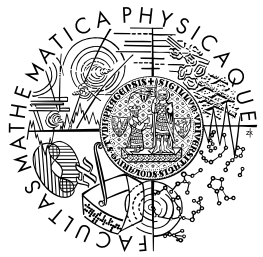

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Soňa Molnárová

Throughput optimization of a multistage data visualisation pipeline

Department of distributed and dependable systems

Supervisor of the master thesis: Mgr. Adam Šmelko
Study programme: Computer science
Study branch: Software Systems

Prague 2023

I declare that I carried out this master 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

I would like to dedicate my thesis to those who played a vital role in my academic journey.

First and foremost, I would like to thank my parents and closest friends for their endless support and belief in my abilities during this challenging time. I would also like to thank my supervisor Mgr. Adam Šmelko for his patience with me, invaluable insights, and constructive criticism. Great words of appreciation also go to my consultant RNDr. Miroslav Kratochvíl, Ph.D. for his guidance, late-night discussions, and words of wisdom that kept me motivated.

Each of them helped me overcome the obstacles and not give up, for which I am incredibly grateful.

Title: Throughput optimization of a multistage data visualisation pipeline

Author: Soňa Molnárová

Department: Department of distributed and dependable systems

Supervisor: Mgr. Adam Šmelko, Department of distributed and dependable systems

Abstract: Recent technological advances allowed vast increases of the size of datasets acquired by flow cytometry. With more cells and features captured, manual exploration and analysis of the data become challenging; multiple unsupervised methods were thus created to simplify the task. This thesis describes BlosSOM, a software that further improves the analysis possibilities by allowing interactive CUDA-accelerated supervision of the most common analysis approaches — dimensionality reduction, clustering, and annotation. The thesis overviews the features of BlosSOM implemented by the thesis author for the previous research project and details new developments, mainly a new dynamic workload balancing approach that allows BlosSOM to easily scale to very complex datasets while maintaining interactive framerate. Together with other improvements, BlosSOM is a production-ready software that offers all essential tools to analyze the flow cytometry data.

Keywords: optimization, gpu, visualisation

Contents

Introduction

Specialized data visualization techniques form a crucial building stone of publishing in virtually any data-driven research field. Analysis of data in cytometry (mainly flow and mass cytometry [1, 2]) has spawned many visual analytics approaches that helped researchers to uncover many complex phenomena that drive the progress in immunology [3, 4, 5] and oncology [6, 7, 8]. Briefly, methods such as UMAP [9] easily provide dataset renderings such as the one in the Figure 1, which may be intuitively interpreted even by users with no background in computational analysis and statistics.

This thesis reports the results which stem from a longer line of research in high-performance dimensionality reduction and clustering analysis of cytometry data: Originating with FlowSOM [10], which gave a fast and precise clustering possibility for the common datasets and some new directions for the visualizations based on drawing the self-organizing maps [11], continuing with the development of EmbedSOM [12] which improved the cell visualization for FlowSOM-style analysis, we developed FLOWer [13] to enable the interactive building of analysis pipelines that use these tools by users. Recently, this was continued by a research project where the author of this thesis developed BlosSOM [14], a highly specialized and optionally CUDA-accelerated tool that combines both methods in a highly intuitive interactive user interface. Most notably, BlosSOM was (to the best of our knowledge) the first tool efficient enough to allow users to manipulate a useful dimensionality reduction of more than 30-dimensional datasets and millions of data points, all in a completely interactive and smooth manner without any noticeable pauses for computation.

A crucial property of the supervision in BlosSOM was the ability of non-programmer domain-expert users to easily interact with the visualization and tune (or debug) it to improve upon the outcomes of the unsupervised methods. For illustration, the Figure 1 shows an example of a dataset visualization that admits a brief reorganization by the user without losing any data visualization fidelity at all. This is complicated to implement in an unsupervised case — the domain expert needs to encode the additional information to an analysis script (which may be complicated for non-programmers) and re-run the dimensionality reduction,

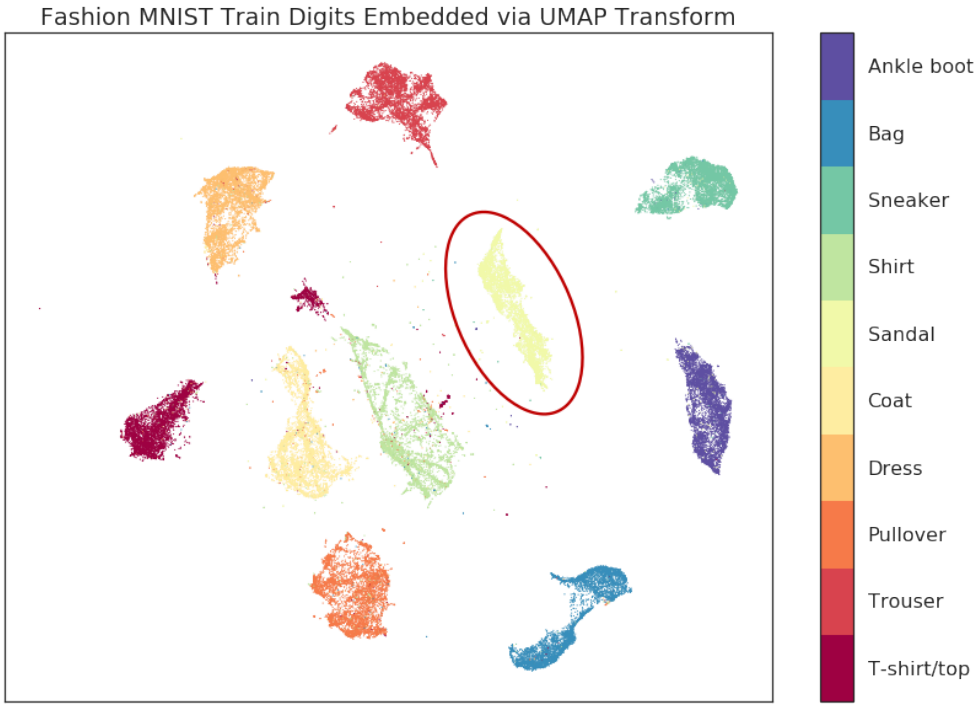

Figure 1 The static embedding of the fashion MNIST dataset with the UMAP algorithm. The resulting embedding shows sandals (the cluster in the red ellipse) to be more similar to shirts (pale green on the bottom left of the ellipse) than to sneakers (darker green on the top right of the ellipse). The user might like to change this result and move the cluster in the red ellipse closer to the right to correspond with the real world. In the static embedding, this is not possible. (The image taken from the UMAP documentation [15])

hoping that the unsupervised algorithm will pick the additional hint "correctly" without any undesired side effects. In BlosSOM, the same reorganization and recomputation are implemented by simply dragging several control points that internally guide the dimensionality reduction, providing an instant preview of the result.

As the main goal and result, this thesis consolidates and extends the properties of the BlosSOM prototype generated in the research project. Problems with BlosSOM prototype were:

- It used a robust rendering engine with too much unnecessary functionality that slowed the rendering and compilation.
- The data were processed in fixed-sized batches in each frame which was not scalable to the used hardware.

The contributions of our thesis are as follows:

- Rewriting the software to a light-weight rendering library.
- Using dynamic workload scheduling of the computational pipeline that computes partial updates to the dataset, which enables BlosSOM to run on various hardware under diverse workloads, properly utilizing the available hardware without the danger of framerate drops.
- Adding several other improvements in BlosSOM include mainly the new possibilities of user interaction, which give more powerful ways to reorganize the visualizations, and the ability to manually mark and annotate various identified parts of the dataset for later statistical analysis.

Notably, we found that for large datasets, most of the work is spent in GPU, and CPU-based parallelism or even asynchronous computation with GPU does not improve the perceived throughput; which directed the further optimization of the rendering pipeline — we created a texture rendering method described in the Section 4.2, which significantly sped up the rendering.

In outcome, the thesis has finalized the features of BlosSOM that are most important for realistic use by cytometry domain experts.

The thesis is organized as follows: Chapter 1 of the thesis gives an overview of properties and typical processing of cytometry data. Chapter 2 details the design and implementation of the BlosSOM tool. Chapter 3 describes the methods for dynamic balancing of batch sizes, and briefly benchmarks the results. Chapter 4 summarizes other software and functionality improvements in the BlosSOM implemented by this thesis. Finally, the conclusion 4.4 provides a summary of the achieved goals and further challenges.

Chapter 1

Background

This chapter focuses on the cytometry data and how they are obtained. The following sections describe the algorithms used for cytometry data visualization and analysis. Further, we describe how they are applied in our software BlosSOM.

1.1 Cytometry data acquisition and visualization

Cytometry represents a biological term for various methods that measure the properties of single cells — such as flow-cytometry [1, 16] and mass-cytometry [2, 17] (for more information about cytometry and methods used for obtaining data and visualization methods, see [13]). In flow cytometry, each cell from the sample is marked by chemicals that bind only to specific molecules and emits different colors. Then each cell, one by one, is excited by a laser and emits different colors of the color spectrum. The amount of the emitted color in different parts of the spectra is then measured and stored in a data matrix. Other techniques have similar principles. These techniques allow us to obtain tens of properties of each cell.

Cytometry data are represented as multidimensional data in the matrix (as shown in the Figure 1.1). Each row is one cell from the sample, and each column represents one measured property of the cell. Each column also represents one dimension of the multidimensional space. Such representation allows us to observe relationships between each parameter by applying various algorithms and techniques used for visualization and analysis.

The most common analysis method among biologists is the linear projection of the properties to two-dimensional space, called gating (see Figure 1.2). While it is straightforward to visualize and read by humans, it can become very error-prone with the increasing number of dimensions. This technique is unsuitable for datasets with dimensions higher than two because many details may be

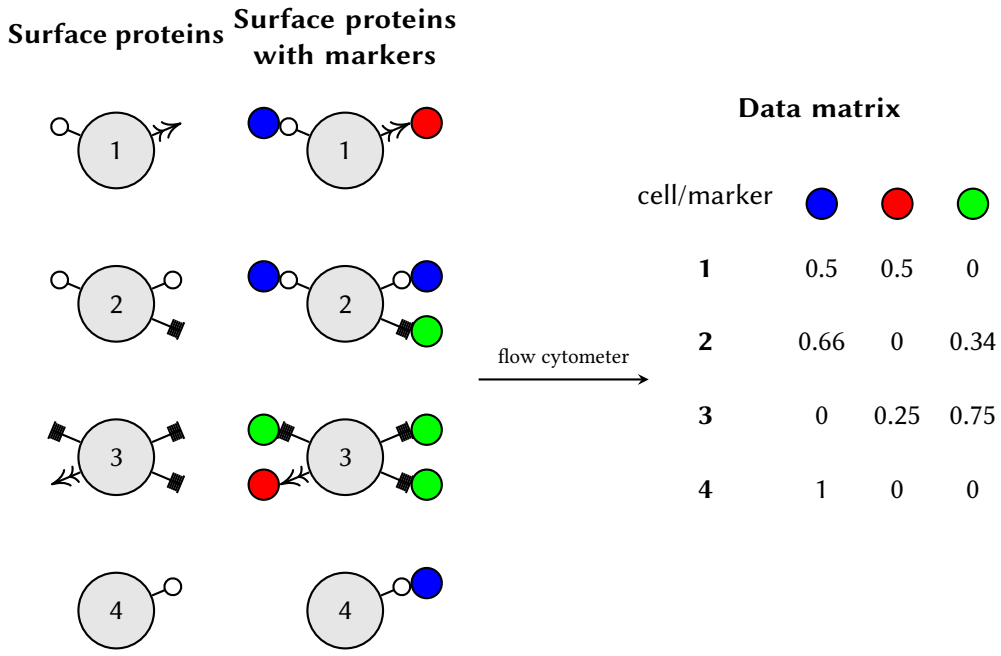

Figure 1.1 Overview of the method of obtaining the flow cytometry data. Cells (the grey circles numbered 1–4) have various specific molecules on the surface (three types of surface proteins are present — different types of arrows in the left column). The surface proteins are marked by chemicals that can bind only to certain specific cell molecules and have a specific distinguishing "color" (i.e., they emit specific light spectra upon excitation — in this case, red, blue, and green). The chemicals are excited by a 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 substances (each bonding to a different cell molecule). Measurements in the example table are idealized, neglecting the physical properties of the process (such as measurement noise and spectral overlaps). (Image originally authored by Molnárová [13])

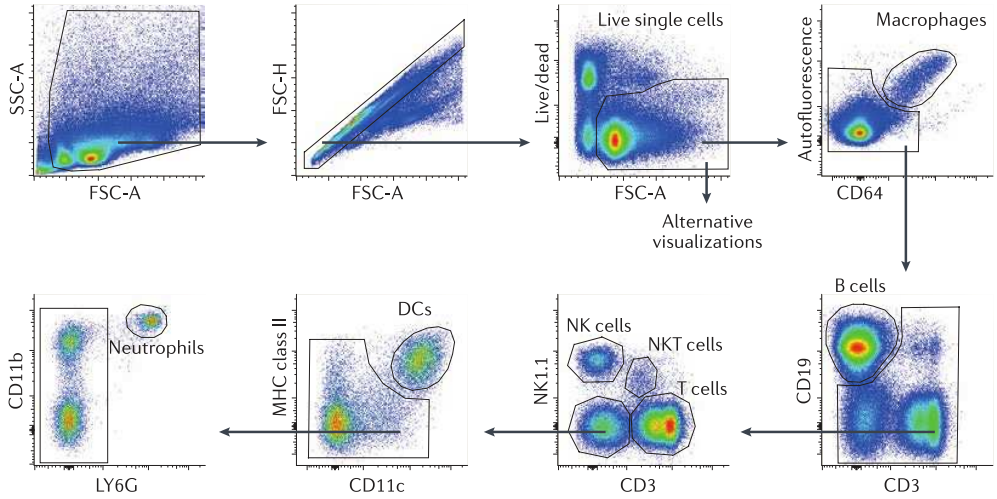

Figure 1.2 Example of the manual gating process — marking similar populations in different subdimensions. (Image originally authored by Saeys, Van Gassen, and Lambrecht [4])

overlooked.

In the last two decades, the non-linear projection algorithms called dimensionality reduction algorithms have helped to analyze datasets with more than two dimensions. These visualization techniques are still human-readable and give a satisfactory view of more advanced features that cannot be observed by gating. However, these algorithms are slow, and therefore user interaction was possible only on small, trimmed datasets with the loss of information. Thus in BlosSOM, we use a performant dimensionality reduction algorithm EmbedSOM [12] that is suitable for interactive visualization without trimming the dataset. An improved CUDA EmbedSOM [14] gives even better results thanks to the GPU-acceleration.

Cytometry data can be visualized in various ways — scatter plot, heatmap, dendrogram, or minimal spanning tree. In BlosSOM, we visualize the data by a two-dimensional scatterplot where each rendered point has its counterpart in high-dimensional data and represents one cell. The scatterplot shows the results of dimensionality reduction algorithms where the cells are arranged in clusters (see Figure 1.3).

1.2 Landmark-based approach to visualization

The main feature of the BlosSOM interactive visualization is a landmark-based approach to visualization. Landmarks are points in the same high-dimensional space as input data with counterparts in the two-dimensional space (see Fig-

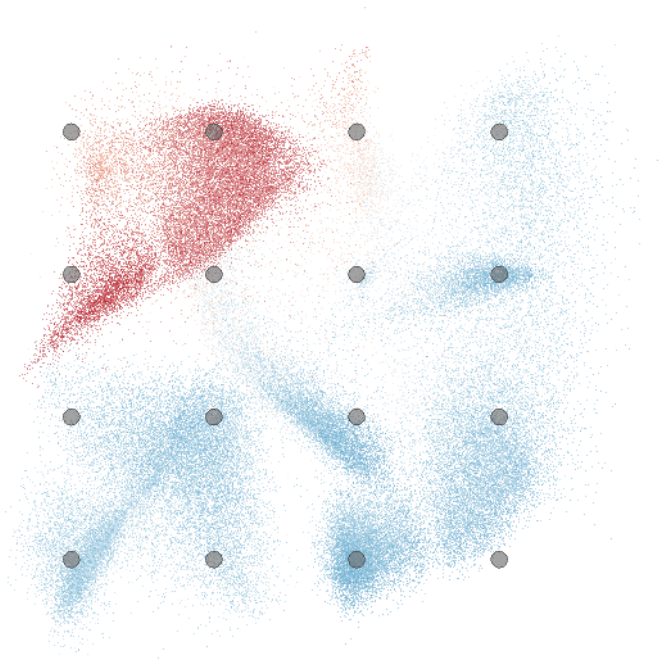

Figure 1.3 The scatterplot from BlosSOM of Levine dataset (authored by Levine et al. [18]) embedded by the EmbedSOM algorithm where cells are arranged in clusters. The scatterplot is colored by the expression CD45 which means that the red area contains cells with high measured values of the CD45 parameter. The gray circles are specific to the EmbedSOM algorithm and are not part of the scatterplot visualization. The gray circles are described in the following Section 1.2.

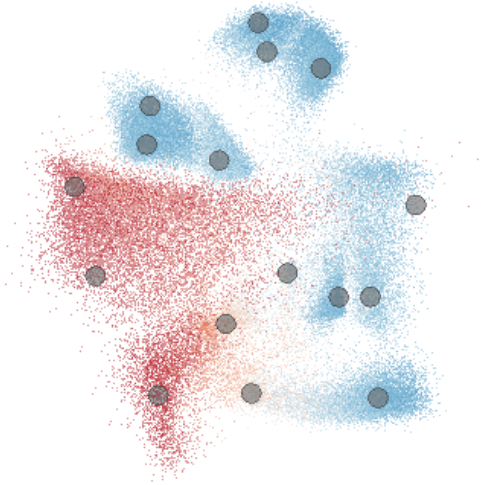

Figure 1.4 Two-dimensional landmarks (gray circles) visualized in BlosSOM with optimized landmark positions to better display the dataset distribution.

ure 1.4). The number of landmarks is significantly lower than the number of data points. Therefore they create a simplified version of the dataset. The underlying embedding algorithm EmbedSOM uses the high-dimensional landmarks to embed the high-dimensional data points to two-dimensional space according to the positions of two-dimensional landmarks.

In the original, non-interactive, unsupervised version of the EmbedSOM, the high-dimensional landmarks are typically chosen as a random subset of data points and are embedded in two-dimensional space by the t-SNE algorithm. Even though t-SNE is a rather time-demanding algorithm, it can be used to embed landmarks because the number of landmarks is relatively small.

BlosSOM is an interactive version of the EmbedSOM algorithm — the user can interact directly with two-dimensional landmarks and change the results of the embedding. Ideally, to make BlosSOM a fully supervised version of EmbedSOM, the user should also be able to move high-dimensional landmarks directly. However, this is an unfeasible task because of the inherent complexity of navigation in high-dimensional spaces. Hence, we use algorithms such as k-Means or SOM to automate the movement of high-dimensional landmarks, possibly also reflecting the user-driven changes in the layout of 2D landmarks. Two-dimensional landmarks can also be moved automatically by algorithms like t-SNE or graph layouting algorithms.

In the following sections, we further describe the algorithms used for the automated movement of high and low-dimensional landmarks and the algorithms for the actual embedding.

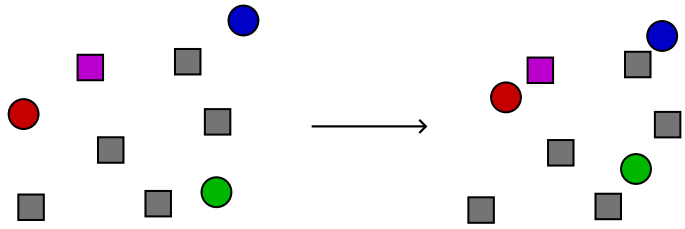

Figure 1.5 Modified k-Means algorithm. The circles are centroids, and the squares are data points. The pink square is a randomly chosen target data point. The closest (red) centroid moves the most to the target (the right picture). The other two (green and blue) centroids move only a little toward the target.

1.2.1 High-dimensional landmark positioning

In this section, the automated ways for high-dimensional landmark positioning are described. The entire dataset must be evenly covered with high-dimensional landmarks so the embedding shows all parts of the dataset. The user can choose which algorithm is running, adjust the parameters of these algorithms or stop them anytime.

k-Means

The k-Means algorithm [19] is usually used in clustering problems. It works in iterations; its input is the number of iterations i, centroids, and data points. In each iteration, each data point is assigned to the closest centroid. New positions of the centroids are computed as the mean of the positions of data points assigned to the centroid. After i iterations, the resulting centroids define dataset clusters. The output of the algorithm is the assignment of data points to the closest cluster.

In our case, we use the modification of this algorithm for the movement of high-dimensional landmarks, which are regarded as centroids. The algorithm is modified so that in i iterations, only one randomly chosen data point is processed in each iteration. Then, all centroids are moved toward the target data point, and the closest centroid is moved toward the target data point more. This algorithm is shown in the Figure 1.5. The $alpha$ parameter affects the moving speed of the closest centroid, and the $gravity$ parameter affects the moving speed of all centroids to the target data point.

Self-organizing maps

The self-organizing map [11](SOM) algorithm is very similar to the k-means clustering algorithm (see Section 1.2.1). The main difference is centroids. In k-means, they are independent, i.e., their positions do not affect the positions of

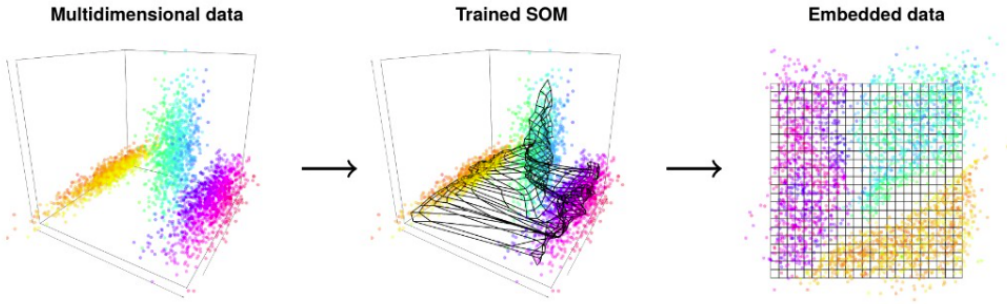

Figure 1.6 Preserved topology of the high-dimensional dataset in the SOM algorithm. On the left, there are high-dimensional data (three-dimensional). In the middle are the positions of neurons in the grid after i iterations. On the right is a low-dimensional (two-dimensional) map used for visualization. The points far from each other in the left picture are also far from each other in the right picture. It is the same with the close points. (Image originally authored by Kratochvil et al. [20])

other centroids. In the SOM algorithm, the centroids are connected and affect positions of each other. In SOM terminology, we use the term neurons rather than centroids. The usual way how to connect the neurons is a regular grid.

The input of the SOM algorithm is the same as for the k-Means algorithm — the number of iterations, neurons, and data points. Each data point is also assigned to the closest neuron in each iteration. However, when the new positions of neurons are computed, they are not computed only from the positions of data points but also from the positions of the neighboring neurons. If the neighboring neuron is too far away, it is attracted, and if it is too close, it is repelled.

The high-dimensional neurons have their counterparts in the low-dimensional space. The neurons are connected in the same grid pattern in both dimensions. In the low-dimensional space, the grid is called a map. High-dimensional points are projected to the map according to the high-dimensional grid. Points in a particular grid area are projected to the same area in the map. Thanks to this, the SOM algorithm preserves the topology of the high-dimensional data (see Figure 1.6). The map is the output of the algorithm — the assignment of the data points to clusters (neurons).

In BlosSOM, we use SOM in the same way as the k-Means algorithm — for positioning high-dimensional landmarks (neurons). We use the same modification as in the k-Means — we do not compute the closest neurons for every data point but only for k randomly selected data points. The only difference is in the computation of the new positions of neurons. In k-Means, the new positions are computed only from the distances between centroids and data points. In the SOM algorithm, the new positions are additionally affected by the connections between

neurons. As we are concerned only about the positioning of the landmarks, the modified algorithm does not output any map.

1.2.2 Two-dimensional landmark positioning

This section describes automated ways to move two-dimensional landmarks. Proper positioning of two-dimensional landmarks is crucial for comprehensible visualization. When a new dataset is loaded, the two-dimensional landmarks are at the default positions and do not move. The user can move each landmark freely with the mouse. The user can also use a positioning algorithm to position landmarks in an automated way. The user can intervene in the algorithm and freely move each landmark while the algorithm is still running and moving other landmarks. The user can choose which algorithm is running, adjust the parameters of the algorithms or stop them anytime.

k-NN graph generation

In BlosSOM, we do not use the k-NN graph generation algorithm for positioning two-dimensional landmarks. It is mentioned in this section because the Spring layout algorithm discussed below uses the results of this algorithm.

The k-NN (k-nearest neighbors) graph generation algorithm produces a graph where each node has edges with at most k other nodes. The input of the algorithm are positions of nodes. It finds k closest nodes for each node and creates an edge between them. It also stores distances between them — usually Euclidean distances. The output is the set of edges and their lengths. Since it has to inspect all pairs of nodes, it is slow on the large set of nodes.

In BlosSOM, the nodes are high-dimensional landmarks. High-dimensional edges are rendered between two-dimensional landmarks only for illustration — to show which landmarks are close to each other in the high-dimensional space (see the Figure 1.7).

Spring layout

The Spring layout algorithm [21] is a force-based graph layout algorithm. The algorithm aims to create a graph layout with acceptably short edges and, simultaneously, well-separated nodes. Its input is a graph with nodes and edges. The edges act as springs that pull both ends together to some extent. The repulsive force is computed from the nodes based on their proximity. The attractive force is computed from the edges based on their lengths. At the end of each iteration, the velocity of each node is updated according to the forces. The velocity is slowed

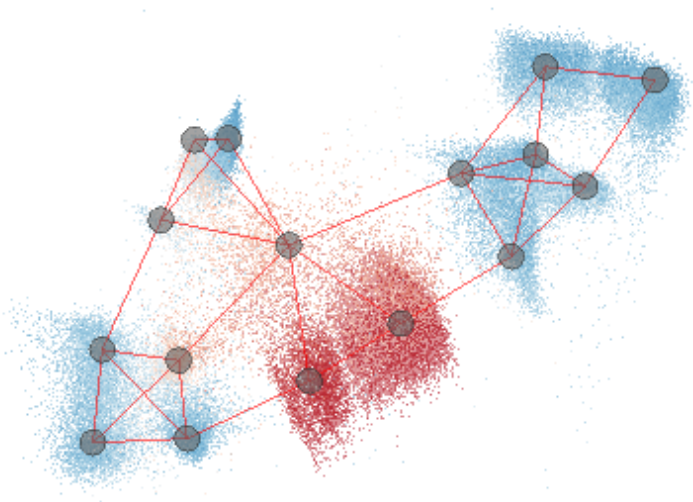

Figure 1.7 An example result of the k-NN graph generation algorithm in BlosSOM with $k = 3$. The gray circles are two-dimensional landmarks, and the red lines between them are edges. Since k-NN is an asymmetric algorithm, each node can have more than k edges.

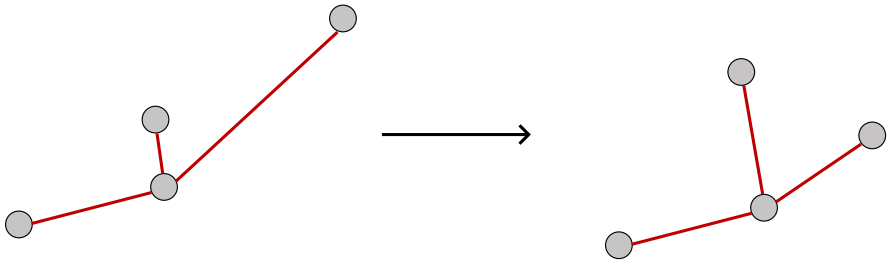

Figure 1.8 One iteration of the Spring layout algorithm. Nodes (gray circles) are repulsed between each other, and nodes connected with edges (red line) are attracted to each other. The leftmost node stays the same because it is at an approximately viable position. The node closest to the middle one is repulsed because the nodes are too close to each other. The node most to the right is attracted to the middle one because the edge between them is too long, so it has to shrink.

down each frame by an arbitrary number, so the layout is stabilized after some time. One iteration of the algorithm is shown in the Figure 1.8.

In BlosSOM, the nodes of the graph are two-dimensional landmarks. The edges are from the k-NN graph generation algorithm (described in the Section 1.2.2). In each frame, landmarks are moved according to the forces.

t-SNE

The t-SNE (t-Distributed Stochastic Neighbor Embedding) algorithm [22] is a dimensionality reduction algorithm used for clustering and analysis of underlying structures in datasets (see Figure 1.9). The algorithm embeds high-dimensional data into low-dimensional space (typically two- or three-dimensional). At the beginning of the algorithm, the matrix of similarities for high-dimensional data points is computed between all pairs of points using Gaussian. In each iteration, the matrix of similarities between low-dimensional data points is computed using Student t-distribution. The goal of t-SNE is to minimize a Kullback-Leibler divergence cost function by the gradient descent method in each iteration, i.e., to minimize the difference between the similarity matrices. As a result, the positions of low-dimensional points are updated. The update of points can be imagined as springs between all pairs of points — the two points are attracted or repelled based on their distance. The iterations of the algorithm are repeated until the convergence of positions of the data in low-dimensional space or after a fixed number of iterations.

In BlosSOM, the t-SNE algorithm is not applied to high-dimensional data points but to high-dimensional landmarks. It works the same as with high-dimensional data points — it embeds high-dimensional landmarks into two-dimensional space. However, the embedding is not used to visualize clusters

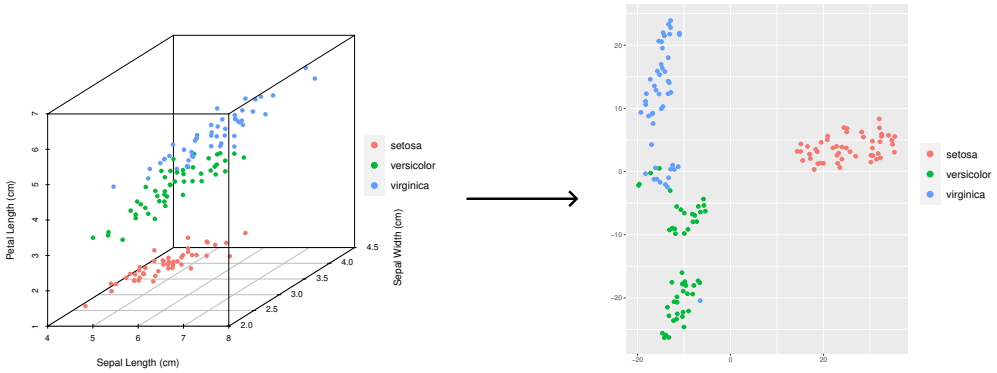

Figure 1.9 Embedding of three-dimensional dataset iris [23, 24] to two-dimensional space using the t-SNE method. It separates the data into three clusters while preserving local structure.

and analyze the data set. It is used for positioning two-dimensional landmarks because it updates the positions of low-dimensional data points in each iteration.

1.2.3 Dimensionality reduction

This section describes dimensionality reduction algorithms applied to high-dimensional data points to create a visualization in the form of a scatterplot. In BlosSOM, the data points must be embedded in each frame. Hence, an algorithm that can swiftly process many data points in a single frame is needed. Therefore, we used the algorithm EmbedSOM which provided speedup of several magnitudes of flow and mass cytometry data visualization compared to other available algorithms. A GPU-accelerated variant of EmbedSOM achieved even more speedup. The user can choose an embedding algorithm at a compile time and adjust the parameters of algorithms at runtime.

EmbedSOM

EmbedSOM [12] is a non-linear dimensionality reduction algorithm. It embeds high-dimensional points into low-dimensional space (WLOG two-dimensional). Its input is high- and two-dimensional landmarks and high-dimensional data points. The pseudocode of the algorithm is in the Algorithm 1. For each high-dimensional data point x, the k-closest high-dimensional landmarks are chosen (on Line 3). The score is computed for all k landmarks based on their distance from x (on Line 4). The closest landmark has the highest score. For all pairs of k high-dimensional landmarks, the point x is projected to the space created by a pair of landmarks (on Line 6). The goal is to find a point in two-dimensional space such that the projection of this point to a space created by corresponding

two-dimensional landmarks is the same. The linear equation, which solution is the wanted point in a two-dimensional space, is created (on Line 7). Such found point is only an approximation but sufficient approximation. The approximations are summed for all pairs of landmarks to create two linear equations with two variables (on Line 8). The solution of this system of linear equations is the position of the embedded point in two-dimensional space (on Line 10).

Algorithm 1 EmbedSOM algorithm pseudocode (the more detailed algorithm is in [12])

```
1:  function EMBEDSOM
2:      for all high-dim data points do
3:          find k-closest high-dim landmarks
4:          assign a score to each of k-closest landmarks
5:          for all k-closest landmark pairs do
6:              project the point to the line created from the pair of landmarks
7:              compute approximation
8:              result_matrix ← add approximation
9:          end for
10:         embedded_point ← solve linear equation(result_matrix)
11:     end for
12: end function
```

In BlosSOM, the EmbedSOM algorithm embeds high-dimensional data points into the two-dimensional space. Each point is embedded independently, and therefore the data points can be processed in subsets, as BlosSOM aims to be interactive (to have a feasible framerate). The user can change the position of the two-dimensional landmarks, thereby changing the embedding.

EmbedSOM algorithm can be accelerated well as the CUDA EmbedSOM algorithm [14] (created for the BlosSOM) demonstrates. CUDA EmbedSOM is a GPU-accelerated version of the EmbedSOM algorithm. The interface and the usage of the algorithm are the same as in the EmbedSOM algorithm. Only two parts of the algorithm are GPU-accelerated — the selection of k-nearest landmarks and projections of points to landmark spaces.

Chapter 2

BlosSOM computation pipeline

BlosSOM started as a research project at Charles University in 2021. The result of the project was the BlosSOM software[1] and the preliminary manuscript [14]. The software was created to demonstrate the functionality of the GPU-accelerated EmbedSOM algorithm. Even though BlosSOM was a by-product, it turned out to be useful for more than just a demonstration. Therefore, we decided to work on BlosSOM in the master thesis and continue with the development.

The original description [14] focuses mainly on the algorithm and not the software. This chapter describes BlosSOM in more detail. The following sections give a short overview of the design and describe the state of the BlosSOM before the thesis.

2.1 Overview

BlosSOM is a visualization software for the interactive dimensionality reduction of large datasets. The datasets are obtained primarily from cytometry (as mentioned in the Section 1.1), but any multidimensional dataset in the FCS [25] or TSV (a special form of the CSV format with the tabulator as a separator) format can be used. An example screenshot of the software is shown in the Figure 2.1.

The software implements a semi-supervised dimensionality reduction algorithm using the EmbedSOM [14] algorithm. The GPU-accelerated EmbedSOM algorithm can be used if the software runs on a device with the CUDA toolkit.

The user can interact directly with the projected dataset and change the results of the embedding in real time. The dataset is rendered as a two-dimensional scatterplot, and the interaction with the dataset is done through landmarks. Multiple algorithms can be applied to landmarks (see Section 1.2).

[1] `https://github.com/molnsona/blossom`

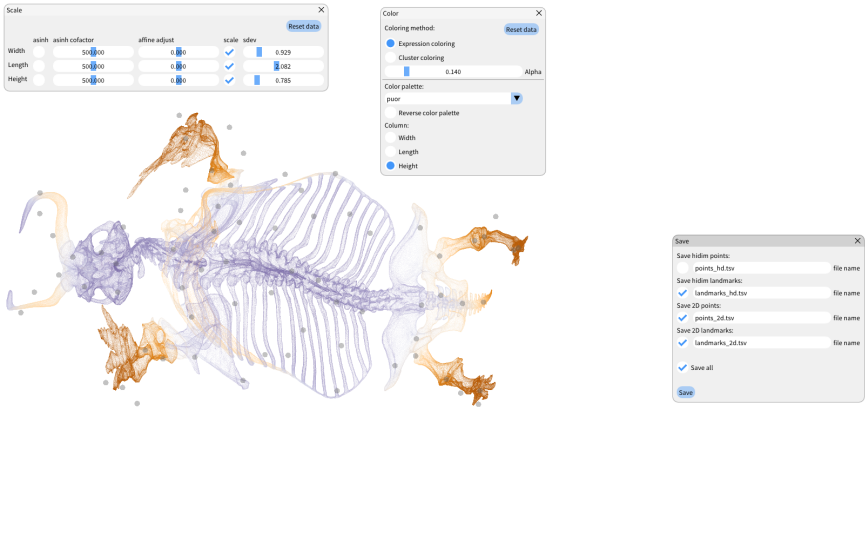

Figure 2.1 An example screenshot of the BlosSOM software. It shows an example dataset with added and moved landmarks (light gray circles) to display the mammoth skeleton properly. There are also tools for scaling, coloring, and saving the dataset (small gray windows).

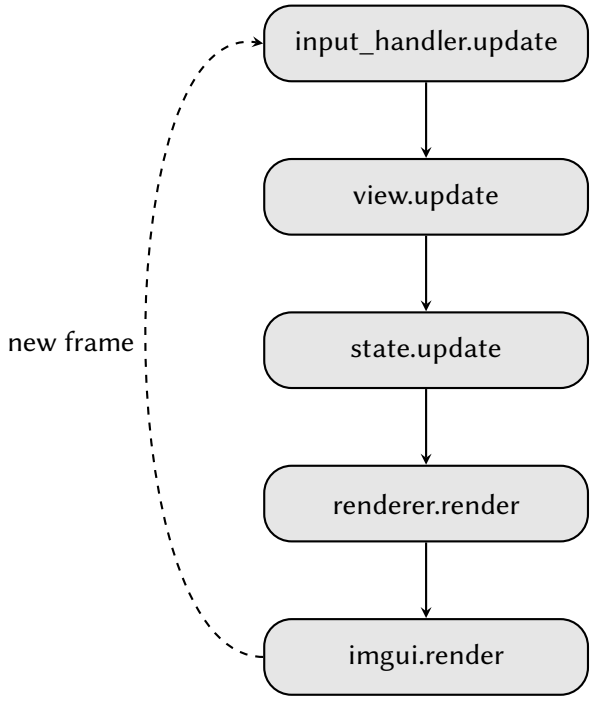

Figure 2.2 Main rendering cycle pipeline. The lines represent the order of the methods.

The user can optimize the position of landmarks by moving them, adding new ones, or deleting existing ones (see Section 2.2.3). Also, the user can move around the scatterplot and zoom in and out. Various coloring settings are provided, including different color palettes, transparency, or different types of coloring. The dataset can be transformed and scaled by the user. The resulting two-dimensional positions of the dataset points can be exported and used to plot the data.

2.2 Design

BlosSOM is an interactive graphical environment that renders data in cycles (frames). The architecture is not very complex because it was created only for demonstration purposes. There is a pipeline of computations performed each rendering cycle (see Figure 2.2).

InputHandler *`InputHandler`* stores input events and, subsequently, processes the input. It informs other parts of the pipeline of new input events so they can update their internal state and react to changes (as shown in the Figure 2.3).

View *`View`* represents a virtual camera. It manages the position and zoom level

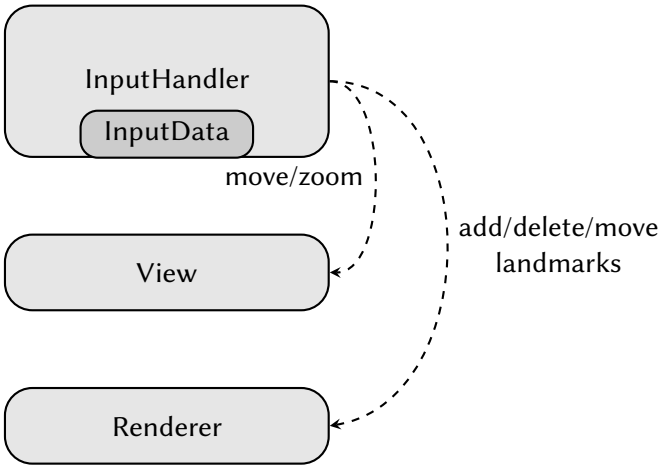

Figure 2.3 The `update` method of the *`InputHandler`*. The dashed lines represent notifying other parts of the pipeline about changes.

of the camera. In each frame, the camera moves closer to the target position and the target zoom level. It also handles the conversion of the coordinates between the screen and the model space and computes the view and the projection matrix needed in the rendering.

State *`State`* represents the current state of the simulation and stores the data of algorithms. It performs steps of the simulation in each frame. The computational pipeline is described in the Figure 2.4.

Renderer *`Renderer`* is responsible for rendering the scatter plot and the landmarks. It receives the updated positions of points and landmarks from the previous stages of the pipeline. It also handles the addition, deletion, and movement events of landmarks.

ImGuiWrapper *`ImGuiWrapper`* renders the user interface and updates the data stored in *`State`* based on the user interaction. It is a wrapper over the user interface library — Dear ImGui[2].

2.2.1 Data flow

The computational pipeline algorithms (see Figure 2.4) work either with high-dimensional data points or landmarks. The orange rectangles in the Figure 2.4 described in the Section 1.2 are the algorithms that work with landmarks. The algorithms that work with high-dimensional data points are in `update` methods of instances of `RawDataStats`, `TransData`, `ScaledData`, `ColorData` and

[2] `https://github.com/ocornut/imgui`

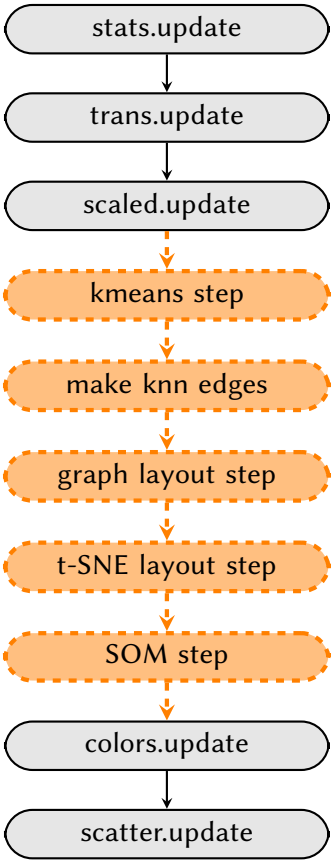

Figure 2.4 The `update` method of the *`State`* represents the computational pipeline. The algorithms in the orange boxes (described in the Section 1.2) only run if the user sets the flags. Each algorithm has its own flag, which can be set in the user interface. The lines represent the order of the algorithms. The algorithms in gray boxes work with high-dimensional data and the algorithms in orange boxes work with landmarks.

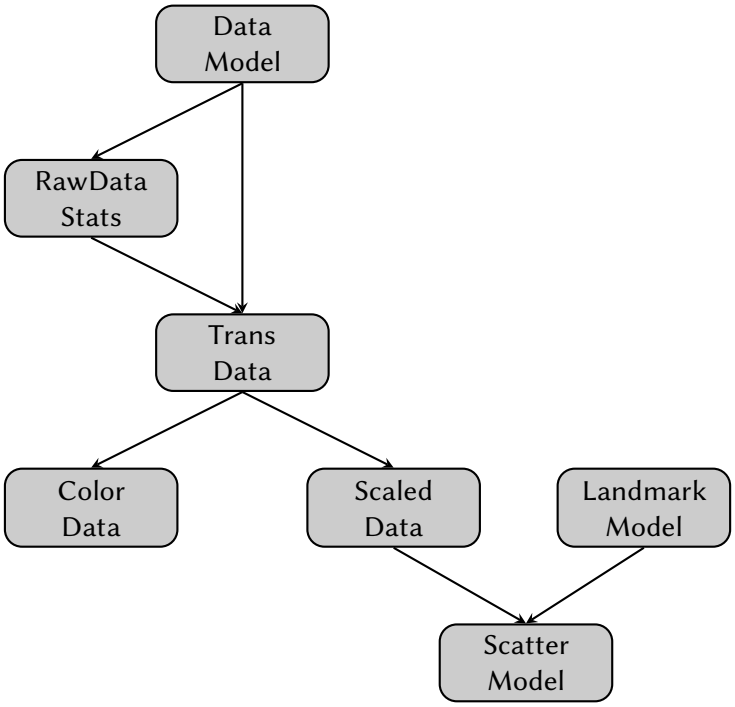

Figure 2.5 Data flow diagram of the computational pipeline (see Figure 2.4). The gray rectangles represent the data with which the algorithms work. The arrows show the direction of the data dependency. The algorithms modify their data according to the data from the direct predecessors. Their data then serves as input data for the successive algorithms.

`ScatterModel`. These algorithms are successors in the computational pipeline (as shown in the Figure 2.4), and they work with and depend on the data from the predecessor. If the predecessor changes the data, all successors have to recompute the data. The data flow is shown in the Figure 2.5.

`DataModel` stores the raw data — cells with their features (high-dimensional data) — loaded directly from the input file. Before the data are transformed, *`RawDataStats`* computes statistics on them. *`TransData`* transforms the raw input data. Subsequently, the data are scaled by *`ScaledData`*. The transformed data are colored by *`ColorData`* where each cell (data point) has assigned a color. *`LandmarkModel`* stores the positions of high- and low-dimensional landmarks. *`ScatterModel`* computes the two-dimensional positions of cells (data points) by EmbedSOM algorithm (see Section 1.2.3) from the high-dimensional scaled data in *`ScaledData`* and from the positions of landmarks in *`LandmarkModel`*.

Data format Input data from the FCS or the TSV file are stored in a vector. Among all the data classes mentioned in the Section 2.2.1, the data are stored in

approximately the same way — as a one-dimensional array storing d-dimensional input data in row-major order. Each row represents one point in the d-dimensional space.

2.2.2 Batch computations

BlosSOM has to compute large data in batches because its main goal is to be interactive. It means that only a part of the data is processed in each frame. Such an approach allows us to maintain a reasonable framerate.

The batch size has been hard coded to the program, but in this thesis, the batch size is dynamic according to the performance of the computer (as described in the Chapter 3). The algorithms that work with high-dimensional data points have to compute the data in batches — namely, the `update` methods of instances of `TransData`, `ScaledData`, `ColorData` and `ScatterModel` classes. Because the datasets can contain millions of points, it would not be feasible to perform complicated operations on each point from the dataset in just one frame. Therefore, we process only a subset of the data points in each frame, which is possible because the operations are independent.

The algorithm knows how much data it has to recompute in every frame and from which index. If it comes to the end of the data, it goes back to the beginning because the processing of the data is cyclic. When all data are recomputed, the algorithm stops. The algorithm starts recomputing again when the input data changes.

2.2.3 Actions with landmarks

The user can interact with the dataset through landmarks (the gray dots on the Figure 2.1). The user can do three actions with the landmarks — add, delete, and move (as shown in the Figure 2.6). More actions with landmarks were implemented in this thesis (see Section 4.3 and Section 4.4). All of the main logic is in the *`InputHandler`*. It notifies *`Renderer`* to do the actions based on the input events that occurred. *`Renderer`* is the middle layer between *`InputHandler`* and *`GraphRenderer`*.

Add The two-dimensional landmark is added by pressing CTRL and clicking LMB anywhere in the window; if the mouse cursor clicks on an already existing landmark, this landmark will be duplicated. The new high-dimensional landmark is created together with the two-dimensional — it finds the closest two-dimensional landmark and duplicates its counterpart in the high-dimensional space. The high-dimensional landmark positioning algorithm (see Section 1.2.1) then arranges the newly created high-dimensional landmark.

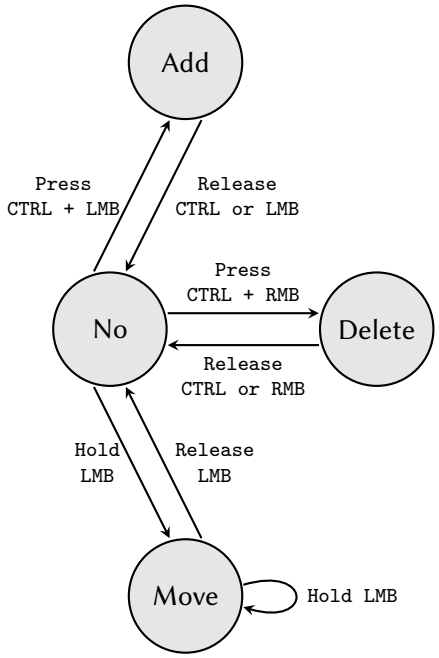

Figure 2.6 State diagram of the actions on landmarks. There are four states (gray circles) — No, Add, Delete, and Move. Arrows show the input event that the user has to do in order to do the action. No state means no action is active, and the application waits for user input to start some action. Add state means a landmark is added after the CTRL and LMB are pressed anywhere in the window. Delete state deletes landmark when the CTRL and RMB are pressed over the landmark the user wants to remove. Move state moves the landmark in the window while the landmark is pressed by LMB and moved.

Delete The two-dimensional landmark is deleted by pressing CTRL and clicking RMB on the landmark; if it does not click on the landmark, it does nothing. It also removes its high-dimensional counterpart. It also removes all existing edges going from the removed landmark.

Move The landmark can be moved around the window if the user clicks with LMB on the landmark and moves the cursor while still holding LMB. The movement is stopped if the LMB is released and other actions can be performed. This action moves only two-dimensional landmarks; therefore, this action is only for modifying the embedding results. The high-dimensional counterparts stay in the same positions.

Chapter 3

Dynamic balancing of workload in BlosSOM

This chapter explains what a dynamically balanced workload is, why it is needed, and by what methods it can be achieved. At the end of this chapter, the implementation of the best method is described and the results with benchmarks are presented.

BlosSOM processes large amount of data. Since it aims to be an interactive tool, it must maintain a reasonable framerate. That is achieved by dividing the input data into batches and processing each in one frame. This approach preserves an illusion of interactivity because the data are updated gradually in each frame, and the user sees partial results and progress. The update of the whole dataset in one frame could cause the freezing of the UI.

In the original version, the algorithms (described in the Section 2.2.2) used a constant batch size independent of the computer on which the application was running. It caused a divergence of framerates on different machines because each processes the same batch size differently. The unpredictable framerate does not meet the interactivity criterion because we want the same level of interactivity on every computer.

This chapter aims to adjust the batch size dynamically according to the performance of the computer. In this approach, the framerate is constant, and batch sizes change. The batch size adapts according to the number of points processed in previous frames and fits the current batch size to the wanted framerate. The following Section 3.1 presents possible methods and the one that was the most suitable for this problem (see Section 3.1.4).

3.1 Methods for dynamic batch size adjustment

This section describes the explored methods for dynamic batch size adjustment (the problem is stated in the Task 1). All methods have been tested for the speed of adaption to a change in a data trend. The speed is measured in the number of data frames (see Figure 3.1). The number of data frames method is shown only to illustrate the functioning and adaptiveness of methods. It would be interesting to compare these methods more rigorously, but for the purposes of this thesis, it is not necessary.

Task 1 (Batch size vs. framerate problem)**.** *Given the constant desired framerate and batch sizes from previous frames, we estimate the amount of work for the next frame so that the overall framerate is as close as possible to the given framerate.*

3.1.1 Linear regression

The first and most straightforward method explored was linear regression. This method is suitable for our data because the time required to process the data increases linearly with the larger batch size. Hence, the fitted line should estimate the trend of the data sufficiently.

Method overview

Since the data are two-dimensional — desired time of the current frame and the estimated batch size — a simple linear regression is used. The formula for the simple linear regression model is

$$y = ax + b$$

where b[1] represents the intercept of the line with the y-axis and a represents the slope of the line. Since the real world is not perfectly linear, the error term must be added to the equation

$$y = ax + b + \varepsilon.$$

The a and b parameters are unknown and, therefore, must be estimated. The ordinary least squares estimation method (further on only OLS) is used for this estimation. The main idea is to minimize all the squared distances of all the

[1]Usually, the b parameter is ignored but in our case, it is a very important part of the estimation because it represents the initialization of the algorithm. It is measured in each frame and additionally, we can turn on and off computational parts of the pipeline and affect the initialization time by that.

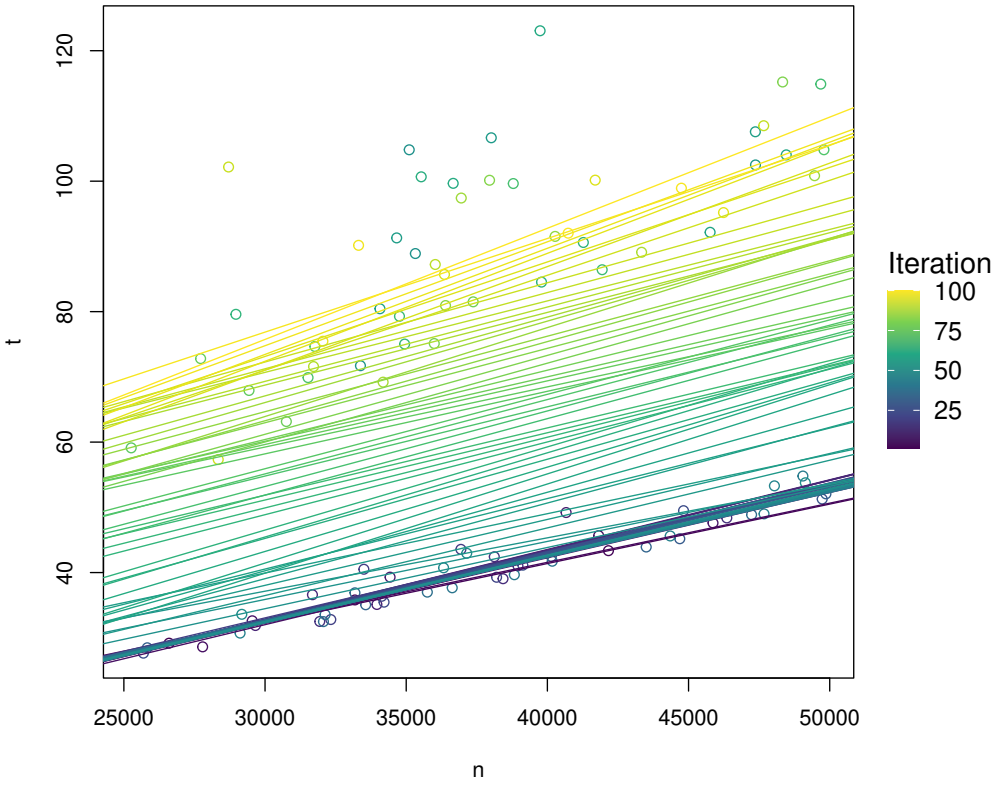

Figure 3.1 The description of the trend line graph used in all methods on the same simulated data — 100 pre-generated numbers of data points with assigned times of computations (100 circles on the graph). The x-axis is the number of data points n. The y-axis is the time t needed to process the data points. Each point (a circle on the graph) represents how long it took to process n data points. The purple points (lower in the graph) simulate a lower computer load. The greenish points (upper in the graph) simulate the change in the computer performance (e.g., some advanced algorithm is running) when the same number of data points is computed slower (it has higher t values than before). The lines represent an estimate of the data trend in each frame. The purpose of this graph is to show how many frames (lines) the algorithm needs to notice a new data trend and start to predict correct values (the lines move from the bottom to the top) — the number of the middle lines (green lines).

measured data points from the fitted line (distances between real and predicted values) called residuals:

$$\min \sum_{i=1}^{n} \varepsilon_i^2.$$

The ε is isolated from the equation of the linear regression as follows

$$\varepsilon = y - ax - b.$$

By substitution of ε we get the following problem

$$\min \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

We will denote S as the sum function

$$S = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

To find a minimum of a function, the derivative of the function must be equal to zero. Since there are two parameters a and b, the partial derivatives must be used. First, we differentiate the function with respect to a:

$$\begin{aligned} \frac{\partial S}{\partial a} &= \sum_{i=1}^{n} 2(y_i - ax_i - b)(-x_i) \\ &= 2 \sum_{i=1}^{n} (-y_i x_i + ax_i^2 + bx_i). \end{aligned}$$

The derivative of the function with respect to b is:

$$\begin{aligned} \frac{\partial S}{\partial b} &= \sum_{i=1}^{n} 2(y_i - ax_i - b)(-1) \\ &= 2 \sum_{i=1}^{n} (-y_i + ax_i + b) \\ &= 2 \sum_{i=1}^{n} (ax_i - y_i) + 2nb. \end{aligned}$$

Now, to find the minimum, the partial derivatives must equal zero. The first

equation isolates the parameter b, and substitutes it into the second equation

$$\begin{aligned} 2\sum_{i=1}^{n}(ax_i - y_i) + 2nb = 0 \\ \sum_{i=1}^{n} y_i = a\sum_{i=1}^{n} x_i + nb \\ b = \frac{\sum_{i=1}^{n} y_i}{n} - a\frac{\sum_{i=1}^{n} x_i}{n} \\ b = \overline{y} - a\overline{x}, \end{aligned}$$

where $\overline{x}$ and $\overline{y}$ are the mean values of the vectors. The second equation for the estimation of the parameter a is

$$2\sum_{i=1}^{n}(-y_i x_i + a x_i^2 + b x_i) = 0,$$

substituting b with the formula above, we get

$$2\sum_{i=1}^{n}(-y_i x_i + a x_i^2 + (\overline{y} - a\overline{x})x_i) = 0,$$

from which we eventually get

$$a = \frac{\sum_{i=1}^{n} x_i(y_i - \overline{y})}{\sum_{i=1}^{n} x_i(x_i - \overline{x})}.$$

A different formula for the estimation of a can be found, but they are equal. More precisely, the different formula is

$$a = \frac{\sum_{i=1}^{n}(y_i - \overline{y})(x_i - \overline{x})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} = \frac{\operatorname{cov}(x, y)}{\operatorname{cov}(x, x) = \operatorname{var}(x)}.$$

We will show the equality of formulas on the equality of numerators and then, in the same way, of denominators

$$\begin{aligned} \sum_{i=1}^{n}(y_i - \overline{y})(x_i - \overline{x}) &= \sum_{i=1}^{n}(y_i - \overline{y})x_i - \sum_{i=1}^{n}(y_i - \overline{y})\overline{x} \\ &= \sum_{i=1}^{n}(y_i - \overline{y})x_i - \overline{x}\sum_{i=1}^{n}(y_i - \overline{y}) \\ &= \sum_{i=1}^{n}(y_i - \overline{y})x_i, \end{aligned}$$

where the first equality is from expanding the first bracket with the second bracket. The second equality is based on the independence of the arithmetic average of x on the variable i and, therefore, can be outside the sum. The third equality stems from the Lemma 1.

Lemma 1. *The sum of deviations from the mean is equal to zero.*

$$\sum_{i=1}^{n}(x_i - \bar{x}) = 0$$

Proof.

$$\sum_{i=1}^{n}(x_i - \bar{x}) = \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \bar{x} = \sum_{i=1}^{n} x_i - n\bar{x} = \sum_{i=1}^{n} x_i - n\left(\frac{\sum_{i=1}^{n} x_i}{n}\right) = 0.$$

□

The equality of denominators uses the same principle as numerators, where the sum will be

$$\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x}).$$

By the computation of a, the substitution of a to b, and consequently, the substitution of a and b to the equation $y = ax + b$, we will get the estimation of the variable y.

Method utilization

The variables from the equation $y = ax + b$ represent:

- y time used for processing n data points
- x number of data points in one batch
- a time used for processing 1 data point
- b time spent on initialization of the algorithm

Our goal is to get the size of the next batch x. We get the value of y as the input of the algorithm, so we consider it a constant. With this assumption, the number of data points x is computed as

$$x = \frac{y - b}{a},$$

where y is constant and a and b parameters are computed from the OLS estimation.

The size of the next batch has to be computed in each frame. Therefore, the method estimates the parameters a and b in each frame.

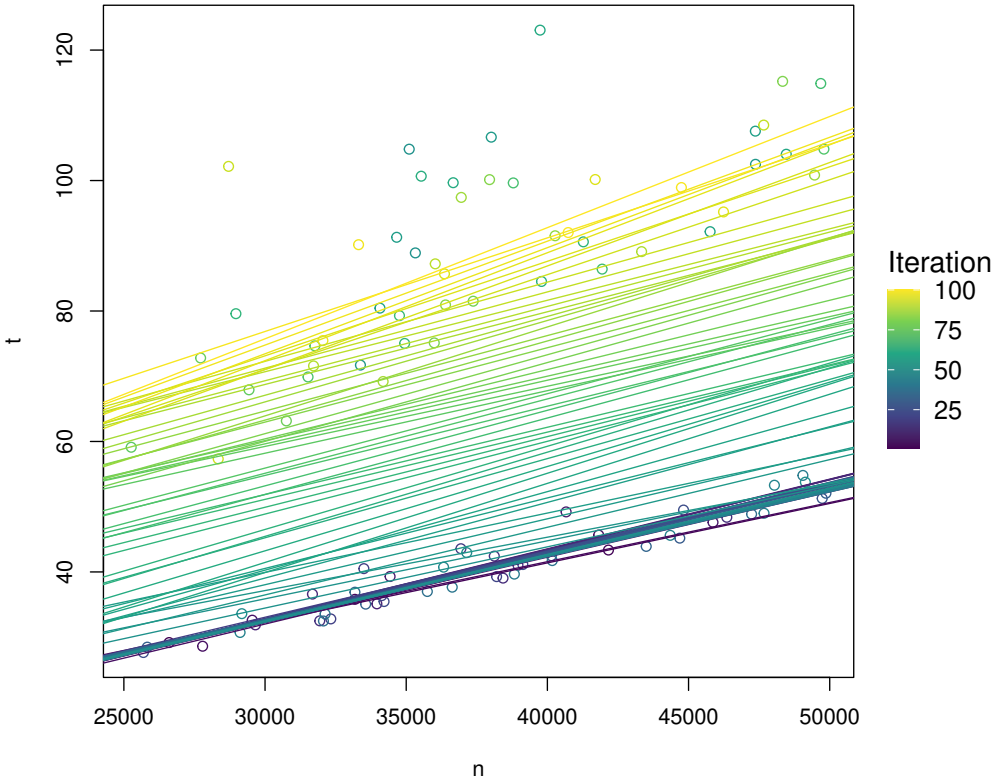

Figure 3.2 The trend line graph for the linear regression method (the graph is described in the Figure 3.1). The correct estimation of the new data trend is made after more than 20 frames (iterations of the algorithm), which shows how slowly this method adapts to the new data trend.

Method evaluation

The simple linear regression method is the best for the first experiment because it is easy to implement, and results are accurate because it considers the last n real measured data. The accuracy has a significant disadvantage — time complexity. It has to go through all previously measured data in each frame, which is not feasible for interactive applications. It also has to store n measured data, which is space-consuming. Moreover, it does not store any additional data which could help with statistical certainty (such as p-value). Also, the method is slowly reacting to the trend change in the dataset, as shown in the Figure 3.2.

3.1.2 Online linear regression

For BlosSOM, we designed a method to improve the time complexity of the linear regression method — online linear regression. It takes the main idea from the

moving average algorithm. The results are cached — the value of the oldest point is subtracted, and the value of the latest point is added. This approach reduces the work done in each frame by processing only two numbers instead of n, as in the linear regression method in Section 3.1.1. We will use the matrix form because the summation form of the OLS estimators is not suitable for the subtraction and addition of the data points. These two forms are equivalent. The results of this method will yield the same results as the simple linear regression in Section 3.1.1.

Method overview

The model is the same as in simple linear regression. Instead of computing all data points in the sum, it stores them all in the matrix.

$$Y = Xb + \varepsilon$$

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} * \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{pmatrix},$$

where n is the number of data points used to compute the regression, and p is the number of estimated parameters. The OLS method estimates parameters in the vector b as the minimization of the dot product of the error vector

$$\min(\varepsilon^T\varepsilon) = \min(\varepsilon_1\varepsilon_1 + \varepsilon_2\varepsilon_2 + \cdots + \varepsilon_n\varepsilon_n) = \min(\sum_{i=1}^{n} \varepsilon_i^2).$$

The ε is isolated from the equation as follows

$$\varepsilon = Y - Xb.$$

We will denote S as the dot product and substitute ε:

$$\begin{aligned} S = \varepsilon^T\varepsilon &= (Y - Xb)^T(Y - Xb) \\ &= (Y^T - b^TX^T)(Y - Xb) \\ &= Y^TY - Y^TXb - b^TX^TY + b^TX^TXb. \end{aligned}$$

We must find and set the derivative of S to zero to minimize it:

$$\begin{aligned} \frac{\partial S}{\partial b} = 0 - X^TY - X^TY + 2X^TXb& \\ -2X^TY + 2X^TXb &= 0 \\ X^TXb &= X^TY \\ (X^TX)^{-1}X^TXb &= (X^TX)^{-1}X^TY \\ b &= (X^TX)^{-1}X^TY. \end{aligned}$$

That is the generalized matrix form of OLS for p parameters.

We will assume $p = 2$ (we estimate two parameters) and $n = 2$ (on two data points) for demonstration purposes:

$$y_1 = b_1 + b_2 x_1$$

$$y_2 = b_1 + b_2 x_2.$$

These equations can be rewritten to the matrix form

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$Y = Xb.$$

We noticed properties of the two matrices $A = X^T X$ and $B = X^T Y$ that helped us deduce what to add and subtract. The former matrix thus becomes:

$$A = X^T X = \begin{pmatrix} 1 & 1 \\ x_1 & x_2 \end{pmatrix} \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} = \begin{pmatrix} 2 & x_1 + x_2 \\ x_1 + x_2 & x_1^2 + x_2^2 \end{pmatrix},$$

which generalizes to larger n as:

$$A = X^T X = \begin{pmatrix} n & \sum x \\ \sum x & \sum x^2 \end{pmatrix}.$$

The latter matrix then becomes:

$$B = X^T Y = \begin{pmatrix} 1 & 1 \\ x_1 & x_2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_1 + y_2 \\ x_1 y_1 + x_2 y_2 \end{pmatrix},$$

which again generalizes as:

$$B = X^T Y = \begin{pmatrix} \sum y \\ \sum xy \end{pmatrix}.$$

These forms allow us to see what to add and subtract in each frame. For the matrix A the updated matrix A' will be:

$$A' = A + \begin{pmatrix} 0 & x_{\text{new}} - x_{\text{old}} \\ x_{\text{new}} - x_{\text{old}} & x_{\text{new}}^2 - x_{\text{old}}^2 \end{pmatrix},$$

where 'old' is the index of the oldest element in the data and 'new' is the index of the newest element in the data. The matrix B' will look like this

$$B' = B + \begin{pmatrix} y_{\text{new}} - y_{\text{old}} \\ x_{\text{new}} \cdot y_{\text{new}} - x_{\text{old}} \cdot y_{\text{old}} \end{pmatrix},$$

where 'old' and 'new' are the same as in the matrix A'. There is no need to create X and Y matrices in each frame; only add and subtract a few numbers. The b parameters are then estimated from the equation

$$b = A'^{-1}B',$$

where the b_1 is the original b from the equation $y = ax + b$ and b_2 is a from this equation. We will get the estimation of y by substituting a and b to the equation.

Method utilization

We will work with $p = 2$ (parameters a and b from the equation $y = ax + b$) and $n = 50$ (the size of the data sample from which the regression is computed). In each frame, we will compute new matrices A' and B' and from them the parameters $b_1 = b$ and $b_2 = a$. All other explanation is the same as in the Section 3.1.1.

Method evaluation

The online linear regression method yields the same results as the simple linear regression (see Section 3.1.1); the graph would be the same as the Figure 3.2. The only difference is the time complexity — in each frame, it does only one inverse and one multiplication of two-by-two matrices instead of summing all n data points. It still has to store n elements in the memory but only for lookup.

3.1.3 Approximate linear regression

The motivation for this method is to save time and space at the expense of accuracy. The main idea is to approximate the parameters a and b using only one previously measured value instead of n previous values.

Method overview

The OLS estimator looks almost the same as in the online OLS (see Section 3.1.2), but the matrices A' and B' have slight differences. The addition and subtraction method is replaced by an approximation of the values with the following formula:

$$\text{new estimate} = \text{old estimate} \cdot (1 - \alpha) + \text{measured value} \cdot \alpha,$$

where α is a *learning rate*. The update of the A' matrix is thus defined as follows

$$A' = (1 - \alpha) \cdot A + \alpha \cdot \begin{pmatrix} 1 & x \\ x & x^2 \end{pmatrix},$$

where n is fixed to 1 because the approximation is done from a single data point. The update of the B' matrix is the following:

$$B' = (1 - \alpha) \cdot B + \alpha \cdot \begin{pmatrix} y \\ y \cdot x \end{pmatrix}.$$

Next, the process is the same as in the online regression (see Section 3.1.2) with $b = A'^{-1}B'$ estimation of the parameters.

Method utilization

The α parameter is set to 0.05. Otherwise, the method is used the same as in the online OLS (see Section 3.1.2), except for n, which is set to 1 here.

Method evaluation

The approximate linear regression method has similar time complexity as the online OLS due to cached partial results. Furthermore, this method has better space complexity — it stores only one newest value instead of n. The graph shows that detecting a new trend is slightly faster — it takes fewer frames to adapt, as shown in the Figure 3.3. This method is less accurate than previous methods because it approximates the result from the most recent data point, not from n data points.

3.1.4 Mean line estimate method MLEM

The motivation behind this method is to not get biased by a permanent noise. The Kalman filter [26] cannot be used because the problem is not linear but hyperbolic. Therefore we designed a method for BlosSOM, which uses a different approach than the previous methods. The main idea comes from the arithmetic average, but it expands it to the two-dimensional space.

Method overview

We still try to find the two parameters a and b from the formula $y = ax + b$ as in the linear regression but not using the OLS estimator. In every frame, the algorithm receives two values — N and T. N is the number of processed data points in the last frame, and T is the computation time of processing N data points. T contains the constant initialization time of the algorithm, and the remaining time is the actual time spent on the computation. The parameter a represents the time spent on the computation of one point, and the parameter b represents the initialization time of the algorithm. If the initialization time is T, then the computation time

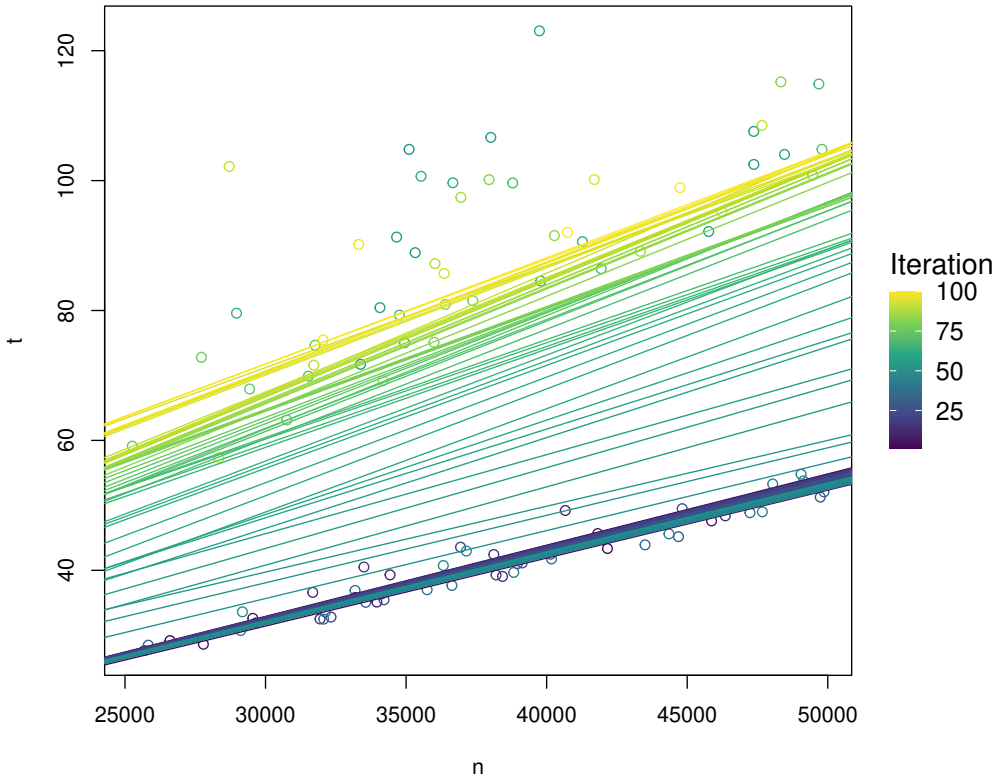

Figure 3.3 The trend line graph for the approximate linear regression method (the graph is described in the Figure 3.1). The correct estimation of the new data trend is made after less than 20 frames (iterations of the algorithm), which shows that this method adapts faster to the new data trend than linear regression and online linear regression shown in the Figure 3.2.

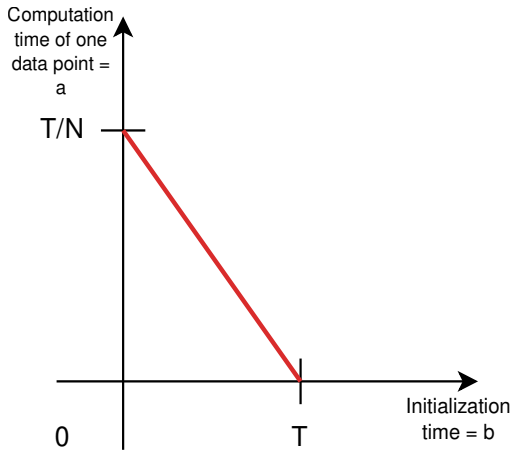

Figure 3.4 The red line represents all possible values of parameters a and b in the formula $y = ax + b$, where a is y-axis and b is x-axis. We need these parameters to estimate the size of the next batch. The two points on the axis represent two extremes — if the initialization time of the algorithm is T, then the computation time of one point is 0, and if the initialization time is 0, then the computation time of one data point is $\frac{T}{N}$.

of one point is 0. Moreover, if the initialization time is 0, then the computation time of one data point is $\frac{T}{N}$. The parameters a and b lie linearly between these two extremes (the line in the Figure 3.4).

The previous paragraph describes only one measurement of values N and T (in one frame). The new values N and T are measured in each frame, and the line in the Figure 3.4 looks different in each frame. These lines do not intersect at one point; it would be the perfect estimate of the parameters a and b if they had exactly one intersection point. Hence, we must find a point closest to all lines, i.e., minimize the squared distances of the point to the lines.

Now we derive the formula for the distance of a point from a line. We work with the line in the Figure 3.4. First, we find the slope of the line, which is $\left(-T, \frac{T}{N}\right)$. Then we find the normal line of the line, which is $\left(\frac{T}{N}, T\right)$ — the vectors (x, y) and $(-y, x)$ are perpendicular because their dot product is 0. Next, we have to normalize the normal vector:

$$(n_1, n_2) = \frac{\left(\frac{T}{N}, T\right)}{\sqrt{\left(\frac{T}{N}\right)^2 + T^2}}.$$

The distance of a point from a line is the dot product of its normal and the point as shown in the Figure 3.5.

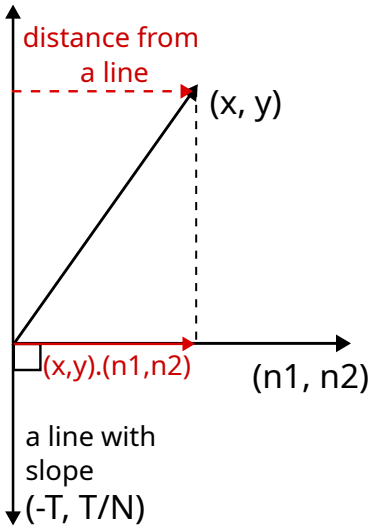

Figure 3.5 Computation of the distance of the point (x, y) from the line with a slope $\left(-T, \frac{T}{N}\right)$. (n_1, n_2) is a normal line. The dot product of the point vector and normal vector is highlighted as a full red line on the normal line. This dot product is the same as the distance of the point (x, y) with the line $\left(-T, \frac{T}{N}\right)$ (shown as the dashed red line on the top).

Next, we have to compute the distance of the line $\left(-T, \frac{T}{N}\right)$ from the point $(0, 0)$ because the distance computation is done at the origin. This distance can be computed either from the point $(T, 0)$ or $\left(0, \frac{T}{N}\right)$, which are the original intersections with axes. So the distance is computed as dot product of $(T, 0)$ and the normal line (n_1, n_2) (the distance is visualized as the blue line in the Figure 3.6):

$$n_3 = (T, 0) \cdot (n_1, n_2) = Tn_1.$$

Now we have to subtract the distance of the line from the $(0, 0)$ and the distance of the point (x, y) from the shifted line to get the actual distance (the actual distance is drawn as the green line in the Figure 3.6). The whole formula for the distance of the point (x, y) from the line with a slope $\left(-T, \frac{T}{N}\right)$ with normal line (n_1, n_2) is:

$$\begin{aligned} \text{dist} &= (x, y) \cdot (n_1, n_2) - (T, 0) \cdot (n_1, n_2) \\ &= xn_1 + yn_2 - n_3. \end{aligned}$$

From now on, the parameters a and b that we discussed at the beginning of this section will be addressed as x and y, and the a and b will be the parameters of the error function described below.

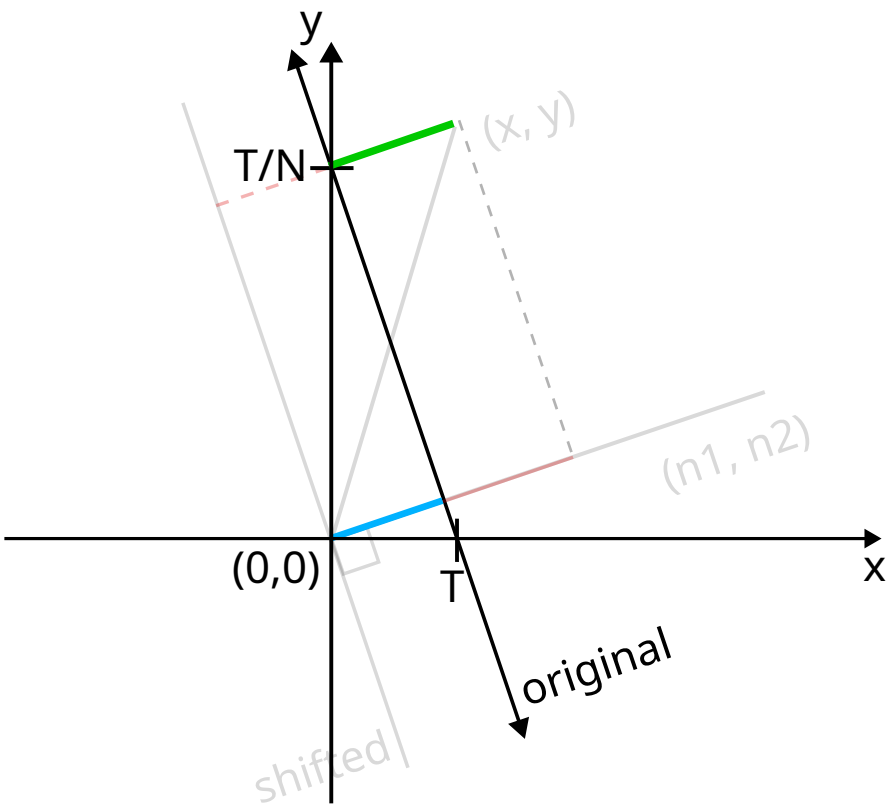

Figure 3.6 The computation of the actual distance of a point from a line. The pale graph in the background shows the Figure 3.5 at the origin. Since the line is shifted, we have to calculate the actual distance of the line from the origin (blue line). Consequently, we must subtract the blue line from the red line to get the green line — the actual distance of the point (x, y) from the original line.

To find the point closest to all the lines (to estimate the parameters x and y), we have to minimize the squared distances from the lines:

$$\min(xn_1 + yn_2 - n_3)^2,$$

in other words, minimize the error function:

$$\min e(x, y).$$

We will expand the error function:

$$\begin{aligned} e(x, y) &= (xn_1 + yn_2 - n_3)^2 \\ &= n_1^2x^2 + n_2^2y^2 + 2n_1n_2xy - 2n_1n_3x - 2n_2n_3y + n_3^2 \\ &= ax^2 + by^2 + cxy + dx + ey + f, \end{aligned}$$

where the parameters a, b, c, d, e and f expand as follows

$$\begin{aligned} a &= n_1^2 \\ b &= n_2^2 \\ c &= 2n_1n_2 \\ d &= -2n_1n_3 \\ e &= -2n_2n_3 \\ f &= n_3^2. \end{aligned}$$

The equation above is a generic formula for a parabola over a two-dimensional plane denoted as

$$z = ax^2 + by^2 + cxy + dx + ey + f.$$

The geometric explanation of the minimization of the error function is to find the minimum of summed parabolas, which is only one and well-defined. Since we approximate the parameters a-f (as in Section 3.1.3), the resulting parabola is the approximated sum of parabolas. To find the minimum of the approximated parabola, we will differentiate the error function $e(x, y)$ and set the derivatives equal to zero. First, we differentiate the function with respect to x:

$$\frac{\partial e(x, y)}{\partial x} = 2ax + cy + d$$

and then with respect to y:

$$\frac{\partial e(x, y)}{\partial y} = 2by + cx + e.$$

Now, we set both equations to zero, and we get the system of linear equations:

$$\begin{aligned} 2ax + \quad cy &= -d \\ cx + 2by &= -e, \end{aligned}$$

which is solved by Cramer's rule as follows

$$\begin{aligned} x &= \frac{ce - 2bd}{4ab - c^2} \\ y &= \frac{cd - 2ae}{4ab - c^2}. \end{aligned}$$

Before the computation of x and y, the parameters a-f must be approximated:

$$\begin{aligned} a &= \mathrm{a}(1-\alpha) + n_1^2\alpha \\ b &= \mathrm{b}(1-\alpha) + n_2^2\alpha \\ c &= \mathrm{c}(1-\alpha) + 2n_1n_2\alpha \\ d &= \mathrm{d}(1-\alpha) - 2n_1n_3\alpha \\ e &= \mathrm{e}(1-\alpha) - 2n_2n_3\alpha \\ f &= \mathrm{f}(1-\alpha) + n_3^2\alpha, \end{aligned}$$

where the parameters a-f are the values from the previous frame and α is the learning rate.

Now, we can calculate parameters x and y with estimated values a-f.

Method utilization

As mentioned in the Section 3.1.1, we use this method to estimate the same parameters a and b simply by finding the minimum-error parameter combination (i.e., the center of the paraboloid, as displayed in the Figure 3.8). However, in this method, the parameters are called x and y, where x is the parameter b and y is the parameter a. The batch size can be estimated in the same way as in the previous methods. We explored a method with probabilistic guarantees but we found out that the complexity of this method is beyond the scope of the thesis. The idea of the method is described below until the critical point where we could not invert a function.

Instead of x, we will denote the batch size N. We want to find N such that, in 95% of cases, the processing time of data points will not exceed the reserved time. We have the set of the Ns that does not exceed the given time and Ns that will cause the algorithm to compute over the given time (as described in the Figure 3.7). According to the MLEM method, we have a paraboloid in each frame that is more

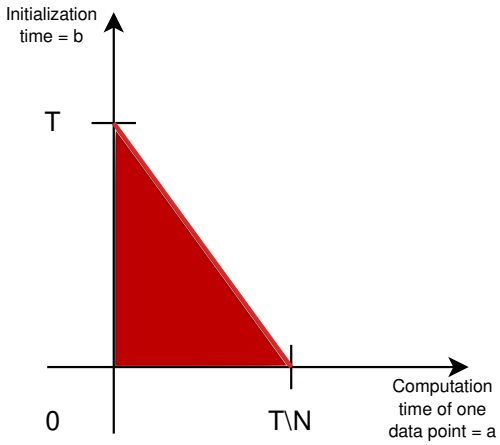

Figure 3.7 The graph is almost the same as Figure 3.4, but the axes are switched WLOG. The red area represents values of N that fit into the reserved time.

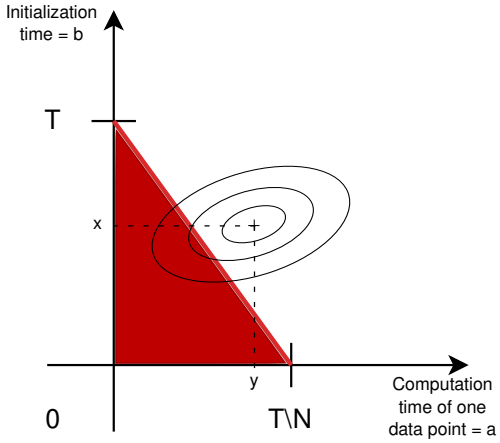

Figure 3.8 The center of the paraboloid is at the (y, x) point. Ellipses represent the depth of the paraboloid.

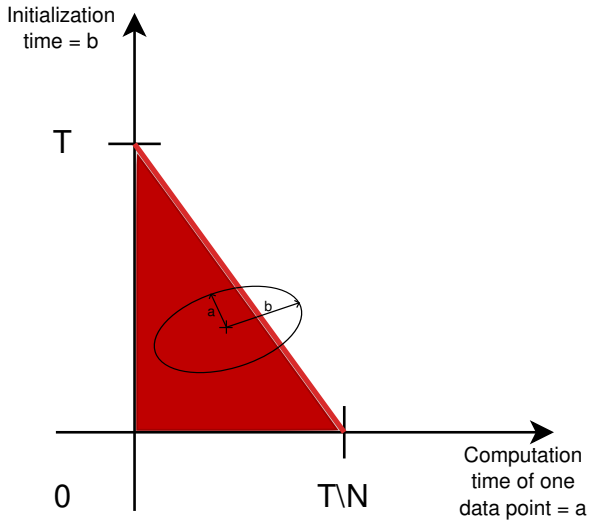

Figure 3.9 The ellipse with axes $\vec{a}$ and $\vec{b}$ has to be divided by the red line to the 95% and 5% of the area. So we find the proper slope of the red line to divide the ellipse correctly. The correct slope of the red line is approximately shown in the figure.

accurate than in the previous frame. This paraboloid has the center at the (y, x) point computed from estimated values a-f (as described in the Figure 3.8).

We can look at the paraboloid as the bivariate normal distribution centered at $(y, x) = (\mu_1 \mu_2)^T = \vec{\mu}$. We want to find the axes of the distribution (the standard deviations in two directions) called $\vec{a}$ and $\vec{b}$. We create a covariance matrix q from the matrix of the quadratic form of the paraboloid with the equation $z = ax^2 + by^2 + cxy + dx + ey + f$, and take the first two rows and columns because it is a two-dimensional distribution:

$$q = \begin{pmatrix} a & \frac{c}{2} \\ \frac{c}{2} & b \end{pmatrix} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

We will perform the SVD algorithm to get the precise values of the standard deviations (zeros off the diagonal). The SVD algorithm decomposes q to three matrices U, Σ, V. The Σ matrix is a diagonal matrix with sizes of the axes on the diagonal. The directions of the axes are the rows of the matrix V.

Now we want to fit the line with a slope $\left(-\frac{T}{N}, T\right)$ to divide the ellipse to 95% and 5% of the area (see Figure 3.9). The line fitting is easier in the one-dimensional normal distribution, where the problem changes to the point fitting.

To get the one-dimensional normal distribution from the bivariate normal distribution, we will project it to the normalized normal line n with a slope $\left(T, \frac{T}{N}\right)$. It is possible because the projection of a normal distribution is also normal. The

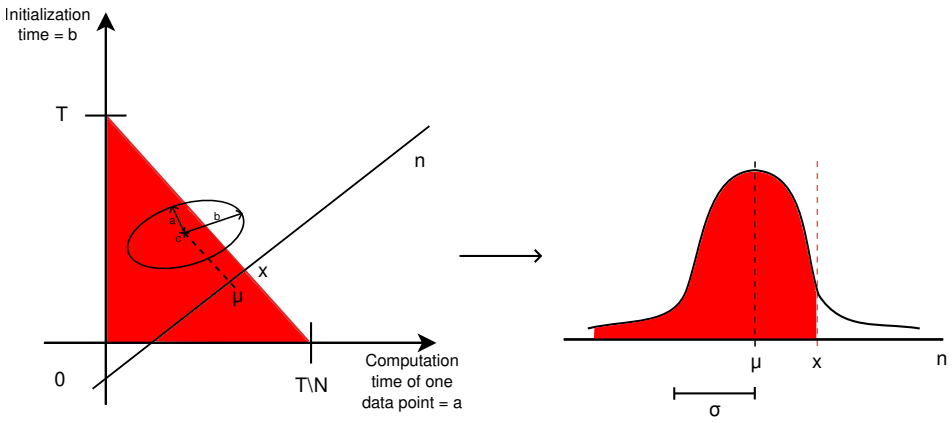

Figure 3.10 The projection of the bivariate normal distribution (the ellipse on the left) to normal n. This projection squashes the two-dimensional normal distribution to the one-dimensional normal distribution (the bell curve on the right). The mean μ of the normal distribution is the projection of the center of the ellipse c. The standard deviation σ is the square root of the sum of projections of the axes of the ellipse a and b to normal n. The point x is a projection of any point from the red line on the left to the normal n (we chose the point $(0, T)$).

mean μ of the distribution is the center of the ellipse c projected to normal n. The standard deviation σ is the square root of the sum of projections of the axes to normal n. The searched point x is a point where the normal is intersected by the line with a slope $\left(-\frac{T}{N}, T\right)$, so the point is a projection of any of the points of the line to the normal (we chose the point $(0, T)$). The mathematical notation of the variables above is as follows:

$$
\begin{aligned}
n &= \frac{(T, T/N)}{\sqrt{T^2 + \frac{T^2}{N^2}}} \\
\mu &= (c \cdot n) \\
\sigma &= \sqrt{\left(n \cdot \vec{a}\right)^2 + \left(n \cdot \vec{b}\right)^2} \\
x &= \left((0, T) \cdot n\right).
\end{aligned}
$$

The graphical representation is in the Figure 3.10.

We want the statistic x to be in the 95% of the normal distribution (its p-value equals 0.05). Therefore, we must convert the normal distribution to the standard normal distribution called z-score:

$$z = \frac{x - \mu}{\sigma}$$

Typically, we know the z-score and find the probability. Now, we do not know the z-score, but we know the probability, so we must invert the function Φ to get the z-score value. The value of the inverse function Φ is from the Z table. So the equation is:

$$\begin{aligned} \Phi(z) &= 95\% \\ z &= \Phi^{-1}(95\%) \\ z &= 1.64 \\ \frac{x-\mu}{\sigma} &= 1.64. \end{aligned}$$

After the substitution, we will get the formula below:

$$\frac{\frac{T}{\sqrt{N^2+1}} - \frac{c_1}{N\sqrt{N^2+1}} - \frac{c_2}{\sqrt{N^2+1}}}{\sqrt{\frac{(a_1^2+b_1^2)N^2 + 2(a_1a_2+b_1b_2)N + (a_2^2+b_2^2)}{N^2+1}}} = 1.64$$

We failed to find a closed form that would extract the desired N for the given parameter probability distribution and desired target time. Therefore, we use MLEM only for the estimation of a and b parameters from the $y = ax + b$ formula (described in the Section 3.1.1) and not for the estimation of the batch size. The batch size N is estimated the same as in the previous methods from the formula

$$x = \frac{y-b}{a}.$$

Method evaluation

The MLEM method has similar time and space complexity as the approximate OLS (see Section 3.1.3) because it does not compute sums in each frame but only approximates the values. It does not store n data points, only parameters from the last frame. But the MLEM method has more precise estimations and is more accurate than the approximate OLS as can be seen in the smooth change of trend lines in the figure. The graph (see Figure 3.11) shows that with the same α (learning rate), the method adapts as fast to the change of the new trend as the approximate OLS (see Figure 3.3).

This method can be modified to sum all previous values as in the Section 3.1.1 or to use the additive and subtract method as in the Section 3.1.2. However, they would have worse time and space complexity, so we directly used the approximative method. This method is also implemented in the BlosSOM tool to estimate the next batch size because it gives the best results and is the fastest method. The implementation of the method is described in the Section 3.2.

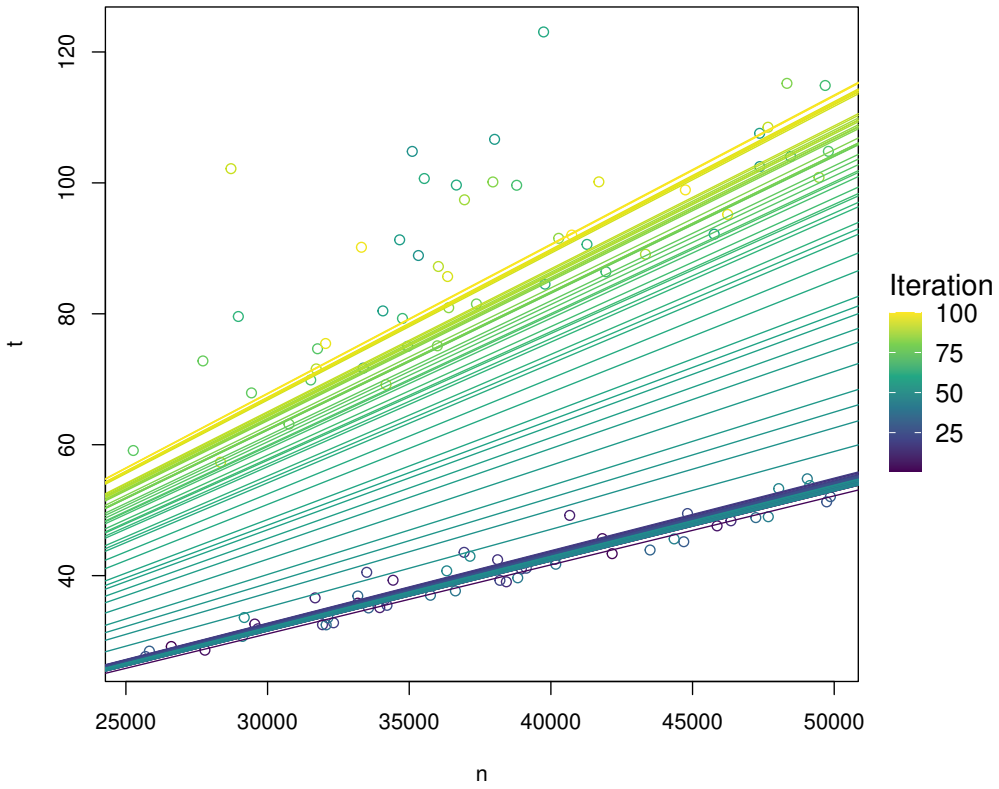

Figure 3.11 The trend line graph for the MLEM method (the graph is described in the Figure 3.1). The correct estimation of the new data trend is made after less than 20 frames (iterations of the algorithm), which is roughly the same as for the approximate OLS (as in the Figure 3.3). In addition, the trend lines change smoothly, not inconsistently, as in the previous methods — the lines are approximately parallel and do not cross each other.

3.2 Implementation

The method we implemented in BlosSOM is MLEM (see Section 3.1.4). It is a straightforward implementation of the mathematics described in the Section 3.1.4 but with the basic estimation of the batch size without the probability guarantees as it turned out to be more complex than we expected. The implementation is in the *Estimator* class. The batch size estimation is used in every stage of the pipeline that is computationally more complex and cannot process all data points in one frame. Namely in the *TransData*, *ScaledData*, *ColorData*, and *ScatterModel* instances where each instance has its own estimator *BatchSizeGen* and therefore has a different batch size. The template of the usage of *BatchSizeGen* is in the Algorithm 2, where T represents the real measured duration of the computation in the last frame, and t is the required computation time in the current frame. The current frame batch size is estimated

Algorithm 2 Batch size estimation

```
function NON-CONST COMPUTATION::UPDATE
    ...
    max_points ← batch_size_gen.next(T, t)
    ...
end function
```

based on these two parameters and the previous batch size.

The time t — how long the computation should last in the current frame — is computed from the duration of the pipeline stages (described in Figure 2.4). Some stages do not need to estimate the batch size, so their computation time is constant. Some stages are non-constant, and their computation time changes according to the batch size. The time reserved for non-constant computations is computed as follows:

$$\text{non-constant time} = dt - \text{constant time},$$

where dt is the duration of the whole frame. The non-constant time has to be divided between all non-constant computations in the pipeline. The non-constant time is divided according to the priorities — the higher the priority, the faster the computation is completed. These priorities and the duration of each non-constant computation are implemented in the `FrameStats::update_times` method. The time needed for computations in the current frame is computed as follows:

$$\text{new duration} = \text{non-const time} \cdot \text{priority},$$

where all priorities are summed to one, so the whole non-constant time is divided between computations.

The MLEM algorithm is reset only when a new dataset is loaded (see Algorithm 3).

Algorithm 3 MLEM reset

```
function NON-CONST COMPUTATION::UPDATE
    ...
    if new dataset is loaded then
        t ← 0.00001f
        batch_size_gen.reset()
    end if
    ...
end function
```

3.3 Pratical results

With the fixed batch size, the framerate was different on each computer, making the software not portable, and we could not guarantee interactivity on every hardware. Also, the framerate differed even on one computer (see Figure 3.12). The dynamically balanced workload utilizes the available hardware to get the most performance while preserving the framerate. Therefore the framerate is the same on each computer, and the software is interactive on every computer. Even with the basic batch size estimation, the batch sizes are computed reasonably well, as shown in the Figure 3.13.

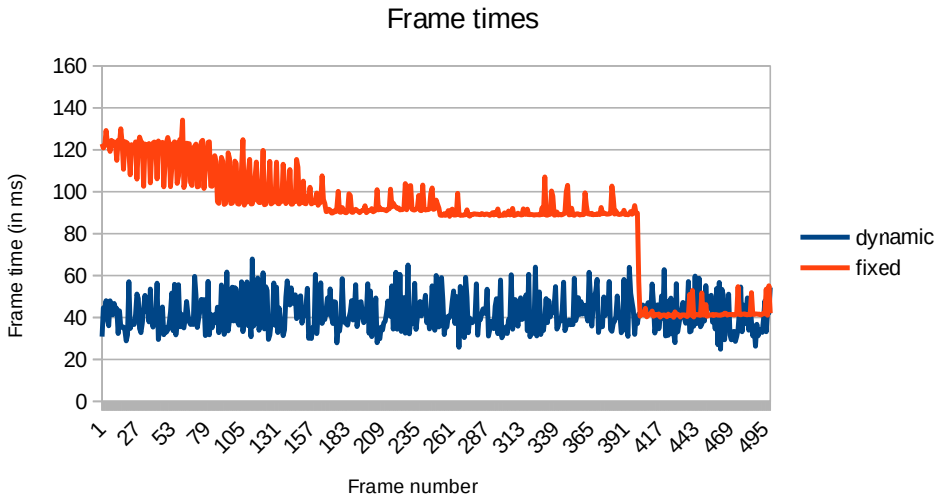

Figure 3.12 Frame times of both fixed and dynamic batch size approaches. For the fixed-batch approach, frame time stabilizes at the desired region only after all computation stages finish, and only thanks to manual tuning of the fixed batch size. The frame time in the dynamic-batch approach is stabilized at the desired region from the beginning, even with all computations running. Variance in the dynamic batch approach is deliberately caused by the algorithm, which scans a wider range of batch sizes to provide better data for the regression.

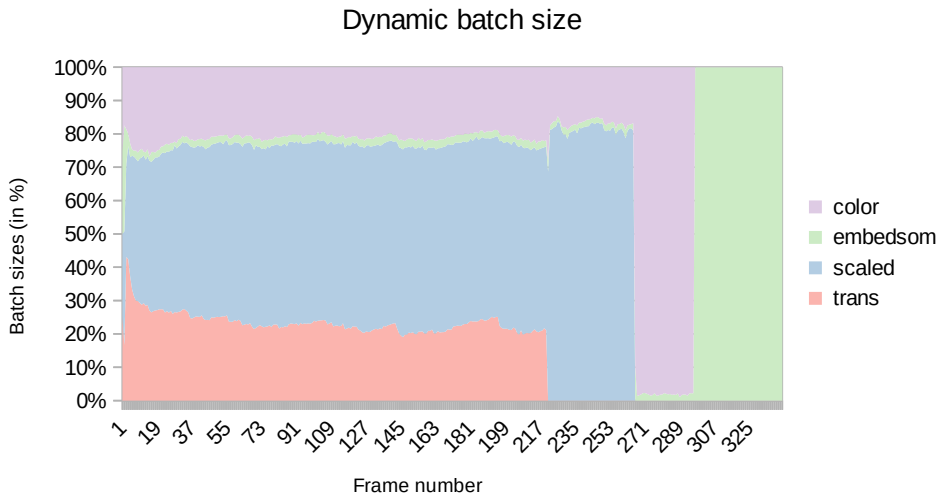

Figure 3.13 The graph depicts how a given non-constant time is divided between all four parts of the pipeline. The batch size is dynamic and it changes a little each frame to fit into the given time. The *trans* and *scaled* update methods have the highest priorities and therefore are computed the fastest (they occupy the largest area at the beginning). After that, *color* has to be recomputed and then the *embedsom* is recomputed with the full possible available batch size. Estimates of batch sizes adjust initially and then stabilize.

Chapter 4

Technical improvements in BlosSOM

In addition to the new main feature (described in the Chapter 3), several other improvements have been implemented, described in the following sections.

4.1 Rendering pipeline based on GLFW

Originally, BlosSOM was implemented in the Magnum engine[1]. However, we found out that it contains too much unnecessary functionality, slowing the compilation time. Therefore, we decided to use only the needed functionality and rewrote the application using the GLFW[2] library.

GLFW is used for handling input and output events and creating windows. The abstraction over OpenGL calls provided by the Magnum engine is no longer available. Hence, the rendering is done directly with OpenGL calls and shaders. The user interface stays the same because Dear ImGui is compatible with GLFW and OpenGL.

Thanks to the GLFW, we now have full control over the rendering cycle, and therefore, the `main` function is used to start the application. Before, there was no `main` function, only methods inherited from Magnum Engine classes. It gives us more freedom in the layout of the `main` function and the possibility to change the order of events.

Because of the change to GLFW, the input and output have to be handled differently; in the Magnum Engine, we just overrode methods. With GLFW, we create GLFW callbacks, register them, collect the values if any callback was triggered, and react to changes in the `InputHandler::update` method. This

[1] `https://magnum.graphics/`
[2] `https://www.glfw.org/`

method is called in each frame at the beginning, and the collected values from callbacks are reset in each frame at the end. The new events are polled after the reset, so in the next frame, the values are up-to-date.

After switching to GLFW, the dependency on the SDL library[3] is no longer needed because all the functionality we need is contained in the GLFW library. Magnum Engine offered the mathematical abstraction for matrices and vectors and operations with them; after removing it from the project, we had to find a replacement — the glm library[4] (a C++ header-only mathematics library). Dear ImGui[5] is the only submodule needed. Other dependencies are binaries installed directly into the operating system.

The rendering is done directly through shaders. That is because the rendering abstraction provided by Magnum Engine is no longer available. The shaders are written in the GLSL language, and objects are rendered in the new OpenGL way — via vertex and array buffer objects. Shaders are written in the *shaders.h* file, and the model, view, and projection matrices are obtained from the *View* instance. We do not use advanced shaders, only basic vertex and fragment shaders that set the position and color of objects.

Although there was some overhead in rewriting the project to GLFW, it made the application more lightweight and reduced compilation time by removing unnecessary features. Due to this change, we decided not to support the Windows operating system because it is no longer a priority since the target audience is mostly Linux users. With more time, it would be possible to restore the support because GLFW is compatible with Windows.

4.2 The speed up of a scatterplot rendering

Rendering the data points is the most complicated part of BlosSOM to solve because a full render cannot be split into two separate frames as it would cause parts of the dataset to flicker. Also, the limitation of the rendering of large datasets is GPU. In each frame, the GPU has to process all data points and render them to the screen. In larger datasets, it prevented us from fitting into the wanted frame time because the rendering took longer than the wanted frame time. We solved this problem by rendering the data to separate textures. We have n data points and t textures. The data points are evenly distributed between the textures, where each texture stores positions of $\frac{n}{t}$ distinct data points. In each frame, only one texture updates its content; all other textures have the old content. The update of only one texture in each frame speeds up the rendering t times because only $\frac{n}{t}$ data

[3] https://www.libsdl.org/

[4] https://github.com/g-truc/glm

[5] https://github.com/ocornut/imgui

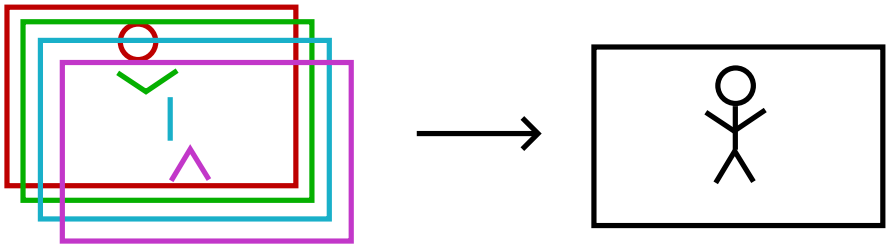

Figure 4.1 Rendering of a scene to multiple textures. Each texture (colored rectangles on the left) stores a distinct part of the resulting data (colored body parts of the stickman on the left). All textures rendered together over themselves give the whole dataset (the picture on the right). In our case, the body parts of a stickman represent different parts of the data points in the scatterplot.

is rasterized instead of all n data points. Textures alternate using a round-robin scheduling. In each frame, all textures are rendered over each other, so it looks like the whole dataset is rendered simultaneously (see Figure 4.1).

This feature is implemented in a newly created class *`TextureRenderer`*. *`ScatterRenderer`* draws points the same as before but only the subset of points — each subset of points to a different texture. The textures are then projected to quads and rendered over each other.

4.3 Multiselection of landmarks

Multiselect is a functionality that allows the user to select more landmarks at once and move them around together. As a main feature for the users, it makes the reorganization of detailed embeddings that depend on many landmarks much more convenient.

The landmarks are selected by drawing a rectangle (see Figure 4.2) by pressing SHIFT and moving the mouse with the pressed left button. The user can freely move the rectangle with landmarks by pressing the left mouse button. To stop multiselect, the user has to press the right mouse button.

When multiselect is active, no other actions with landmarks can be performed. The multiselect mode has three states (as shown in the Figure 4.3):

No multiselect In this state, multiselect is inactive, and other actions can be performed, such as adding or deleting landmarks (as shown in the Figure 2.6).

Active multiselect To get to the Active state, the user has to press and hold SHIFT and LMB (left mouse button). It will activate the selection creation (gray rectangle in the Figure 4.2). While the user holds SHIFT and LMB and moves the mouse, it specifies the selection area. In this state, the user

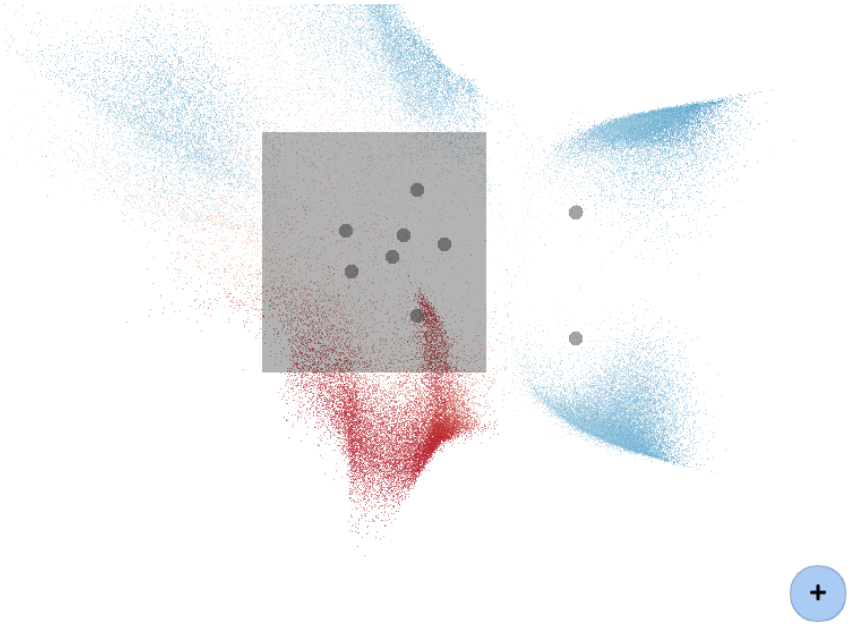

Figure 4.2 The multiselection of landmarks in BlosSOM. Landmarks (gray circles) inside the gray rectangle will move together when clicked and dragged.

can only create the rectangle, nothing else. The SHIFT or LMB must be released to leave this state and switch to the Passive multiselect state.

Passive multiselect When the user creates the selection (releases SHIFT or LMB), it switches to the Passive state. In this state, the user can only move the selection (click LMB on the rectangle and move the mouse with pressed LMB). As in the move action of landmarks, the selection only moves with two-dimensional landmarks, and high-dimensional counterparts stay the same. All other actions are not possible in this state. To end this state, the user has to press RMB, and it returns to the No multiselect state where all other actions can be performed.

The logic of the multiselect is implemented in the `process_mouse_button` method of *`InputHandler`*, which triggers *`Renderer`* when input events occur. *`Renderer`* serves as a middle layer between *`InputHandler`* and *`UiRenderer`*. *`UiRenderer`* creates the rectangle, checks which landmarks are within the rectangle, moves landmarks together with the rectangle, and renders the rectangle. It also stores all flags, which determine the state of multiselect.

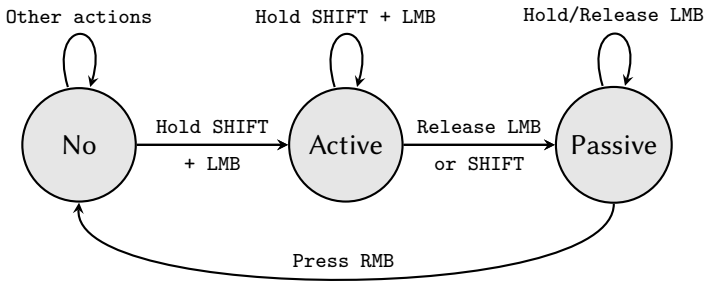

Figure 4.3 State diagram of multiselect. It consists of three states (gray circles) — No multiselect, Active multiselect, and Passive multiselect. The lines show transitions between states using mouse and keyboard buttons. No multiselect state means the multiselect functionality is inactive, and other actions can be performed, such as adding or deleting landmarks (as shown in the Figure 2.6). The Active multiselect state represents the creation of the gray rectangle (as shown in the Figure 4.2). Passive multiselect allows only the movement of the selected landmarks (the movement of the gray rectangle).

4.4 Brush coloring

Brushing is a coloring method that allows users to color data points indirectly through landmarks. The user can color the landmarks by "spraying" the color. The color of each data point is the same as the closest high-dimensional landmark. The color represents the cluster to which the landmark and corresponding data points belong. The cluster is also represented by a name. The user can set both cluster attributes — color and name — in the user interface. An example of the brushing coloring is in the Figure 4.4.

The "spraying" of the color is done by choosing the cluster color and moving the mouse with pressed LMB above the landmarks. The mouse cursor has a circle radius within which the landmarks will be colored (as shown in the Figure 4.5). The user can change the size of the radius. The clusters can be added or deleted as the user wants. When the cluster is deleted from the tool, all landmarks and closest data points that belonged to that cluster are colored back to the default color without any cluster assigned. When the brushing is active, no other action can be performed with landmarks.

Data points are colored in each frame by batches (as mentioned in the Section 2.2.2). In each frame, the closest high-dimensional landmark is found for each data point in the batch, and its color is assigned to the data point. The logic of the "spraying" is in the `process_mouse_button` method of *`InputHandler`*, which informs *`Renderer`* about the change in the input and *`Renderer`* notifies *`UiRenderer`*. *`UiRenderer`* renders the radius circle around the mouse cursor and checks which landmarks are within the circle. *`InputHandler`* also notifies

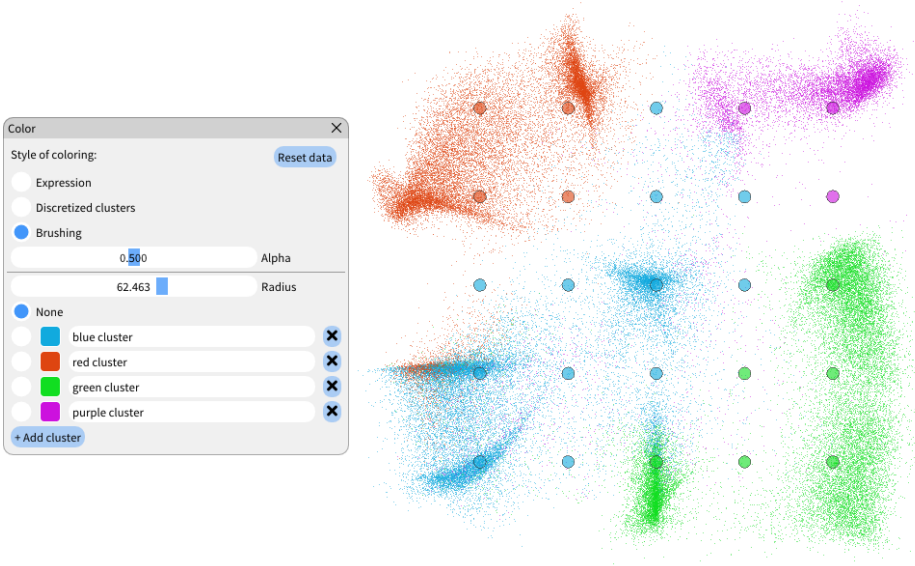

Figure 4.4 Landmarks assigned to clusters by brushing coloring technique. Each cluster has its name and color, which are editable by the user (in the Color tool on the left). Each data point has the color of the closest high-dimensional landmark.

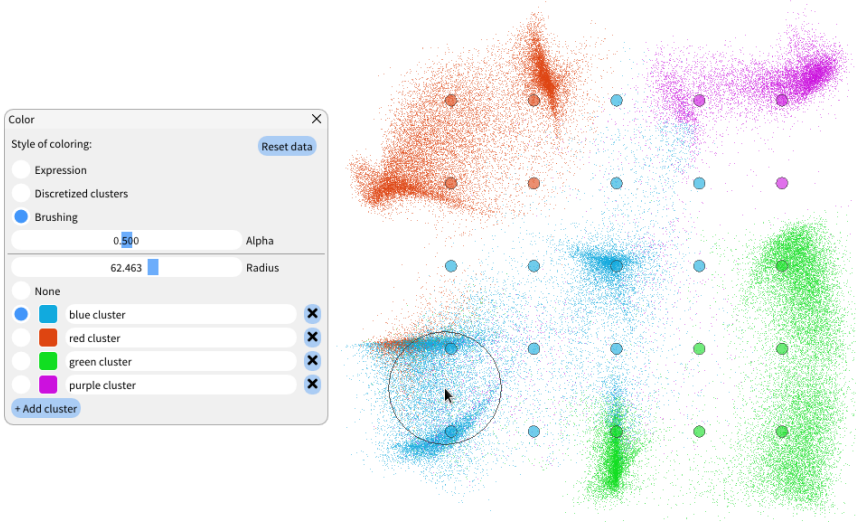

Figure 4.5 The "spraying" of the color is done by choosing the cluster with its color (the blue cluster in the picture) and then moving the mouse with pressed LMB. The landmarks within the circle radius around the mouse cursor (the black circle around the cursor) will be colored. The slider in the Color tool on the left sets the radius size.

Listing 1 Format of the TSV file that stores exported clusters of landmarks.

```
...
landmark_id r g b cluster_name
...
```

the *ColorData* class about the landmarks that should be colored according to the active cluster in *ClusterData*. The *ColorData* stores the colors and IDs of the clusters for each landmark. The *ClusterData* stores information about the clusters added by the user and about the active cluster — which color will be used for coloring.

The results of the brushing coloring can be exported to the TSV file. One line of the exported file contains the ID of the landmark together with the cluster name and color (for the exact format of the exported file see Listing 1). It has as many lines as there are landmarks. The results can be exported from the Save tool with the "Save clusters" option ticked.

Conclusion

The thesis improved the proof-of-concept software BlosSOM to the production-ready state thanks to the following features:

- BlosSOM is now implemented with the light-weight rendering library GLFW which offers only the functionality we need and nothing more. It speeds up the compilation time and reduces the complexity of the software, making it easier for modification and customization.
- The dynamic batch size enables the same framerate on different computers and hence it maintains interactivity.
- Two new functionalities were added to BlosSOM — multiselect and brush coloring. Thanks to these two new features, the software now has all the essential tools needed to explore and analyze datasets.

Some of the original goals of the thesis proved to be rather problematic; but we found sufficiently good alternative solutions:

- We found that the planned CPU-based parallelism actually did not contribute much to the overall throughput of the pipeline, because most of the latency is already present in GPU-accelerated processing. To further offload the GPU, we added a multiple-buffer solution (see Section 4.2) for caching partial rendering results, allowing the rendering of huge datasets to be split into multiple frames with negligible impact on user experience.
- While we managed to construct a method MLEM (see Section 3.1.4) for very precise estimation of the batch size including the confidence intervals, we found that the involved complexity may not be practical – the computation of MLEM either requires inverting a 4th-degree rational polynomial function, or estimating the value of the inverse numerically. BlosSOM now thus uses the simpler variant of MLEM which is not able to account for confidence regions, but can be computed much more easily. At least on the common tasks, we were not able to detect any rendering problem caused by the loss of statistical precision.

Establishing the mathematics required to decide whether MLEM may be computed efficiently is left for future work.

Bibliography

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

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

[3] Guillaume Monneret et al. “How clinical flow cytometry rebooted sepsis immunology”. In: *Cytometry Part A* 95.4 (2019), pp. 431–441.

[4] 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.

[5] Stephan Fuhrmann, Mathias Streitz, and Florian Kern. “How flow cytometry is changing the study of TB immunology and clinical diagnosis”. In: *Cytometry Part A* 73.11 (2008), pp. 1100–1106.

[6] Janghee Woo, Alexandra Baumann, and Vivian Arguello. “Recent advancements of flow cytometry: new applications in hematology and oncology”. In: *Expert review of molecular diagnostics* 14.1 (2014), pp. 67–81.

[7] Neelam Varma and Shano Naseem. “Application of flow cytometry in pediatric hematology-oncology”. In: *Pediatric blood & cancer* 57.1 (2011), pp. 18–29.

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

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

[10] 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.

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

[12] Miroslav Kratochvíl, Abhishek Koladiya, and Jiří Vondrášek. “Generalized EmbedSOM on quadtree-structured self-organizing maps”. In: *F1000Research* 8 (2019).

[13] Soňa Molnárová. “Interactive environment for flow-cytometry data analysis”. In: (2020).

[14] Adam Šmelko et al. “Scalable semi-supervised dimensionality reduction with GPU-accelerated EmbedSOM”. In: *arXiv preprint arXiv:2201.00701* (2022).

[15] *UMAP for Supervised Dimension Reduction and Metric Learning*. July 4, 2023. URL: `https://umap-learn.readthedocs.io/en/latest/supervised.html`.

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

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

[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] J. A. Hartigan and M. A. Wong. “Algorithm AS 136: A K-Means Clustering Algorithm”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108. ISSN: 00359254, 14679876. URL: `http://www.jstor.org/stable/2346830` (visited on 07/05/2023).

[20] Miroslav Kratochvil et al. “Som-based embedding improves efficiency of high-dimensional cytometry data analysis. bioRxiv”. In: (2019).

[21] William Thomas Tutte. “How to draw a graph”. In: *Proceedings of the London Mathematical Society* 3.1 (1963), pp. 743–767.

[22] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

[23] Edgar Anderson. “The species problem in Iris”. In: *Annals of the Missouri Botanical Garden* 23.3 (1936), pp. 457–509.

[24] Ronald A Fisher and Michael Marshall. “Iris data set”. In: *RA Fisher, UC Irvine Machine Learning Repository* 440 (1936), p. 87.

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

[26] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

Appendix A

How to build BlosSOM software

The BlosSOM software currently supports only Unix-like systems and not Windows. For further details, see README on Github[1].

A.1 Dependencies

The BlosSOM has to be built with Cmake[2] build system. GLFW[3] OpenGL library and GLM[4] mathematics library are needed for rendering and mathematical computations. For a compilation of a version with CUDA EmbedSOM, the NVIDIA CUDA toolkit[5] is needed.

These dependencies can be installed as `libglfw3-dev`, `libglm-dev`, and `libgl-dev` binaries on Debian-based systems. The names may differ depending on the Linux distribution.

A.2 Compilation

A.2.1 From a git repository

1. Clone git repository of BlosSOM project

   `https://github.com/molnsona/blossom`

2. Change to the project repository

   `cd blossom`

[1] `https://github.com/molnsona/blossom`
[2] `https://cmake.org/`
[3] `https://www.glfw.org/`
[4] `https://github.com/g-truc/glm`
[5] `https://developer.nvidia.com/cuda-zone`

3. Clone git sub repositories

   ```
   git submodule update --init --recursive
   ```

4. Create build subdirectory and change to build directory

   ```
   mkdir build
   ```

   ```
   cd build
   ```

5. Run Cmake

   A variant without CUDA:

   ```
   cmake .. -DCMAKE_INSTALL_PREFIX=./inst
   ```

   A variant with CUDA (make sure to use correct gcc/g++ compilers compatible with your version of CUDA/nvcc):

   ```
   cmake .. -DCMAKE_INSTALL_PREFIX=./inst -DBUILD_CUDA=1
   -DCMAKE_C_COMPILER=/usr/bin/gcc-10
   -DCMAKE_CXX_COMPILER=/usr/bin/g++-10
   ```

6. Compile the application

   ```
   make install
   ```

A.2.2 From a zip file

1. Unzip the file

2. Change to blossom directory

   ```
   cd blossom
   ```

3. Create build subdirectory and change to build directory

   ```
   mkdir build
   ```

   ```
   cd build
   ```

4. Run Cmake

   A variant without CUDA:

   ```
   cmake .. -DCMAKE_INSTALL_PREFIX=./inst
   ```

   A variant with CUDA (make sure to use correct gcc/g++ compilers compatible with your version of CUDA/nvcc):

   ```
   cmake .. -DCMAKE_INSTALL_PREFIX=./inst -DBUILD_CUDA=1
   -DCMAKE_C_COMPILER=/usr/bin/gcc-10
   -DCMAKE_CXX_COMPILER=/usr/bin/g++-10
   ```

5. Compile the application

   `make install`

A.3 Running

1. Run the application with the directory specified in the `-DCMAKE_INSTALL_PREFIX` flag (in our case `inst`)

   `./inst/bin/blossom`

   Or a CUDA version

   `./inst/bin/blossom_cuda`

2. The application is ready! Look at the How to section (in Appendix B) to start.

Appendix B

How to use BlosSOM

B.1 Quickstart

1. Open the menu by clicking on the bottom right plus button.
2. Click on the first item in the menu to open a dataset.
3. Choose any dataset from `./demo_data/` folder.
4. Now, you can freely interact with the dataset by actions with landmarks (see Section 2.2.3).
5. Also, you can apply different algorithms on landmarks (see Section 1.2) from the training settings tool in the menu.
6. Or change the color of the dataset from the color settings tool in the menu.

A more detailed description of the tools and an example of a demo dataset is on Github[1].

[1] `https://github.com/molnsona/blossom/blob/master/HOWTO.md`