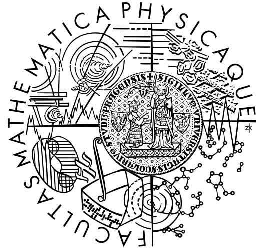


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tereza Kulichová

Configurable point rasterization for large scatterplots

Department of Software Engineering

Supervisor of the bachelor thesis:	doc. RNDr. David Hoksza, Ph.D.
Study programme:	Computer Science
Study branch:	Computer Graphics, Vision and Game Development

Prague 2023

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

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

In . . . . date . . . .

Author’s signature

Dedication. I am very grateful to David Hoksza and Miroslav Kratochvíl for their consistent advice and help. Thanks should also go to my family for their support during my studies.

Title: Configurable point rasterization
for large scatterplots

Author: Tereza Kulichová

Department: Department of Software Engineering

Supervisor: doc. RNDr. David Hoksza, Ph.D., Department of Software Engineering

Abstract: Scattermore is a simple R package used for scatterplot visualizations. Before its reinvention, it gained popularity with the cytometric community because of its functionality and speed. The new version of scattermore offers a highly customizable API. Again, it is much faster than the R standard plot function because some parts of the code are implemented in C language. Plotting points or lines, combining the data in various ways, and avoiding overplotting simultaneously are possible. Except for that, the conducted analysis offers potential speed optimizations regarding cache utility and parallelization.

Keywords: scatterplots, visualization, rasterization, R language

Contents

Introduction 3

1 Contemporary use of scatterplots 7
- 1.1 Common challenges . . . 8
  - 1.1.1 Displaying density . . . 8
  - 1.1.2 Interpretation of correlation and causation . . . 9
  - 1.1.3 Overplotting . . . 10
- 1.2 Improving visualization with image kernels . . . 14
  - 1.2.1 Gaussian filtering and its optimization . . . 14
- 1.3 Histograms for scatterplot summarization . . . 16

2 Redesign of scattermore 19
- 2.1 Input data formats . . . 20
  - 2.1.1 Data formats for intermediate processing . . . 22
- 2.2 Data operations . . . 23
  - 2.2.1 Expanding pixels with image kernels . . . 23
  - 2.2.2 Merging and blending . . . 25
- 2.3 Data formats for output and plotting . . . 25
  - 2.3.1 Data format conversions . . . 26
  - 2.3.2 Plotting and output . . . 26
- 2.4 Rendering performance and parallelization possibilities . . . 27
  - 2.4.1 Performance effects of data layout . . . 27
  - 2.4.2 SIMD for image processing . . . 27
  - 2.4.3 Cache effects and interface to R . . . 29
  - 2.4.4 Available parallelism . . . 30

3 Results and discussion 31
- 3.1 Performance of point scattering . . . 33
- 3.2 Throughput of format conversion . . . 34
- 3.3 Overhead of the R API . . . 34
- 3.4 Speedup of kernel application . . . 35

3.5 New use cases for high-definition scatterplots 39
3.6 Practical usage of scattermore 42

Conclusion **49**

Bibliography **53**

A Using Scattermore **57**

Introduction

R is a popular programming language that is used primarily in statistical analysis and data visualization [1]. According to `TIOBE index`, R still belongs to one of the most used programming languages[1]. R has many interesting packages that are capable of analyzing large and complicated biological data, such as from scRNAseq and single-cell cytometry in packages `Seurat` and `FlowSOM`, respectively [2, 3].

Users of such packages typically expect to be able to plot the huge processed datasets. In the case of `Seurat` and `FlowSOM`, the common output form is a scatterplot where every point represents one of the millions of cells, usually extended by extra information in the form of colors and other point parameters.

Goals Scatterplotting large amounts of data without specialized tools is quite challenging. It may take minutes to render the image with common R plotting methods (such as ggplot2 [4] and the base plotting functionality). This is rather uncomfortable for the users, especially when they need to interactively examine many views of the data.

The R package `scattermore` has previously contributed to the solution to this deficiency. `Scattermore` is sufficiently fast, owing to its optimized scatterplot rasterization implemented in C. Solving the immediate problem, the package gained popularity and became a recommended way to plot large-scale data in several packages.

However, despite the utility, the first version of `scattermore` was a simple, inflexible single-purpose package that only supported only the one most common way of plotting the data. This thesis aims to redesign and reimplement `scattermore` to provide a highly flexible API that would allow users to construct complex plotting pipelines freely. The redesigned version should be able to, e.g., produce good scatterplot density visualizations, provide data for plotting contours, and allow customization of the scattered point shapes. Additionally, as the aim of the `scattermore` is to provide good plotting performance, the thesis explores the available sources of speedups in scatterplot processing (such as parallelization and cache efficiency) and implements several of them.

[1] `https://www.tiobe.com/tiobe-index`

Outcomes As the main result of this thesis, we produce a new version of scattermore that includes composable functionality such as summarizing and colorizing histograms, applying kernels to both picture and histogram data to provide smoothed-out and visually more desirable results, plotting specific point shapes and lines, and combining multiple plotting results to produce more informative layered scatterplots. We describe and implement a pixel format RGBWT, tailored to prevent plotting-order issues with alpha-blending, mitigate the overplotting (section 1.1.3) in the resulting plots, and provide additional flexibility.

Further in the thesis, we conduct experiments and measurements to get a systematic overview of the performance and limits of the new implementation. We observe that the data storage formats and data processing reorganization have helped utilize the memory caches more efficiently. We further parallelize the kernel application (typically used for blurring), gaining a significant speedup for plotting scatterplots with large points.

The results of the thesis to this day are available in the GitHub repository[2] (as of May 2023) and are scheduled to be released to the official R CRAN package repository.

The rest of the thesis is organized as follows: chapter 1 describes the usage of scatterplots, their everyday use in science, and several known approaches to address the commonly occurring issues. Chapter 2 describes the redesign of scattermore, introducing all newly available operations and data structures and showing the opportunities for performance improvements. Chapter 3 then summarizes the results from optimization experiments and displays several examples and use cases for high-volume scatterplotting.

Related work

Scattermore is currently used in several R packages such as ShinySOM [5], scHOT[3] and Seurat[4]. At the same time, it is used in assorted bioinformatics pipelines, such as the one for the creation of a retina single-cell transcriptome dataset[5].

Similar packages Many programming languages and environments have packages available that address similar problems as scattermore. As an example,

[2] https://github.com/teri934/scattermore-thesis

[3] https://github.com/shazanfar/scHOT

[4] An up-to-date list of reverse dependencies of scattermore can be found at https://cran.rstudio.com/web/packages/scattermore/index.html.

[5] https://iovs.arvojournals.org/article.aspx?articleid=2773412

Datashader[6] is a Python package that can visualize extensive data fast, utilizing Numba [6] to compile the computation-intensive code written directly in Python to optimize machine code, enabling it to approach the speeds of C or FORTRAN[7]. Similarly, Julia ecosystem has GigaScatter.jl[8] which can efficiently rasterize points, enlarge them using kernels, and combine multiple rasters.

Use cases The main motivation for this thesis is software that needs to render scatterplots quickly, typically for interactivity reasons. Correspondingly, the original motivation for scattermore was single-cell data rendering in ShinySOM package.

The web-based frontend of ShinySOM[9] provides functionalities similar to FlowSOM, but in a more straightforward and graphical form that is suitable for biologists. ShinySOM aimed to minimize the computational delays by utilizing and visualizing highly efficient versions of the analysis based on self-organizing maps [7] from FlowSOM. In this setting, the R plotting routines form a major bottleneck that severely limits the interface responsiveness.

Seurat[10] is an R package designed for the analysis of single-cell RNA-seq data, aiming to identify and interpret sources of heterogeneity in single-cell transcriptomic measurements and to integrate single-cell data from different experiments [2]. Typically, Seurat users view multiple renderings of the single-cell data colored by different genes based on user selection, again placing a strong requirement on fast scatterplotting.

GigaSOM.jl[11] [8] is used for distributed clustering and analysis of extensive cytometry data using Julia. It is similar to FlowSOM in its functionality and thus faces the same plotting performance challenges. GigaScatter was implemented as an immediate solution, allowing users to quickly plot partial scatterplots from data slices residing at multiple distributed processes and efficiently merge them at the coordinating process. Due to the simple implementation, GigaScatter also suffers from overplotting issues.

[6] https://datashader.org
[7] https://numba.pydata.org
[8] https://github.com/LCSB-BioCore/GigaScatter.jl
[9] https://github.com/exaexa/ShinySOM
[10] https://github.com/satijalab/seurat
[11] https://github.com/LCSB-BioCore/GigaSOM.jl

Chapter 1

Contemporary use of scatterplots

Scatterplots are plots that use dots to visualize values dependent on two numeric variables. The position of a dot on the horizontal and vertical axis determines the value for an individual data point. There is a lot of information to be gained from scatterplots (figure 1.1) because they can show patterns in the data in addition to the individual values. They are useful for various purposes, such as dividing data points into groups according to how closely points are and forming clusters. Scatterplots can also point out unexpected gaps in the data and find possible outlier points[1].

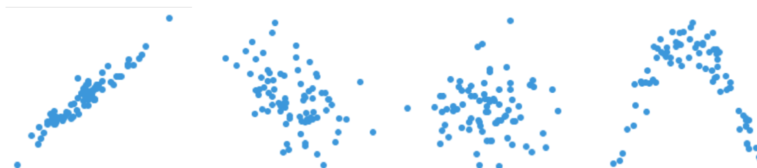


Figure 1.1 Examples of relationships between variables in scatterplots. From left to right: strong, positive, linear; moderate, negative, linear; no relationship; strong, non-linear. Scatterplots provide an intuitively comprehensible picture of this relationship. Taken from chartio.com[1].

Primarily, scatterplots are not suitable for showing *density* in the data. Density in a scatterplot is the number of points assigned to an individual pixel. For the purpose of displaying density, histograms and contour-based visualization may supplement scatterplots to illustrate the density better, using color scales and lines. This is one of the reasons why it is useful to transform scatterplots into histograms.

[1] https://chartio.com/learn/charts/what-is-a-scatter-plot

1.1 Common challenges

Scatterplots are useful in various fields and are a very good tool for data visualization. However, some issues in the visualization process need to be handled. We chose to analyze three of these challenges that we deemed to be the most important in most use cases:

- Misinterpretation of *correlation* and *causation* in section 1.1.2.
- Speed. It is challenging for a generic plotting package to work with a billion points in a reasonable time. That is why speed is a significant challenge. For software where tasks require some interactions, e.g., `ShinySOM` [5], it can even be problematic to analyze over 10 million points. However, `scattermore` is suitable for this task because of its partial implementation in C (chapter 3).
- *Overplotting* and *density representation* in section 1.1.3. The phenomenon of overplotting is related to the already mentioned density representation – the problem of dealing with many points on the same spot. 2D histograms (section 1.3) offer a solution for this situation and are better for plotting density than colorful scatterplots, but at times better techniques need to be considered.

1.1.1 Displaying density

When analyzing data visualizations, one often has to choose between being able to distinguish individual points and a good density estimation. With scatterplots, smaller point sizes and low alpha values are suitable for satisfactory distribution visualization, but it is challenging to see outliers. High alpha values with larger point sizes offer a good discriminability of outliers; however, density estimation is impossible (figure 1.2).

One of the possible solutions for this is to use hexagonal binning or scatterplots combined with contour plots where density is depicted with changing intensity values (section 1.1.3).

Alternatively, in this thesis, we argue that in an interactive visualization environment, the ability to quickly rerender scatterplots based on user choice of point size and alpha solves this problem for many use cases. If available, the user may dynamically adjust the density rendering, allowing observation of the desired density range.

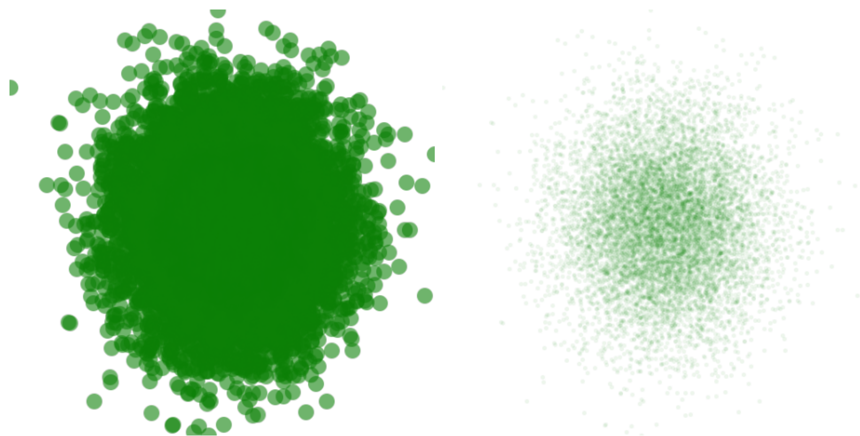


Figure 1.2 Trade-offs between displaying density and outliers with plain scatterplots. One possibility is to use a high alpha value with larger points (left) to make them more distinguishable. Another way is to make them smaller with a low alpha value (right) to ensure good density estimation.

1.1.2 Interpretation of correlation and causation

Correlation does not imply causation. Still, it is difficult not to forget the knowledge when presented with actual data – assuming that the reason for that is the combination of misunderstanding and expectations hoping to come true. Bergstrom and West [9] say "correlations only hint at how the world might work; causal relationships directly answer that question" and even allow to suggest possible actions to reach the desired outcome.

This fact is crucial when working with a scatterplot because its relationships tend to misinterpret easily. The fundamental problem with scatterplots is their depiction on the Cartesian plane, often used to display "cause-and-effect or dependency relationships." Especially students tend to assume that the variable on the horizontal axis somehow influences the value on the horizontal axis [9].

One of the solutions for the problem is to display the same data using *diamond plot* (figure 1.3) visualization. The axes are rotated by $45°$, so both have the same importance. There is no distinction between the dependent and independent variables, and the new format requires greater attention in the attempt to draw causal conclusions [9].

Fast replotting is important when there is a need to redraw data because of this possible misinterpretation.

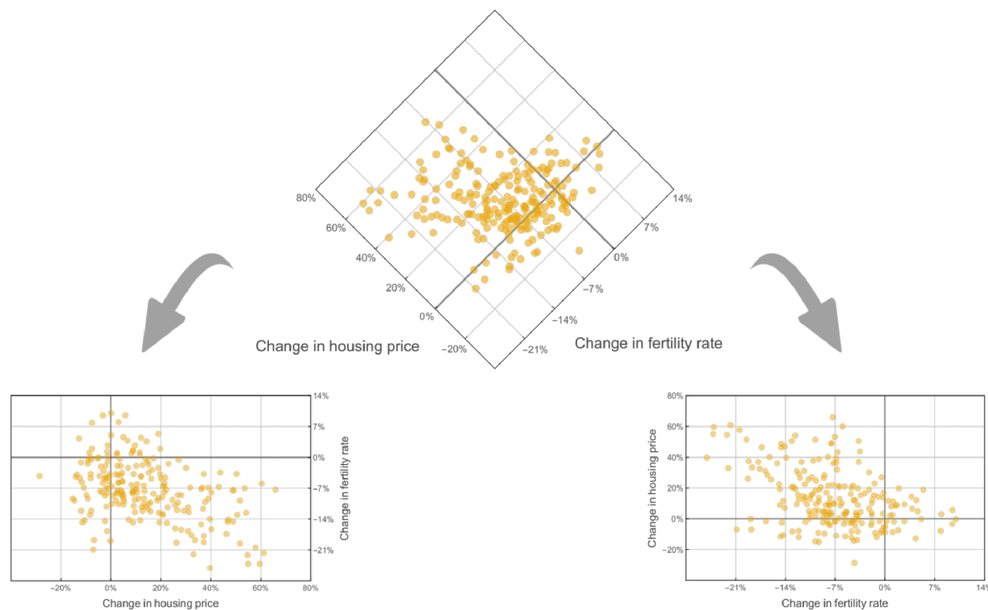


Figure 1.3 An example of a diamond plot and 'ordinary' scatterplots with the same data. The diamond plot does not have the tendency to create 'false' relationships between variables. Taken from the article about diamond plots [9].

1.1.3 Overplotting

Overplotting is a scatterplot visualization problem where data points overlap so much that it becomes challenging to distinguish the features in the data. Many points, multiple data sets, and high-density regions create this problem. Even when overplotting is not a prominent issue in some cases, the large amount of information leads to *visual clutter*. Here, we illustrate three known methods for reducing some of the adverse effects of overplotting: the combination of *point filtering* and *density* or *contour plots*, *binning*, and *point order mixing*.

Points and plots Mayorga and Gleicher [10] introduced an innovative technique to avoid visual clutter when designing scatterplots. It also offers a nice way to analyze density at the same time solving the problem mentioned in section 1.1.1. In this method, two approaches:

- density or contour plots
- point filtering

are used to visualize the results as well as possible.

The phases are the following; density estimation and thresholding are applied to aggregate the points into contours. Then the distance transform eliminates

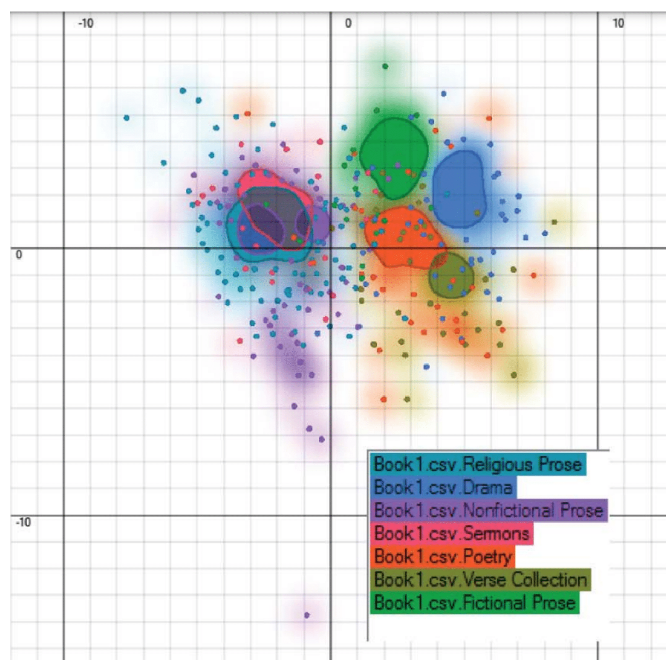


Figure 1.4 An example of scatterplot visualization employing contour plots and point filtering to reduce overplotting. Taken from the article about this technique [10].

the points lying too close to the contours. The next step is to color dense regions forming a bounded smooth shape. Color choices must be easily distinguishable and coexist well with the blending strategy. The CIE Lab color space is employed because it is easier to differentiate between colors whose perceptual distances can correspond to Euclidean distances [10].

Finally, the algorithm combines all sub-results and creates the scatterplot. Changes in lightness and chroma (purity of color) parameters together with alpha help to indicate between overlapping sets. After that, an operation must identify outlier points to prevent visual clutter and overdrawing. Then a decision occurs on which points participate in the sampling process [10]. This procedure generates a new kind of visualization (figure 1.4).

Binning Another way to reduce overplotting uses hexagonal binning that plots rather density than points, and colors display the number of points per a heaxagon[2]. This method also solves the problem of density.

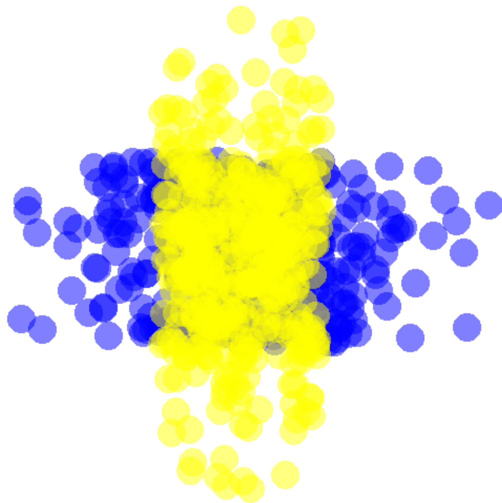
Compared to the previous one, this more straightforward technique is more proper for visualizing cells when dealing with single-cell RNA-sequencing data. In the process, Freytag and Lister [11] divide the plotting area into a regular grid

[2] `https://datavizproject.com/data-type/hexagonal-binning`

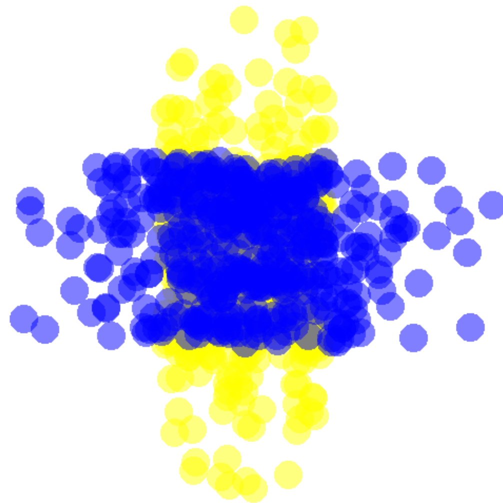
of hexagons, and then cells are allocated to a 'bin' they belong to. The main disadvantage of this technique is the resolution decrease. The package `schex` in R makes the hexagonal plotting accessible to the scientific community [11].

Poing order mixing The simplest and the most popular method for reducing overplotting utilizing point order mixing is point order mixing visualized by figure 1.5. The technique is based on randomizing the plotting order of points to mitigate this undesired phenomenon because it primitively simulates the color mixing when laying the points of alternating color on each other. Nevertheless, the resulting effect does not reflect reality well enough.

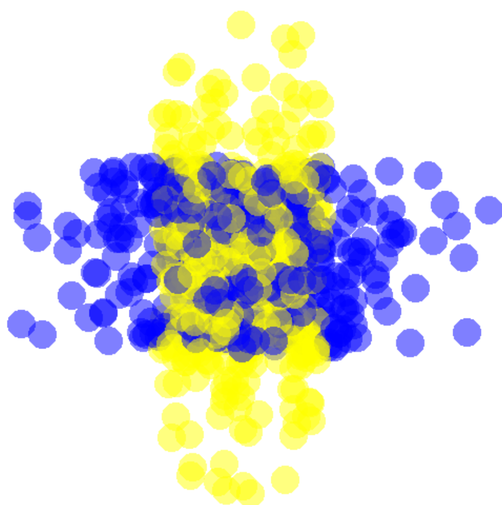
The figure 1.5 also illustrates the approach chosen by this thesis; mitigation of overplotting with RGBWT matrix ensuring correct color mixing. For details, please see chapter 2.



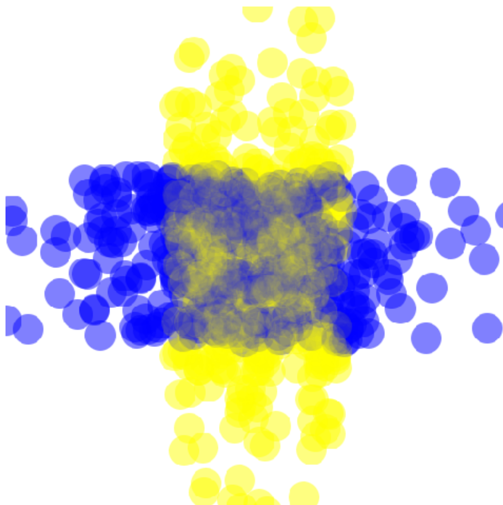
(a) Overplotting — yellow cluster completely shadows the blue one.



(b) Symmetric case of overplotting.



(c) Simple mitigation of overplotting by randomizing the point plotting order.



(d) Overplotting mitigation by using RGBWT matrix.

Figure 1.5 Example of overplotting and techniques dealing with it. The first three images are plotted using R's default method, and the image **(d)** is plotted with `scattermore`.

1.2 Improving visualization with image kernels

Image *kernel* is a 2D matrix filter used for traversing an image and getting information from it. The filters have versatile usage when employing *convolution* in the space domain. The process is shown in figure 1.6.

Kernels have been used for various purposes, such as detecting edges based on first or second derivatives or reducing image noise. This is achieved by image *smoothing*, removing high-frequency content, like edges and corners, from an image [12].

With scatterplots, kernel filtering may be used to expand or smooth out the points to improve visualization and to smoothen the point histograms in order to make them more suitable for other visualization methods, such as heat maps or contour plots.

For some convolution filters, it is essential to precompute them to avoid costly repetitive computations. For example, evaluating the exponential in a Gaussian function may be quite expensive, so the filtering cost can be quite high using this technique. That is why approximation methods mentioned in the next section are utilized.

1.2.1 Gaussian filtering and its optimization

A *Gaussian filter* is a filter whose impulse response approximates a Gaussian function.

Definition 1 (A Gaussian function)**.** *Let σ be a non-zero and μ a real constant. Then the function*

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

is a Gaussian function[3].

Computer vision often uses Gaussian smoothing: $smoothed_image(x) = \frac{\sum_y value_y * g(|x-y|)}{\sum_y g(|x-y|)}$.

Applying the Gaussian filter tends to be computationally more expensive because the weight assigned to each pixel is calculated with the use of exponential and square root operations (if the kernel is not precomputed). Additionally, the Gaussian kernel has a larger kernel size compared to the simple square kernel of ones to achieve the same level of smoothing; kernel support has to be large enough to fit the essential part of the Gaussian [13].

That is why especially higher-resolution images require minimizing their filtering costs. One of the solutions for that is to approximate the Gaussian

[3] `https://en.wikipedia.org/wiki/Gaussian_function`

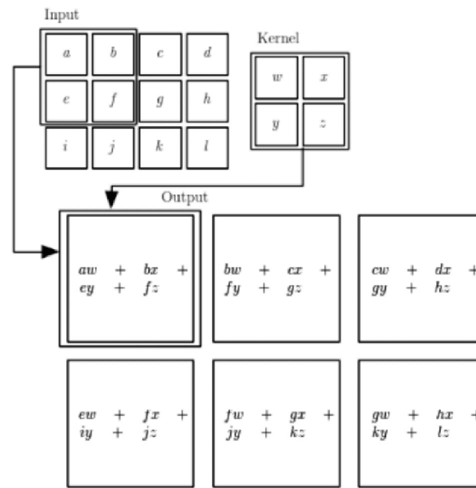


Figure 1.6 An example of 2D convolution without kernel flipping. The kernel is 'placed' over the input data, and the output is computed according to the expressions in the image [15].

filter through multiple averaging filters. Kovesi [14] describes the possibilities to compute these filters in the space domain as follows.

Integral image, or a summed area table, "can be generated by computing the cumulative sums along the rows of an image and then computing the cumulative sums down the columns." So the value at any point (x, y) in the integral image corresponds to the sum of all the image pixels above and the left of (x, y), including the point itself. Then the sum of all image pixels within an arbitrary rectangle is computed and divided by the number of pixels in the rectangle resulting in an averaging filter [14].

Separability of the averaging filter and performing repeated moving average filtering on the rows and columns of the image is another way to achieve the desired results. The computational cost can reach similar results to the method using integral pictures in case some division operations on the repeated averaging passes are united. The possible solution is to have two filter sizes and decide the number of passes according to σ [14].

When using integral images, three repeated averagings achieve a passable approximation to a Gaussian, and the approximation becomes very good beyond four repeated averagings. In the other case, e.g., for the approximation of a Gaussian with $\sigma = 40$, five passes were sufficient, with the actual standard deviation being 39.983 [14].

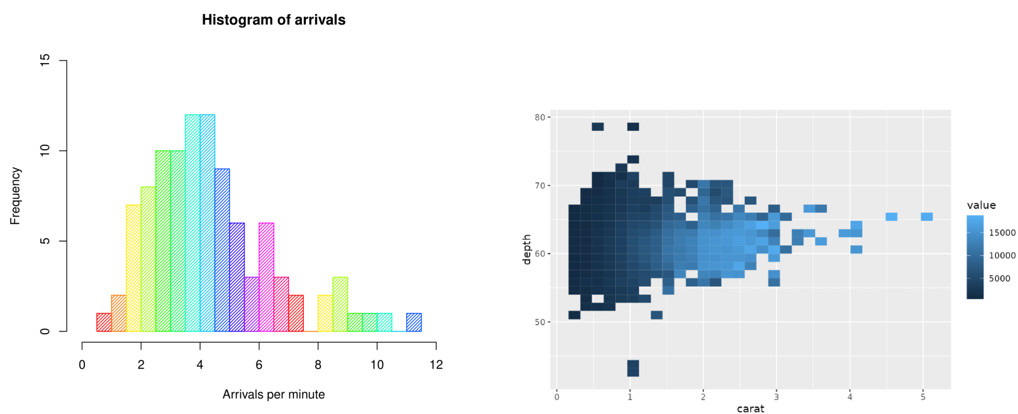


Figure 1.7 Examples of a simple and 2D histogram[4][5].

1.3 Histograms for scatterplot summarization

"A *histogram* is a picture history of a statistical distribution" [16]. Histograms consist of 'bins' created by dividing the entire range of values into a series of intervals and then counting how many values fall into each gap. If the bins have equal sizes, there is a corresponding rectangle to each 'bucket' with a height proportional to the number of cases in each bin (figure 1.7). However, if the containers do not have equal width, then the frequency is calculated from the area of the rectangle[4].

Histograms are suitable for estimating the precise data density compared to scatterplots, providing good data structure visualization. However, most of the time, both of the parameters are needed. Please see figure 1.8 for a possible solution in R.

In addition to the use in scatterplot simplification, histogram data may serve other purposes. These include, e.g., *histogram equalization*, which improves the perceived contrast in the pictures [17] and may be beneficial in data science for choosing good levels for contours. A method resulting in a good contrast provides more gray-level values or more saturation values for the different color tones in the image, but there is almost no image degradation.

Although histogram equalization gives good results, it is not a fool-proof method, so other techniques exist to achieve an even better outcome. Wang and Ye [18] introduce a process called brightness-preserving histogram equalization, reaching a new image with similar brightness levels to the original one.

[4]https://en.wikipedia.org/wiki/Histogram
[5]https://ggplot2.tidyverse.org/reference/stat_summary_2d.html
[6]https://ggplot2.tidyverse.org/reference/geom_density_2d.html

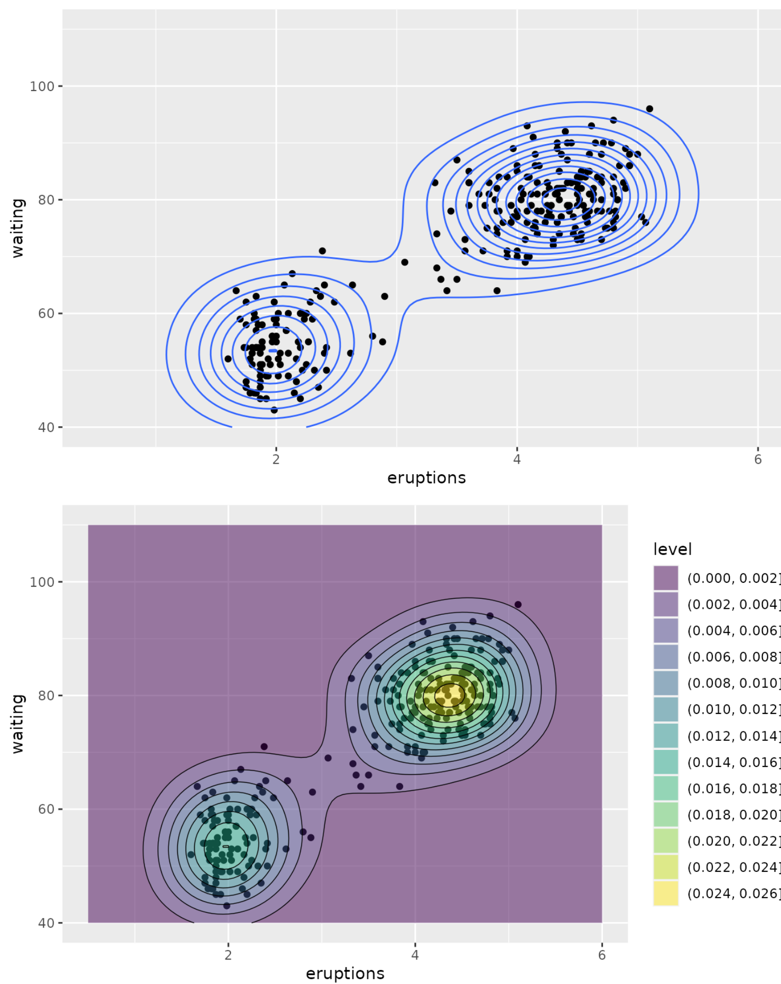


Figure 1.8 Contours of a 2D density estimate[6]. R uses contours and alternating colorful regions for density estimation, which can be useful for overplotted regions.

Then another way is to apply a procedure based on piece-wise linear transformation where the smooth version of the histogram created by a Gaussian filter passing low frequencies finds the total number of segments [17].

Chapter 2

Redesign of scattermore

The original version of `scattermore` offered fast plotting of millions of points. Compared to the R's default plot function, it took only a few moments[1]. However, the original API did not allow any further flexibility in configuring the rendering procedure. Ideally, one would expect features such as combining scatterplots with different datasets, improving scatterplot visualization with custom kernels, and drawing lines.

In the next part, we describe our reimplementation of `scattermore` internals, which features a comprehensive API and thus enables high customizability. This chapter mainly concerns the structure of this API, the speed of the underlying implementation, and the optimizations that we applied.

[1] `https://academic.oup.com/bioinformatics/article/36/10/3288/5734646?login=false`

2.1 Input data formats

This section introduces `scattermore`, its input formats, and the logic behind their implementation.

Proper methods must accept data, either with or without color input, to be able to visualize something later. The figure 2.1 shows this initial decision. The two approaches to accomplish that are drawing points or lines, each having its appropriate functions:

- *Point drawing functions* convert 2-coordinate point data into a raster grid of a given size. The input for points consists of two coordinates, and the given points 'fall' into the raster grid with assigned sizes. If there was no color input, the process creates a histogram or, in the other case, an RGBWT data structure. For more information, please see section 2.1.1.
  - `scatter_points_histogram` outputs histogram data
  - `scatter_points_rgbwt` also accepts a color description of points and outputs the RGBWT matrix
- *Line drawing functions* convert line data into a raster grid of a given size. The approach is similar to the case of only points but with something to add. Information about lines comes from two pairs of start and end coordinates. Bresenham's line algorithm ensures the drawing of an arbitrary number of lines. Its task is to approximate a straight line between two points conveniently. It works only with cheap computer operations such as integer addition, subtraction, and bit shifting[2].
  - `scatter_lines_histogram`
  - `scatter_lines_rgbwt`

Drawing lines

We specifically chose to provide the explicit line drawing primitives because sometimes it is not enough to plot only input points because the result does not reflect expectations. With a given number of points, one can visualize the image 'perfectly' only if the bitmap size is sufficiently small. But, as figure 2.2 shows, with the support for drawing lines, it is more likely to meet the expectations because, for every pair of points, new points lying on a line between them are drawn too.

[2] `https://gitlab.cecs.anu.edu.au/pages/2018-S2/courses/comp1100/assignments/02/Bresenham.pdf`

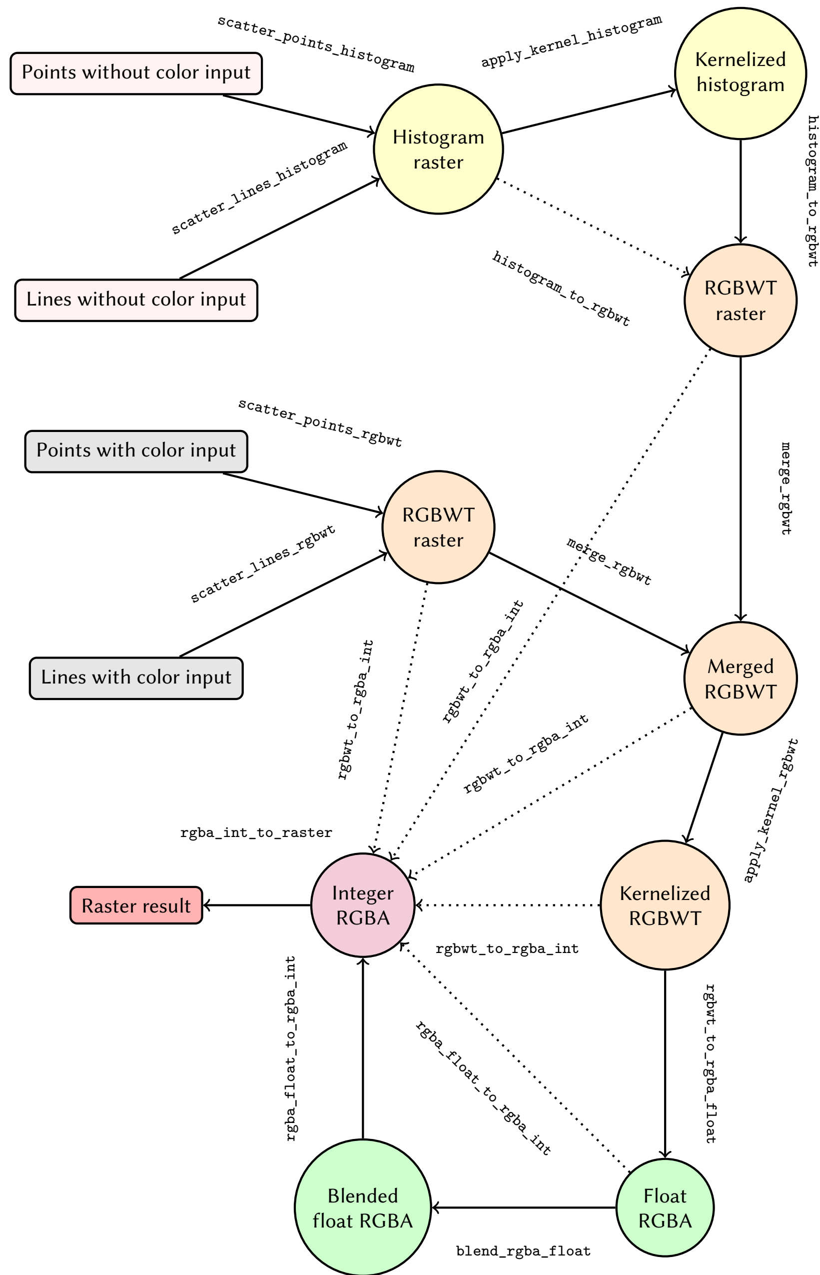


Figure 2.1 Overview of functions and data format conversions implemented in `scattermore`.

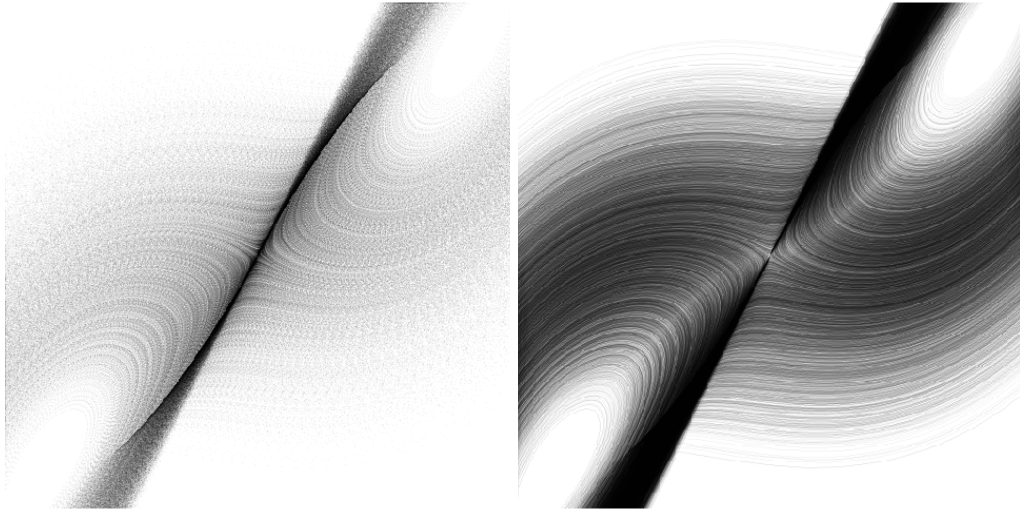


Figure 2.2 Rendering the Lorenz attractor using points (left) and lines (right). Rendering multiple lines as a series of points creates visual artifacts that confuse the perception of the structure and the apparent density is confusingly diminished in areas where the attractor approximation 'moves' quickly. Both problems are most apparent in the top-left area of the rendering.

2.1.1 Data formats for intermediate processing

The methods in `scattermore` are responsible for storing data in various ways according to their usage. The primary data formats are *histogram*, *RGBWT* matrix, and *RGBA* matrix.

Histogram

In `scatermore`, we use histograms represented by 2D arrays of integers that count the number of points that were assigned to the particular bucket (or pixel) in the grid. This format is useful when there is no available or required color input. For more information about histograms, please see section 1.3. By assigning a palette of colors to the histogram, an RGBWT matrix comes into existence.

RGBWT matrix

The RGBWT matrix is a 2D array of pixels where each pixel has 5 channels. Each pixel has entries about red, green, blue, and alpha color channels[3].

- The R, G, and B entries include information about the result pixel coloration for each channel and their opacity (A as alpha channel). They are generated

[3]`https://en.wikipedia.org/wiki/RGBA_color_model`

by multiplying the color channel's value with the alpha channel's value for each pixel and eventually summing them together. This *premultiplying* of alpha eliminates the need for additional computations when combining more transparency values, which would involve repetitive multiplying and dividing by A.

- The W entry stands for weight, and it stores the total color amount assigned to the pixel, i.e., a sum of alpha values of all assigned points for each pixel.
- The T entry describes the total pixel transparency – pixels start with transparency 1, and the transparency is multiplied by transparency (calculated as 1-A) of all incoming pixels.

This data storage system preserves essential information lost in other formats, e.g., the RGBA matrix (section 2.2.2). Crucially, separating the transparency and weight value enables precise, order-independent mixing of individual pixel colors. This solves the problem in figure 1.5 in section 1.1.3.

RGBA matrix

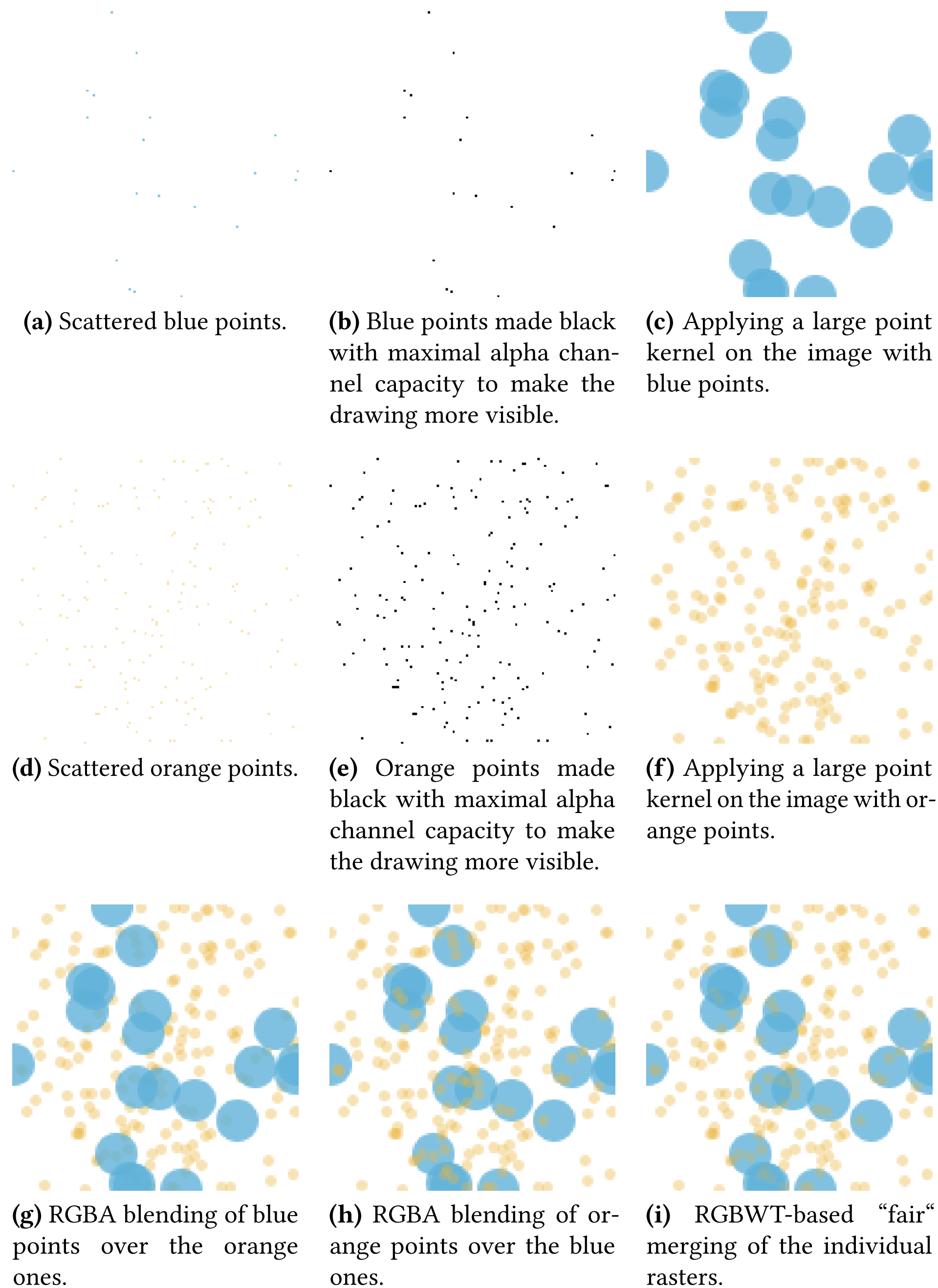
Like the RGBWT matrix, this data format stores information about the result pixel coloration and its opacity for each channel. However, the information about entries W and T from the RGBWT matrix is present in R, G, B, and A entries in the new matrix form. Except for the alpha channel, the color channels store information about alpha too, so A is again premultiplied to simplify possible blending operations. This format is present in a float and an integer variant.

2.2 Data operations

This section describes possible data operations such as smoothing and two ways of image combining.

2.2.1 Expanding pixels with image kernels

With `scattermore`, applying a kernel to the data is possible. Available filters are in the shape of a square, circle, or Gaussian filter (section 1.2.1), and there is an opportunity to design your mask. The kernel operation may be applied to the histogram and RGBWT matrices, using methods `apply_kernel_histogram` and `apply_kernel_rgbwt` (figure 2.1). The algorithm for smoothing alone is implemented in C to provide sufficient performance.



(a) Scattered blue points.

(b) Blue points made black with maximal alpha channel capacity to make the drawing more visible.

(c) Applying a large point kernel on the image with blue points.

(d) Scattered orange points.

(e) Orange points made black with maximal alpha channel capacity to make the drawing more visible.

(f) Applying a large point kernel on the image with orange points.

(g) RGBA blending of blue points over the orange ones.

(h) RGBA blending of orange points over the blue ones.

(i) RGBWT-based "fair" merging of the individual rasters.

Figure 2.3 Example of possible operations with `scattermore`.

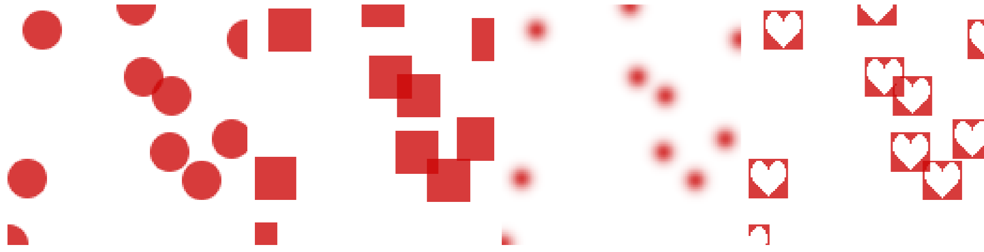


Figure 2.4 Example of available masks in `scattermore` (left to right: circle, square, Gaussian, and custom a custom user-defined heart mask.

Kernel application may be easily used to expand small single-pixel points (e.g., the outputs of the scattering functions) to ones with a specific larger shape (figure 2.4).

It can be especially useful when the number of points needed to be visualized is larger than the number of pixels because the blurring is done on a bitmap of already processed points, so a large number of points is not a speed hindrance. However, even with fewer points, the post-expansion of points using a kernel has a performance benefit because the kernel operation convolution can be easily parallelized. Please see section 3.4 for details.

2.2.2 Merging and blending

`Scattermore` also offers a possibility to combine more visualizations together; the methods `merge_rgbwt` and `blend_rgba_float` take care of the process.

When combining two RGBWT matrices, the result is an equivalent color mixture, so there is no overplotting. All the matrix's R, G, B, W, and T entries are equally summed or multiplied together in the case of T, creating a new matrix. This form of storing data is crucial for preventing overplotting (figure 1.5).

In contrast, when blending two RGBA matrices, one entry is more prioritized than the other one depending on the value of A because when combining them, the complement of A of the first one stands as the A value for the other. The resulting visualization shows the inequality (figure 2.3).

2.3 Data formats for output and plotting

This section describes the possible conversions among formats, their usage, and the eventual plotting with a simple example.

Listing 1 Example of plotting and saving a simple image.

```
library(scattermore)
library(magrittr)

png(filename = "your_path")
par(mar = c(0,0,0,0))
pts <- cbind(rnorm(1e5), rnorm(1e5))

# high-level interface compatible
# with original scattermore
pts %>% scattermore %>% plot(interpolate = F)

# low-level customizable interface
pts %>% scatter_points_rgbwt %>% rgbwt_to_rgba_int %>%
rgba_int_to_raster %>% plot(interpolate = F)
dev.off()
```

2.3.1 Data format conversions

Conversions ensure high variability of the visualization process and its eventual plotting. The histogram has to be converted to an RGBWT matrix using the method `histogram_to_rgbwt`. The algorithm is written in C to allow its fast conversion.

Then the RGBWT matrix can transform into the float RGBA matrix using `rgbwt_to_rgba_float` and then `rgba_float_to_rgba_int` methods, or it is possible to convert it into the integer RGBA matrix straight away with the function `rgbwt_to_rgba_int`. The first case is valid when one wants to apply some blending before the following visualization (figure 2.1).

The final transformation entails the change into the raster format with `rgba_int_to_raster` utilizing R's default function to support plotting.

2.3.2 Plotting and output

R takes care of the actual plotting with its `plot` function accepting a raster form of the data[4]. The method is not just a straightforward function but a means to call other methods dealing with the plotting or different process according to the input parameters. `Scattermore` uses the plot function only at the end of the process, changing the input data structure to a raster format, so no potentially slow algorithms are employed on the data.

[4]https://www.rdocumentation.org/packages/graphics/versions/3.6.2/topics/plot

Listing 2 Example of a structure of arrays. All individual R, G, and B components of a pixel are grouped together and then stacked next to each other.

```
struct RGB_list {
    int R[N];
    int G[N];
    int B[N];
};
struct RGB_list colors;
int get_channel_R(int i) { return colors.R[i]; }
```

2.4 Rendering performance and parallelization possibilities

There are various ways to improve the performance of implemented algorithms and, consequently, the resulting plotting. Some methods include cache utility optimization using different storage formats, *SIMD*, and parallelization techniques for image blurring.

2.4.1 Performance effects of data layout

Structure of arrays or *SoA* 2 and *array of structures* or *AoS* 3 are data storage formats. It is helpful to group data together in the best suitable form to improve spatial locality. The spatial locality ensures that elements close to each other are stored in the same memory block[5]. AoS is a more conventional data storage method supported by most programming languages, whereas when using SoA, there is a possibility to use SIMD instructions (section 2.4.2).

Array of structures of arrays 4 is a hybrid approach between the two formats. It enables using SIMD while offering a good cache locality. The idea here is to benefit from the locality at the outer level and unit-stride at the innermost level[6].

2.4.2 SIMD for image processing

SIMD (Single Instruction/Multiple Data) is a computing method that enables multiple data processing with a single instruction. It uses only one instruction to achieve the same result as conventional scalar operations. Data types used for SIMD operations are called vector types[7].

[5] https://en.algorithmica.org/hpc/external-memory/locality

[6] https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html

[7] https://en.algorithmica.org/hpc/simd

Listing 3 Example of an array of structures. The pixel elements are stacked alongside and all R, G, and B components of an individual pixel are stored next to each other.

```
struct RGB {
    int R;
    int G;
    int B;
};
struct RGB colors[N];
int get_channel_R(int i) { return colors[i].R; }
```

Listing 4 Example of an array of structures of arrays. It is a hybrid approach between the structure of arrays and the array of structures formats.

```
struct RGBx8 {
    int R[8];
    int G[8];
    int B[8];
};
struct RGBx8 colors[(N+7)/8];
float get_channel_R(int i) { return colors[i/8].R[i%8]; }
```

In `scattermore`, we used SIMD in the SoA format to improve the speed of the C method `histogram_to_rgbwt`. As seen in listing 5, SIMD instructions enable access to more elements in the array at once. For example, `_mm_loadu_ps` extracts four 32-bit floating-point elements[8]. The listing 6 shows that the 'SIMD' methods are also effective in assembly because they have their own special instructions.

However, SIMD uses larger register files, increasing power consumption and required chip area. Next, there can be restrictions on data alignment[9]. Then the simultaneous access advantage makes it possible to use only a specific predefined processing pattern, meaning it is impossible, e.g., to add some elements from two groups of data and subtract others from the same two groups in a single processing instruction[7].

Nevertheless, there is a possibility to solve these issues. VeGen, a vectorizer generator, enables practical usage of non-SIMD vector instructions making it possible to use non-isomorphic and cross-lane operations. LLP (Lane Level Parallelism) captures the parallelism model with SIMD and non-SIMD vector instructions. It "automatically generates a vectorization pass to uncover target-

[8] `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`

[9] `https://www.wikiwand.com/en/Single%20instruction,%20multiple%20data#Disadvantages`

Listing 5 Example of the SIMD usage in scattermore. The code is taken from histogram_to_rgbwt method implemented with vector instructions.

```
__m128 histogram_values = _mm_loadu_ps(histogram);
// normalize histogram_values
histogram_values =
  _mm_mul_ps(_mm_sub_ps(histogram_values, minimums),
  differences);
// palette indices multiplied by 4
__m128i palette_indices =
  _mm_cvtps_epi32(_mm_mul_ps(histogram_values, bins));
// correct if one of the indices is equal to size_palette
palette_indices = _mm_min_epi32(palette_indices,
sizes_palette);
```

Listing 6 Assembly code of the SIMD usage example5. The translated 'SIMD' functions are represented in assembly by specific instructions.

```
vmovups (%rcx), %xmm3
addq   $16, %rcx
vsubps  %xmm17, %xmm3, %xmm0
addq   $16, %rdx
vmulps  %xmm5, %xmm0, %xmm0
vmulps  %xmm18, %xmm0, %xmm0
vcvtps2dq %xmm0, %xmm0
vpminsd %xmm16, %xmm0, %xmm0
vmovd %xmm0, %eax
sall   $2, %eax
cltq
```

architecture-specific LLP in programs while using only instruction semantics as input." This new approach to non-SIMD instructions can accelerate applications in fields such as machine learning, image, and digital signal processing [19].

2.4.3 Cache effects and interface to R

The cache efficiency is an essential factor to be considered when analyzing performance. When drawing a single point, more data must be loaded from the memory because the processor cannot load individual bytes, so it has to be done by chunks of data.

As mentioned in section 2.4.1, the structure of arrays and the array of structures enable different approaches for cache access, meaning that the resulting

speed depends on the storage format. Accessing the SoA storage format more optimally is possible using SIMD. These data storage formats also influence the conversion speed from R to C and another way around.

In our experiments, we also consider the kernel algorithm to improve the cache utility, especially the looping movement of the kernel through the image.

2.4.4 Available parallelism

Another way how to improve the performance of operations in `scattermore` is multithreading. Applying this concept to the scattering process and the kernel algorithm could be helpful.

To parallelize the points scattering safely and avoid a race condition, it is necessary to divide the image bitmap so that each part is accessible only by a single thread and, in the end, merge the results. However, for every operation, a random cache line has to be loaded to the memory, so the threads must wait for it. Except for that, with sufficient threads, the memory bandwidth becomes a bottleneck.[10].

Nevertheless, the parallelization of the kernel algorithm is possible and relatively straightforward. We do multithreading by dividing the image bitmap into many blocks, assigning the neighboring block to another thread, and then repeating this process. This means that the threads do not fight over the available cache space.

[10] `https://en.algorithmica.org/hpc/cpu-cache`

Chapter 3

Results and discussion

In this chapter, we verify the assumptions from section 2.4 by measuring the performance of the following C methods in various scenarios.

- `scatter_histogram`
- `scatter_singlecolor_rgbwt`
- `histogram_to_rgbwt`
- `kernel_histogram`
- `kernel_rgbwt`

To summarize, the areas of `scattermore` taken into account are point scattering, conversion from histogram to RGBWT format, and kernel application. The results of the following benchmarks can tell us how much it is valuable and convenient to implement these optimizations to `scattermore`.

For measuring the C methods, the C++ class `high_resolution_clock` was employed, and for its R interface, the method `system.time` was utilized. All of the measurements were performed on the hardware with the following specifications:

- Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 MHz, 4 Core(s), 8 Logical Processor(s)
- RAM: 16,0 GB (15,7 GB usable)

Performance evaluation methodology Most of the operations run quickly, and many factors influence the benchmarking process: that is why we saw large variances in the resulting speeds. Therefore, with the help of linear regression, we decided to employ strategy from Haskell library `Criterion`[1] to determine the speed of operation from several measurements with different problem sizes.

Briefly, the time t needed to solve the problem composed of the initialization part (taking time b) and N algorithm repetitions for solving N objects (each taking time a) is $N * a + b + e$ where e is the randomly distributed error. We measured most of the benchmarks in chapter 3 using a large number of problem sizes (with different N), and we estimated values a a b with the help of OLS algorithm [20, 21]. Consequently, we reported time a in the plots.

Furthermore, OLS allowed us to estimate the accuracy of a and to report it as the amount explained into variance – statistically R^2. In generated plots, we also reported R^2 with the help of point size, e.g., figure 3.2, which enabled us to recognize parameters or conditions of the algorithm where the total time was mostly determined by random events, e.g., such as page fault.

[1] `https://github.com/haskell/criterion`

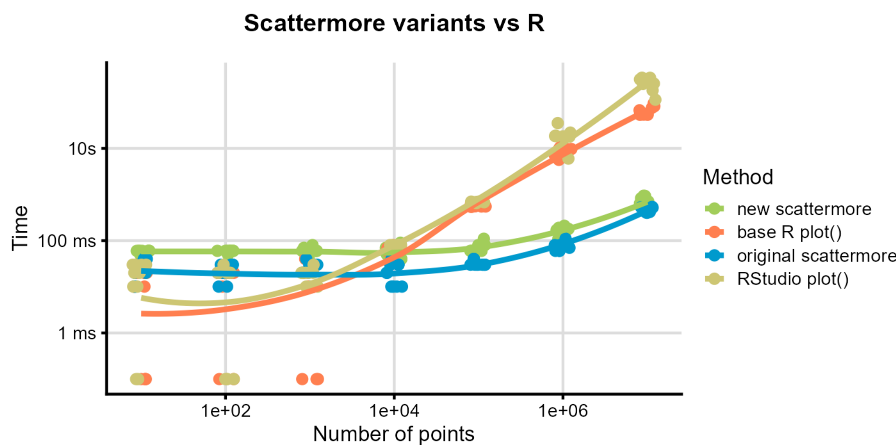


Figure 3.1 Comparison of the speed difference when plotting points among R, RStudio, original, and new `scattermore`'s version.

3.1 Performance of point scattering

The figure 3.1 clearly shows the contrast between plotting (without additional operations) with `scattermore` and R's default plot function. `Scattermore` is slightly slower with smaller data, likely because of its API overhead. However, when there are many points, the roles get reversed, and with millions of points, R is slower by tens of seconds.

The original version of `scattermore` is slightly faster than the new one. The reason for that, presumably, is the API overhead of the more complex processes and bitmap allocations in the newer implementation.

Except for that, the figure compares the usage of R's standard plot function in R alone and its IDE, RStudio[2]. RStudio plotted the images into its window interface, with many points being much slower than R plotting them and saving them as png files.

The way how to parallelize the points scattering mentioned in the section 2.4.4, when there are millions of them, does not help to improve the performance of `scattermore` significantly, and to employ more threads with less-sized data seems to be contra-productive (figure 3.2). Presumably, there is a significant overhead for generating and assigning threads, so generally, multithreading is not an optimal form to speed up the point scattering. However, when plotting millions of points, it can be helpful to consider it.

In the plot, we measure scattering points to the RGBWT matrix with 1024 rows and columns using a single color, but other variants with color for each point input and color palette would be similar.

[2]`https://www.rstudio.com/products/rstudio`

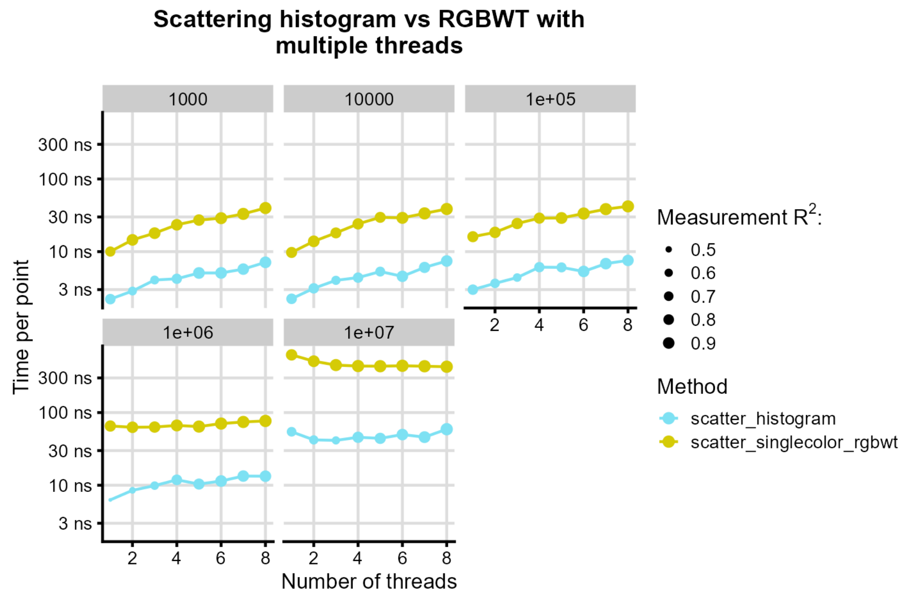


Figure 3.2 Comparison of the speed of the two C methods for scattering points to the histogram and the RGBWT matrix using a single color. The size of the bitmap is 1024x1024 pixels.

3.2 Throughput of format conversion

The conversion from histogram to RGBWT matrix implemented by the C method `histogram_to_rgbwt` belongs to the parts of `scattermore` that can be optimizable.

In the benchmark, we analyze the speed of the C `histogram_to_rgbwt` method with increasing bitmap size. Changing the data storage format results in different function speeds (figure 3.3). Naturally, as described in section 2.4.3, the SoA version with SIMD is faster than only the SoA version because of vector instructions. Next, it appears that the proximity of data in the AoS format, ensuring more positive accesses to a cache, is still worse than using the SoA format with SIMD.

3.3 Overhead of the R API

When benchmarking the conversion from histogram to RGWBT matrix with increasing bitmap size and considering the R part, the resulting algorithm is fastest when implemented with the SoA format, which is improved by SIMD (section 2.4.2). The vector instructions enable working with more data simultaneously without additional overhead, and the SoA format provides instant access to

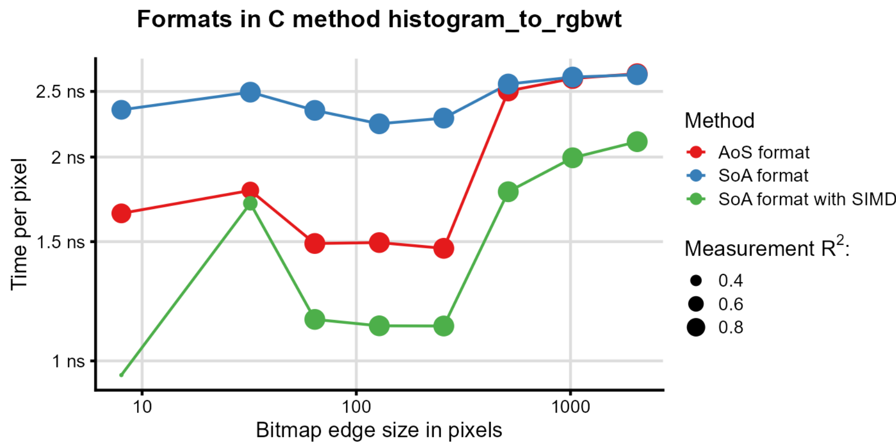


Figure 3.3 Comparison of the speed of the C method `histogram_to_rgbwt` using different storage formats: structure of arrays with or without SIMD and array of structures.

them (figure 3.4).

The plot shows that the conversion to column format, which is natural in R because it enables working with more data at once, is costly; after the run of the C `histogram_to_rgbwt` employing the AoS format, there must be conversion to the column format before ultimately leaving the R method, so that is why the SoA format is now faster than the AoS one in comparison to the previous case (figure 3.3).

We can also see that small bitmaps take the longest time to convert because they require the same R resources, but because of the bitmap size, no optimizations are employed.

3.4 Speedup of kernel application

In the kernel algorithm for either histogram or RGBWT matrix, there is a possibility to change the order of the for loops, so instead of the iterating mask over the image bitmap, the image bitmap iterates over the mask. The figures 3.5 and 3.6 illustrate the speed difference between the implementations concerning the increasing bitmap size for four kernel sizes. There are two cases: the mask iterates over the image bitmap, or the image bitmap iterates over the mask.

In both cases, the plots show that variant with the mask over the image bitmap is faster. This result is not surprising because when traversing the image bitmap over the mask, some pixels do not have to be in a cache anymore because the image size is much greater than the mask size.

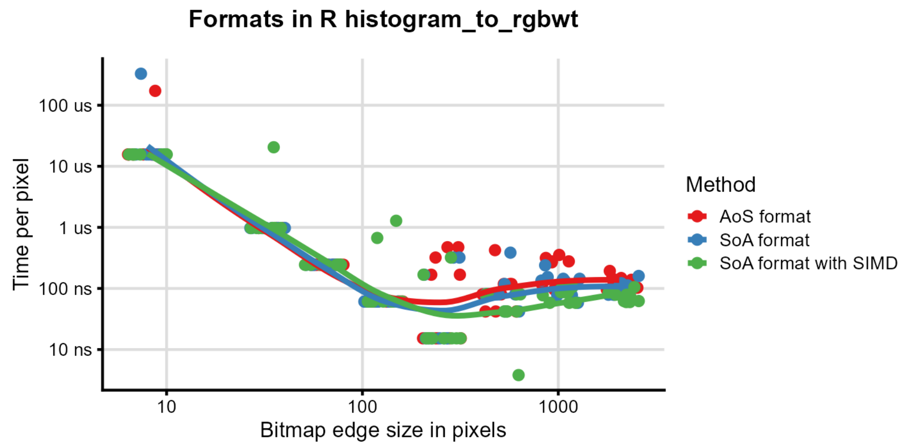


Figure 3.4 Comparison of the speed of the R method `histogram_to_rgbwt` which internally calls the C `histogram_to_rgbwt` method. The following storage formats are considered: structure of arrays with or without SIMD and array of structures.

When applying multithreading to the kernel algorithm, it is crucial to consider the distribution of the image bitmap to the threads. The goal is that the threads are 'close enough', as explained in section 2.4.4.

In the plot, we measured the histogram and RGBWT version of the algorithm with increasing threads and changing kernel size (figure 3.7). The results show that two threads bring the most enormous speedup, and with more threads, the benefit is less significant in both cases with every kernel size. This fact concludes that the more overhead of the threads overwhelms their advantage.

It is also important to underline the fact that the new version of `scattermore` starts to be faster than the original one when blurring one million points. The reason is that the original version blurs the points one by one, whereas, in the new one, only individual bitmap pixels are being blurred. In figure 3.8, we can also see that when employing multithreading, the speed does not depend much on the size of the kernel radius. The blurring operation is pretty regular so there is only a constant overhead; most of it, presumably, is the calling of more methods to perform the desired operations in the new API.

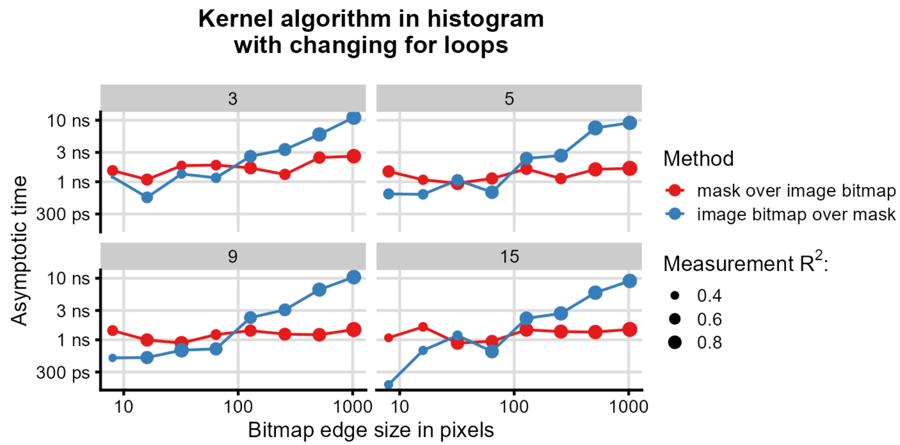


Figure 3.5 Comparison of the speed of the C `kernel_histogram` method in the relationship between the bitmap size and kernel size (gray blocks). This measurement has two variants depending on the order of the for loops.

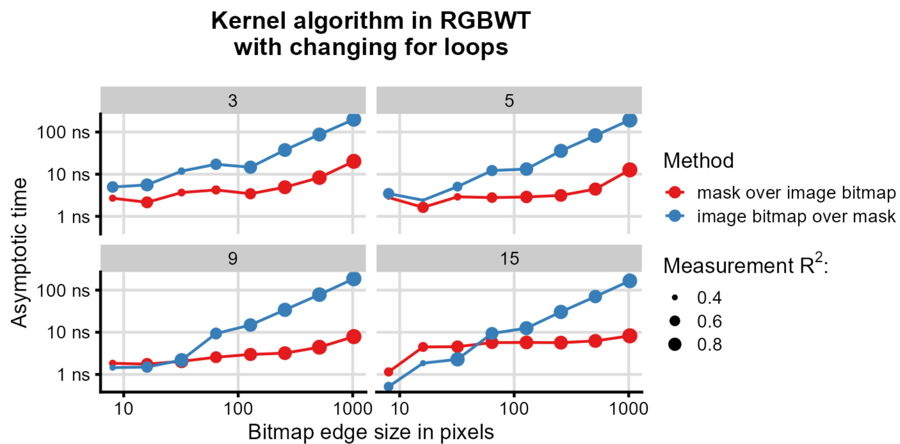


Figure 3.6 Comparison of the speed of the C `kernel_rgbwt` method in the relationship between the bitmap size and kernel size (gray blocks). This measurement has two variants depending on the order of the for loops.

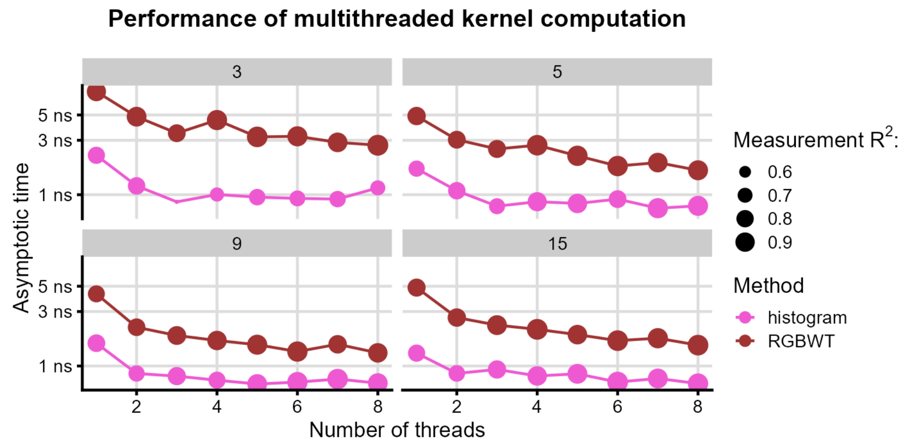


Figure 3.7 Comparison of the speed of the two C methods for applying kernel to the histogram and the RGBWT matrix in the relationship between the number of threads and kernel size (gray blocks). The size of the bitmap is 1024x1024 pixels.

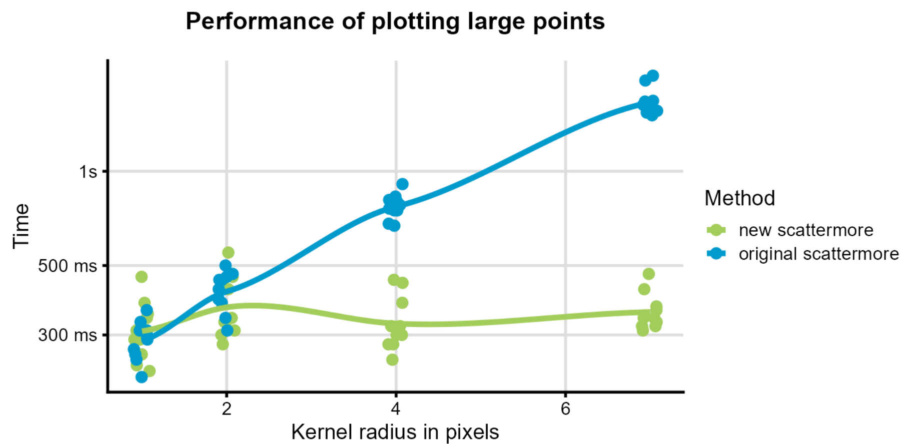


Figure 3.8 Comparison of the speed of the new and original version of `scattermore` with increasing kernel radius. The size of the bitmap is 512x512 pixels, and always a million points are plotted. In this case, we also employ multithreading in the new version of `scattermore`. In this benchmark, eight concurrent threads were utilized.

3.5 New use cases for high-definition scatterplots

Fast plotting of scatterplots is a useful tool for discovering new relationships in the data. To illustrate, we describe a surprising example that we found coincidentally during the preparation of the graphics for figure 3.12 (waterways).

Plotting the Czech waterway data points as a single long line (originally by mistake) clearly reveals a block structure in the points. We found that this visualization in figure 3.9 corresponds to the order in which the points are stored in the dataset file, where the waterways are apparently organized by groups separated by meridians.

Identifying such a dataset property would arguably be complicated unless we knew the property in advance. Conversely, simple visualization of the dataset has enabled us to generate a viable hypothesis about the data layout, which would be later easy to test.

Next, creating a histogram from the data and applying a kernel to it enables us to see the density of the data. Specifically, the image in figure 3.10 was created from data where each entry consisted of two points forming a line and then a Gaussian kernel with size ten. The black borders are contours that visualize the density of the data, and their location corresponds to the changing colors in the histogram palette so it is possible to use `scattermore` for density estimation. For more information about original data visualization and their plotting speed, please see section 3.6.

Density estimation is helpful in various other cases. One of them is shown by figure 3.11. The line plotting of 102 400 entries in scattermore takes a few tenths of a second compared to R, where the process is over two minutes long. Although we plot the lines with not full opacity in the first image, it is pretty unclear what the line/error distribution looks like in the red area. However, the situation can be rectified again when representing data as a histogram and then using a palette.

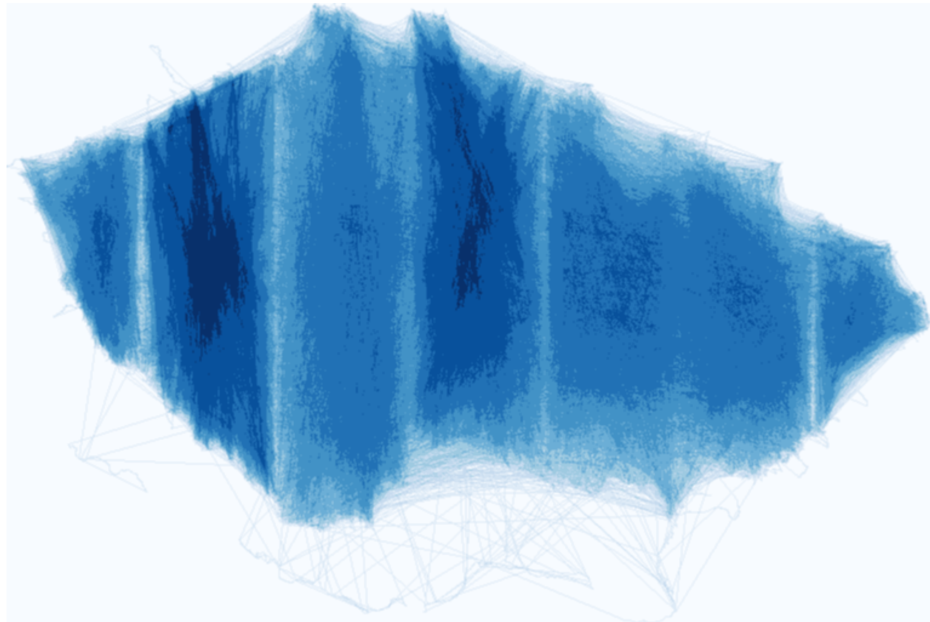


Figure 3.9 Depiction of unequal data distribution concerning waterways in the Czech Republic. Input data consist of 2 599 332 points used to create line entries for a histogram. The bitmap size is 768x512 pixels.

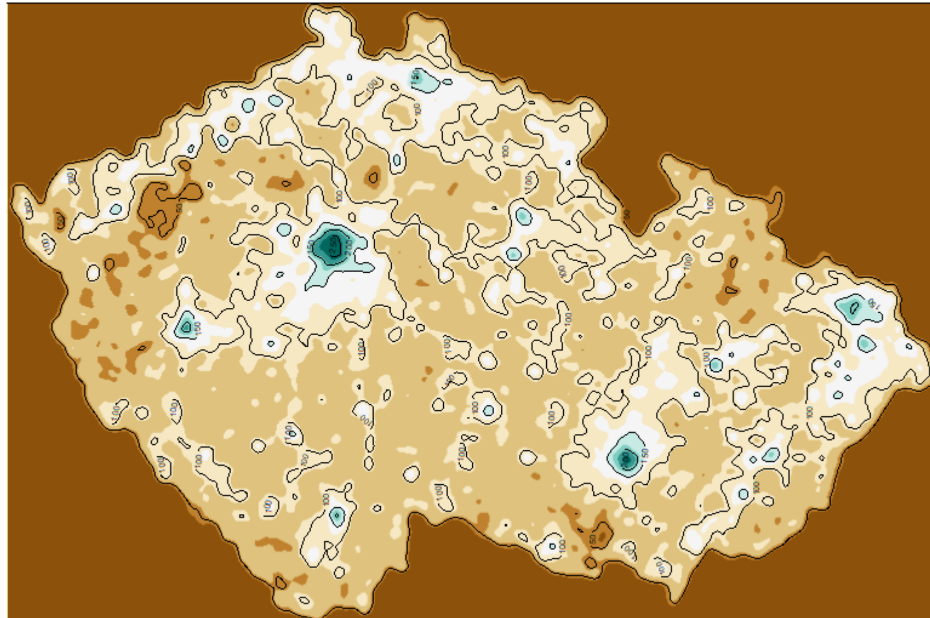
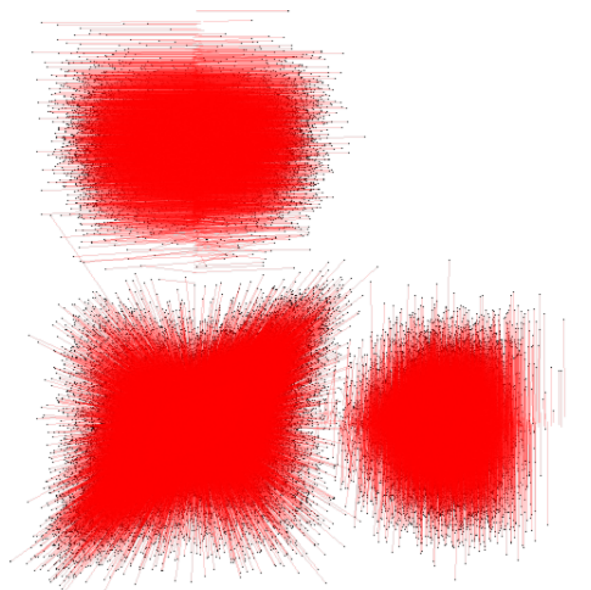
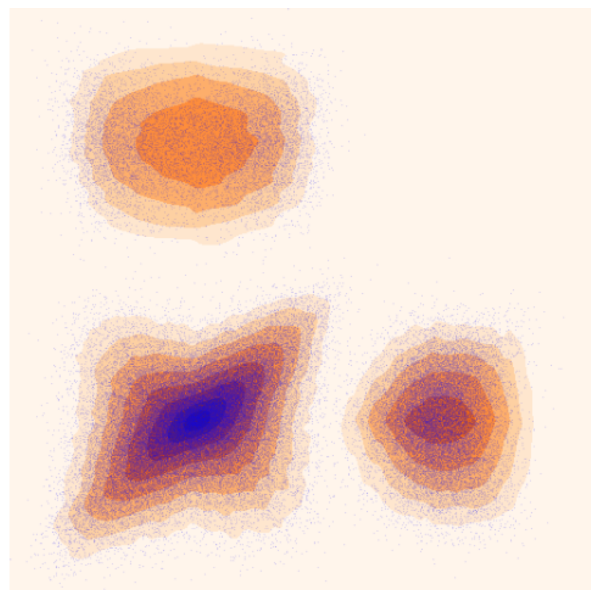


Figure 3.10 Density visualization of highways in the Czech Republic. Input data consist of 10 180 391. Then a palette colorized the data. The basis for the contours was also done by `scattermore`. The bitmap size is 768x512 pixels. For original visualization, see figure 3.12.



(a) Original image created by line scattering.



(b) Line density converted to a histogram, clearly communicating that the total error is most severe in the lower left cluster.

Figure 3.11 The original image visualizes the correct position of data using red lines. The data, black points, are not clearly visible because of many misalignments. A histogram displays the number of lines in space to show their density, and blue points depict the original data. Merging the points and the line histogram, we get the second image. Bitmap sizes are 512x512 pixels.

3.6 Practical usage of scattermore

This section shows examples of nice outputs generated by `scattermore`. As already mentioned in section 2.1, figure 3.13 demonstrates the usage of drawing lines when plotting Lorenz attractor. Plotting the image with over one and a half million line entries using a histogram took 0.27s $\pm$ 0.03s.

Again figure 3.12 depicts the usage of a histogram and lines together. Similarly to the Lorenz attractor, both images' visualization process is very fast. It took between 0.18s and 0.23s to plot waterways and between 0.39s and 0.45s to plot highways in the Czech Republic.

Figure 3.14 visualizes an example of trajectory inference [22] in single-cell data. In order to achieve this, the R package `tviblindi`[3] puts concepts from graph theory and algebraic topology to practice. It took approximately 0.15s $\pm$ 0.06s to draw the blurred (Gaussian kernel of size five) point version. For the line data, the time interval was similar.

In figure 3.15, the importance of blurring the image by an image kernel is visible. After creating a histogram and applying a palette, the results in both cases can be pretty unclear. Nevertheless, after applying a kernel of size five, it is easier to observe the data density. Plotting the images without kernel took 0.13s $\pm$ 0.04s, and with applying the kernel, on average, it was about 0.16s $\pm$ 0.05s.

We demonstrate the use of `scattermore` on very large datasets. Figure 3.16 shows visualizations of other attractor kinds. With `scattermore`, the plotting process is relatively fast compared to the default R function. Specifically, it took only between 0.75s and 1.05s to scatter 10 million points shown in figure 3.16 whereas the process takes over 100 seconds with employing the base R plot function.

To illustrate realistic requirements on plotting speed, figure 3.17 shows an example use of scatterplots for displaying results of data analysis from flow cytometry. The scatterplot shows several types of cells organized into visually distinguishable clusters in 2D by an embedding algorithm (the original dataset has more than 30 dimensions). Importantly, scientists often need to replot the dataset with a different coloring scheme to highlight features of interest. That easily becomes a bottleneck of the data analysis — in the original study that only utilized base R graphics [23], the plotting took more than ten minutes per scatterplot.

Detailed descriptions of the image generation parameters are available in the captions of the individual figures. The examples mentioned take much longer to plot without `scattermore`; sometimes, the whole process can last for two minutes. For repetitive replotting, this time interval is too long for practical use.

[3] `https://github.com/stuchly/tviblindi`

(a) Highways.

(b) Waterways.

Figure 3.12 Depictions of highways and waterways in the Czech Republic. Input data consist of 10 180 391 and 2 599 332 line entries, respectively. Both images used a palette to colorize the data. The bitmap sizes are 768x512 pixels[5].

The scripts for image visualizations from this section and from section 3.5 are available in the GitHub repository[4].

Figure	Objects	Type	Plotting time
figure 3.12	10,180,391	lines	0.39s – 0.45s
	2,599,332	lines	0.18s – 0.23s
figure 3.13	1,666,667	lines	0.24s – 0.30s
figure 3.14	99,948	points	0.09s – 0.21s
	280,103	lines	0.11s – 0.22s
figure 3.15	999,778	points	0.11s – 0.22s
	499,470	points	0.12s – 0.20s
figure 3.16	10,000,000	points	0.75s – 1.05s

Table 3.1 Summarized results for some figures plotted with `scattermore`. Due to the number of components in the complete plotting pipeline (including the R graphics backend), the plotting times are hard to estimate precisely; we thus report the expectable range of total plot rendering times.

[4] `https://github.com/teri934/scattermore-examples`
[5] `http://download.geofabrik.de/europe/czech-republic.html`
[6] `https://holoviews.org/gallery/demos/bokeh/lorenz_attractor_example.html`
[7] `https://3d.si.edu`
[8] `https://examples.pyviz.org/attractors/attractors.html`

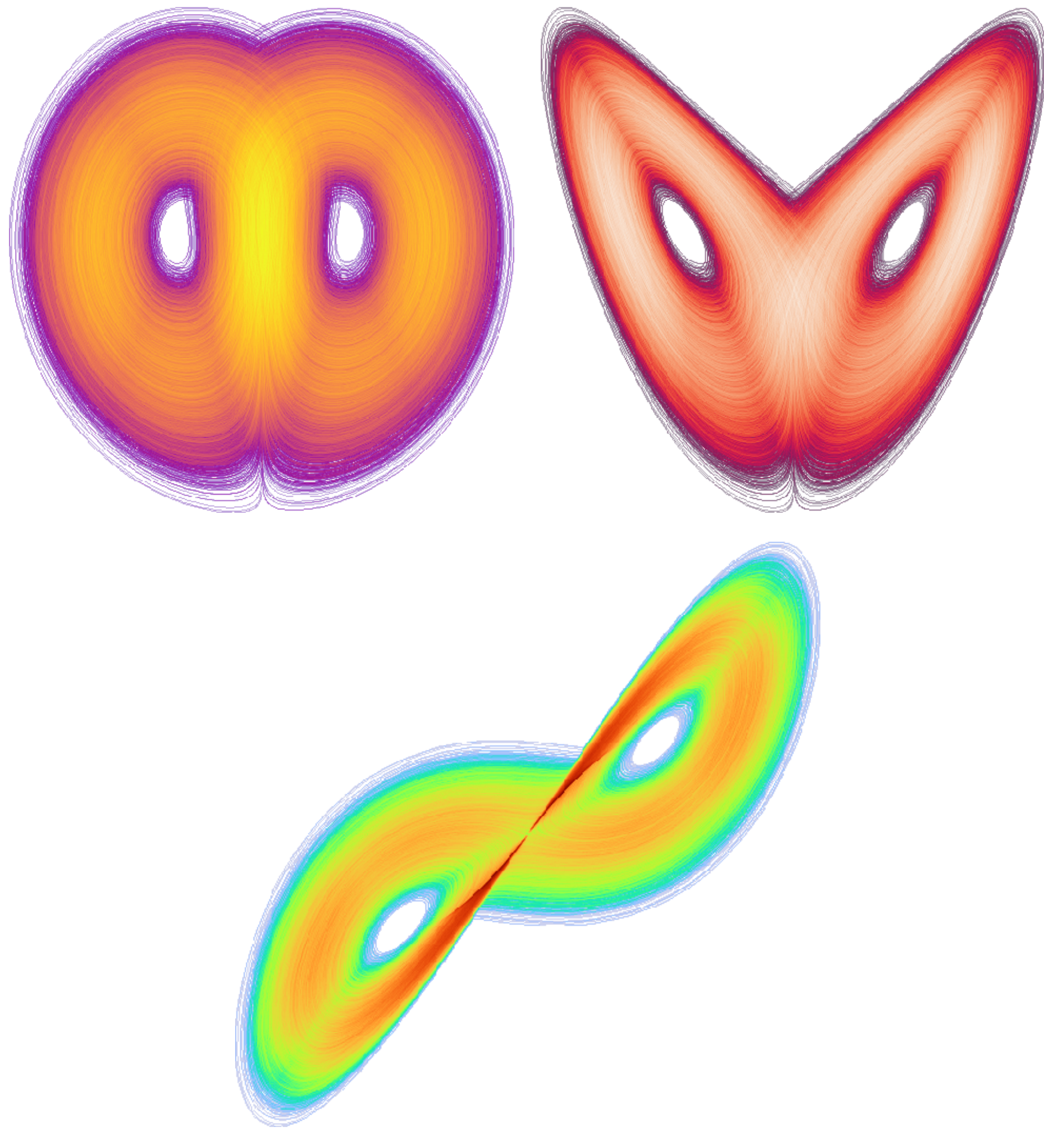
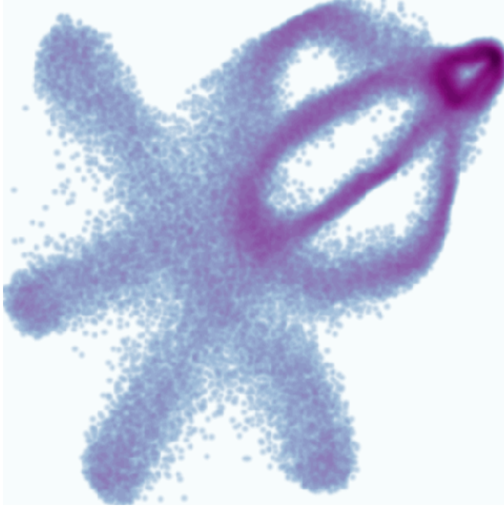
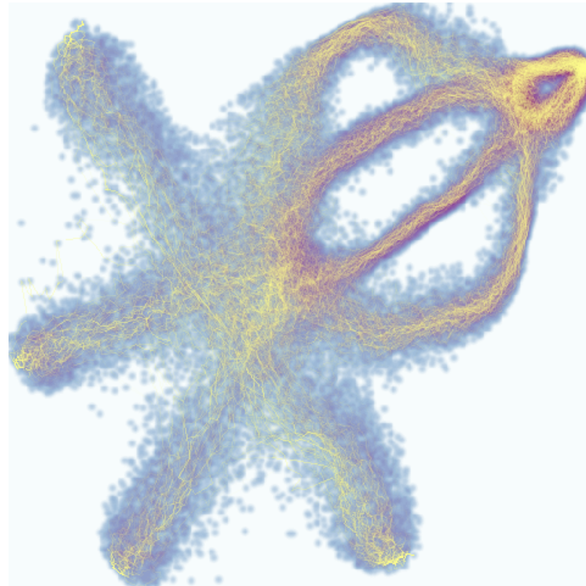


Figure 3.13 Lorenz attractor example from different angles. Histograms are created using line drawing without color input and then applying a palette. Input data include 1 666 667 line entries, each consisting of two points, and the bitmap size is 512x512 pixels. Inspired by an available example[6].



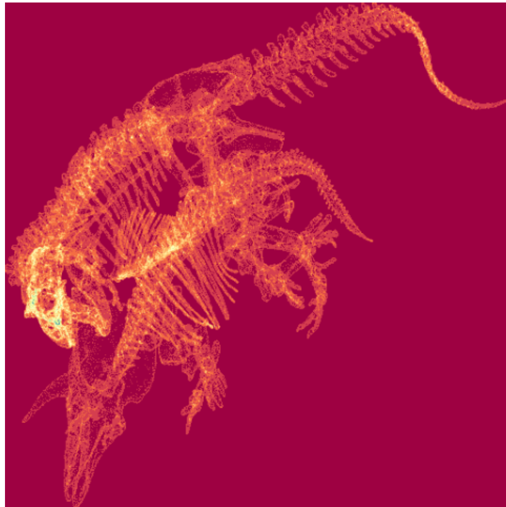
(a) Blurred single-cell point data to estimate density.

(b) Trajectory inference using line plotting.

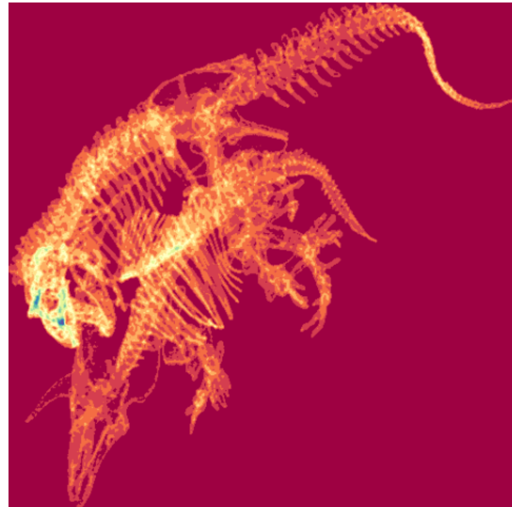


(c) Blend of the previous two.

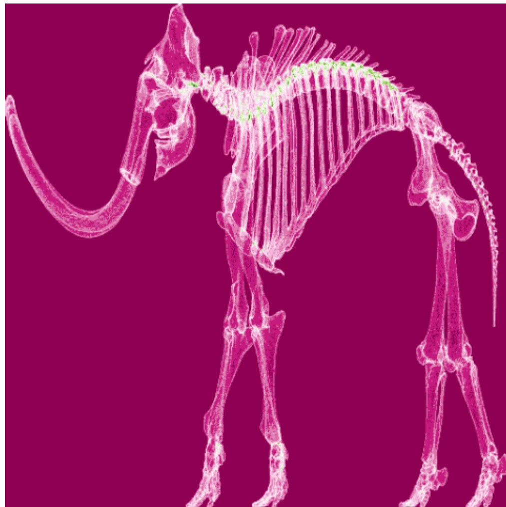
Figure 3.14 Visualizing projection of synthetic cellular-development pathway data using package `tviblindi`[3.6]. Input data consist of 99 948 point and 280 103 line entries, respectively. Both images used a palette to colorize the data. The bitmap sizes are 512x512 pixels.



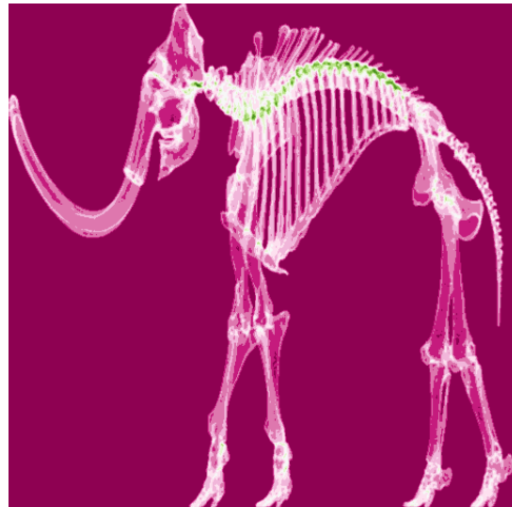
(a) T-Rex skeleton.



(b) Blurred T-Rex skeleton.

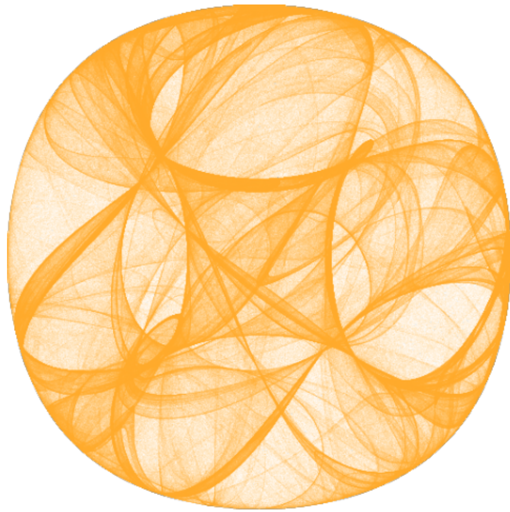


(c) Mammoth skeleton.

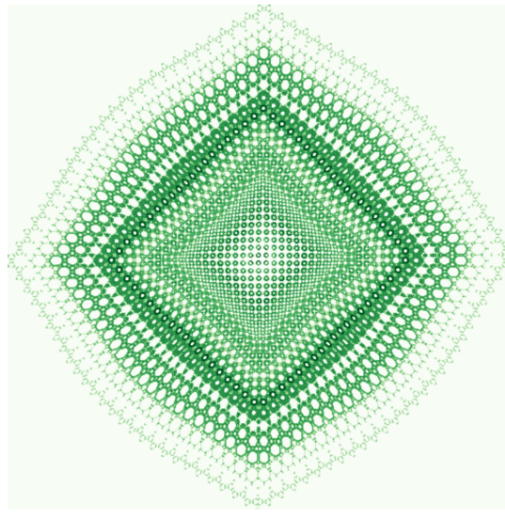


(d) Blurred mammoth skeleton.

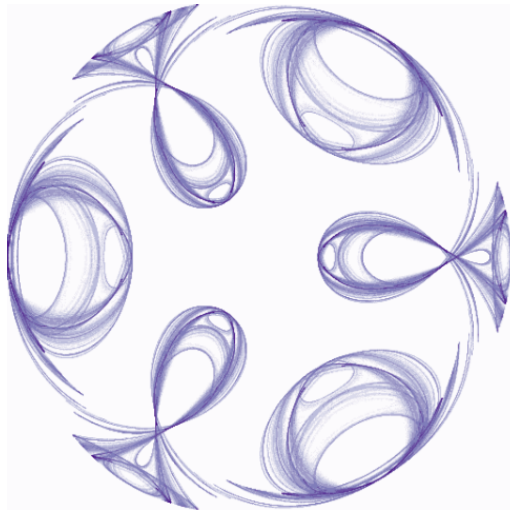
Figure 3.15 Images created using points without color input and then applying a palette and a circle kernel to blur them. Data about mammoth and T-Rex skeletons have 999 778 and 499 470 points, respectively. The bitmap sizes are 512x512 pixels[7].



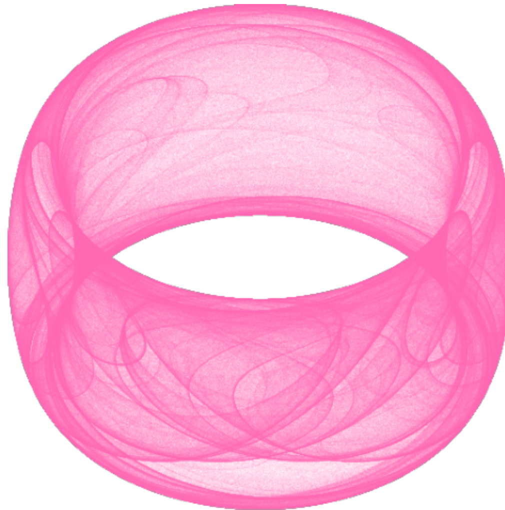
(a) A Clifford attractor.



(b) A Hopalong attractor.



(c) A Symmetric icon attractor.



(d) A Svensson attractor.

Figure 3.16 Created using 10 000 000 points with color input for each attractor. The bitmap sizes are 512x512 pixels. Parameters for individual attractors:
(a) 0, 0, -1.3, -1.3, -1.8, -1.9
(b) 0, 0, 7.8, 0.13, 8.15
(c) 0.01, 0.01, 10.0, -12.0, 1.0, 0.0, -2.195, 3
(d) 0, 0, 1.4, 1.56, 1.4, -6.56.
Inspired by available examples and algorithms about attractors[8].

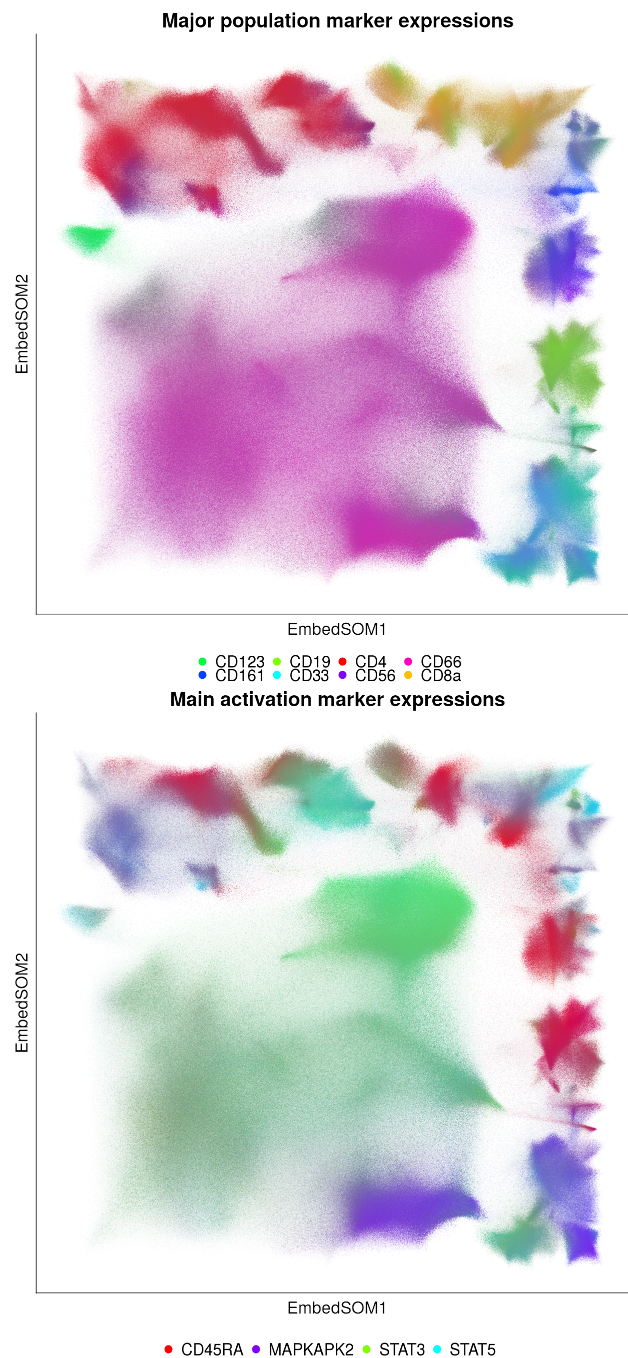


Figure 3.17 Example use of scatterplots to display high-volume data from cytometry experiments. The scatterplot shows a 2D embedding of the multidimensional dataset that describes the contents of peripheral blood in pregnant women; each of the 24 million points corresponds to a single blood cell. Image provided by Kratochvíl et al. [23].

Conclusion

In this thesis, we have described a redesigned version of the R package `scattermore` for fast rendering of scatterplots, which vastly improves the range of supported use cases. As the main result, we have created a highly flexible API that allows describing and running many new useful operations related to the rendering of scatterplots.

The version of `scattermore` produced by this thesis is at this point merged into the official repository of `scattermore`[9] and is available to the users. The main new features available after the redesign are as follows:

- New RGBWT pixel format described in section 2.1.1 solves the problem of overplotting and allows order-independent alpha-blending, which can help when combining many scatterplots.

- Layered API for scatterplotting enables the users to mix density-based operations with alpha-blending operations in a single plotting pipeline. Examples of applications can be found in section 2.2.

- Application of 2D image kernels benefits many common plotting tasks, such as faster rendering of large points in scatterplots and computation of smoothed histograms for contour drawing (section 2.2.1).

- Drawing of more complex primitives, mostly custom patterns (figure 2.4) and lines (section 2.1), which improves the rendering of primarily line-based data, such as Lorenz attractor visualizations or real-world maps (section 3.6).

Furthermore, we implemented all computationally difficult operations in C language, which is transparently called from R enabling efficient execution of most of the rasterization procedure. As a result, `scattermore` can be over several hundred times faster when visualizing millions of points compared to the R standard plotting function. Section 3.4 details the speedup compared to the original version in several selected specific cases.

[9] `https://github.com/teri934/scattermore-thesis`

The other goal of the thesis was to analyze and apply additional performance optimizations of some operations in `scattermore`. We determined that parallelization of the base point-scattering operations is actually contra-productive on typical off-the-shelf CPUs, mainly because the operation is memory-bound (section 3.1). Nevertheless, we improved the speed of several data conversion routines by applying data structure and SIMD instruction optimizations (section 3.2). Most importantly, we parallelized the kernel application (the most computationally intensive operation) and found that when using the correct data layout and access patterns, the result vastly outperforms the methodology used in the original `scattermore` (figure 3.8), and scales sufficiently well to properly utilize the full potential of the commonly available CPUs properly.

To summarize, the new version of `scattermore` is a high-performance R package for plotting scatterplots that offers a broad range of customizable operations using a comprehensive API. We believe that it will provide a good, scalable, and sustainable base for plotting huge, detailed scatterplots by the scientific packages implemented in R.

Future work

Although we observed that the expense of data copying using the R .C function is not large, it still poses some overhead. The future version of `scattermore` could contain routines that work directly with 32-bit float data in a way similar to R package `tibble` [24], avoiding the frequent conversions to 64-bit floats used in R. This way, most of the unnecessary data conversion might be eliminated entirely.

The SIMD version of conversion between the histogram and RGBWT data is faster than the version without vector instructions, but it is not optimal in all cases and platforms. Although this is not currently an issue, future use cases dealing with massive scatterplot processing might require different trade-offs and thus substantiate additional future optimization. Similarly, since many operations easily become memory-bound on CPU architectures, it would be interesting to port these to GPUs, which offer much-improved memory data transfer rates.

As a natural extension, users might demand additional features such as plotting Bezier curves and, more importantly, filled polygons. Although we did not find a good use case for plotting exuberant amounts of small and overlapping polygons, fast algorithms exist for optimized polygon plotting that can be implemented. For example, GPUs typically employ tiling [25, 26] to improve the cache efficiency of the rendering vastly. On CPU hardware, we expect that aggregating the updates of large filled regions in a more complex data structure (such as a quadtree that structures the bitmap into hierarchical regions that may aggregate the updates [27]) might provide optimal performance. For drawing polygons,

various other algorithms are already used by the current plotting packages for similar purposes; for example, the aforementioned Datashader uses trimesh-based rasterization [28].

Bibliography

[1] Ross Ihaka and Robert Gentleman. "R: a language for data analysis and graphics". In: *Journal of computational and graphical statistics* 5.3 (1996), pp. 299–314.

[2] Yuhan Hao et al. "Integrated analysis of multimodal single-cell data". In: *Cell* 184.13 (2021), pp. 3573–3587.

[3] 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.

[4] Hadley Wickham. "ggplot2". In: *Wiley interdisciplinary reviews: computational statistics* 3.2 (2011), pp. 180–185.

[5] Miroslav Kratochvíl et al. "ShinySOM: graphical SOM-based analysis of single-cell cytometry data". In: *Bioinformatics* 36.10 (Feb. 2020), pp. 3288–3289. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/btaa091`. eprint: `https://academic.oup.com/bioinformatics/article-pdf/36/10/3288/33204463/btaa091.pdf`. URL: `https://doi.org/10.1093/bioinformatics/btaa091`.

[6] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[7] T. Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. DOI: `10.1109/5.58325`.

[8] Miroslav Kratochvíl et al. "GigaSOM. jl: High-performance clustering and visualization of huge cytometry datasets". In: *GigaScience* 9.11 (2020), giaa127.

[9] Carl T. Bergstrom and Jevin D. West. "Why scatter plots suggest causality, and what we can do about it". In: *CoRR* abs/1809.09328 (2018). arXiv: `1809.09328`. URL: `http://arxiv.org/abs/1809.09328`.

[10] Adrian Mayorga and Michael Gleicher. "Splatterplots: Overcoming Overdraw in Scatter Plots". In: *IEEE Transactions on Visualization and Computer Graphics* 19.9 (2013), pp. 1526–1538. DOI: 10.1109/TVCG.2013.65.

[11] Saskia Freytag and Ryan Lister. "schex avoids overplotting for large single-cell RNA-sequencing datasets". In: *Bioinformatics* 36.7 (Dec. 2019), pp. 2291–2292. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz907. eprint: https://academic.oup.com/bioinformatics/article-pdf/36/7/2291/33027528/btz907.pdf. URL: https://doi.org/10.1093/bioinformatics/btz907.

[12] Raman Maini and Himanshu Aggarwal. "Study and comparison of various image edge detection techniques". In: *International journal of image processing (IJIP)* 3.1 (2009), pp. 1–11.

[13] Ruchika Chandel and Gaurav Gupta. "Image filtering algorithms and techniques: A review". In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3.10 (2013).

[14] Peter Kovesi. "Fast Almost-Gaussian Filtering". In: *2010 International Conference on Digital Image Computing: Techniques and Applications*. 2010, pp. 121–125. DOI: 10.1109/DICTA.2010.30.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] D Rufilanchas. "On the origin of Karl Pearson's term" histogram". In: *Revista Estadística Española* 59.192 (2017), pp. 29–35.

[17] Gabriel Thomas, Daniel Flores-Tapia, and Stephen Pistorius. "Histogram Specification: A Fast and Flexible Method to Process Digital Images". In: *IEEE Transactions on Instrumentation and Measurement* 60.5 (2011), pp. 1565–1578. DOI: 10.1109/TIM.2010.2089110.

[18] Chao Wang and Zhongfu Ye. "Brightness preserving histogram equalization with maximum entropy: a variational perspective". In: *IEEE Transactions on Consumer Electronics* 51.4 (2005), pp. 1326–1334. DOI: 10.1109/TCE.2005.1561863.

[19] Yishen Chen et al. "VeGen: A Vectorizer Generator for SIMD and Beyond". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, 902–914. ISBN: 9781450383172. DOI: 10.1145/3445814.3446692. URL: https://doi.org/10.1145/3445814.3446692.

[20] Henk AL Kiers. "Weighted least squares fitting using ordinary least squares algorithms". In: *Psychometrika* 62 (1997), pp. 251–266.

[21] Rasmus Bro, Nicholaos D Sidiropoulos, and Age K Smilde. "Maximum likelihood fitting using ordinary least squares algorithms". In: *Journal of Chemometrics: A Journal of the Chemometrics Society* 16.8-10 (2002), pp. 387–400.

[22] Wouter Saelens et al. "A comparison of single-cell trajectory inference methods". In: *Nature biotechnology* 37.5 (2019), pp. 547–554.

[23] M Kratochvıl et al. "SOM-based embedding improves efficiency of high-dimensional cytometry data analysis". In: *biorxiv* (2019).

[24] Hadley Wickham et al. "Welcome to the Tidyverse". In: *Journal of open source software* 4.43 (2019), p. 1686.

[25] Ola Olsson and Ulf Assarsson. "Tiled shading". In: *Journal of Graphics, GPU, and Game Tools* 15.4 (2011), pp. 235–251.

[26] Chang Xu, Steven R Kirk, and Samantha Jenkins. "Tiling for performance tuning on different models of GPUs". In: *2009 Second International Symposium on Information Science and Engineering*. IEEE. 2009, pp. 500–504.

[27] Hanan Samet. "Algorithms for the Conversion of Quadtrees to Rasters". In: (1982).

[28] Juan Pineda. "A parallel algorithm for polygon rasterization". In: *Proceedings of the 15th annual conference on Computer graphics and interactive techniques.* 1988, pp. 17–20.

Appendix A

Using Scattermore

To be able to use `scattermore`, you need to install R software first[1]. `Scattermore` was developed using R version 4.3.0. You can additionally install RStudio[2] (R's GUI).

Installing and running `Scattermore`

You may install the official CRAN version of `scattermore` by running the following in the R console (until the release of the new `scattermore` version, this will install the original one with limited options):

```
install.packages("scattermore", repos = "http://cran.r-project.org",
build_vignettes = TRUE)
```

Another way is to install the development version from GitHub:

```
devtools::install_github("teri934/scattermore-thesis", build_vignettes = TRUE)
```

As an alternative, please see the enclosed zipped folder where the `scattermore` folder is located. After unzipping the compressed folder, it is possible to run the following in the local `scattermore` folder with its source files:

```
devtools::install("../scattermore", build_vignettes = TRUE)
```

To use the `scattermore` package, you need to type `library("scattermore")` in the R console.

Now you should be able to use `scattermore` in R. You can use the following code7 and the code from the thesis1 to try it. Or you can have a look at the vignettes by writing `browseVignettes("scattermore")` in the R console:

[1] `https://cran.r-project.org`
[2] `https://www.rstudio.com`

Listing 7 Try scattermore with this code.

```
library(scattermore)
library(magrittr)

# scatter points into a histogram

pts <- cbind(rnorm(1e5), rnorm(1e5))
pts %>% scatter_points_histogram %>%
{. ->> hst} %>% image

# blur them with Gaussian kernel

hst %>%
apply_kernel_histogram(filter = "gauss", radius = 4) %>%
{. ->> gauss_blurred_hst} %>% image

# convert and plot them

gauss_blurred_hst %>% histogram_to_rgbwt %>%
rgbwt_to_rgba_int %>% rgba_int_to_raster %>%
plot(interpolate = F)

# scatter points

pts %>%
scatter_points_rgbwt(RGBA = c(0,128,192,50)) %>%
{. ->> rgbwt1}
pts %>%
scatter_points_rgbwt(RGBA = c(192,128,0,50)) %>%
{. ->> rgbwt2}

# merge two RGBWT matrices

merged <- merge_rgbwt(list(rgbwt1, rgbwt2))
merged %>% rgbwt_to_rgba_int %>%
rgba_int_to_raster %>% plot(interpolate = F)

# blend two RGBA float matrices

frgba1 <- rgbwt_to_rgba_float(rgbwt1)
frgba2 <- rgbwt_to_rgba_float(rgbwt2)
blended <- blend_rgba_float(list(frgba1, frgba2))
blended %>% rgba_float_to_rgba_int %>%
rgba_int_to_raster %>% plot(interpolate = F)
```