool*.

4. The third argument is the name of the new class (*NewToolClass*).

New data type

1. The new data type can be implemented by following the data inheritance structure in the section 2.6 if the provided data types are not sufficient. If the new data type works with the temporary file on the disk, the same pattern can be used as in the classes *FloatMtxData* or *IntVectorData*.

2. The new type in *DataTypes* enum (in common.h) must also be added together with a new string representation of this type in the `types` variable in Tool.h.

Listing 2 Example NewToolClass.

```
#include "../FloatMtxData.h" // optional
#include "../IntVectorData.h" // optional
#include "../Tool.h"
#include "../ToolRegistry.h"

class NewToolClass: public Tool
{
 public:
  NewToolClass() : Tool("New tool name")
  {
    // One input
    inputs.emplace_back("matrix", DataTypes::FLOAT_MATRIX);
    // One output
    outputs.emplace_back("vector", DataTypes::INT_VECTOR);
  }

  void on_render() override { /*Some ImGui widgets*/ }
  void input_changed(const data_vector&) override
    { ask_recompute(); }
  recomp_fn_pair get_recompute() override;
  void tool_recomputed(Parameters*) override
    { /* New tool was recomputed.*/ }

  json to_json() override {return json();}
  void from_json(json&) override {}
};

struct NewToolParams : public Parameters
{};

static bool some_comp_fn(
  const data_vector&, data_vector&, Parameters*);
{
  // Some computations
  // if something went wrong
  // return false;
  // else
  return true;
}

recomp_fn_pair NewToolClass::get_recompute()
{
  return std::make_pair(
    some_comp_fn, std::make_unique<NewToolParams>());
}

REGISTER_TOOL("category", "New tool name", NewToolClass)
```