FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Visualization of 3-dimensional solar surface data
Student:	Vojtěch Tomas
Supervisor:	Ing. Radek Richtr, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

The thesis goal is to propose a method of suitable visualization of the solar surface and to realize it using the chosen technology. Visualization should be physically correct and user-friendly. The visualization must also offer appropriate ways of interaction.

1) Perform a search for existing ways to display similar 3D vector data.
2) Design a way of visualizing 3D vector solar surface data.
a. Focus on physical accuracy, clarity, performance, and portability.
b. The user must be able to: choose how to approximate values between specified values and change the view, scale, make a 3D data slice or export data to the 3D monitor.
3) Analyze the technological possibilities of displaying solar surface data.
4) Create the data visualizing prototype according to the proposed method.
5) Test the created prototype appropriately.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 24, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Visualization of 3-dimensional solar surface data

Vojtěch Tomas

Department of Software Engineering
Supervisor: Ing. Radek Richtr, Ph.D.

May 9, 2019

Acknowledgements

I want to express my thanks to my supervisor Ing. Radek Richter, Ph.D. for his guidance and advice throughout the entire two years it took to complete this thesis. Additional thank you belongs to doc. Mgr. Michal Švanda, Ph.D. for his valuable insight. I would also like to thank my family for supporting me throughout my studies.

Declaration

In Prague on May 9, 2019 …………………

Citation of this thesis

Tomas, Vojtěch. *Visualization of 3-dimensional solar surface data.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Práce se zabývá možnostmi vizualizace vícerozměrných dat. Cílem práce je vybrat a implementovat vhodné vizualizační metody. V práci jsou dále rozebrány výpočetní metody numerické integrace, interpolace a konstrukce geometrie, které se pojí s vizualizačními metodami. Vybrané metody a algoritmy jsou implementovány ve formě prototypu, jehož design se zaměřuje na fyzikální přesnost zobrazení, srozumitelnost, možnosti aproximace hodnot, výkon, ale i portabilitu. Prototyp je implementován pomocí webových technologií (JavaScript a WebGL), výpočetní část je implementována v jazyce Python. Prototyp umožňuje zobrazení na LCD stěně v prostředí SAGE2 a na 3D monitoru. Součástí prototypu je implementace Python balíčku umožňující rychlou integraci a interpolaci. Implementovaný prototyp byl uživatelsky otestován. Testování obsahuje také porovnání rychlosti implementovaných matematických metod s implementací totožných metod v balíčku SciPy.

Klíčová slova vektorové pole, vícerozměrná data, vizualizace dat, interpolační metody, srozumitelnost vizualizace

Abstract

The thesis deals with the possibilities of multidimensional data visualization. The aim is to select and implement a suitable vector field visualization methods. The thesis includes an analysis of numerical integration methods, interpolation methods, and geometry construction algorithms. Based on the analysis, the thesis presents the design of a prototype visualization application. The design focuses on physical accuracy and clarity of the visualization, value approximation, performance, and also portability. The prototype is implemented using web technologies (JavaScript and WebGL) and works also in SAGE2 environment; the computation components of the application are implemented in Python. The thesis includes the evaluation of the user tests as well as a performance comparison of the implemented mathematical methods with the corresponding methods implemented in SciPy module.

Keywords vector field, multidimensional data, data visualization, interpolation methods, clarity of visualization

Contents

List of Figures

List of Tables

Introduction

As a result of recent technological development, people are surrounded by an ever-growing amount of data. Some of the data is created during scientific simulations and experiments. The amount of infromation often exceeds a comprehensible quantity; however, the most profound understanding is still acquired by seeing and observing. This led to the development of scientific visualization as an independent scientific discipline.

In this thesis, the visualized model is the outer envelope of Sun — it is a plasma model positioned near the solar surface. The visualized data represents a vector field of speed values; the dataset is a product of numerical simulations and real observation analysis.

The goal of the visualization is to provide insight and allow the user to create a mental image of the flow inside the field. Instead of large tables, the graphical representation is capable of providing clear information and revealing the hidden structures and processes.

There are several visualization tools available for scientific or information visualization. Regardless of the primary focus of the tool, the program has to respect a universal balance between physical accuracy, usability and the amount of displayed data. The common practice is to prioritaze one or two of these attributes over the others. The main shortcoming of a tool providing highly accurate visualization can be an incomprehensible graphical presentation and a lack of user interface. The aesthetic side of the visualization cannot be omitted since it undoubtedly contributes to the overall impression.

Visualization is more than just a pure data representation. There are additional factors influencing the quality of the output visualization. The key to producing a satisfiable result is to find the right balance. A motivation for this thesis is the lack of adjustability of the existing programs. The User interface (UI) is often too complicated, and it is not manageable to scale the application onto different devices. The goal is to develop a prototype correcting these shortcomings.

Goals

The goal is to develop a prototype allowing to visualize a vector field. The design of the prototype is based on a previous analysis of frequently used visualization techniques and related mathematical methods. The implemented prototype provides a way to interpolate between dataset points and use more than one visualization method. The used technology allows for portability between the leading OS platforms.

Visualized Properties The vectors of the vector field have, among other properties, the magnitude and direction. An additional feature is the topological continuity of the vector field. The prototype has to provide a way to observe these properties.

Visualization Methods The analysis covers several visualization techniques. The visualization techniques often require the use of additional mathematical methods. The goal is to implement suitable visualization and numerical methods efficiently.

Selecting Area of Interest The visualization tool has to provide a way to choose a specific region in the complete dataset. The expected supported areas of interest are 3D and 2D slices of space.

Used Technology The application has to be portable and scalable for different environments. Besides the standard desktop, the app has to support the Scalable Amplified Group Environment (SAGE2), enabling the projection on big grids of screens. Additionally, the application has to support the rendering of the content on the 3D monitor.

Chapter 1

Multidimensional Data

This chapter introduces the concept of a vector field and data representation. The goal of this chapter is to provide an overview of specific topics, for further detailed information, please see referenced materials. The first section describes vector fields, followed by a part presenting means of interpolation. The third section introduces methods for estimating partial derivations, and section four closes this chapter with an overview of numerical methods for ordinary differential equations.

1.1 Vector Fields

An excellent example of a vector field can be a flowing fluid. The fluid flows at varying speed and in different directions. According to [1], the vector field can be modeled as a continuous function V associating a vector $V(x, y, z)$ with each point (x, y, z) in a certian region. Precisely, a vector field can be defined as

$$V\left(x,y,z\right) = V_x\left(x,y,z\right)\mathbf{i} + V_y\left(x,y,z\right)\mathbf{j} + V_z\left(x,y,z\right)\mathbf{k}, \tag{1.1}$$

$\mathbf{i}, \mathbf{j}, \mathbf{k}$ are unit vectors parallel with each axis. It is possible to obtain a vector for any point in space from the function V. Further information about the vector field can be obtained from operators such as divergence and rotation. In this thesis all further calculations are performed in the cartesian coordinate system unless stated otherwise. For clarity, all spatial points will be further marked as $\mathbf{p} = (x, y, z)$ and function values of a vector field function as $\mathbf{v} = V\left(\mathbf{p}\right)$.

Divergence The divergence of a vector field V is a scalar field assigning each point a quantity of vector field's source. The value of the vector field's divergence describes the flow of the vector field.

$$div\, V = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} + \frac{\partial V_z}{\partial z} \tag{1.2}$$

Areas with positive divergence are called sources; negative areas are called sinks. Following the example with fluid flow, positive divergence would mean more fluid enters a given area than leaves; it spreads outwards. Negative divergence shows that fluid keeps accumulating in a specific area; more leaves than enters. Zero divergence means no fluid spreads or gets sucked, for further information see [1, 2].

Curl Curl (also called *vorticity* or *rotation*) represents the size and direction of rotation in each point of the vector field. The curl of a vector field is also a vector field. The operator curl is defined as

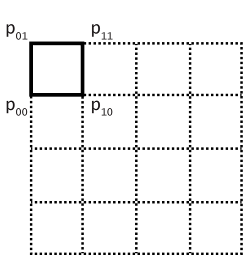
$$rot\ V = \left(\frac{\partial V_z}{\partial y} - \frac{\partial V_y}{\partial z}, \frac{\partial V_x}{\partial z} - \frac{\partial V_z}{\partial x}, \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y}\right). \tag{1.3}$$

According to [2], vector $rot\ V(\mathbf{a})$ determines the axis of the vector field's rotation in the area around point $\mathbf{a}$. Size of $rot\ V(\mathbf{a})$ is equal double the rotation speed (in angular rate).

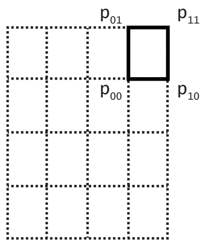
1.2 Sampling and Interpolation

Various scientific experiments often produce a dataset which does not contain complete information about the vector field, such as a symbolical representation of the analytical function. More commonly, those datasets contain values sampled in certain spatial points. In those cases, it is required to interpolate the values between sampled points and reconstruct a particular part of the original vector field function. There are three interpolation methods presented in this section — nearest neighbor, linear (trilinear), and cubic (tricubic) interpolation.

Original sample points are usually organized into a structured or unstructured grid consisting of cells and nodes or vertices. Figures 11, 12 and 13 illustrate possible arrangements in two and three dimensions. This thesis focuses on interpolation inside regularly and uniformly structured grids. All equations in this section assume that the data is sampled along a regularly structured grid (all of the sample points are equally distant).

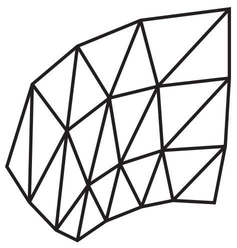


(a) Regular grid with square cells

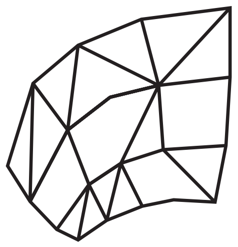


(b) Uniform grid with rectangular cells

Figure 11: Structured grids



(a) Regular grid with triangular cells



(b) Irregular grid

Figure 12: Unstructured grids

Nearest Neighbor Interpolation A nearest-neighbor method is the fastest interpolation method and also the least accurate from those presented here. As the title suggests, it uses the value of the nearest neighbor as the approximation. In three dimensions, when the points are distributed on a regular grid, the interpolation requires determining the coordinates of a local data cell consisting of 8 vertices and then finding the nearest one.

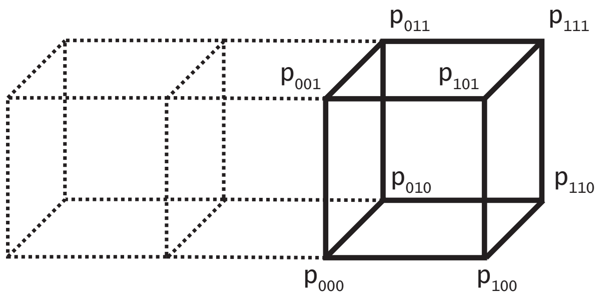


Figure 13: Regular structured grid with cube cells

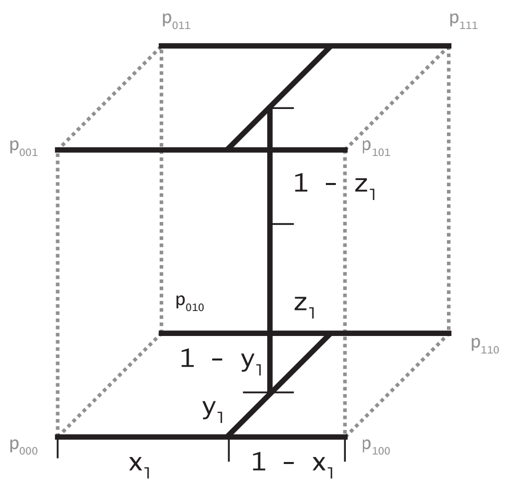


Figure 14: Illustration of trilinear interpolation

Linear interpolation A value of a function can be interpolated as a sum of values of the closest points weighted by the relative distances inside the local cell. Hence, the linear interpolation in one dimension can be written as

$$\mathbf{v}_x = \mathbf{v}_0 \left(1 - x\right) + \mathbf{v}_1 x, \tag{1.4}$$

where $V\left(\mathbf{p}_x\right) = \mathbf{v}_x$ and $\mathbf{p}_x$ lays inside an interval of known points $\mathbf{p}_0$ and $\mathbf{p}_1$. In three dimensions, it is possible to perform trilinear interpolation inside a local cell of 8 nearest points. Trilinear interpolation can also be understood as one linear interpolation between two bilinear interpolations; in other words, it is seven linear interpolations. Following expression is the general equation of trilinear interpolation in a matrix form

$$V\left(x, y, z\right) \approx \mathbf{QC}, \tag{1.5}$$

where $\mathbf{C}$ is a matrix obtained from known vectors (vertices of the local cell) and $\mathbf{Q}$ is a single-line matrix of coefficients calculated from distances inside the local cell [3].

$$\mathbf{C} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{v}_{000} \\ \mathbf{v}_{001} \\ \mathbf{v}_{010} \\ \mathbf{v}_{011} \\ \mathbf{v}_{100} \\ \mathbf{v}_{101} \\ \mathbf{v}_{110} \\ \mathbf{v}_{111} \end{pmatrix} \tag{1.6}$$

$$\mathbf{Q} = \begin{pmatrix} 1 & x_l & y_l & z_l & x_l y_l & y_l z_l & x_l z_l & x_l y_l z_l \end{pmatrix}$$

Cubic Interpolation Cubic interpolation approximates values using cubic polynomials. There are two possible strategies for cubic interpolation; first uses values and tangents in two surrounding points and the second utilizes the value of four surrounding points. The first method is closely related to Hermite cubics. If the tangents remain unknown, it is desirable to estimate them from four surrounding points, which leads to the second approach to cubic interpolation, closely related to Catmull-Rom splines. The Hermite spline is defined as

$$Q\left(t\right)=\begin{pmatrix}t^3 & t^2 & t & 1\end{pmatrix}\begin{pmatrix}2 & -2 & 1 & 1\\ -3 & 3 & -2 & -1\\ 0 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}\mathbf{v}_0\\ \mathbf{v}_1\\ \mathbf{t}_0\\ \mathbf{t}_1\end{pmatrix}. \tag{1.7}$$

Continuity of curves constructed by combining multiple Hermite splines is guaranteed by the identity of tangents in the endpoints, see [4]. If the tangents are unknown, it is possible to approximate them using finite differences. The interpolation then uses four values instead of two values and two tangents. The figure 15 illustrates the tricubic interpolation. The following paragraph demonstrates the relation between Hermite and Catmull-Rom splines, showing how the tangents are approximated.

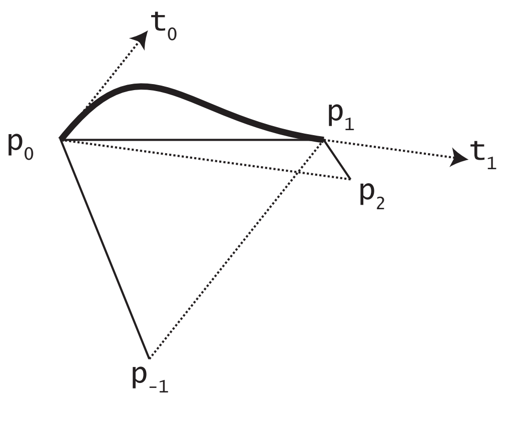


Figure 15: Illustration of cubic interpolation

It is possible to modify the formula (1.7) and substitute the tangents and values in the right-hand side vector with values of four surrounding points $\mathbf{v}_{-1}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$. Assuming the distance between each point is one, the tangents can be represented such as

$$\mathbf{t}_0=\frac{\mathbf{v}_1-\mathbf{v}_{-1}}{2}, \mathbf{t}_1=\frac{\mathbf{v}_2-\mathbf{v}_0}{2}. \tag{1.8}$$

It is possible to express coefficients of the polynomial $Q(t)$ in (1.7) as a system of equations and substitute the tangents with the expressions from (1.8).

$$\begin{aligned} a &= -\tfrac{1}{2}\mathbf{v}_{-1} + \tfrac{3}{2}\mathbf{v}_0 - \tfrac{3}{2}\mathbf{v}_1 + \tfrac{1}{2}\mathbf{v}_2 \\ b &= \quad \mathbf{v}_{-1} - \tfrac{5}{2}\mathbf{v}_0 + 2\mathbf{v}_1 - \tfrac{1}{2}\mathbf{v}_2 \\ c &= -\tfrac{1}{2}\mathbf{v}_{-1} + \tfrac{1}{2}\mathbf{v}_1 \\ d &= \quad \mathbf{v}_0 \end{aligned} \tag{1.9}$$

The system of equations (1.9) can be converted back into matrix form. The result of the transformation is the following expression also known as Catmull-Rom spline.

$$\mathbf{v}_x = 0.5 \begin{pmatrix} x^3 & x^2 & x & 1 \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_{-1} \\ \mathbf{v}_0 \\ \mathbf{v}_1 \\ \mathbf{v}_2 \end{pmatrix} \tag{1.10}$$

Boundary Effects There are two approaches toward interpolating between first two or last two elements of a list. According to [5], one option is to repeat the first and last value of the list, therefore $\mathbf{p}_{-1} = \mathbf{p}_0$ and $\mathbf{p}_1 = \mathbf{p}_2$. The other approach is to approximate missing values using a line fitting first two and last two points, thus $\mathbf{p}_{-1} = 2\mathbf{p}_0 - \mathbf{p}_1$ and $\mathbf{p}_2 = 2\mathbf{p}_1 - \mathbf{p}_0$.

Interpolation in 3D Interpolation in three dimensions utilizes values in points outside of the local cell (those values correspond with values $\mathbf{v}_{-1}$ a $\mathbf{v}_2$) Consequently, there are 21 interpolations needed in total. This method is not suitable for repeated interpolation; a better solution is proposed in [6], utilizing pre-calculated coefficients for the local cube to speed up repeated interpolation.

1.3 Partial Derivatives Approximation

As seen in the previous sections, sometimes it is necessary to obtain a derivative of a function. Obtaining a derivative of a symbolical function is a relatively straightforward process. Unfortunately, sometimes there is no symbolical description of the function available, and it is necessary to work with sampled function values instead.

There are two main possible approaches — the first is to interpolate the function itself from function values and then find its derivative or to estimate the derivative directly from known function values. The second approach was already mentioned and used in section 1.2 Cubic Interpolation and is also known as finite difference approximations of derivatives.

Finite Difference The idea behind this method is to substitute the derivative with an approximation. The formula for a finite difference can be derived from Taylor's theorem. Suppose there is a uniform distance h between known points $x_0, x_1, x_2, \ldots$, thus $x_{i+1} = x_i + h$ and suppose a function f is $(n+1)$ times differentiable at a point x. According to Taylor's theorem

$$f(x+h) = f(x) + \frac{f'(x)}{1!}h + \ldots + \frac{f^{(n)}(x)}{n!}h^n + R_{n+1,x}(x+h). \qquad (1.11)$$

It is possible to modify (1.11) and express the first derivative. It is also possible to express the error of this approximation as $\mathcal{O}(h)$. Suppose $\mathcal{O}(h)$ is small enough, the first derivative can be approximated by

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \qquad (1.12)$$

This expression is also called the forward difference. Similarly, there is an expression for a backward and central difference.

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}, \; f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad (1.13)$$

Likewise, finite differences can approximate partial derivatives of a multivariable function [7].

$$\frac{\partial f}{\partial x} \approx \frac{f(\ldots, x+h, \ldots) - f(\ldots, x-h, \ldots)}{2h} \qquad (1.14)$$

Approximating Derivative from an Interpolated Function The other approach is first to interpolate the function from sample points and then differentiate the interpolating function. When choosing the interpolation method, it is essential to take into account the way how the sample data was obtained. For example, it is possible to estimate the derivative value by modifying the formula (1.7) for tricubic interpolation.

$$Q'(t) = \begin{pmatrix} t^2 & t & 1 \end{pmatrix} \begin{pmatrix} 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \mathbf{t}_0 \\ \mathbf{t}_1 \end{pmatrix} \qquad (1.15)$$

1.4 Numerical Methods for ODEs

One of the visualization techniques uses particle tracing to visualize the flow of a vector field. The tracing requires evaluating an integral which can be reduced to an initial value problem for an ordinary differential equation. Since it is required to solve such equations for sampled vector fields, it is necessary to solve these equations numerically. There are three methods discussed in this section, which can be applied to the differential equations and thus solve the integral, allowing to trace particles inside a vector field. All discussed methods are members of the Runge-Kutta family of Ordinary differential equation (ODE) solvers[1] [8, 9].

Euler Method Euler method divides time into uniform intervals h called time steps. Suppose there is a particle at an initial position $\mathbf{y}_0$ in a vector field V at a time t_0. In each run, the method moves in the direction of the tangent vector multiplied by the time step.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\,V\left(\mathbf{y}_n\right) \tag{1.16}$$

The error made in one step also known as the local truncation error for the Euler method is $\mathcal{O}\left(h^2\right)$ and can be calculated from the remainder term in Taylor's theorem in Lagrange form.

Heun's Method Heun's method[2] is based on the Euler method. It uses the result of the Euler method as an intermediate value $\mathbf{y}^{int}$ and then combines it with the value obtained at the next integration point. The final value is calculated as

$$\begin{aligned} \mathbf{y}_{n+1}^{int} &= \mathbf{y}_n + h\,V\left(\mathbf{y}_n\right) \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \tfrac{1}{2}h\left(V\left(\mathbf{y}_n\right) + V\left(\mathbf{y}_{n+1}^{int}\right)\right). \end{aligned} \tag{1.17}$$

Thanks to the correction step, the estimate of the local truncation error is $\mathcal{O}\left(h^3\right)$.

Runge-Kutta Method As was mentioned earlier, all methods listed here are members of the Runge-Kutta (RK) family. To be precise, the techniques presented here are all explicit methods. Although each explicit RK method is conceptually the same, each of them differs by order of accuracy. The general concept of RK methods takes into account the integration time, allowing for integration in a time-dependent[3] vector field $V\left(t, \mathbf{p}\right)$. When working with a time-independent vector field, which acts the same way at all times, it is possible to use the same integration methods and ignore the time parameter,

[1] There is another family of methods for solving ODEs called linear multistep methods, but they are not utilized in this thesis.

[2] also known as improved or modified Euler's method

[3] A time-dependent vector field function V takes one more time-related parameter t.

therefore, $V(t,\mathbf{p}) = V(\mathbf{p})$. The general framework for explicit RK methods is given by

$$\begin{aligned}\mathbf{y}_{n+1} &= \mathbf{y}_n + h\sum_{j=1}^{s} b_j V(t_n + c_j h, \mathbf{z}_j)\\ \mathbf{z}_i &= \mathbf{y}_i + h\sum_{j=1}^{i-1} a_{i,j} V(t_n + c_j h, \mathbf{z}_j),\ i = 1,\ldots,s\,.\end{aligned} \tag{1.18}$$

The coeficients are usually displayed in a table called Butcher tableau:

$$\begin{array}{c|ccccc} c_1 & & & & & \\ c_2 & a_{2,1} & & & & \\ c_3 & a_{3,1} & a_{3,2} & & & \\ \vdots & & & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} \tag{1.19}$$

It is possible to prove that the estimate of the local truncation error of the method of order m is $\mathcal{O}(h^{m+1})$ [9]. Probably the most popular is the standard fourth-order procedure with following intermediate steps:

$$\begin{aligned}\mathbf{z}_0 &= \mathbf{y}_n\\ \mathbf{z}_1 &= \mathbf{y}_n + \tfrac{1}{2}h\,V(t_n, \mathbf{z}_1)\\ \mathbf{z}_2 &= \mathbf{y}_n + \tfrac{1}{2}h\,V\left(t_n + \tfrac{1}{2}h, \mathbf{z}_2\right)\\ \mathbf{z}_3 &= \mathbf{y}_n + \quad h\,V\left(t_n + \tfrac{1}{2}h, \mathbf{z}_3\right),\end{aligned} \tag{1.20}$$

therefore,

$$\begin{aligned}\mathbf{y}_{n+1} = \mathbf{y}_n + \tfrac{1}{6}h\,(V(t_n,\mathbf{z}_1) + 2V\left(t_n + \tfrac{1}{2}h, \mathbf{z}_2\right)\\ +2V\left(t_n + \tfrac{1}{2}h, \mathbf{z}_3\right) + V(t_n + h, \mathbf{z}_4)).\end{aligned} \tag{1.21}$$

The Butcher tableau for the fourt-order RK method is:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{1.22}$$

Using any of the presented methods, it is neccesary to choose the coeficient h. Smaller coefficient leads to smaller error, but it is necessary to take more steps to cover the same amount of space, which makes the integration more computationally expensive. Adaptive RK methods offer a solution to this problem. The concept is to estimate the local truncation error by using two methods, one with order m and the other with order $m-1$. These two methods have common integration steps. Consequently, the computational cost of using two methods is near negligible. Coefficients b of the lower-order methods are here marked as b^* and can be found under the coefficients b of the higher-order method in Butchers tableau.

$$\begin{array}{c|ccccc}
c_1 & & & & & \\
c_2 & a_{2,1} & & & & \\
c_3 & a_{3,1} & a_{3,2} & & & \\
\vdots & & & \ddots & & \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b_1^* & b_2^* & \cdots & b_{s-1}^* & b_s^*
\end{array} \tag{1.23}$$

The local truncation error can be estimated as

$$e_{n+1} = \mathbf{y}_{n+1} - \mathbf{y}_{n+1}^* = h \sum_{j=1}^{s} \left(b_j - b_j^*\right) V\left(t_n + c_j h, \mathbf{z}_j\right). \tag{1.24}$$

This allows keeping the error under a defined threshold. When the error is too small, h can be increased to save time, and when it is too high, the integration step can be repeated with smaller h. [10] The method used in this thesis is one of the adaptive RK methods called Dormand–Prince method [11], which produces fourth- and fifth-order accurate solutions.

Chapter 2

Visualization Methods

The second chapter introduces the concept of visualization pipeline, followed by sections presenting selected visualization methods. This chapter draws mostly from sources [12, 13, 14].

2.1 Visualization Pipeline

The visualization pipeline is a model, dividing the process of visualization into several independent parts. The separation allows for modularity on both conceptual and design levels and has been adopted by most of the existing visualization apps. [12] The idea is to divide the whole visualization process into four following parts — data importing, enriching, mapping and rendering. The pipeline is illustrated in figure 21.

Data Importing Importing data usually means correctly loading the particular dataset. This operation often depends on the implementation of the original dataset format and ideally is equivalent to reading the dataset file from some storage (e.g., database, disk). In case the import involves resampling or modifying the dataset, it is essential to take into account that all modifications performed this early in the visualization process have a significant impact on the quality of the output visualization.

Filtering and Enriching The imported dataset often contain excessive amount of information, from which only a small fraction is what the user wants to visualize. Another approach is to perform some transformation, allowing to visualize structures hidden inside the dataset. These two actions are performed in the second phase of the visualization pipeline and are called data filtering and enriching. As a result of filtering, the output of this step should be a relevant subset of the complete dataset. Enriched dataset will be easier to use in later steps of the visualization pipeline. It is crucial to

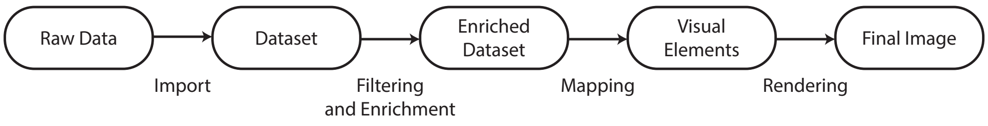


Figure 21: Illustration of the visualization pipeline

take into account the size of the dataset since it influences the memory and processing performance in later steps of the pipeline. Larger sets also require more filtering; otherwise, the visualization results in convoluted images.

Mapping The third step involves mapping enriched dataset onto visual elements, such as three-dimensional geometry. Various visual elements have different degrees of freedom or features, which can encode one or more attributes of the enriched dataset. The selected mapping should have several desirable properties. Mapping should be invertible, meaning it should be possible for the user to assign the state of the visual element (e.g., specific color) to the original attribute value. This property implies that each attribute value should have a unique encoding. Moreover, the mapping should not be invertible just from the mathematical point of view; it should be easy for the user to interpret the visualization, which can be achieved by selecting the appropriate combination of visual features. Mapping should also respect the organization of attribute values. Some attributes are by their nature not comparable. Such attributes are called nominal attributes; contrary to comparable attributes, which are called ordinal attributes.

Rendering Mapping process creates a virtual scene consisting of various visual elements. In the last step of the visualization pipeline, the last parameters are added, such as viewport dimensions, lights, camera positions. The output of the last step is the actual visualization image. It is common to allow users to modify specific parameters of the scene and then recompute the output image, allowing for simple modifications, e.g., camera movement.

2.2 Vector Glyphs

The idea behind vector glyphs is to represent vectors in isolated points using specific visual elements. Each type of glyphs has multiple degrees of freedom (e.g., color, size, rotation), which can be ultimately used for visualizing one of the vector attributes (e.g., direction, temperature, speed). According to the number of degrees of freedom, glyphs can be classified into several classes, the simplest being line glyphs, followed by more complex arrow glyphs and cone glyphs.

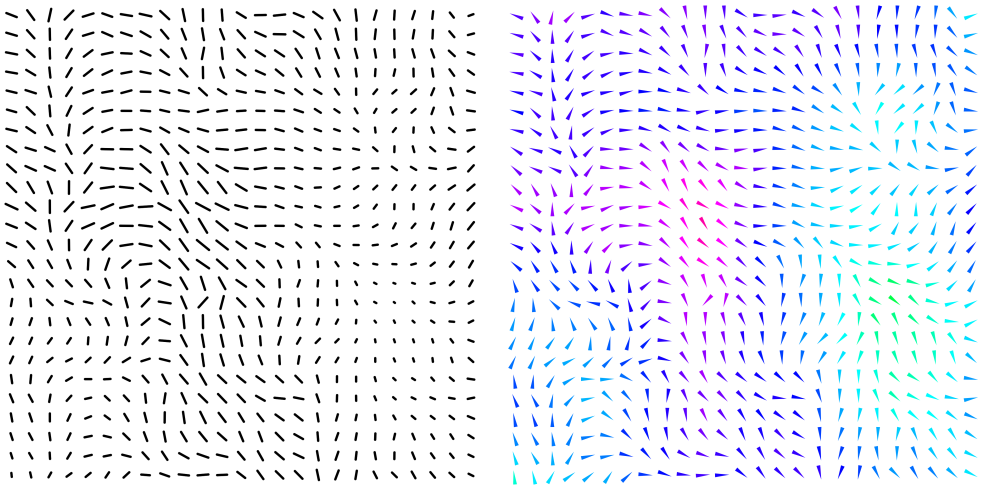


Figure 22: Example of glyph visualization in 2D — line glyphs and modified 2D cone glyphs

Line Glyphs The first and also the simplest class of glyphs are line glyphs with the following degrees of freedom.

- length
- rotation[4]
- color and transparency
- thickness

The length of the glyph usually represents the length of the vector. Alternatively, the glyphs are all scaled to be about the same size, and vector length is coded primarily by the glyph color. In both cases, it is suitable to scale the glyphs to prevent clutter but also to prevent the glyphs from being too small. Additional highlighting of the critical regions can be achieved by choosing a suitable mapping, such as exponential or logarithmical. The essential drawback of the line glyphs is the impossibility to derive the exact vector direction from it.

Cone and Arrow Glyphs Line and arrow glyphs share multiple degrees of freedom, but arrow and cone glyphs finally add the possibility to differentiate glyphs with the exact opposite directions.

- length
- direction
- color and transparency
- thickess
- cone base dimensions

[4]Not to be confused with direction, since the rotation of a line glyph is by its nature unsigned; thus, it is not possible to unambiguously distinguish the exact direction of the vector represented by the glyph.

Although the cone base dimensions can be counted as individual degrees of freedom, it is not recommended to use them for encoding additional attributes since the cone[5] already conveys the direction of the corresponding vector and any further manipulations could obliterate the conveyed information. The essential advantage of arrow glyphs compared to line glyphs is their ability to encode the vector direction. One of the drawbacks is the fact that arrow glyphs require more space and thus create clutter more easily.

Vector Glyph Visualization Problems There are several problems connected with the usage of glyphs; some of them were already mentioned earlier. Following list contains a discussion regarding two of the most common problems - glyph density and spatial distribution. These two characteristics are somewhat similar, but each of them is problematic in a slightly different situation.

Density The glyph density is closely related to sampling. The higher the sampling, the smaller area is reserved for each glyph. Hence, higher sampling leads to a denser cluttered image. The second approach is to preserve the higher density and scale each glyph down to reduce clutter. The issue with images produced by the second approach is that smaller glyphs are difficult to interpret.

A new problem with occlusion arises in three dimensions, when near glyphs obscure further ones. In other words, it is not possible to see *inside* the dataset. This problem can be solved with further subsampling or by using transparency. Using transparency should, in theory, highlight denser regions.

Distribution A distribution problem emerges in situations when glyphs are located in a regular grid. With scalar values, the human eye has no problem performing visual interpolation between regularly distributed points. The situation is different when it is required to interpolate between vectors. When the surrounding vectors point in different directions, it is challenging if not impossible for human vision to estimate how the interpolated vector should look like, especially if all of the surrounding vectors also vary in size.

Random distribution offers a solution to this problem. There is a chance that both of the problems mentioned above will be eliminated and the output image will be perceived as more natural.

[5]or cone head in case of the arrow glyph

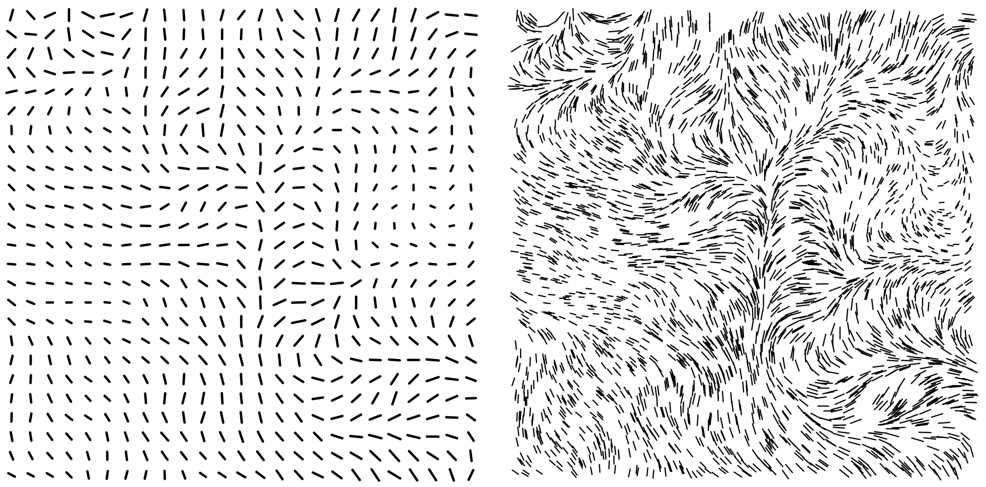


Figure 23: Comparison of visualization using regularly and randomly distributed glyphs

2.3 Streamlines and Streaklines

An entire family of visualization methods is based around tracing particles inside a vector field and constructing curves along traced paths. Streamline is a curve visualizing flow inside a stationary field; streaklines visualize a flow over time in non-stationary fields.

A streamline P in a vector field V can be modeled as a parametrical curve $P : [0, e] \to \mathbb{R}^3$ when $0 < e \in \mathbb{R}$. With $t \in [0, e]$, streamlines can be defined with the expression

$$\frac{dP(t)}{dt} \times V(x, y, z) = 0. \tag{2.1}$$

The formula (2.1) describes the streamline as a parametrical curve tangent to $V(x, y, z)$ vectors of the vector field in each point (x, y, z). That can also be expressed as an ordinary differential equation

$$\frac{dP(t)}{dt} = V(x, y, z) \tag{2.2}$$

with initial condition $P(0) = \mathbf{s}_0$. By integrating (2.2) over t, it is possible to obtain streamline $S = \{P(t), t \in [0, e]\}$, where

$$P(t) = P(0) + \int_0^t V(P(s))\, ds \tag{2.3}$$

with $P(0) = \mathbf{s}_0$. [12] Parameter t represents integration time and should not be confused with the time parameter of time-dependent vector fields. It is possible to solve the equation (2.2) numerically using any of the methods introduced in section 1.4 Numerical Methods for ODEs.

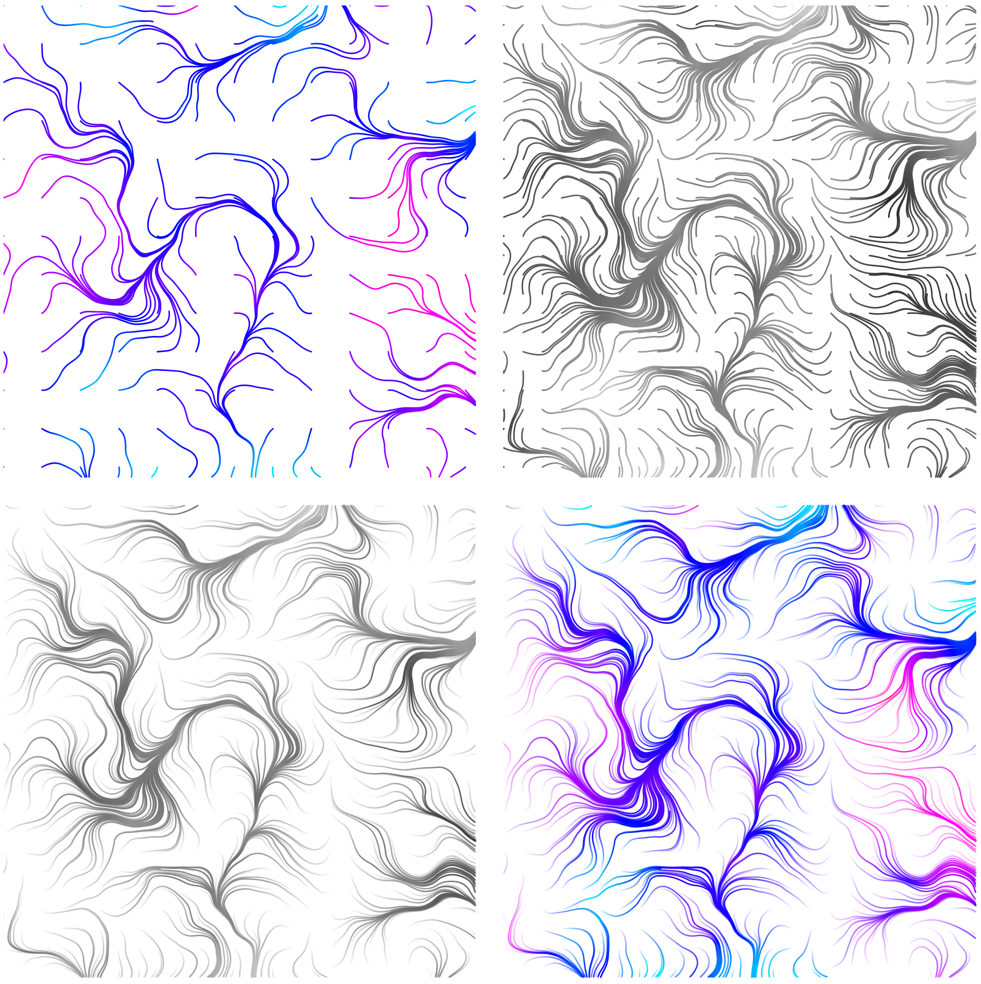


Figure 24: Example of streamline visualization

There are several technical parameters which need to be considered; two of them are discussed below — an appropriate selection of seeding points and a method of geometry construction. Figure 24 is attached as a demonstration of streamline visualization. There are four versions of the same vector field visualization utilizing a different combination of color coding and sampling points.

Geometry Construction Integrating the streamline generates a list of points positioned along the curve. The number of points forming a streamline is inversely proportional to the step size and directly proportional to the upper bound of the time parameter. Therefore, the density of the geometry can be influenced by changing the step size or by adjusting the error tolerance limits to influence the choice of the step size in case a method with adaptive step size is used. On the other hand, taking longer steps will likely result in more significant deviations of the computed streamline from the expected path.

Drawing densely sampled curves is computationally more expensive. The solution to this problem is to separate the geometry construction from the integration process. Based on the output of the initial integration, it is possible to construct the geometry by iterating over the output points and adding a new line between two consecutive points when the distance between them exceeds a chosen minimum value. In case any two consecutive points are too close, the point further in the list is skipped, and the process is repeated with the following point.

Geometry Seeding There is a range of techniques for choosing appropriate seeding points. According to [15] there are three main criteria for choosing relevant seeding points — coverage, uniformity, and continuity.

Coverage Each point in space should be within a certain distance of the closest streamline. This requirement assures that all interesting regions and unique phenomenons are covered. Further, it is required that streamlines cover the entire vector field. Coverage is simply achieved by generating new streamlines, which is not acceptable in convergent regions, where the streamlines join together generating clutter.

Uniformity Streamlines should be uniformly distributed over the entire area. That is especially difficult to achieve in three-dimensional space since the seeding points would be view-dependent. This issue has been further discussed in [16].

Continuity Longer streamlines are preferred both from the aesthetic and user-friendly points of view. It is easier to interpret longer streamlines; they give an impression of a continuous flow. The drawback is that longer streamlines tend to come together in convergent regions and thus fail the previous uniformity criterion. Hence, it is needed to balance the continuity with the coverage and uniformity.

Probably the most straightforward approach is to choose regularly or randomly distributed seeding points. Both of those methods are valid; the only drawback is that they both fail the uniformity criterion. Also, note the fact that close streamlines tend to be parallel and do not contribute any new information into the image. There has been additional research regarding strategies for choosing appropriate seeding points [15, 17]. These strategies opt for choosing seeding points based on the result of detection of areas called critical points.[6] Such strategies are often called *flow-guided streamline seeding.*

[6]Explanation of what critical points are can be found in [18].

2.4 Scalar Visualization

Even though this thesis is mostly focused around visualizing vector fields, it is often convenient to visualize a scalar field derived from the original vector field. The scalar field could describe a quantity such as absolute speed value or temperature. The area of scalar visualization is a vast topic, and this section discusses only three selected techniques — scalar glyphs, isosurfaces, and planes. For further information, see [14].

Scalar Glyphs Scalar glyphs are similar to vector glyphs, although their features (degrees of freedom) should be adequate for representing not only the values but also the meaning of the explained variable. The fundamental features of scalar glyphs are size and color. It is possible to map different attributes to different features and look for any dependencies.

Isosurfaces Isosurfaces use a surface as a feature for modeling and connecting areas in space where the quantity or attribute is equal to a fixed value. A well-known algorithm for constructing isosurfaces is *Marching cubes.*

Planes Another way to visualize scalar values in space is to construct a plain and color-code the values on its surface. The construction of suitable colormaps is discussed in the following section.

2.5 Color

Color remains one of the features (degrees of freedom) of each of the visualization methods presented above. There are certain principles associated with designing an appropriate colormap, which can be further applied to other degrees of freedom behaving similarly to color. Colormap M can be generalized as a function associating each value $v \in V$ to a color $c \in$ Colors.

$$M : V \rightarrow \text{Colors} \tag{2.4}$$

It is a common practice to limit the range of values V to $[v_{min}, v_{max}]$ and design the color mapping based on this limited range. This approach is not always suitable; using globally determined limits could render some of the areas as uniformly shaded regions when in reality the values inside the areas could vary by an amount too small to be distinguishable when compared to the global range of the colormap. This phenomenon is particularly problematic in combination with zooming. In this case, it is more appropriate to determine the local limits and adapt the color mapping to the displayed area.

When working with vector visualization techniques, color usually represents one of the vector attributes[7], most often it is the size of the vector. In

[7]velocity, temperature or any other scalar value

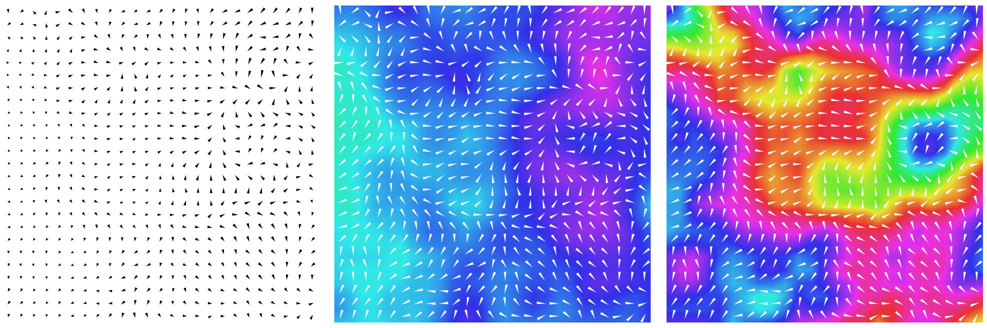


Figure 25: Glyph visualization with color-coded size and direction

combination with introduced methods, e.g., vector glyphs in two dimensions, it is possible to encode the vector orientation using the HSV (hue, saturation, value) color model (HSV). In three dimensions, it would be necessary to use a color model which could be projected onto a sphere. An example of this method is figure 25 displaying one vector with different techniques. In the second image, the size of the vectors is coded by color. In the last image, the background corresponds to the vector rotation which can be easily verified by comparing the glyph orientation to the matching position on the color wheel in the figure 26.

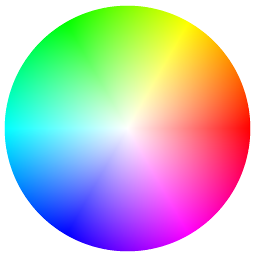
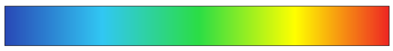
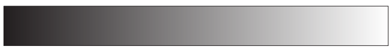


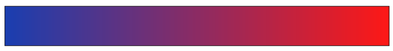
Figure 26: Slice of the HSV color space mapped onto a circle

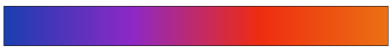
Principles of Designing a Colormap According to [12], *"a color-mapping visualization is effective if, by looking at the generated colors, we can easily and accurately make statements about the original dataset that was color mapped".* There are five fundamental principles which should be respected in order to achieve an optimal color mapping:

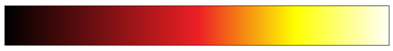
1. It has to be possible to derive the original dataset value at all points. Consequently, the mapping function must be injective; in other words, each color must represent only one value. The colors must be visually distinct from one another.


(a) Rainbow map


(b) Grayscale map


(c) Two-hue map


(d) Four-hue map


(e) Heat map

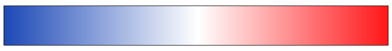

(f) Diverging map

Figure 27: Example colormaps

2. The mapping must allow comparing[8] values at different points. This principle does not enable determining the amount of difference; it only allows identifying which value is larger or smaller.

3. It must be possible to determine what is the difference between two given points. Contrary to the previous point, the order of the original values is not relevant.

4. The mapping should enable locating areas with any given value. Once again, this requires the mapping to be injective. Assigning a corresponding color to the given value may require a color legend.

5. It should be possible to approximate the rate of change[9] between two given points.

The colormap design must reflect the visualization purpose and the character of the dataset. One of the examples could be a visualization of a dataset containing gradually changing values. In case the user expects linear changes in the image, the used color mapping should indicate that.

Colormap designs Certain colormaps, e.g., the rainbow colormap, are often overused. As a more suitable alternative, [19] introduces diverging colormaps, presents their advantages and proposes an algorithm allowing their construction. Possible colormap designs are:

Rainbow Colormap The rainbow map is a widely adopted colormap despite several significant setbacks. Besides the varying luminance of the colors, the user has to know the order of tones used in the rainbow map.

[8]based on the color

[9]The mathematical analogy of this is the gradient expressing the amount and direction of a change. For this thesis, it is sufficient to estimate the rate of change purely based on the visualization.

Grayscale Colormap A grayscale map is cleaner than the rainbow map. On the negative side, comparing two grayscale values is not easy; the colormap does not provide enough visually unique values.

Two-hue and Four-hue Colormaps A two- or four-hue map should cover the disadvantages of grayscale maps by introducing a hue into the system. Hue allows easier ordering of the values. However, the colormap still offers the same dynamic range. The four-hue map could solve this problem by introducing a pair of entirely distinctive colors; however, the result could end up closer to the rainbow map, lacking clarity.

Heat Colormaps A heat map is a specific type of color map designed to symbolize the temperature of the visualized object. It uses luminance to highlight high values; other values are mostly suppressed.

Diverging Colormaps A diverging map is suitable for data with diverging values. The map is similar to the two-hue maps with an added third color for the neutral spectrum (usually the values around zero). This construction allows highlighting the extreme values at both ends of the spectrum, suppressing the mean values.

CHAPTER 3

Techonology

This chapter presents the available visualization technology. The initial section discusses platforms which have to be supported by the developed application. The second part introduces the available visualization software. The final section contains additional notes on the available software and presents some of the alternative solutions.

3.1 Target Platforms

According to the assignment, it is required that the application will be scalable to support visualization on two unconventional platforms — SAGE2 and 3D monitor. Here is a description of the platforms for later analysis.

SAGE The SAGE environment was designed to enable teamwork in front of large shared screens. The task would usually require displaying an immense amount of information in a high resolution. The SAGE2 environment is an implementation based on web technologies; applications for this environment are likewise developed using web technologies and JavaScript programming language [20].

3D monitor In order to support 3D televisions and monitors, the system must be capable of producing an image and a depth map of the scene. Alternatively, a rendering of the scene from two different points is expected. Some of the monitors allow viewing the space "behind" the displayed object - in those cases, a combination of images from different angles and a depth map is required. Due to missing documentation for some of the devices, determining the best parameters for the visualized scene requires some experimentation.

3.2 Available Software

It is possible to divide the available software into two categories. The first group represents applications enabling visualization; however, it is not the main functionality of the software. The second group contains software focused solely on visualization. Well-known representatives of the first group are Matlab or Mathematica; the second group is represented by Paraview, MayaVi or VisIt. For brevity, the first group will be referred to as computational software, the second group as visualization software. Using existing software enables

- utilizing implemented algorithms,
- incorporating extensions into working visualization pipeline, and
- building a visualization on top of working renderer and frontend.

Computational Software Computational systems, such as Mathematica, are based around their proprietary programming language, which allows for effective manipulation and additional transformations of the original dataset. Depending on the available features of the computational software, it is possible to use already built-in visualization tools. These functionalities usually require preprocessing the original dataset into a custom format; however, the transformation can be performed using other built-in methods. Matlab supports visualizations using all methods mentioned in chapter 2, both in two and three dimensions. Mathematica has the same capabilities, except for visualizing streamlines in three dimensions. Although, it is possible to add that functionality using other built-in methods [21]. The systems support creating a custom user interface for managing the visualization.

Visualization Software Programs Paraview, MayaVi, and VisIt are open-source projects based on open-source software called The Visualization Toolkit (VTK) [22, 23, 24]. This software is a state-of-the-art tool for 3D visualization rendering and implements a vast range of algorithms and visualization methods. The VTK project is written in C++ and offers an interface to various other languages such as Python. The core of the VTK is a pipeline working on similar principles as those introduced in section 2.1 [25]. There are several approaches to creating a visualization with visualization software (VTK):

Standalone Application It is possible to use VTK for developing a standalone application. VTK offers an API in several programming languages. A new application could implement custom UI or build extensions on top of the fixed VTK visualization pipeline.

Plugins and Extensions The alternative is to implement a plugin into an existing application, allowing to read a custom file format or implement custom dataset enrichment; the application would handle the rest of the visualization pipeline, including UI manipulations. The general approach is to extend the available applications by adding new functionalities.

The design of VTK is heavily object-oriented.[10] It supports many file formats[11], although it is common to write a custom reader to support non-standard file formats. Custom readers are also suitable for parsing standard formats that support custom data organization, e.g., FITS file format.

3.3 Alternative Resources

This section presents resources for designing an independent application. A different approach is to design an entirely new visualization application without utilizing any existing visualization frameworks. One of the downsides of using an existing framework or computational system is that it requires the knowledge of the system. VTK is known for its steep learning curve, and most of the computational systems are proprietary, which precludes any development of a standalone application. Implementing an application without using a visualization framework

- provides control over the entire visualization pipeline,
- enables using technologies already familiar to developers, and
- removes any design limits otherwise posed by the underlying framework.

The following list presents a variety of alternative technologies, modules, and frameworks suitable for developing standalone applications.

SciPy and NumPy SciPy provides numerical routines for numerical integration, and NumPy is the fundamental package for scientific computing with Python, among other things it contains tools for manipulating multidimensional arrays and integrating C/C++ and Fortran code [27, 28].

OpenGL and WebGL OpenGL is widely adopted API designed for high-performance graphics software applications. It serves as a layer between the application and the graphics hardware exposing its features and enabling parallel computations. WebGL is a multi-platform web standard for low-level APIs for 3D graphics, based on OpenGL ES. WebGL 2.0 context allows rendering using an API which conforms closely to the OpenGL ES 3.0 [29].

[10]The structure is perhaps slightly overdesigned.

[11]For more information regarding file formats, see [26].

three.js three.js is a framework based on WebGL that implements some of the visual primitives. Managing the geometry of the scene, it speeds up the development process by providing an abstract layer above the WebGL API. [30]

p5.js p5.js is a JavaScript library enabling sketching with code. The library draws inspiration from Processing[12], and it is based on similar principles [31]. All images in chapter 2 were created with p5.js.

[12]For more information see processing.org.

Chapter 4

Analysis

The visualization application is a combination of mathematical and visualization methods implemented using suitable technology. This chapter presents an analysis and evaluation of these methods based on specific criteria defined by the assignment. Since some of the criteria are linked, or relate to more than one of the aspects of the visualization process, the discussion needs to be organized to provide a more natural understanding. The organization is based on the visualization pipeline. The criteria used throughout the analysis include physical accuracy and adjustability of used methods, clarity of the output visualization, performance, and portability.

4.1 Importing

The first step in the visualization pipeline is the data import. The import involves transforming the raw data into an internal representation. During this step, no data or meta information should be lost. The internal representation should be designed to provide adequate performance for the most frequently performed actions. Consequently, it is necessary to determine what the most frequently performed actions are. The following visualization methods are considered:

Glyphs The glyphs are constructed based on the values in isolated points. The internal representation must allow obtaining the value in any arbitrary point inside of the dataset.

Streamlines Streamlines are constructed with numerical integration starting in the seeding points. During each step of the integration, it is necessary to retrieve the value in the current and other surrounding points, based on the chosen numerical method.

The fundamental and the most frequently performed operation is the interpolation. The first chapter presented three possible interpolation methods.

method	intermediate values[14]	local values	accuracy
Nearest-neighbor	0	8	low
Trilinear	6	8	medium
Tricubic	20	64	high

Table 41: A table of considered interpolation methods

Each of the methods provides results with different accuracy. The trade-off of higher accuracy is a higher computational complexity. The actual interpolation consists of two stages.

1. In the first stage, the values inside of the local cell are obtained. In case a streamlined parallel version of the interpolation method[13] is used, the local coefficients are precalculated.
2. In the second step, the calculation is performed based on the local coordinates.

The choice of the interpolation method depends on whether the repeated interpolation is performed inside single or multiple cells.

Single Cell In case the repeated interpolation takes place inside a single data cell, the first step of the interpolation — the lookup of the local values — can be performed only once and the second step is repeated for every individual set of local coordinates. Furthermore, the interpolation can be sped up by performing the calculation in a matrix form[15], which can be efficiently calculated using dedicated hardware. In this case, it is viable to employ more sophisticated and physically more accurate interpolation methods such as tricubic interpolation.

Multiple Cells When the repeated interpolation takes place among different local cells, the lookup of the values precedes each calculation. Since a parallel version of the calculation would require recalculating the local coefficients for every local cell, speeding up the actual calculation using dedicated hardware is no longer beneficial. In this case, it is advisable to use methods which work with a smaller number of local values. The nearest-neighbor method does not require any precalculated values, and

[13]See sections 1.2 Linear interpolation and 1.2 Interpolation in 3D.

[14]The number of intermediate values is always one less than the total number of one-dimensional interpolations. In case a parallel version of the computation is performed, instead of calculating the intermediate values, two matrix multiplications are performed.

[15]The parallel computation of a trilinear interpolation is based on the equations (1.5) and (1.6) from the section 1.2 Linear interpolation. The matrix $\mathbf{C}$ is calculated only once, and the matrix $\mathbf{Q}$ is recalculated for every set of local coordinates. Matrices used for tricubic interpolation are different, but the principle remains the same.

the trilinear method requires calculating six intermediate values. The trilinear method appears to be the best option, balancing both performance and accuracy.

The data provided with the assignment is a regularly sampled dataset. The sampling is not uniform in each axis, the step in x- and y-direction is 1 and in the z-direction it is -0.1. The number of values in the dataset is $600 \cdot 600 \cdot 206$. The speed values are indicated in millions of meters per second; the positions in millions of meters. Unless it is required to visualize a small segment of the dataset, the interpolation will mostly take place among many different local cells. Thus, the most suitable method would be the trilinear interpolation.[16]

Additional Grid Types Support This thesis primarily focuses on regularly and uniformly sampled data. Additionally, it might also be possible to extend the support to other types of grids. The unstructured grids require a completely different approach; the methods manipulating such data were not introduced and reach beyond the scope of this thesis. With the introduced methods, the support can be easily extended to rectilinear grids or any other three-dimensional grids where all of the axes are perpendicular to each other.

4.2 Filtering and Enriching

The internal representation of the dataset supports a specific class of grids and a fast and accurate interpolation. The second stage of the visualization pipeline involves transforming the imported dataset. The dataset transformation leads to filtering out all of the insignificant parts and highlighting important areas. Additionally, the data structure is adjusted to guarantee ease of use in the following stages of the visualization pipeline.

4.2.1 Filtering

First, it is needed to filter out all the insignificant areas and recognize what should be left in the dataset. The critical factors are once again the performance and physical accuracy of the output dataset. There are two approaches:

Automatic Detection The first option is an automatic recognition of the critical parts based on dataset analysis. Segmenting the important areas based on curl or divergence is possible. This procedure could be for example followed by filtering out the areas with lower velocities. A similar effect can be achieved by adopting techniques of *flow-guided* visualization.

[16]In case the repeated interpolation inside a single cell is required, the trilinear interpolation will perform the same as if it was interpolating many times among different cells; thus the performance should be sufficient.

These techniques mentioned in section 2.3 Geometry Seeding try to identify *critical points* in the vector fields with more sophisticated methods.

Manual Selection The second, more straightforward approach is to let the user decide which parts should be filtered out. It is possible to highlight the significant parts based on the property represented by the vector field itself, e.g., highlight the areas where the temperature or speed reaches the highest values. This effect can be understood as *visual filtering* enabled by correct mapping of the values onto the visual elements in the third step of the visualization pipeline. It is also possible to introduce the concept of thresholding, which is not strictly applicable only to the manual approach. Thresholding allows to filter out values below or above a particular value.

Relying on an automated process to recognize significant events or areas is quite common, especially if the amount of the data exceeds a comprehensible quantity. The first method relies on automated detection of the critical points in the dataset. The second approach utilizes presumably weaker tools such as thresholding and relies mainly on the third step of the visualization pipeline — the mapping step — to highlight only the significant areas.

Since the first approach utilizes more complex filtering algorithms, it could bring more satisfying results, although the methods used for *flow-guided* visualization are once again beyond the scope of this thesis. For simplicity, the second approach is chosen here. Based on the previous decision, the available tools for dataset filtering are:

- thresholding
- manual selection of a subset of the dataset
- *visual filtering* in the mapping stage of the pipeline

4.2.2 Enriching

As a part of the enrichment process, a new dataset is generated by transforming the filtered data. The structure of the enriched set should satisfy the following conditions:

- ease of use in the following pipeline stages
- balancing memory and performance efficiency
- physical accuracy

Ease of use The first condition can be satisfied by determining the data requirements of the individual visualization methods. In the ideal state, the enriched dataset would include only the data required by the used visualization method.

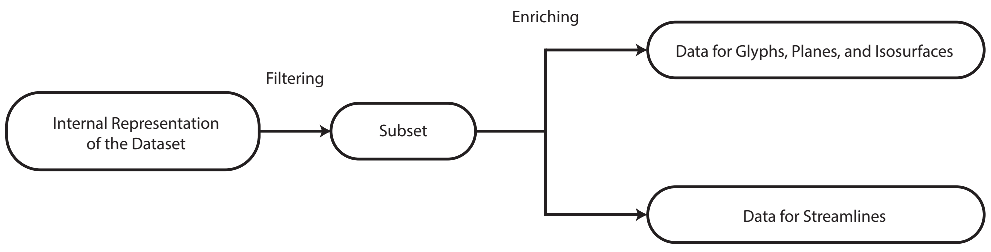

Enriching
Filtering
Internal Representation of the Dataset
Subset
Data for Glyphs, Planes, and Isosurfaces
Data for Streamlines

Figure 41: Illustration of the filtering and enrichment process

Glyphs Glyphs, both vector and scalar, require values and positions of the visualized points.

Streamlines Each streamline requires a list of corresponding positions, values and timestamps.

Planes and Isosurfaces Planes and isosurfaces require values and positions.

Balancing Efficiency and Physical Accuracy It is necessary to decide how the data is going to be obtained and stored. There are two solutions to this problem.

1. The first approach separates the calculation from the stored data. In terms of representation, the data can be stored in the form of arrays of floating point numbers. The data is obtained from a source, e.g., the interpolator.
2. The second approach dynamically retrieves the specific value every time it is requested.

The dynamic representations for individual data types are the following:

Positions The randomly sampled points can be stored in the form of a pseudo-random generator with a fixed seed. Regular sampling can be described in terms of the boundaries and actual sampling values, enbling easy reconstruction.

Values The values can be stored as the interpolator itself, dynamically performing the interpolation every time any value is requested.

Times Storing the times with a dynamic representation means the streamline integration would have to be performed every time a timestamps value is requested.

If appropriately implemented, a dynamic representation of values and points could be the solution enabling visualizations with a massive number of visual elements. The trade-off of memory efficiency, in this case, is the computational performance. An additional question is whether a memory efficient representation is needed after all. As illustrated in the figure 41, the structure of data for streamline visualization is different from the data structure used for glyphs, planes, and isosurfaces, therefore it will be treated as a separate case.

Data for Glyphs, Planes, and Isosurfaces It is likely that a visualization containing a higher number of glyphs or huge isosurfaces will lack clarity.[17] The other fact is that data in the form of arrays of floating point numbers are simpler to manipulate and thus prefered in the following stages of the visualization pipeline. The physical accuracy of both approaches is the same. Consequently, the data prepared for glyph and isosurface visualization are best represented as arrays of the actual values and points.

Data for Streamlines Transforming the dataset for streamline visualization involves evaluating the integral (2.3) from section 2.3 Streamlines and Streaklines with initial values corresponding to chosen seeding points. The seeding points are obtained during the filtering step. As was presented in the section 1.4 Numerical Methods for ODEs, there are several integration methods with a varying degree of accuracy. Methods with higher accuracy take more intermediate steps and are more computationally expensive. Using lower order methods, the same level of accuracy can be achieved by taking smaller integration steps, which will consequently lead to higher computational cost. Since physical accuracy is the priority, it is suitable to use a method of higher order to ensure the right balance between performance and accuracy.

Integration produces a list of positions and times modeling a parametric curve. The values of the vector field along the curve can be additionally obtained from the interpolator. The use of the introduced dynamic representation would require integrating the streamline every time a position of the curve is requested. Since the integration is a very computationally intensive task, it is more efficient to integrate the curve once and store the values in the form of floating point arrays.

4.3 Mapping

In the third stage, the data is mapped onto a geometry of selected visualization methods. The choice of the visualization methods and their attributes should be based on the following criteria:

[17]Thus, the filtering of the dataset was probably not sufficient.

method	features	pros	cons
Line Glyphs	length direction color transparency thickness	simple implementation	do not convey direction
Cone Glyphs	length direction color transparency thickness cone base dimensions	convey direction	take up more space
Arrow Glyphs	length direction color transparency thickness arrow dimensions	convey direction	take up more space, complicated geometry
Streamlines	length trajectory color transparency thickness	good representation of the flow	complicated geometry
Plane Surfaces	color transparency	simple geometry	able to convey only one attribute
Isosurfaces	position color transparency	able to visualize data on non-planar geometry	the visualization could lack clarity

Table 42: Considered visualization methods

- the ability of the method to convey a required attribute of the dataset
- clarity of the visualization

The table 42 documents the features of the individual methods. The attributes of the dataset to be represented by the methods are:

- direction of the vectors
- magnitude of the vectors
- a topological continuity of the vector field

The table 43 presents possible mapping between the data attributes and features of the methods. The following step is to determine which methods represent individual attributes the best.

methods	direction	magnitude	topological continuity
Line Glyphs	rotation	length color transparency thickenss	groups can create an effect of continuous flow
Cone and Arrow Glyphs	direction		
Streamlines	trajectory	color transparency thickess	geometry of the streamline natively represents the flow of the vector field
Plane Surfaces	–	color transparency	–
Isosurfaces	–	position color transparency	–

Table 43: Mapping method features to data attributes

4.3.1 Direction

This section discusses possible representations of the direction attribute. The goal is to determine the most suitable visualization methods. The table 45 sums up the results of the following analysis.

Glyphs Glyphs produce the most straightforward visualization of direction at specific points. When it is difficult to visually interpolate between neighboring glyphs, the situation can be solved with the following methods:

- random distribution
- subsampling the regular grid

Subsampling the regular grid reduces the space allocated for each glyph. If the vector field is not very turbulent, it might be possible to find *the sweet spot* of sampling without making the glyphs too small. Unlike line glyphs, arrow or cone glyphs can convey signed direction. The drawback is that they both take up more space, and arrow glyphs have a more complex geometry. Since line glyphs take up less space, the allocated area for individual glyphs can also be reduced. Line glyphs are more suitable for situations in which a denser sampling is required.

Streamlines Streamlines use their trajectory to represent the direction. The streamline visualization can be adjusted by modifying the seeding points and changing the length of the curves. As was introduced in section 2.3 Geometry Seeding, there are three possible seedings:

- regular seeding
- random seeding
- more sophisticated seeding produced by *flow-guided* methods

The criteria presented in section 2.3 Geometry Seeding are not satisfied when a regular seeding is used. The problem is that longer streamlines cover only the areas connected to the seeding points and the regular seeding does not take into account the future trajectory of the streamlines. Also, the length of the streamlines contributes to the problem. Longer streamlines tend to come together in convergent areas and continue with the same trajectory. Shorter streamlines do not cover much space; however, they might produce a more uniformly distributed visualization.

Random sampling might perform better than the regular sampling. The methods of *flow-guided* visualization offer a more general solution to this problem. A significant drawback is that implementing these methods would require more in-depth analysis.

Plane Surfaces and Isosurfaces A method allowing direction encoding with color was presented in section 2.5 Color; however, the presented method is rarely used. Isosurfaces work better with scalar visualization; they are not suitable for presenting a vector quantity, such as direction.

4.3.2 Magnitude

The goal is to find the most suitable features for visualizing the vector magnitude. Magnitude is also the attribute which will be preferably used for the *visual filtering*, as was discussed in section 4.2.1 Filtering. The areas are going to be highlighted based on the magnitude of the contained vectors. The table 45 contains a score for every method based on the following analysis.

Glyphs The magnitude is often visualized with the thickness or overall size of the glyph. Alternatively, all of the glyphs are set to be the same size, and color mapping is used.[18] The resizing enables *visual filtering*. On the other hand, using color mapping without resizing produces clearer images. Combining both approaches is possible. An additional problem appears when the the area of interest contains high contrast values. Because of the *visual filtering* effect, the values at one of the ends of the spectrum are going to become unreadable. This problem can be solved by inverting the size rule, highlighting the regions with the values at the opposite end of the spectrum. Color coding should also be invertible. The analysis of suitable colormaps is in a separate section.

[18]Transparency can be used as a separate feature, although, it can be understood as a component of the color.

Streamlines Streamlines' color can be used to convey the vector magnitude. Alternative feature with the ability to encode the magnitude is thickness; however, introducing the streamlines' thickness as an additional feature would cause clutter since streamlines already take up a lot of space.

Plane Surfaces and Isosurfaces Plane surfaces usually use color to encode the vector magnitude and thanks to the simplicity of the geometry, the visualization usually works quite well. The only downside is that the visualization is limited to the area of the plane.[19] Isosurfaces can represent the magnitude with their geometry — the surface leads through positions with the same magnitude. However, constructing these surfaces requires algorithms which were not introduced in this thesis. For simplicity, only the color mapping will be used.

Designing a Colormap A set of principles for designing a colormap were introduced in the section 2.5 Principles of Designing a Colormap. Also, the following effects need to be considered:

- Warmer colors attract more attention than cooler tones.
- The perceived luminance of two colors can differ despite them having the same luminance value in the HSV color model. Because of this effect, users can be more attracted to some of the colors over the others.
- There are several well-known colormaps, e.g., rainbow map, or heat map, which are more suitable for certain types of context.

Figure 27 in section 2.5 Color illustrates six well-known colormaps. The most common actions involve highlighting the distribution of the velocities and locating the areas containing extreme values.

The table 44 contains the evaluation of the presented colormaps based on the criteria introduced in 2.5 Principles of Designing a Colormap. According to every criterion, the methods are ordered from worst to best. Based on the results of the evaluation, it is preferable to implement maps such as diverging or heat colormap. Since the final scores of all hue-based maps were quite close, it is desirable to develop a tool allowing manual construction of any hue-based colormap.

4.3.3 Topological Continuity

Topological continuity of the vector field is a property related to perceiving the vector field as a whole. The visualization reveals the hidden structure of the

[19]The high score in the table 45 of plane surfaces does not reflect the fact that the visualization is limited only to a single plane.

colormap	reading values	ordering	difference	locating areas	rate of change	$\sum\uparrow$
Rainbow	4	1	2	2	2	11
Grayscale	1	2	1	1	1	6
Two-hue	1	3	2	2	1	9
Heat	2	3	2	3	2	12
Diverging	3	4	3	3	1	14

Table 44: Evaluation of colormap designs

method	direction score	magnitude score	topological continuity score	$\sum\uparrow$
Line Glyphs	1	2	2	5
Cone Glyphs	2	2	1	5
Arrow Glyphs	1	2	1	4
Streamlines	2	1	3	6
Plane Surfaces	2	3	0	3
Isosurfaces	2	1	0	1

Table 45: Presumed usability score of methods for visualizing individual attributes.

flow inside of the dataset, and the visualization methods have to capture this structure. According to the table 43, only glyphs and streamlines have degrees of freedom with the ability to visualize it. The table 45 assesses the ability of the methods to convey the topological continuity based on the following analysis.

Glyphs The flow structures can emerge from a visualization containing a higher number of glyphs. This effect relates to the discussion about direction. Visualization with more glyphs generally tends to contain sections where all elements point in the same direction. However, as was presented, denser grids also reduce the space allocated for each glyph. If the visualization of topological continuity is the primary goal, it is once again advisable to use line glyphs which need less space.

Streamlines Streamlines by definition follow the flow of the vector field. There are two drawbacks, the one regarding coverage was already discussed in section 4.3.1 Direction. The other one is computational complexity. Nevertheless, the streamlines are the most powerful tool for visualizing the topological continuity.

4.3.4 Summary

The table 45 contains the final usability score for each method. The score can be interpreted as an indicator of how universal each method is. Most of the results stay in the range 4–6. The similarity of the final scores denotes that each of the attributes is best conveyed by a different method. It is desirable to implement the methods which scored the highest amount of points for individual attributes. Based on this criterion, the selected methods are cone and line glyphs, plane surfaces, and streamlines.

4.4 Rendering

The goal is to develop a rendering part of the application, which will run in the SAGE2 environment and will support depth rendering. The renderer should produce images with sufficient clarity. Additionally, the renderer should support the following actions:

- viewport manipulation, including scaling, rotation, changing the viewport size
- minor adjustments of the geometry, e.g., changing the thickness of visual elements, adjusting the length of streamlines
- saving rendered images
- generating a depth map for a 3D monitor

The underlying technology and the overall application architecture dictate the structure of the renderer. As was introduced in section 3 Techonology, there are three distinct approaches to implementing the application — using an existing framework (VTK), computational software (Mathematica), and creating a custom application without any visualization framework (SciPy + WebGL). The table 46 assesses[20] all three approaches according to the following criteria:

Documentation The score reflects the quality, completeness, and clarity of the documentation. Bonus points are awarded for step-by-step examples.

Learnability The score indicates how straightforward achieving tasks is and reflects the steepness of the learning curve.

Licensing Reflects whether the software uses open source license (preferable).

Community Points are awarded based on the extent of the existing community. The community is evaluated based on available examples and support on the internet.

[20]The selection of the assesment criteria is inspired by [32].

criterion	VTK	Mathematica	SciPy + WebGL
Documentation	1	2	2
Learnability	1	1	1
Licensing	1	0	1
Community	1	1	2
Accessibility	1	0	1
Portability	2	1	2
Extendability	1	2	2
Supported formats	1	1	0
Development complexity	2	3	1
Familiarity	0	1	2
$\sum \uparrow$	11	12	14

Table 46: Assessment of implementation options

Accessibility To what extent is the software freely available, are there binary distributions available or is the source code available?

Portability Can the application be run on Linux, Windows, macOS and is it possible to develop an application for SAGE2 environment?

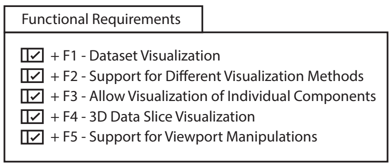
Extendability How easily can the existing software be extended with additional parts? Is it possible to implement additional visualization methods?

Supported formats How vast is the range of formats natively supported by the software? How easy is it to develop a plugin enabling import of additional formats?

Development complexity What is the overall complexity of developing the application using the chosen software?

Familiarity Reflects how familiar the author of this thesis is with the selected technology.

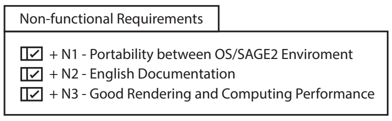
The final results are close, most of the differences balance out and the decisive factor might but just the last criterion — the familiarity. However, without the last factor, the combined score still points to the third option — developing the application without an underlying visualization framework. The only major downside of this approach is the development complexity.



Functional Requirements

- ☑ + F1 - Dataset Visualization
- ☑ + F2 - Support for Different Visualization Methods
- ☑ + F3 - Allow Visualization of Individual Components
- ☑ + F4 - 3D Data Slice Visualization
- ☑ + F5 - Support for Viewport Manipulations

(a) Functional requirements



Non-functional Requirements

- ☑ + N1 - Portability between OS/SAGE2 Enviroment
- ☑ + N2 - English Documentation
- ☑ + N3 - Good Rendering and Computing Performance

(b) Non-functional requirements

Figure 42: Requirements model

4.5 Summary

The dataset is loaded into an internal representation. The internal representation has to allow a fast interpolation using the trilinear method. The supported type of grid is primarily the regular grid; nevertheless, the support can be extended to other types of grids, where the use of trilinear interpolation is possible. Chosen filtering techniques include thresholding, manual selection of a subset of the dataset and mainly *visual filtering* in the mapping stage of the pipeline. The enriching process involves transforming the dataset into a format suitable for the selected visualization method.[21] Each method requires different data inputs. The physical accuracy and performance of glyph and surface methods are given by the used interpolation method. The streamline method requires integration, which is performed by higher order numeric integration methods. The subtypes of individual visualization methods chosen for implementation are cone and line glyphs, plane surfaces, and streamlines. It is recommended that the application enables creating various hue-based colormaps. The preferred design of the renderer uses only WebGL as the underlying library, and the computational part of the application utilizes SciPy modules for performing calculations.

4.6 Functional and Non-functional Requirements

Functional requirements target the visualization capabilities of the application. The requirements ordered from the most general to the more detailed include dataset visualization, support for different visualization methods, enabling visualization of individual components, enabling visualization of a 3D slice, and support for viewport manipulations.

Dataset Visualization The application will provide tools for visualizing a dataset. The dataset will have a specific format, and the application will be capable of producing a rendering of the dataset visualization.

[21]Based on the chosen libraries, the used integration method is the Dormand–Prince method from 1.4 Runge-Kutta Method.

Support for Different Visualization Methods The app will support multiple visualization methods. The supported methods are glyph, streamline and layer methods.

Allow Visualization of Individual Components The application will enable visualizing individual vector components.

3D Data Slice Visualization The application will support methods for selecting a subset of the complete dataset and enable visualization of a 3D subset of the complete dataset.

Support for Viewport Manipulations The application renderer will be capable of handling viewport manipulations, such as zooming and changing point of view.

The non-functional requirements target the requests which do not influence the functionality of the application; however, their satisfaction is necessary for a correct functioning of the application.

Portability between OS/SAGE2 Environment The application will be portable between different OS' and SAGE2 environment. The portability can be guaranteed by using web technologies for the rendering part of the application. The supported OS' are Windows, Linux, and macOS.

English Documentation English user documentation will be provided for the application.

Good rendering and Computing performance The application will be designed and implemented in a way that the performance of the application will not cause any usability problems.

4.7 Use Cases

Use cases further illustrate the functional requirements. Figure 43 lists the analyzed use cases.

UC1 — Data Visualization The goal is to create a *visualization* of a new *dataset.* The user utilizes an editor for setting up the pipeline and a renderer for viewing the visualization.

The main script Visualizing the dataset

1. The user wants to visualize a new *vector field dataset.*
2. The user uploads the dataset into the application.
3. The application checks the format of the *dataset.* This script continues if the format of the uploaded file is valid.

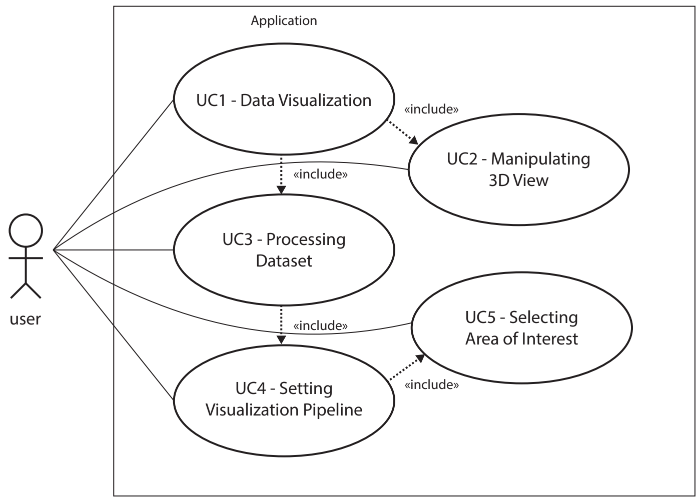


Figure 43: Use case model

4. Include (Processing Dataset)
5. The user opens the output file in the renderer.
6. Include (Manipulating 3D View)

Alternative Route Failed upload

1. This script starts in the third point of the original script.
2. If the format of the uploaded file is not valid, the application updates the status of the *dataset* to invalid.
3. The application leaves the file on the server.

UC2 — Manipulating 3D View The renderer allows viewing and manipulating the output models. The rendered properties of the *visualization* can be changed in a menu.

1. With opened renderer, the user is viewing the *3D models* of a particular *visualization*. The goal is to manipulate the *visualization*.
2. The user navigates to the target action in the menu and changes the current value.
3. The application instantly shows the effects of the performed action.

UC3 — Processing Dataset Processing the dataset involves assembling the description of the *visualization pipeline* and executing the *pipeline*. The output of the *visualization pipeline* is a file containing the calculated geometry.

1. With an opened editor, the user wants to process a *dataset*.
2. Include (Setting Visualization Pipeline)
3. The user launches the execution of the *visualization pipeline*.

4. User awaits the output, the application processes selected *dataset* according to the description of the *pipeline*.
5. The user can directly display the output *visualization* in renderer or download the output stored in a file.

UC4 — Setting Visualization Pipeline The purpose of the *visualization pipeline* is to define the order of performed operations, e.g., selecting *areas of interest*, using *visualization methods*. The description can be assembled in the editor.

1. With an opened editor, the user wants to set up the description of the *visualization pipeline*.
2. The user selects a dataset.
3. Include (Selecting Area of Interest)
4. The user selects a *visualization method* appropriate for the selected *area of interest*. Following combinations are suitable:
   a) A 2D slice is suitable for constructing *layer model*.
   b) A 3D slices is suitable for constructing *glyph* and *streamline models*.
5. Optionally, the user can set up additional attributes for the *3D models*.
6. Optionally, the user can assign a custom *colormap* to the selected *3D models*.
7. In the last step, the user adds or selects an existing output *scene*.
8. The use case is finalized by saving the description of the *visualization pipeline* in the application.

UC5 — Selecting Area of Interest Selecting the *area of interest* enables visualizing only a subset of the dataset. This ability is particularly useful when dealing with big datasets.

1. With an opened editor, the user wants to select the *area of interest*.
2. The user needs to decide whether the *area of interest* is a 2D or 3D slice of space. All methods can operate inside any *area of interest*. However, the layer method produces only a model of a single layer for a given area. The glyph and streamline methods are more suitable for 3D data slices
3. If needed, the user specifies the sampling of the *area of interest*.
4. The editor displays a representation of the selected *area of interest*.

4.8 Domain model

This section presents the classes in the domain model. The domain model illustrates objects from the real world and relations between them related to the topic of visualization. The relations are further illustrated in figure 44.

Vector Field Dataset The class represents a package of data. The data was obtained as a result of measurements or simulations. The dataset consists of positions and values. The *vector field values* attribute represents the values of the contained vectors. *Data positions* are the spatial positions of individual vectors. The dataset has the following states:

Checking The application is performing the validation.
Valid The dataset can be used by the application.
Invalid The format is corrupted, and the application cannot use the dataset.

Visualization Pipeline The class represents a set of methods applied to a datatset. Each pipeline contains a *description* of the visualization pipeline used for the processed dataset. The processing is stored in a notebook file, and during the processing, various visualization methods are used. The processing has the following states:

Created The processing was created, might be empty.
Running The processing is running; the methods are performed according to the included description.
Finished The processing is completed, 3D models for used methods are available.

Interpolation Method The class represents an interpolation method used in a visualization method. The methods differ by order of *accuracy*.

Integration Method The class represents an integration method used for streamline construction. The methods differ by order of *accuracy*. The integration is performed for a time range specified by attributes *start* and *end time*.

Area of Interest The class represents an area of operation for connected visualization methods. Preferred areas of interest lie inside the vector field dataset positions; however, it is possible to construct other areas. The area is a rectangular space with a cuboid shape. It is defined by the *position* of one of its corners and the *dimensions* of its edges.

Visualization Method The class represents visualization methods used in the processing. Each visualization method uses an interpolation method to approximate the data between known values. The visualization method works in a selected area of interest.

Streamline Method The class represents a use of the streamline method. The method constructs streamlines starting based on the *sampling* of the area of interest. The trajectory of each streamline is obtained from an integration method.

Glyph Method The class represents the use of the glyph visualization method. The method selects the values based on the *sampling* of the area of interest and maps the values onto a glyph geometry.

Layer Method The class represents the use of visualization on a plane surface. The surface is a slice of the area of interest and is defined by the *slice axis* and *slice coordinate.* The method performs interpolation in a given area and maps the values onto a plane geometry.

3D Model The class represents a model created by a particular visualization method. Besides geometry, each model stores filtered *values* and *positions* which are mapped onto the geometry. *Color mode* refers to the type of used encoding; the color can encode the absolute value of stored values or selected individual components.

Streamline Model The class represents a set of 3D streamline objects. As a product of the streamline method, every streamline model contains *integration times* (timestamps). Every timestamp corresponds to a single *point* and *value*; in other words, the number of timestamps is equal to the number of *points* or *values.* The class has an attribute specifying the *thickness* of the streamlines and a *segment geometry* — a mesh corresponding to a segment between two points.

Glyph Model The class represents a set of 3D glyph objects. Each glyph has a the same initial *size* and utilizes a basic *glyph* geometry. The geometry of individual glyphs is transformed according to the represented value.

Layer Model The model represents a single layer object. Additionally, the layer only contains a specification of the *layer geometry.*

Colormap The class represents a single colormap, specifying the range of colors used for value encoding. The *sampling* attribute represents the number of used colors. Every colormap utilizes two colors at a minimum. The attribute *color order* gives the order of colors.

Color The class represents a single color and stores the *color values.*

Scene The 3D models are assembled into scenes. The models contained in a scene determine the *scene dimensions.* The contents are scaled to fit the *scaled dimensions.* The data mapped onto 3D models as a set contain minimal and maximal values, which are extracted and marked as scene *limit values.*

Visualization The class represents an entity performing the visual rendering of the scene. The visualization has a *viewport size* and renders the scene based on the *point of view.*

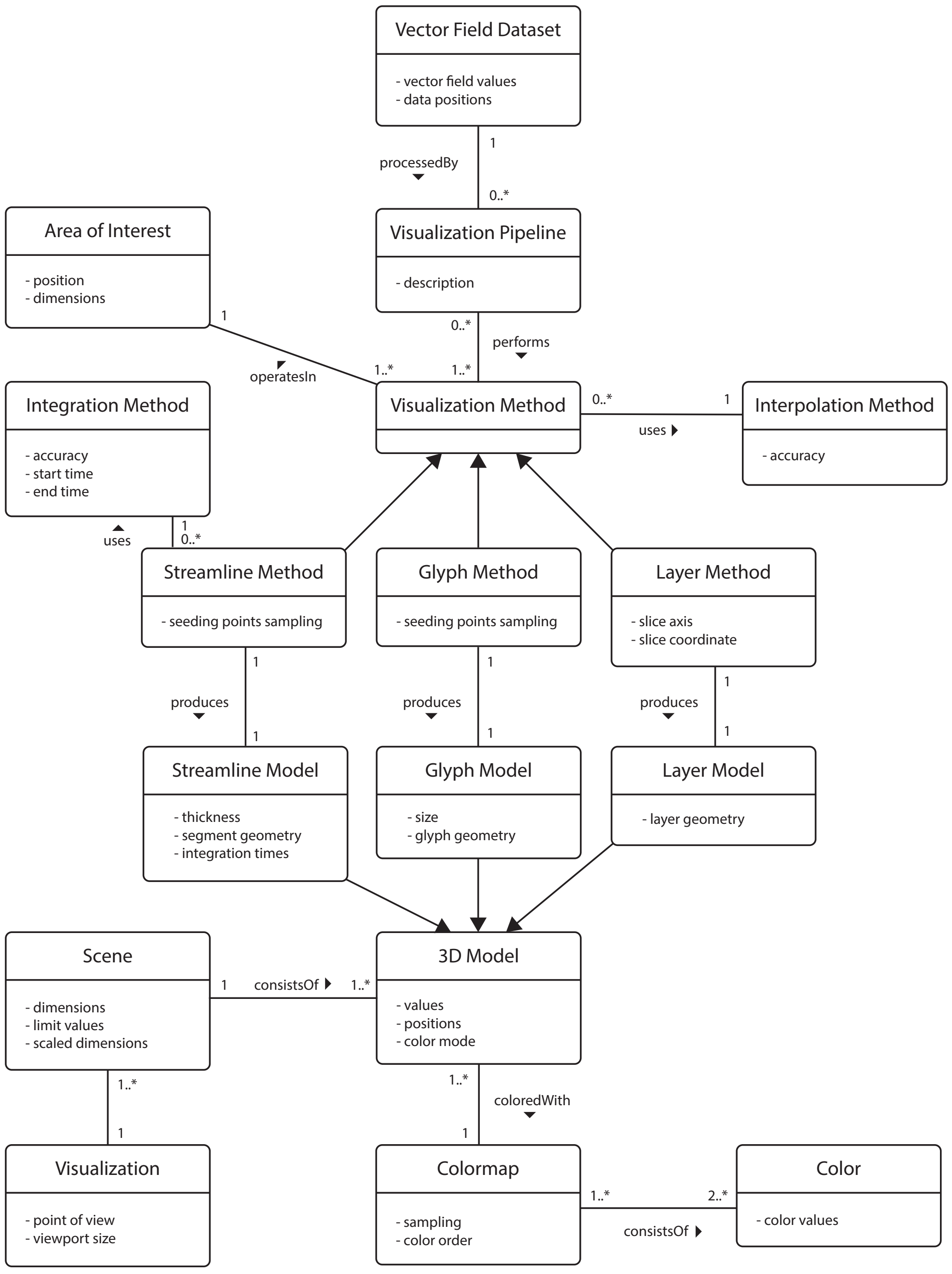
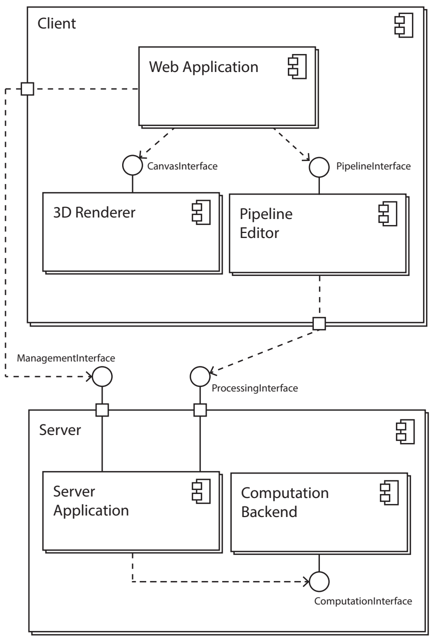


Figure 44: Domain model

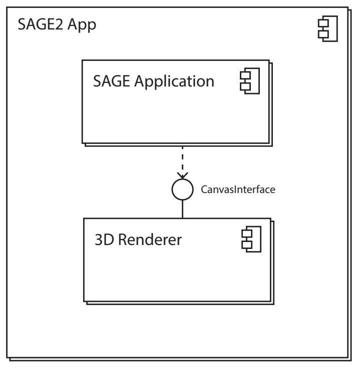
Chapter 5

Design

The design of the application mostly follows the relationships established in the domain model. The app is designed as a server-client project. This design allows separating the intensive computations from the renderer and other additional components. The layout and relationships between individual components are illustrated in the figure 51. Additionally, the solution for SAGE2 environment is illustrated in the same figure. The design of the SAGE application introduces one additional component, which is closely similar to the web application component.



(a) Web app



(b) SAGE2 app

Figure 51: Components and interfaces

5.1 Computation Backend

The computation backend component implements the first two stages of the visualization pipeline — loading and enriching. The overall structure of the computation backend is illustrated in the figures 52 and 53. The structure of the computation component isolates the underlying mathematical operations (numeric module) from pipeline management (editor module).

5.1.1 Editor module

The structure of the editor component is based on the layout of the visualization pipeline. The pipeline consists of atomic procedures, e.g., loading dataset, performing an integration. Furthermore, the pipeline can be designed to follow a fixed order of operations or support dynamic ordering of procedures.[22]

Fixed pipeline Fixed pipelines process all datasets with the same order of operations. Since the order of the procedures is known in advance, the design of such pipeline can be quite straightforward.

Dynamic pipeline The other option is to implement a dynamic pipeline. This design allows determining the order of procedures at runtime; thus, the user can choose the procedures and their order according to his or her needs. However, this design is more intricate.

For customisability reasons, the design of a dynamic pipeline is used. The pipeline is best represented as a directed acyclic graph, as is illustrated in figure 54. The nodes in the graph represent atomic operations, and edges represent the flow of input and output data between individual operations. The sources in the graph correspond to procedures requiring no input, such as loading dataset, or specifying the area of interest. The sinks represent the only process with no data output — the rendering procedure. The table 51 lists the supported procedures, corresponding data structure representations, and the required inputs and outputs. Some of the procedures can take a mix of different types as an input. Round brackets indicate the input is optional. When all inputs are optional, at least one is required, regardless of its kind.

Pipeline processing The pipeline processing is demonstrated in the algorithm 1. First, the algorithm checks the pipeline description and constructs the graph. In the second step, a topological sort is performed to check for cycles and determine the order of node executions. In the last stage, the algorithm iterates over the sorted nodes. For each node, the inputs of the previous nodes are linked to the current node and the execution routine is called.

[22]Terms *operations* and *procedures* refer to the same subject.

procedures	data structure (node)	input types	output type
loading dataset	DatasetNode	–	dataset
selecting 3D slice	PointsNode	–	points
selecting 2D slice	PlaneNode	–	plane
thresholding	FilteringNode	dataset, points	points
constructing glyphs	GlyphsNode	dataset, points	glyphs
constructing streamlines	StreamlineNode	dataset, points	streamlines
constructing layer/plane	LayerNode	dataset, plane	layer
modifying glyph geometry	GlyphGeometryNode	glyphs	glyphs
modifying streamline geometry	StreamlineGeometryNode	streamlines	streamlines
scaling values or positions	ScaleNode	(glyphs), (streamlines), (layer)	(glyphs), (streamlines), (layer)
translating values or positions	TranslateNode	(glyphs), (streamlines), (layer)	(glyphs), (streamlines), (layer)
adjusting the colormap	ColorNode	(glyphs), (streamlines), (layer)	(glyphs), (streamlines), (layer)
rendering output	DisaplyNode	(glyphs), (streamlines), (layer)	–

Table 51: Operations, corresponding nodes and in/out types

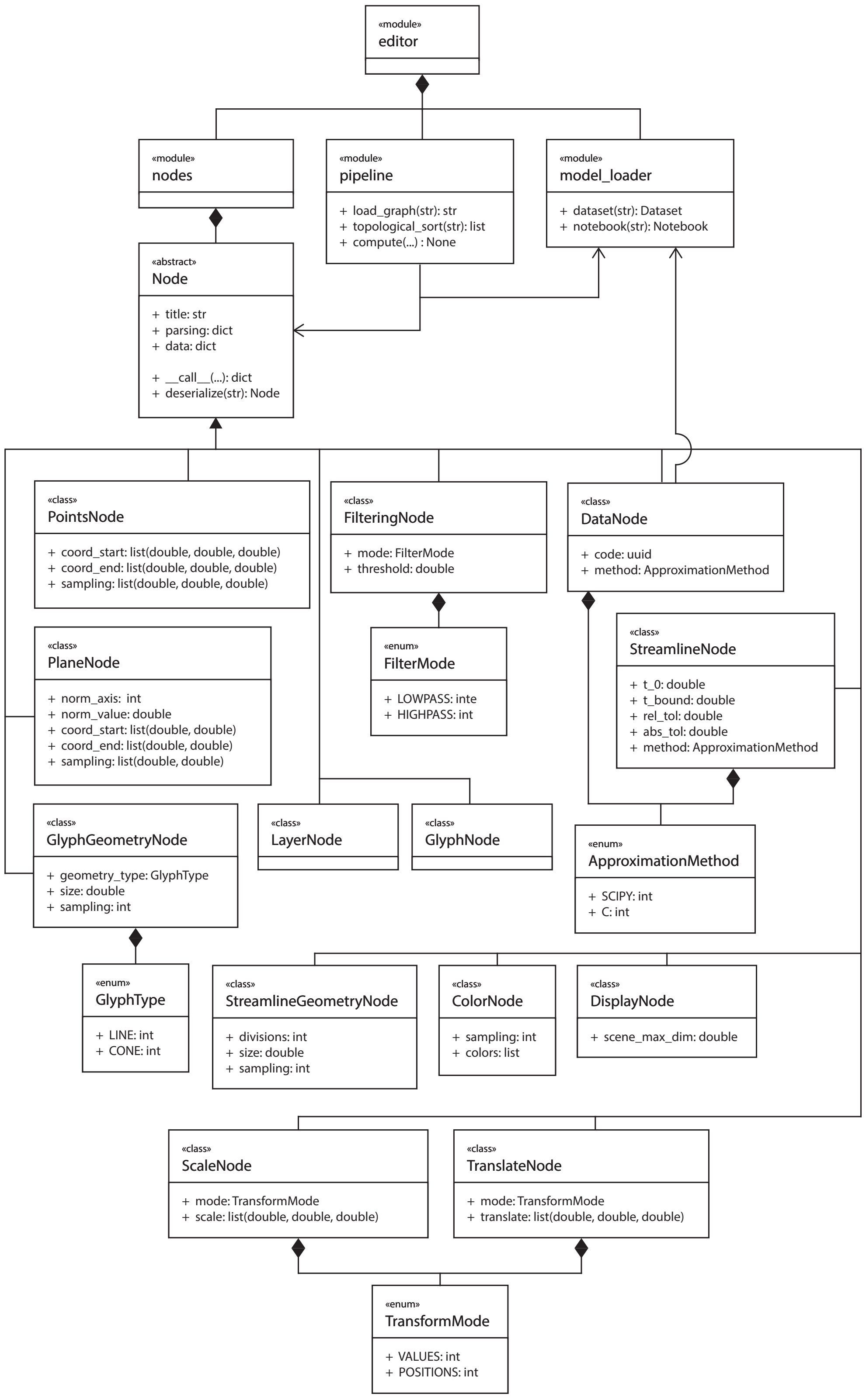


Figure 52: Design of the editor module

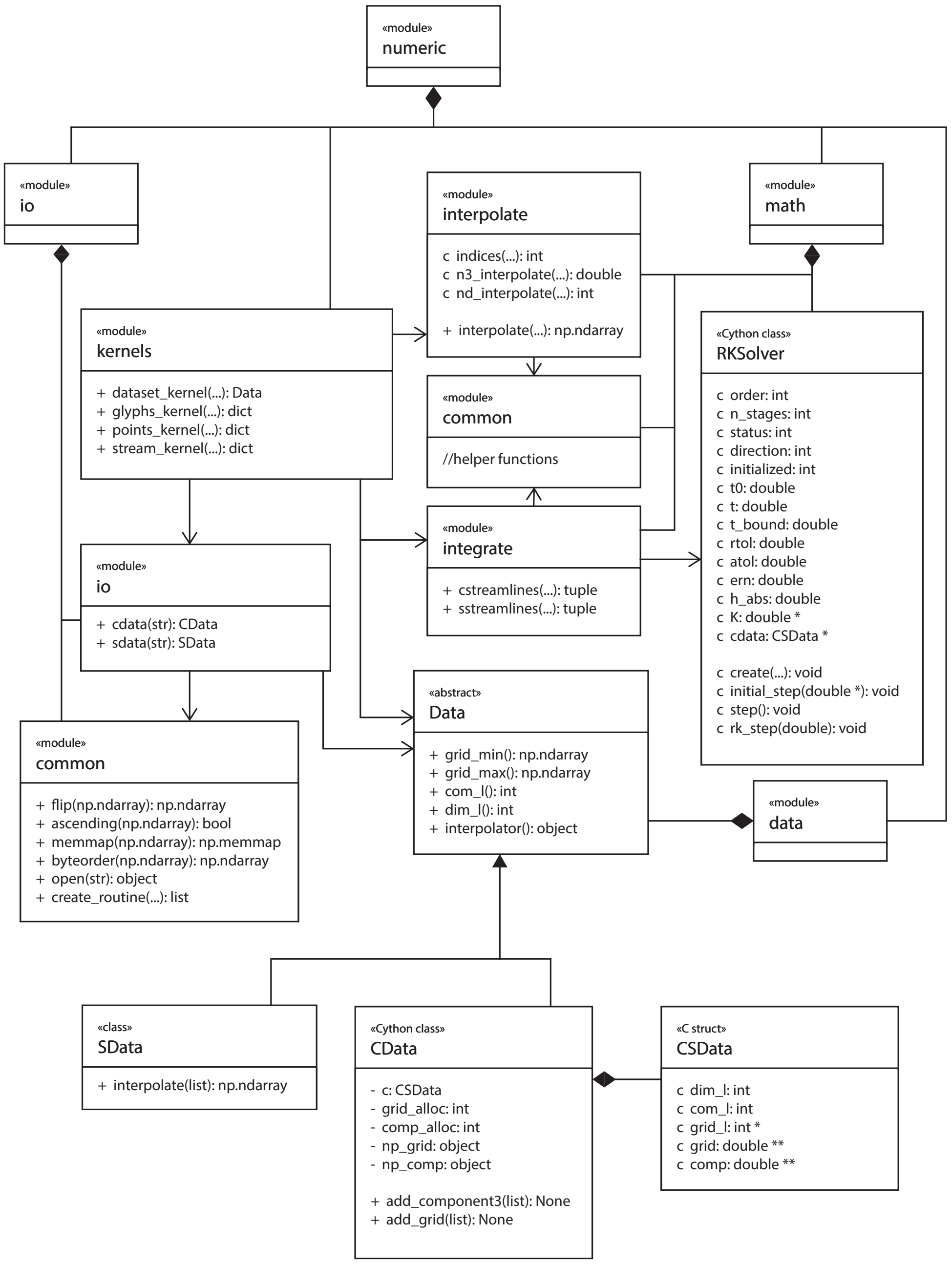


Figure 53: Design of the numeric module

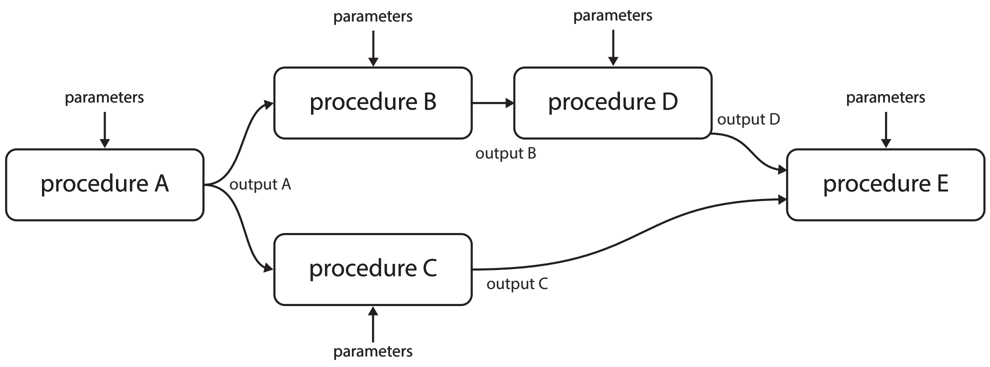


Figure 54: Illustration of pipeline representation as a graph

```
Input: pipeline description
Output: none, the display node saves the data to disk

class {
    inputs — input node ids
    outputs — output node ids
    parameters — additional node paremeters
} Node

nodes ← load_graph(description)
order ← topological_sort(nodes)
outputs ← []

foreach $o \in order$ do
    inputs ← $\emptyset$
    foreach $in \in nodes[o].inputs$ do
        $inputs \leftarrow inputs \cup outputs[in]$
    end
    $outputs[o] \leftarrow nodes[i]$.call(inputs)
end
```

Algorithm 1: Pipeline execution

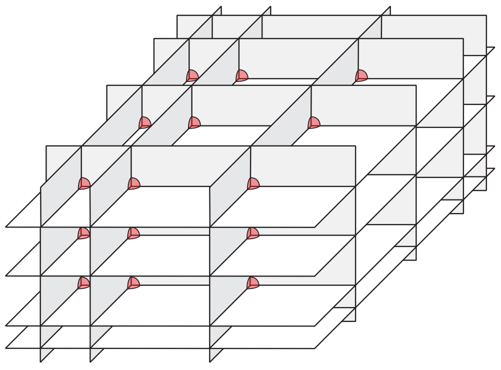


Figure 55: Illustration of the supported grid type

5.1.2 Numeric module

The numeric module can be further divided into three parts — data representation, interpolation, and integration. The goal is to enable real-time calculations and quickly provide physically accurate results. Therefore, all functionalities have to be optimized for both memory and performance efficiency. One approach is to use an existing implementation of the required algorithms. Based on the previous analysis, the decision is to utilize NumPy and SciPy functions. The early testing included the following procedures:

1. interpolating with a grid interpolator *RegularGridInterpolator*
2. constructing streamlines (solving initial value problem with *solve_ivp*) in a field represented by the grid interpolator

Early testing revealed that the SciPy interpolator is sufficient for interpolating a vast number of points at once; however, the efficiency is inversely proportional to the number of interpolated points. Unfortunately, a single value is interpolated during every integration step multiple times. For performance reasons, it is necessary to provide an alternative to SciPy implementation. The alternative solution involves implementing custom interpolator, solver, as well as a custom class for data representation.

Data Representation To simplify the interpolation, the format of the new underlying data structure represents a cartesian space sampled by parallel planes with a plane normal equal to one of the space axes. The sampled points lay in the intersection of three planes. The data structure is illustrated in figure 55. As the figure suggests, this definition allows for non-uniform sampling along the axes but preserves some regularity of the overall structure.

Interpolator The interpolator has to fully replace and speed up the functionality of the original SciPy interpolator for three-dimensional data. The

```
Input: $dataset$, $points$
Output: interpolated $values$

class {
    $values$ — array with dataset values
    $points$ — array of sampling points along individual axis
    $sampling$ — the numbers of sampling points along each axis
} $Dataset$

Function $guess\_index(p,\ points\ sampling)$ is
    $min \leftarrow points[0]$
    $max \leftarrow points[sampling - 1]$
    $step \leftarrow (max - min)/sampling$
    $index \leftarrow (p - min)/step$
    return $index$
end

$values \leftarrow []$
foreach $p \in points$ do
    $local\_cell \leftarrow []$
    foreach $i \in 0..2$ do
        $sampling \leftarrow dataset.sampling[i]$
        $points \leftarrow dataset.points[i]$
        $index \leftarrow$ guess_index($p$, $points$, $sampling$)
        if $p >= points[index] \wedge p <= points[index + 1]$ then
            $local\_cell[i] \leftarrow index$
        else
            $local\_cell[i] \leftarrow$ binary_search($points[i]$, $p[i]$)
        end
    end
    $values \leftarrow$ cell_interpolate($dataset.values$, $local\_cell$, $p$)
end
return $values$
```

Algorithm 2: Interpolator

new interpolator follows the algorithm 2. The lookup is sped up by binary search; thus a single interpolation can be achieved in $\mathcal{O}(log(x_{sampling}) + log(y_{sampling}) + log(z_{sampling}))$. Additionally, the interpolator is optimized to look up the local cell in regularly sampled datasets in constant time. The optimization is possible thanks to a procedure which tries to estimate the position of the local cell.

```
Input: $dataset$, $points$, $t_0$, $t_{bound}$
Output: $streamlines$

$solver \leftarrow RKSolver$
$solver$.init($dataset$, $t_0$, $t_{bound}$)
$streamlines \leftarrow []$
foreach $p \in points$ do
    $positions \leftarrow [p]$
    $times \leftarrow [t_0]$
    $solver$.init_step($p$)
    while not $solver.finished$ do
        $solver$.step()
        $positions \leftarrow solver.position$
        $times \leftarrow solver.time$
    end
    $streamlines \leftarrow [position, times]$
end
return $streamlines$
```

Algorithm 3: Integration

Integration and Solver The algorithm 3 used for integration has a similar structure to the original SciPy function *solve_ivp*. The design of the original solver used for integration is satisfactory; however, the SciPy RK solvers are implemented purely in Python and lack the performance of the solvers implemented in compiled languages. One option is to use already existing solver written in a compiled language. The other option is to port the Python code into a compiled language. In both cases, the solver has to be compatible with the new data structure and take advantage of the new fast interpolation routine.

To improve the performance, probably not the simplest but an elegant solution is to port the Python code into Cython, a Python extension enabling memory management and declaration of types. The Cython code can be directly translated into C. In the purest form, the Cython can eliminate the use of Python interpreter, which dramatically improves the performance. Since the code has to be optimized for performance, the final decision is to implement critical parts of the code in Cython.

Enriched Dataset Format Since it is necessary to transfer the computed data, the enriched dataset should have a well-defined format. The format has to be flexible and easy to process since the third stage of the pipeline will be performed in the browser by the renderer. The final choice is to use JSON format. The arrays of floating point numbers can be stored as Base64 encoded strings, which disables user inspection of the values but ensures that no precision is lost by conversion.

5.2 Server Application

The server application serves as a bridge between the computation backend and the client-side application. The responsibilities of the server application involve managing dataset uploads, storing the datasets and pipeline descriptions and managing the queue of computation requests. Since the computation backend is written in Python, the server application is designed as a Django application to enable a smooth connection between these two components. The figure 56 presents the database model of the server application.

The internal Django infrastructure manages user requests. The server application contains the database model and serves individual views of the client-side application. Another Django extension handles the communication between clients and computation backend — Django Channels, which enables web socket communication. The user requests fall into one of the following categories – management request or processing request.

Management Requests The management requests are always HTTP requests and involve creating, updating, and deleting a database model, or serving a view. The management requests are managed via the management interface.

Processing Requests There are only three distinct types of processing requests - start or stop a computation and list possible request types. The processing requests are served strictly via web sockets since the server needs to send back the information about the progress of the computation.

Internally, the server keeps a queue of the computation requests and processes them one by one. The server application is designed to be entirely separated from the computation backend and runs in a separate process, which ensures that any possible problems caused by the computation backend do not bring the server application down. The communication between the server app and computation backend is performed via web sockets.

5.3 Web Application

The web application consists of views and controllers. The web application is designed as a single-screen application; however, the single view is divided into multiple subviews. The notable subviews are dataset and notebook views. Each listed subview contains a set of control elements which are also treated as separate views. This hierarchy allows assigning an independent controller to each view, which simplifies the overall design.

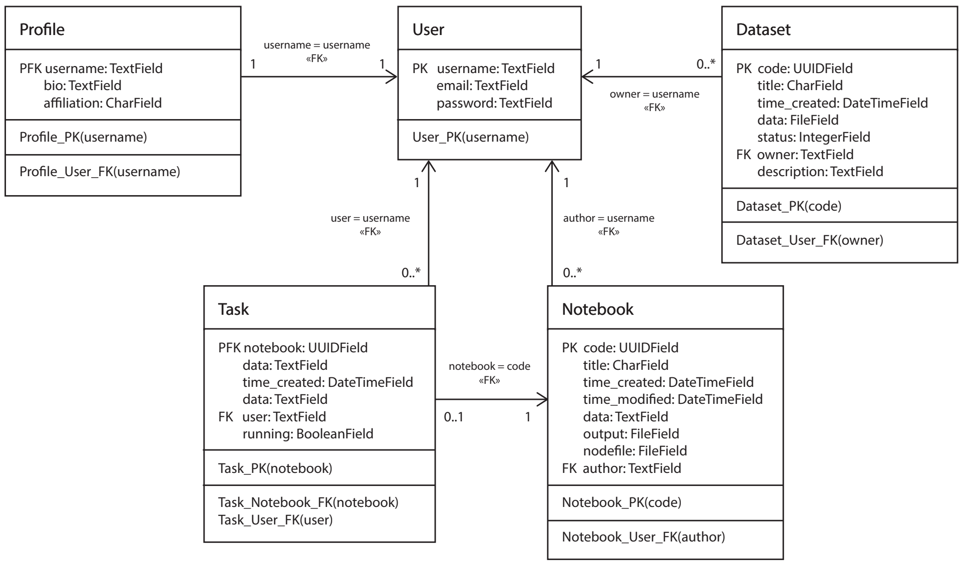
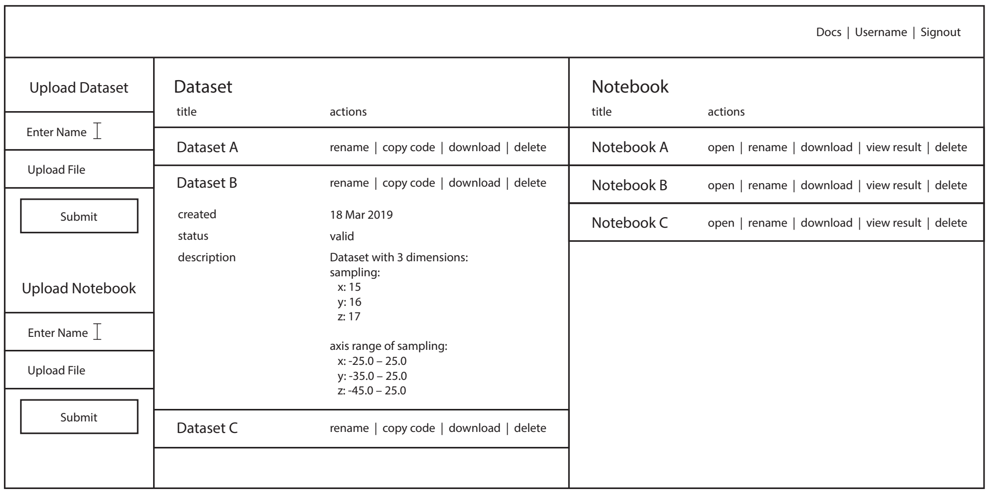


Figure 56: Database model of the server application


Docs | Username | Signout

Upload Dataset

Enter Name

Upload File

Submit

Upload Notebook

Enter Name

Upload File

Submit

Dataset

title	actions
Dataset A	rename | copy code | download | delete
Dataset B	rename | copy code | download | delete
created	18 Mar 2019
status	valid
description	Dataset with 3 dimensions: sampling: x: 15 y: 16 z: 17 axis range of sampling: x: -25.0 – 25.0 y: -35.0 – 25.0 z: -45.0 – 25.0
Dataset C	rename | copy code | download | delete

Notebook

title	actions
Notebook A	open | rename | download | view result | delete
Notebook B	open | rename | download | view result | delete
Notebook C	open | rename | download | view result | delete

Figure 57: Mockup of the main application screen

The figure 57 contains a mockup of the single screen with labeled sub-views. The second figure 58 displays the structure of the web application design. Since the application is supposed to function as a website, the views are implemented as HTML elements and the controllers as scripts in JavaScript.

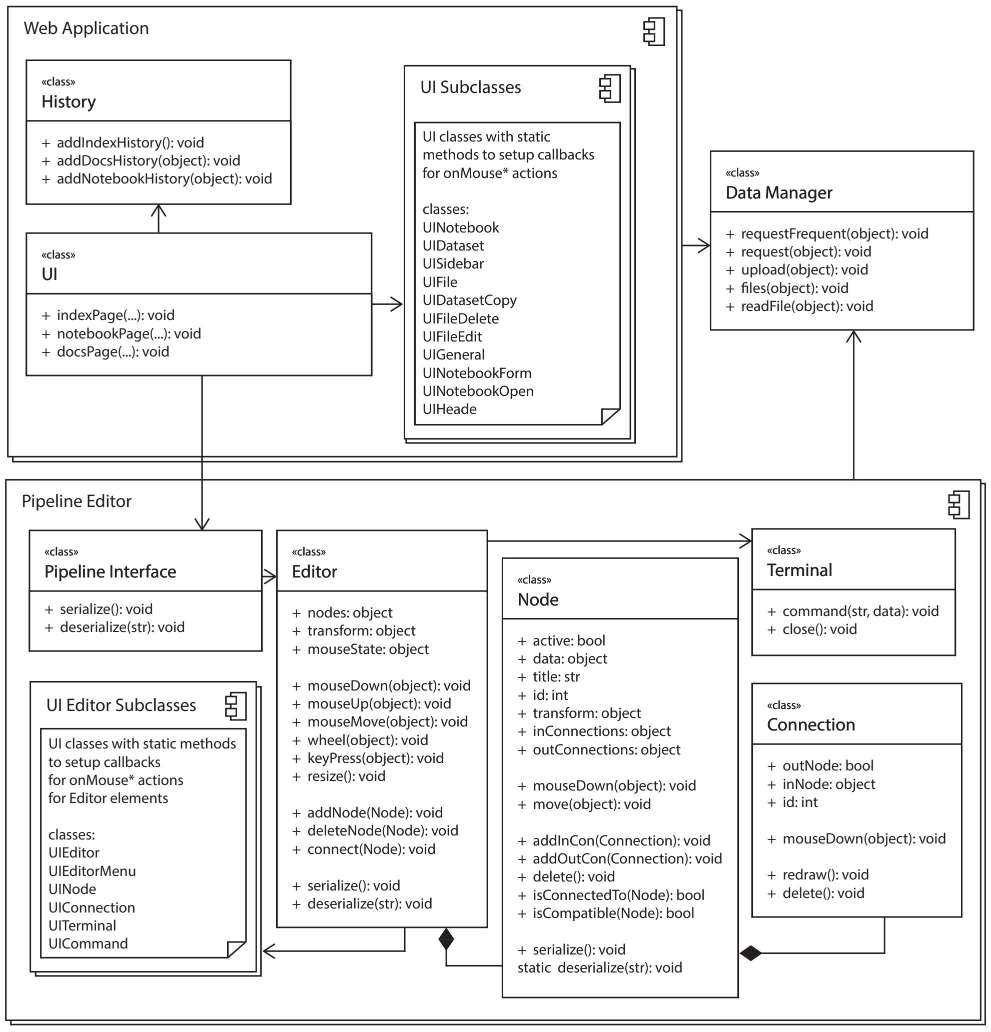


Figure 58: Design of client components web application and pipeline editor

5.4 Pipeline Editor

The pipeline editor consists of two views — the graph editor and a terminal. It is isolated from the rest of the web application and serves a single purpose — it allows assembling the description of the dynamic visualization pipeline.

Graph Editor The design of the editor is based on a metaphor of the pipeline being a graph. Each procedure is represented as a single visual node, and the editor allows manipulating and dynamically reconnecting compatible nodes. Besides the input data from other procedures, each procedure requires additional parameters. These parameters are not treated as data flowing from one node to another. This locality needs to be represented by the design of the user interface. The figure 59 illustrates the

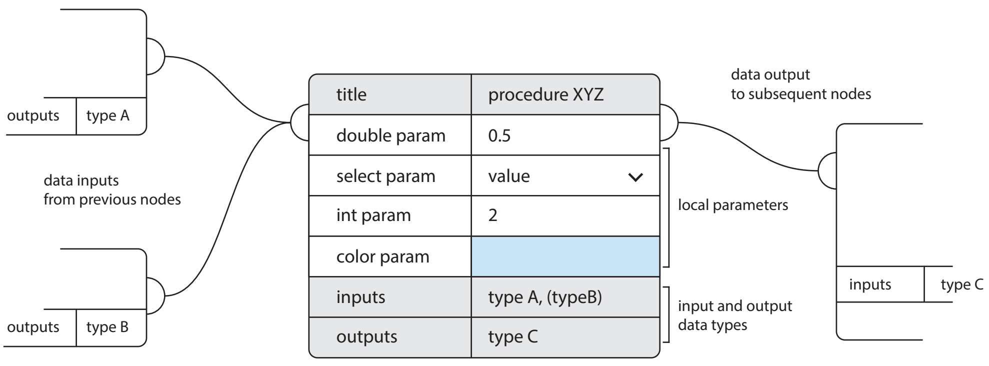


Figure 59: Mockup of the UI of a single node from the pipeline editor

visual design of a single node from the pipeline editor. The nodes have to follow the design structure of the computation backend which also includes the rules about supported input and output types. Therefore, the visual nodes also display required and optional data types.

The figure 58 illustrates the design structure of the pipeline editor. The underlying controllers take care of manipulating nodes, adding connections and updating values of the parameters. A significant responsibility of the editor is serializing and deserializing the description of the pipeline. The nodes are designed as JavaScript objects; however, it is necessary to store only a selection of the object's attributes.

The serialization produces an array of objects which can be stored in a JSON format. The inverse operation involves reconstructing the visual representation of the graph from the JSON format.

Terminal The terminal serves for communication with the server application and the computation backend. The terminal sends the entered command to the server along with the serialized description of the pipeline and displays the server response. The request is sent to the processing interface of the server application.

5.5 Renderer

The renderer component is one of the more complex components. As its title suggests, the component enables rendering of the visualization. The component accepts the enriched dataset produced by the computation backend. The entire renderer is encapsulated in a single class. The main class further contains three member classes which are responsible for handling the user input, managing user interface, and drawing on the screen. The first two tasks are not very technically challenging.

Input Interface The interface class handles the user input. The class keeps a record of user actions such as key press and mouse drag and makes this data available for other classes. The user inputs connected to scene manipulation are used during each frame update.

User Interface The user interface is encapsulated in a single class which further distributes the control over individual parts among a group of other classes.

Graphics The graphics class performs the most technically challenging operations. The class is responsible for rendering the enriched dataset, and the used algorithms have to balance memory and performance efficiency. The lowest level graphics API available — WebGL — is used to enable hardware acceleration of the rendering.

WebGL Principles The rasterization pipeline implemented by WebGL has fixed and programmable parts. The basic use of WebGL for static models can be explained in two steps. The first step involves storing data into buffers. In the second step, programs called shaders take a set of data from one or more buffers and process it. The renderer utilizes only two types of shaders — vertex shader and fragment shader. The performance boost is gained by running the shaders in parallel on dedicated hardware. Further reading about WebGL and OpenGL can be found in [33].

5.5.1 Visualization methods

Based on the brief introduction of WebGL, there are two design problems which need to be addressed in order to utilize the WebGL API.

- How to map the dataset onto a geometry stored in the buffers?
- How do the shaders handle the buffer data?

The renderer has to support three unique visualization methods — glyphs, streamlines, and planes. Each method requires a slightly different approach. The design has to reflect what the eriched dataset contains and what the visualization method is from the perspective of the renderer.

Glyphs The dataset represents glyphs with points, corresponding values, and some additional metadata about colormap and the geometry type. The glyphs are a repeated instance of the same object (glyph) scaled and colored according to the local value of the vector field. The most straightforward solution is to create a model of a single glyph, transform the model for every position and value, and store the transformed geometry along with the corresponding color data. The shader would take the transformed geometry and only apply the predetermined color. This approach is very ineffective, more efficient approach is to use a technique called instancing. This technique allows for rendering the same model multiple times with slightly different parameters. The changing parameters have to be stored in a separate buffer and WebGL needs to know how often it should swap the current set of parameters with a new one. The spatial geometry transformation is performed in the shaders along with the color interpolation.

Streamlines The streamlines are represented in the dataset as a list of points, values, and timestamps. A simple representation of a streamline could be implemented using only lines. However, this visualization is not sufficient; it is difficult to determine the actual position of a single line in 3D space. Thus, the streamlines have to be visualized with 3D curves of non-zero thickness. There are several ways how to implement 3D curves; the design chosen here is inspired by a method called *extrusion along a curve*.

The curve is divided into several segments. The algorithm processes each segment individually. The segment is bounded by a pair of points — starting and ending point. A tube can be used as a geometry for the segment. There is a weight assigned to each vertex of the geometry, which indicates the position of the vertex relative to the ending point of the segment. The distribution of weights is illustrated in the figure 510. The algorithm is trying to *bend* the geometry along the segment of the curve. First, the algorithm approximates the local tangent vector and the local position of the curve. In the second step, the vertex is translated to make the tangent vector perpendicular to the radius of the tube. Finally, the vertex is translated according to the local position of the curve.

A simple version of the algorithm linearly approximates the curve; however, there are a few additional improvements possible. The geometry of the segment can be further divided along the curve. The vertices between both ends can be transformed with the same algorithm. If a more sophisticated method for approximation of the local values is used, e.g., the cubic interpolation, the additional sampling of the curve should produce a more smooth reconstruction of the original streamline. The algorithm can be further improved to allow adjusting the thickness of the curve.

Further, the endpoint timestamps can be used for adjusting the rendered portion of the streamline. The vertices outside the visible range are filtered out by determining the visibility of each vertex. The visibility information is passed into the fragment shader, and the fragments lying outside of the visible region are discarded.

The final design of the algorithm had to be additionally modified. The cubic interpolation requires two points, two tangents, and the local factor for interpolating the local point. The points are obtained during the integration of the streamline. The tangents are the vectors of the vector field

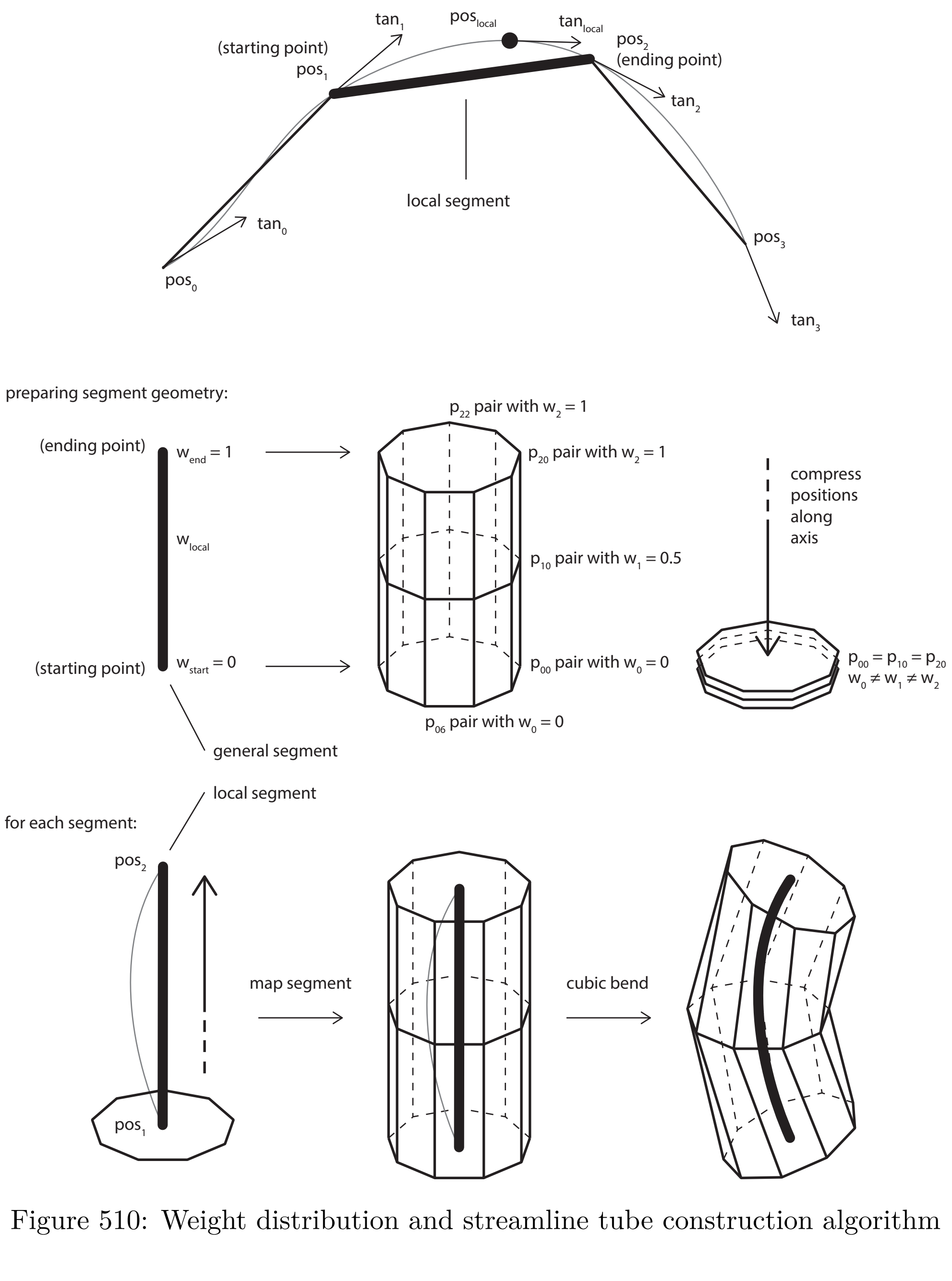


Figure 510: Weight distribution and streamline tube construction algorithm

```
Input: $pos_{vertex}$ — vertex position
       $w_{local}$ — local weight
       $tan_0$, $tan_1$, $tan_2$, $tan_3$ — 4 surrounding tangents
       $pos_0$, $pos_1$, $pos_2$, $pos_3$ — 4 surrounding points
       $t_0$, $t_1$, $t_2$, $t_3$ — timestamps of 4 surrounding points
       $thickness$ — radius of the streamline
       $t_{start}$, $t_{end}$ — visible range of timestamps
Output: the real-world position of the vertex mapped along the
        curve

$t \leftarrow \mathbf{cubic_{scalar}}(t_0, t_1, t_2, t_3, w_{local})$
$visibility \leftarrow t_{start} \leq t \wedge t \leq t_{end}$

$tan_{start} \leftarrow 0.5\,|(pos_2 - pos_0)| \cdot \mathbf{normalize}(tan_1)$
$tan_{end} \leftarrow 0.5\,|(pos_3 - pos_1)| \cdot \mathbf{normalize}(tan_2)$
$pos_{local} \leftarrow \mathbf{cubic_{vector}}(pos_1, pos_2, tan_{start}, tan_{end}, w_{local})$
$tan_{local} \leftarrow \mathbf{cubic_{derivate}}(pos_1, pos_2, tan_{start}, tan_{end}, w_{local})$

$p \leftarrow pos_{vertex} * thickness$
$p \leftarrow \mathbf{rotate_{tan}}(p, tan_{local})$
$p \leftarrow p + pos_{local}$

return $p$
```

Algorithm 4: Streamline construction algorithm

in the points obtained during integration. The problem arises in turbulent vector fields. The higher order numeric integration methods produce physically accurate results even in more turbulent vector fields; however, the cubic interpolation is not capable of accurately estimating the curve trajectory based on the interpolated tangents. Therefore, the design of the final algorithm is adjusted to utilize four surrounding points and values instead of two. The complete algorithm with the additional improvements is presented in algorithm 4 and 5.

Layers The dataset stores the layer as a grid of points and corresponding values. The layer consists of rectangular cells which can be again efficiently rendered with instancing. However, a more straightforward and equally efficient approach is to use a different method called elements rendering. An additional buffer of indices has to accompany the buffer of positions and values. A sequence of four numbers in the additional buffer corresponds to the indices of the positions and values forming a single rectangular cell.

```
Function $cubic_{scalar}(v_0, v_1, v_2, v_3, w)$ is
    $c_1 \leftarrow v_1$
    $c_2 \leftarrow -0.5v_0 + 0.5v_2$
    $c_3 \leftarrow v_0 - 2.5v_1 + 2v_2 - 0.5v_3$
    $c_4 \leftarrow -0.5v_0 + 1.5v_1 - 1.5v_2 + 0.5v_3$
    return $(((c_4w + c_3)w + c_2)w + c_1)$
end

Function $cubic_{vector}(v_0, v_1, t_0, t_1, w)$ is
    $c_1 \leftarrow v_0$
    $c_2 \leftarrow t_0$
    $c_3 \leftarrow -3v_0 + 3v_1 - 2t_0 - t_1$
    $c_4 \leftarrow 2v_0 - 2v_1 + t_0 + t_1$
    return $(((c_4w + c_3)w + c_2)w + c_1)$
end

Function $cubic_{derivate}(v_0, v_1, t_0, t_1, w)$ is
    $c1 \leftarrow t_0$
    $c2 \leftarrow -6v_0 + 6v_1 - 4t_0 - 2t_1$
    $c3 \leftarrow 6v_0 - 6v_1 + 3t_0 + 3t_1$
    return $((c_3w + c_2)w + c_1)$
end
```

Algorithm 5: Streamline construction algorithm — functions

As the figure 511 illustrates, the methods are represented by individual classes. Each of the methods contains additional primitives — colormaps, bounding boxes, and labels.

Colormaps Colormaps are used for color coding the vector components. There are four color coding modes — one for each of the components of the vector and one for the magnitude of the complete vector. The current colormap is always displayed on the screen along with labels describing the limit values.

Bounding Boxes The purpose of bounding boxes is to mark the area containing a single visualization method. The box enables more natural orientation and provides a space for labels.

Labels The labels inform the user about the position of the geometry in space.

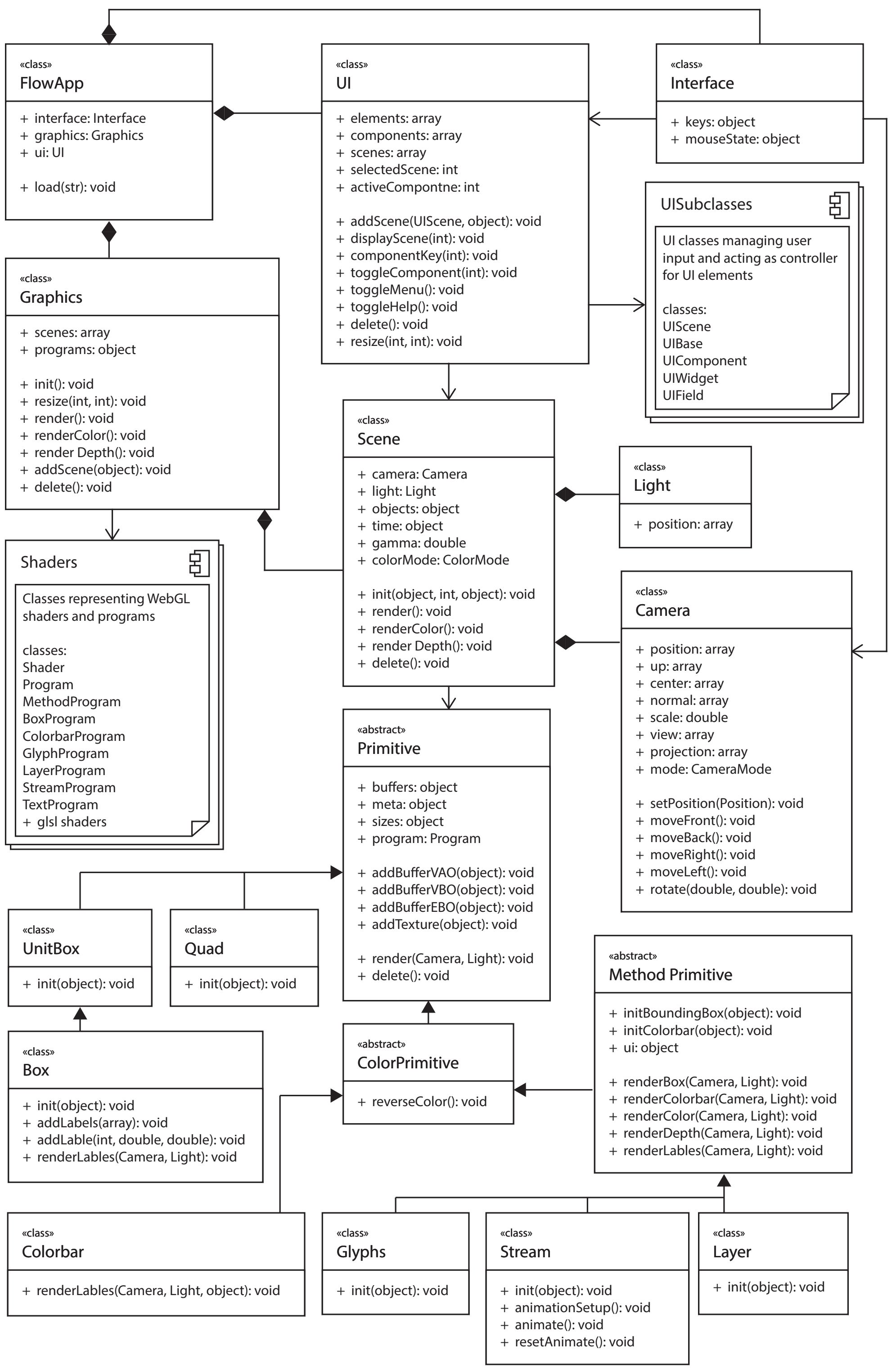


Figure 511: Design structure of renderer component

5.5.2 Significance value

Section 4.2.1 Filtering introduced a method called *visual filtering.* The method uses scaling and color coding to highlight the visual elements representing significant values. The importance is expressed with a new unit called significance value.

When all of the values represented by the visual elements are either positive or negative, the significance can be calculated as

$$\sigma = \frac{value - value_{max}}{value_{max} - value_{min}} . \tag{5.1}$$

When the ends of the range fall into different sides of the number line, the calculation can be kept the same or an alternative formula is available. The alternative maps the significance of zero value to sigma 0.5. The formula for the alternative calculation is the following expression.

$$\sigma = \frac{value + max(value_{max}, -value_{min})}{2\, max(value_{max}, -value_{min})} \tag{5.2}$$

The alternative approach could be more suitable for displaying a single component of vectors. The design of the application allows the user to use any of these two modes. The significance value is used for scaling the geometry of glyphs, and for setting the color of all models. The color is obtained by mapping the significance value onto the colormap.

5.5.3 Rendering strategy

The most straightforward strategy is to draw on the screen every frame. The disadvantage of this approach is that the renderer draws the frame even when the image has not been changed. This behavior might slow other tasks running on the computer and affect the usability of the renderer in combination with other computationally demanding application.

The solution is to check every frame whether the update is necessary. The update is ultimately needed in two situations — when the camera has been moved, or a specific parameter of any of the displayed models has been changed. The camera already keeps internally the state from the previous frame in order to enable smooth camera transitions. Thus, checking if the viewpoint or angle has changed is no problem. Checking if any value has changed can be trivially enabled by introducing the *changed* flag to each object.

5.6 SAGE Application

The last component of the project is a layer allowing the renderer to run in the SAGE2 environment. The component acts as a bridge between SAGE2 server and the renderer, utilizing the canvas interface. The application has to manage three tasks — reading local files with enriched datasets, pass the interface inputs from the server to the renderer, and synchronize the state of individual SAGE2 clients.

The enriched datasets are stored locally on the SAGE2 server, and the SAGE2 interface provides a method for loading local files. The user inputs are picked up by the SAGE2 app and passed to the renderer through the canvas interface. The state synchronization between clients should be possible with the SAGE2 internal state variable. The internal SAGE2 mechanisms should keep the state variable consistent in the entire grid of clients by distributing the most recent value via the server. However, unexpected behavior was encountered during the testing of this feature. The clients do keep their local state variable consistent during repeated restarts and the application loads with the last saved state; however, the state is not synchronized between individual clients. To solve this problem, an additional mechanism is introduced. One of the clients is tagged as a master client. The state of the master client is distributed via a broadcast every three seconds, and the other clients receive the synchronized value and update the internal state.

Chapter 6

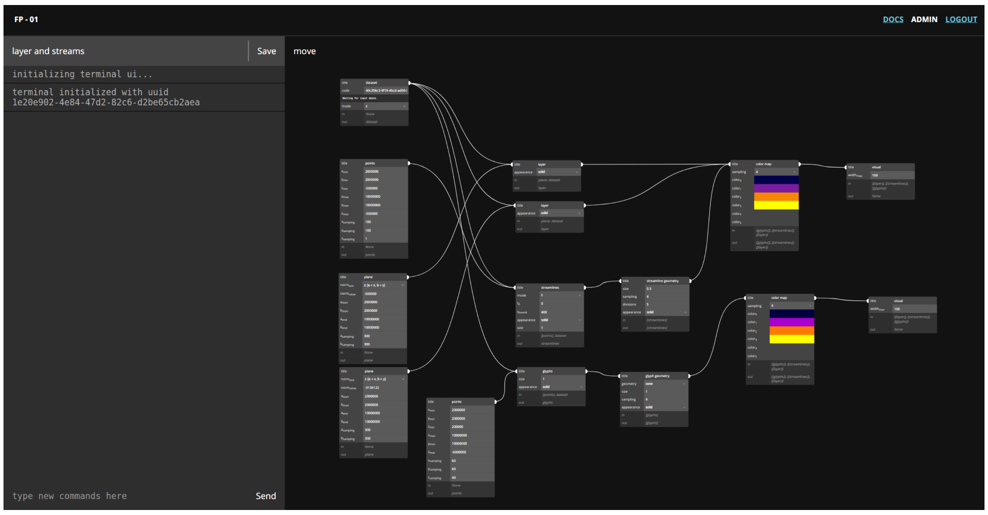
Implementation

The application is implemented according to the design from the previous chapter. The sections of this chapter present the known bugs, and missing or added features, as well as technical details which were omitted in the previous chapter. A vision of future development is presented in the appendix A. All of the missing features can be easily added during the next application update. The user manual can be found in the final distribution of the application.

6.1 Computation Backend and Pipeline Editor

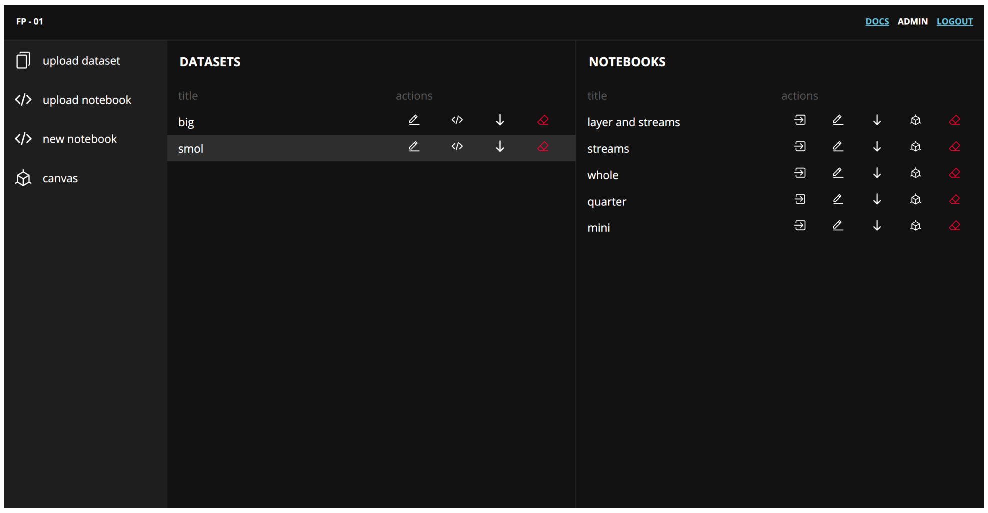
The backend has been implemented correctly according to the presented design with a few minor exceptions listed below. The figure 61 displays an example screen of the pipeline editor with a running computation. The critical sections of the numeric module are implemented in Cython. The use of Python interpreter in the solver, interpolator and integration routine code was mostly eliminated. For further details, see the HTML files containing an analysis of the compiled code. The analysis files are created during building procedures of the numeric module and can be found in the same folders as the files of the numeric module. The missing features include:

- The points node allows visualizing a regularly sampled 3D area. The node is missing a control element enabling random sampling.
- The internal structure allows passing the output of the plane node into streamline and glyph nodes; however, the pipeline editor does not allow connecting the mentioned nodes.
- Filtering node was not implemented.



```
initializing terminal ui...
terminal initialized with uuid
1e20e902-4e84-47d2-82c6-d2be65cb2aea
```

Figure 61: Screenshot of the pipeline editor



DATASETS

title	actions
big	
smol	

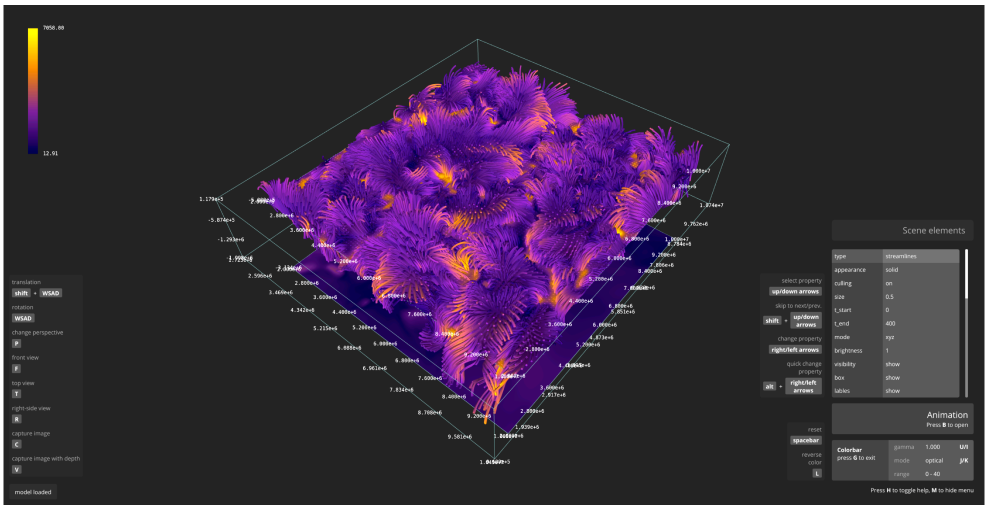
NOTEBOOKS

title	actions
layer and streams	
streams	
whole	
quarter	
mini	

Figure 62: Screenshot of the application main screen

6.2 Web and Server Application

The web and server applications were both implemented according to the proposed design; additional adjustments could improve the usability and accessibility of the application. Figure 62 presents the current design of the central screen.



type	streamlines
appearance	solid
culling	on
size	0.5
t_start	0
t_end	400
mode	xyz
brightness	1
visibility	show
box	show
lables	show

Colorbar press **G** to exit

gamma	1.000	U/I
mode	optical	J/K
range	0 - 40	

Figure 63: Screenshot of the renderer

6.3 Renderer

The renderer has been implemented to be usable both in the browser and in the SAGE2 environment. The renderer has two additional features, which were added to the initial design.

Animation The streamlines can be animated to simulate the flow. The animation effect is achieved by updating the visible range of each streamline every frame. The animation feature has a dedicated UI control panel.

Gamma correction The application enables transforming the used colormap with an implementation of a custom gamma correction. Additionally, inverting the colormap is possible. The gamma correction is useful for highlighting only the extreme values when the formula (5.1) for determining the significance is used. A dedicated UI widget allows the user to perform the transformation of the colormap.

The figure 63 displays the screen of the renderer.[23] The renderer is also capable of producing images which allow viewing the scene on the 3D monitor, which is done by rendering the scene and the depth map separately and merging the outputs into a single image.

6.4 Proprietary Formats

The application introduces two proprietary formats - one for storing the pipeline description and the second for storing the enriched dataset. As was

[23]As of May 2019, additional images produced by the renderer can be found on http://flow.vojtatom.cz. or in the appendix B.

discussed in 5.1.2 Enriched Dataset Format and 5.4 Pipeline Editor, both formats utilize the JSON format and store the content as serialized JSON. The description of the pipeline (notebook) is marked with a suffix *.docflow* and the enriched dataset as *.imgflow*. A detailed description of the proprietary formats can be found on the attached medium.

6.5 Dependencies

One of the goals was to minimize the number of external dependencies of critical parts of the application. The following list contains all dependencies of the individual components. There is a minimal version of the installation requiring fewer dependencies presented in the user guide.

- Python modules
  - Django
  - Django Channels
  - Cython
  - NumPy
  - SciPy
  - Whitenoise
  - Gunicorn
  - Astropy
- Nginx
- Redis

Chapter 7

Testing

The prototype of the visualization application has been tested for performance and for user accessibility and usability. This chapter is divided into two sections; the first section presents the results of the user testing; the second section compares the performance of SciPy interpolation and integration implementation and the custom Cython code. As a part of the development process, several tests were taken to measure the performance of various methods. The tests and summary of the results were already mentioned in chapters 4 Analysis and 5 Design. The results of the final tests presented here closely resemble the results of the tests taken during development.

7.1 User Testing

The users were challenged with a set of smaller tasks which involved registration, navigating the main screen and managing the uploaded data. These trivial tasks served as a preparation for two additional tasks which tested the core of the application — setting up the visualization pipeline and user manipulation of the output visualization. The scenarios of individual tests with the answers and notes of the users are available on the enclosed medium. The most frequently encountered issues were the following:

- The status of uploaded dataset is not understandable. A text labels indicating the status of the uploaded dataset (checking, valid, invalid) would improve the accessibility.
- A UI for managing running tasks would improve usability and simplify administrative tasks.
- A link for downloading the latest computation result for each notebook should be provided. Currently, the link is only available in the terminal when the computation finishes.

- Support for adding custom labels to the scene in renderer would improve usability.
- Incorporating a grid indicating the exact position of the labels in the visualization scene would improve usability and accessibility of the rendered images. However, these tasks can be currently done during local post-processing by the user.
- The renderer was designed to function in both the browser and in the SAGE2 environment. One of the compromises made during the implementation included disabling mouse controls for the UI. It is possible to manipulate the scene with the cursor; nevertheless, the UI is manipulated strictly by keys. Adding the ability to manipulate the UI with a cursor would improve usability.
- The pipeline editor is missing any help/guide labeling the keybord shortcuts for adding and deleting nodes.
- The users forget to save the notebook, the save button is not very accesible. Alternative design should be used to highlight the button.

The users who were familiar with the domain and those who were briefly introduced to the topic without any previous knowledge were less confused by the design of the pipeline editor. The design of the editor is not universally understandable without the understanding of how the background of the application works. Therefore, it is advisable to incorporate a description of the background processes into the user guide.

7.2 Performance of the Numeric Module

This section presents the benchmark results of the used calculation methods. The two tested functionalities are interpolation and integration. Each of the benchmarks compares the SciPy implementation with the custom Cython code. The initial tests which were taken during the development process suggested the SciPy interpolation method does not perform the best when a repeated interpolation of a single point is requested. The computer used for benchmarking had the following specifications:

- Intel Core i7-7700, Quad-Core, 3.6GHz
- RAM HyperX 8GB DDR4 2133MHz
- MSI GeForce GTX 1060 6GT

7.2.1 Interpolation

Measuring the interpolation performance involved generating random vector field with randomly sampled axes and interpolating a certain number of random points inside the dataset. The interpolation of the same set of points was repeated multiple times.

The table 71 contains the measured times, as well as the ratio between the execution times of the SciPy and Cython/C implementations. The same procedure was repeated with a random field where the distance between individual sampling points was uniform. The table 72 presents the results. The custom Cython/C performs significantly better than the SciPy implementation especially when a lower number of points is interpolated. The interpolation in a vector field with uniform distances is faster as expected since the Cython/C interpolator is optimized to take advantage of the uniform sampling. There is a slight difference between the values produced by the Cython/C and the SciPy methods. A rounding error probably causes the difference; however, the variation is negligible. The values of the random vector field fall into the range [0, 100] and the results of both methods satisfy the NumPy *allclose*[24] test with the default parameters.

7.2.2 Integration

Benchmarking the integration involves constructing a random vector field, selecting a set of random points and performing the integration. The two compared alternatives are the SciPy integrator (*solve_ivp* with default solver *RK45*, see documentation[25] and custom Cython/C integrator (also using a custom implementation of *RK45*). The methodology of the tests is similar to the previous interpolation benchmarks. The tables 73 and 74 present the times and ratios of the individual implementations. As expected, the SciPy suffers from slow interpolation of a single point. Results of both methods are no longer comparable by the NumPy *allclose* function since the rounding error of individual interpolations influences the future steps of the integration. However; the differences between the outputs of two methods are still within a reasonable range, and the difference is visually indistinguishable.

[24]See the function *numpy.allclose* in NumPy documentation [27].
[25]See the *solve_ivp* in SciPy documentation [28].

		repetitions													
		1		10		100		1000		10000		100000		1000000	
# of points		Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy
1	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	< 0.01	0.15	0.01	1.56	0.09	15.43	0.87	156.52
	ratio	25.89		118.09		175.58		183.81		182.23		175.64		180.47	
10	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	< 0.01	0.18	0.02	1.72	0.19	16.97	–	–
	ratio	14.31		57.57		91.15		97.63		98.72		91.63		–	
100	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	0.01	0.22	0.12	2.20	–	–	–	–
	ratio	5.33		13.78		17.89		16.62		18.38		–		–	
1000	times	< 0.01	< 0.01	< 0.01	0.01	0.02	0.06	0.16	0.64	–	–	–	–	–	–
	ratio	2.51		3.74		3.96		3.98		–		–		–	
10000	times	< 0.01	0.01	0.03	0.07	0.25	0.69	–	–	–	–	–	–	–	–
	ratio	2.66		2.72		2.73		–		–		–		–	
100000	times	0.03	0.07	0.27	0.80	–	–	–	–	–	–	–	–	–	–
	ratio	2.68		3.00		–		–		–		–		–	
1000000	times	0.28	0.76	–	–	–	–	–	–	–	–	–	–	–	–
	ratio	2.67		–		–		–		–		–		–	

Table 71: Interpolation times (in seconds) in a randomly sampled dataset

		repetitions													
		1		10		100		1000		10000		100000		1000000	
# of points		Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy	Cython	SciPy
1	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	< 0.01	0.15	0.01	1.52	0.08	15.04	0.80	149.94
	ratio	21.65		61.81		158.90		188.90		186.42		180.17		186.75	
10	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	< 0.01	0.16	0.01	1.58	0.12	16.40	–	–
	ratio	14.96		81.39		124.42		130.59		132.14		135.44		–	
100	times	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.02	0.01	0.20	0.05	1.97	–	–	–	–
	ratio	4.78		20.15		36.15		35.16		36.76		–		–	
1000	times	< 0.01	< 0.01	< 0.01	0.01	0.01	0.06	0.09	0.62	–	–	–	–	–	–
	ratio	2.99		5.71		7.08		7.15		–		–		–	
10000	times	< 0.01	0.01	0.02	0.07	0.18	0.70	–	–	–	–	–	–	–	–
	ratio	3.55		3.72		3.95		–		–		–		–	
100000	times	0.02	0.07	0.20	0.73	–	–	–	–	–	–	–	–	–	–
	ratio	3.57		3.75		–		–		–		–		–	
1000000	times	0.20	0.78	–	–	–	–	–	–	–	–	–	–	–	–
	ratio	3.86		–		–		–		–		–		–	

Table 72: Interpolation times (in seconds) in a uniformly sampled dataset

		repetitions					
		1		10		100	
# of points		Cython	SciPy	Cython	SciPy	Cython	SciPy
1	times	< 0.01	0.04	< 0.01	0.22	0.01	5.71
	ratio	299.99		418.54		677.80	
10	times	< 0.01	0.27	< 0.01	3.16	0.03	26.12
	ratio	702.11		884.21		950.00	
100	times	< 0.01	2.18	0.03	23.73	0.22	223.60
	ratio	864.92		927.76		1034.14	
1000	times	0.03	24.55	0.27	247.22	–	–
	ratio	897.68		925.23		–	
10000	times	0.28	244.01	–	–	–	–
	ratio	877.44		–		–	

Table 73: Integration times (in seconds) in a randomly sampled dataset

		repetitions					
		1		10		100	
# of points		Cython	SciPy	Cython	SciPy	Cython	SciPy
1	times	< 0.01	0.02	< 0.01	0.21	< 0.01	2.23
	ratio	179.87		452.36		556.51	
10	times	< 0.01	0.26	< 0.01	1.58	0.02	26.66
	ratio	977.40		1266.79		1648.98	
100	times	< 0.01	2.47	0.01	24.78	0.14	249.64
	ratio	1324.99		1720.52		1795.53	
1000	times	0.02	26.37	0.17	247.85	–	–
	ratio	1443.12		1474.39		–	
10000	times	0.17	260.67	–	–	–	–
	ratio	1490.96		–		–	

Table 74: Integration times (in seconds) in a uniformly sampled dataset

Conclusion

The thesis described and analyzed selected visualization techniques, as well as related mathematical methods and presented a design of application effectively implementing the visualization methods. The application was implemented as a client-server project, creating a portable online tool for vector field visualization.

The initial research presented several visualization methods which were suitable for visualization of the selected vector field attributes. It was necessary to analyze the underlying mathematical methods such as interpolation and numerical integration to make the implementation of the visualization methods possible. The analysis revealed that the performance of the used mathematical methods is one of the critical parts for producing a satisfactory visualization. Additionally, the concept of the visualization pipeline was introduced.

Based on the visualization pipeline, the prototype of the application was divided into four parts. Pipeline editor allows the user to program a custom visualization pipeline; computation component provides a custom implementation of the essential mathematical methods and underlying structures; renderer enables responsive and scalable rendering of the visualized scene; last — the web and server application manage the connections between the other application components. The modular design allows optimising selected code and reusing the implemented algorithms. The complete application is a scalable tool for vector field visualization.

The final user testing shows that the application still requires additional changes to provide an intuitive interface; however, the performance of the new underlying framework proves that it is possible to implement a powerful visualization tool without any significant dependencies. As the following list illustrates, all of the goals set at the beginning of the thesis were reached.

Visualized Properties The application supports the visualization of individual points, as well as plains and streamlines. The combination of these three visualization methods provides a way to display any of the vector field properties listed in the introduction.

Visualization Methods As the results of the final testing illustrate, the performance of the custom implementated methods exceed the performance of implementation in some widely-used Python packages; thus, the implementation can be considered as efficient.

Selecting Area of Interest The application provides a way to select 2D and 3D slices of space for visualization. Moreover, the user can specify the exact points he wishes to visualize.

Used Technology Since the computation part was moved onto the server, the used technology allows using the application on any operating system with a browser, where WebGL technology is supported. The app is capable of producing images for 3D monitor, and the renderer is scalable as a SAGE2 application.

Bibliography

[1] Davis, H. F.; Snider, A. D.; et al. *Introduction to vector analysis.* Boston: Allyn and Bacon Boston, Mass, fourth edition, 1961, ISBN 0-205-07002-7.

[2] Klaus, R.; Hrivňák, D. *Breviář vyšší matematiky.* Ostravská univerzita v Ostarvě, first edition, 2001, ISBN 80-7042-819-8.

[3] Kang, H. R. *Computational color technology.* Spie Press Bellingham, 2006, ISBN 978-0-819-46119-3.

[4] Žára, J.; Beneš, B.; et al. *Moderní počítačová grafika.* Computer press, 2004, ISBN 80-251-0454-0.

[5] Breeuwsma, P. Cubic interpolation - Paulinternet.nl. [online], 1995, [cit. 2018-03-08]. Available from: `http://www.paulinternet.nl/?page=bicubic`

[6] Lekien, F.; Marsden, J. Tricubic interpolation in three dimensions. *International Journal for Numerical Methods in Engineering*, volume 63, no. 3, 2005: pp. 455–471, ISSN 0029-5981.

[7] Olver, P. J. *Introduction to partial differential equations.* Springer, 2014, ISBN 978-3-319-02099-0.

[8] Iserles, A. *A First Course in the Numerical Analysis of Differential Equations.* Cambridge: Cambridge University Press, 2008, ISBN 978-0-521-73490-5.

[9] Atkinson, K.; Han, W.; et al. *Numerical solution of ordinary differential equations.* John Wiley & Sons, 2009, ISBN 978-0-470-04294-6.

[10] Press, W. H.; Teukolsky, S. A. Adaptive Stepsize Runge-Kutta Integration. *Computers in Physics*, volume 6, no. 2, 1992: p. 188, ISSN 08941866, doi:10.1063/1.4823060. Available from: `http://scitation.aip.org/content/aip/journal/cip/6/2/10.1063/1.4823060`

[11] Dormand, J. R.; Prince, P. J. A family of embedded Runge-Kutta formulae. *Journal of computational and applied mathematics*, volume 6, no. 1, 1980: pp. 19–26.

[12] Telea, A. C. *Data visualization: principles and practice*. Boca Raton: CRC Press, Taylor & Francis Group, second edition, 2015, ISBN 9781466585263.

[13] Komura, T. Vector Field Visualisation. 2008, [cit. 2018-04-29]. Available from: `http://www.inf.ed.ac.uk/teaching/courses/vis/lecture_notes/lecture12_2008.pdf`

[14] Beneš, B.; Felkel, P.; et al. Skripta Vizualizace. [online], 1997, [cit. 2018-04-30]. Available from: `https://www.fi.muni.cz/~sochor/VIZ/`

[15] Verma, V.; Kao, D.; et al. A flow-guided streamline seeding strategy. In *Proceedings of the conference on Visualization'00*, IEEE Computer Society Press, 2000, pp. 163–170.

[16] Marchesin, S.; Chen, C.; et al. View-Dependent Streamlines for 3D Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, volume 16, no. 6, Nov 2010: pp. 1578–1586, ISSN 1077-2626, doi: 10.1109/TVCG.2010.212.

[17] Ye, X.; Kao, D.; et al. Strategy for seeding 3D streamlines. In *Visualization, 2005. VIS 05. IEEE*, IEEE, 2005, pp. 471–478.

[18] Globus, A.; Levit, C.; et al. A tool for visualizing the topology of three-dimensional vector fields. In *Proceeding Visualization '91*, Oct 1991, pp. 33–40, doi:10.1109/VISUAL.1991.175773.

[19] Moreland, K. Diverging color maps for scientific visualization. In *International Symposium on Visual Computing*, Springer, 2009, pp. 92–103.

[20] Marrinan, T.; Aurisano, J.; et al. SAGE2: A new approach for data intensive collaboration using Scalable Resolution Shared Displays. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on*, IEEE, 2014, pp. 177–186.

[21] Inc., W. R. Mathematica, Version 11.3. [online], champaign, IL, 2018. Available from: `https://reference.wolfram.com/language/guide/VectorVisualization.html`

[22] Henderson, A.; Ahrens, J.; et al. *The ParaView Guide*. Kitware Clifton Park, NY, 2004, kap. Introduction.

[23] Childs, H.; Brugger, E.; et al. VisIt: An end-user tool for visualizing and analyzing very large data. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, 2012, [cit. 2018-05-10]. Available from: `http://www-vis.lbl.gov/~hrchilds/paper.pdf`

[24] Ramachandran, P.; Varoquaux, G. Mayavi: 3D visualization of scientific data. *Computing in Science & Engineering*, volume 13, no. 2, 2011: pp. 40–51.

[25] Avila, L. S. (editor). *The VTK User's Guide*. Clifton Park, NY: Kitware, 11th edition, 2010, ISBN 978-1-930934-23-8.

[26] Schroder, W. J.; Martin, K. M.; et al. VTK User's Guide-VTK File Formats, chapter 14. [online], 2000, [cit. 2018-05-15]. Available from: `https://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf`

[27] Oliphant, T. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–2019, [cit. 2019-04-09]. Available from: `http://www.numpy.org/`

[28] Jones, E.; Oliphant, T.; et al. SciPy: Open source scientific tools for Python. [online], 2001–2019, [cit. 2019-04-09]. Available from: `http://www.scipy.org/`

[29] Marrin, C. Webgl specification. [online], 2011, [cit. 2018-05-16]. Available from: `https://www.khronos.org/registry/webgl/specs/latest/2.0/`

[30] Cabello, R.; et al. Three.js. [online], [cit. 2018-05-05]. Available from: `https://threejs.org/docs/`

[31] McCarthy, L.; Reas, C.; et al. *Getting Started with P5. js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Inc., 2015.

[32] Jackson, M.; Crouch, S.; et al. Software Evaluation: Criteria-based Assessment. *Governance*, 2011.

[33] Wright Jr, R. S.; Haemel, N.; et al. *OpenGL SuperBible: comprehensive tutorial and reference*. Addison-Wesley Professional, fifth edition, 2010, ISBN 0-321-71261-7.

Appendix A

Future Development

This appendix shortly presents the vision of the future development of the visualization application designed and implemented in this thesis. The introduction of this thesis mentioned a balance of three factors — usability, physical accuracy and the amount of displayed data. The developed application had to overcome many obstacles arising from the absence of any underlying framework and implement the required parts. As a result, three unique independent components were created — the pipeline editor, the computation backend and the renderer. The computation backend uses SciPy and provides a faster alternative implementation of the same methods. The quicker implementation might have some technical shortcomings; however, the goal to balance the physical accuracy and response time was achieved. The same goes for the renderer. The main problem is that the server-client design is not suitable for users-scientists, who want to visualize a dataset and do not have access to any free server.

Therefore, the decision is to take the unique components of the application and fuse them to a single Python module, allowing for local computation and rendering. This is going to require some redesigning; however, the majority of the code will remain the same. The goal is to create a simple Python API anyone could use in their scripts. This functionality is already supported by Mayavi or VTK; however, the design of their API is based on the internal VTK structure which has been previously discussed. The implemented prototype avoided using any significant underlying frameworks to keep modularity, and the future versions are going to stay true to this design decision. The renderer can be embedded into the Python module as a single JavaScript file, and the current design of the renderer allows viewing models locally on the computer without access to the internet. Some additional mechanisms to simplify the calculation and rendering will be considered; however, the development of the visualization application (or module) will be indeed continued.

Appendix B

Images

The appendix contains example images produced by the renderer. The figures display three scenes; each of them is a different subset of the solar dataset. The visualized subsets are labeled as medium, quarter and whole, the name corresponds to the size of the visualized area; thus, the images of the whole subset are indeed a visualization of a 3D layer across the entire dataset. The images B5 and B6 present a rendering of the scene with a depth map, which is used for displaying the dataset on the 3D monitor.

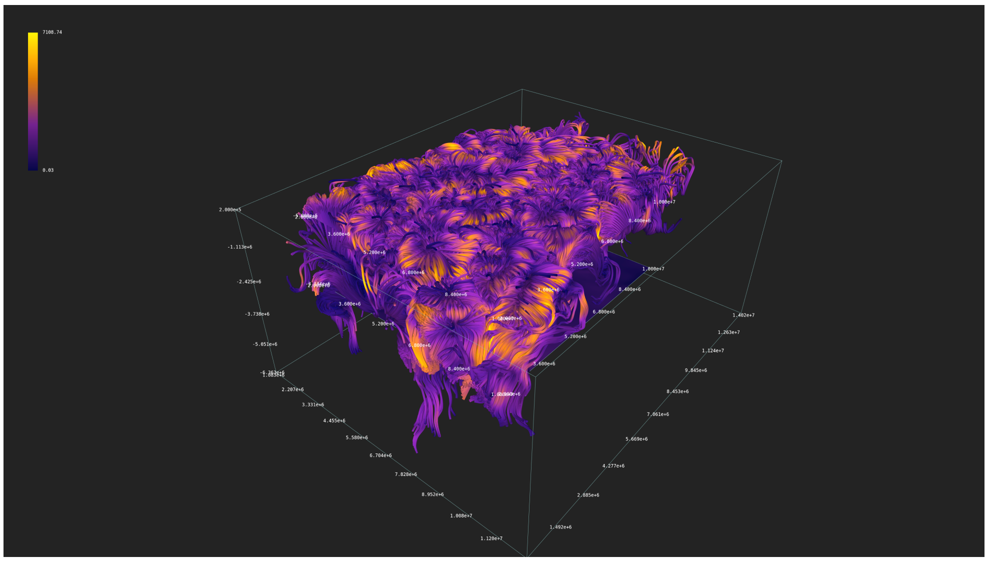


Figure B1: Medium dataset, top side view, visualized with streamlines and layers

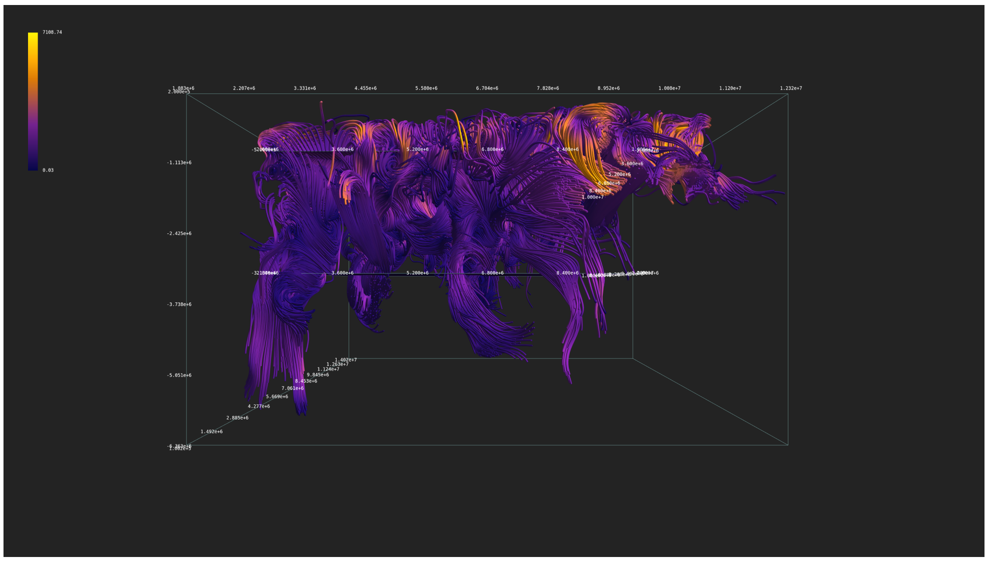


Figure B2: Medium dataset, side view, visualized with streamlines and layers

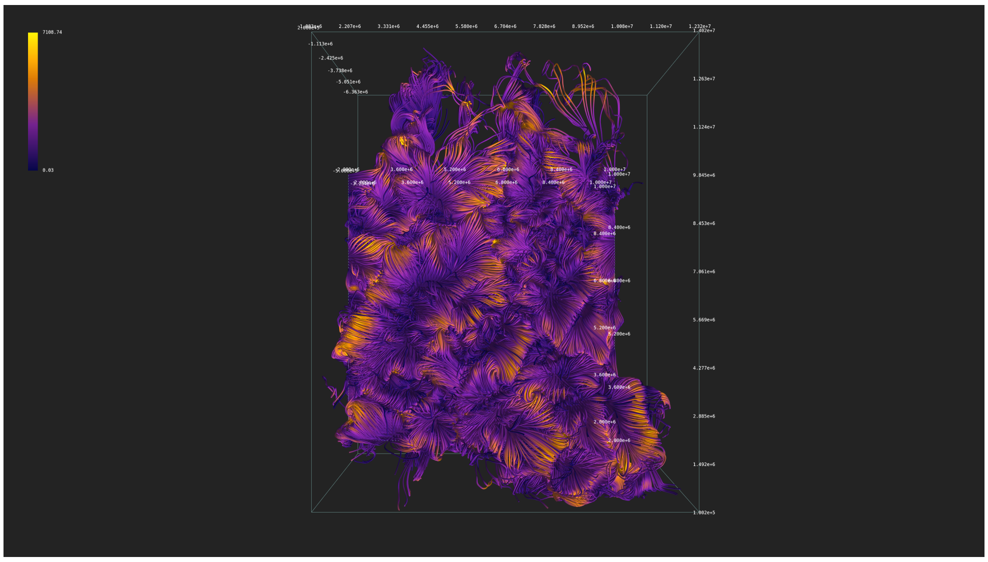


Figure B3: Medium dataset, top view, visualized with streamlines and layers

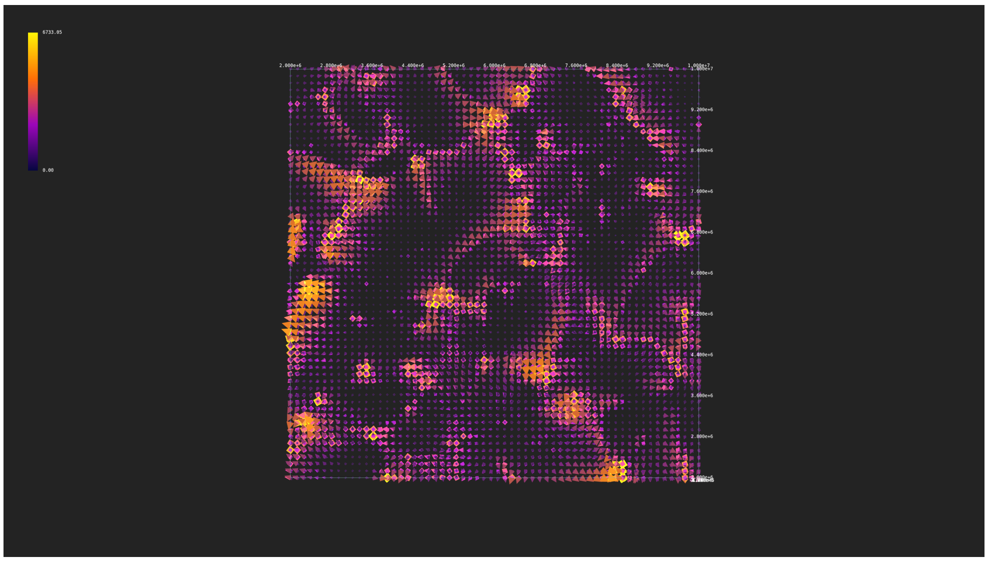


Figure B4: Medium dataset, top view, visualized with glyphs

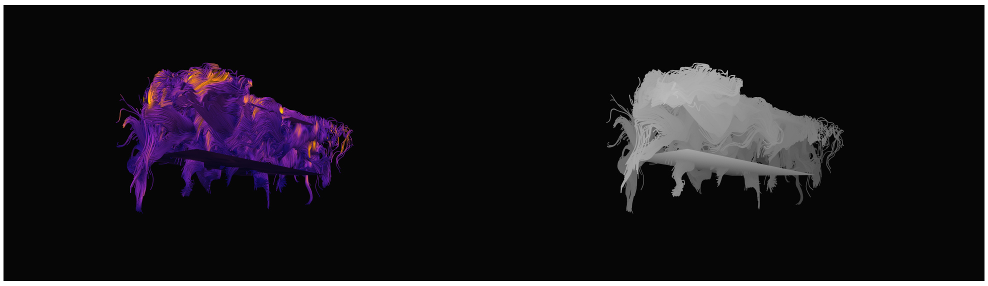


Figure B5: Medium dataset, side view, rendering with a depth map

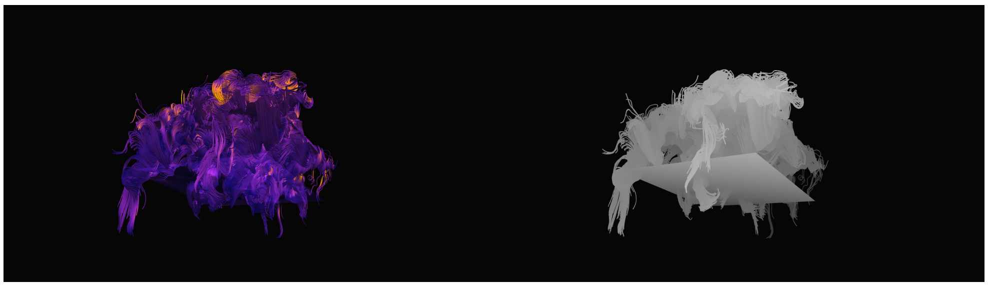


Figure B6: Medium dataset, bottom view, rendering with a depth map

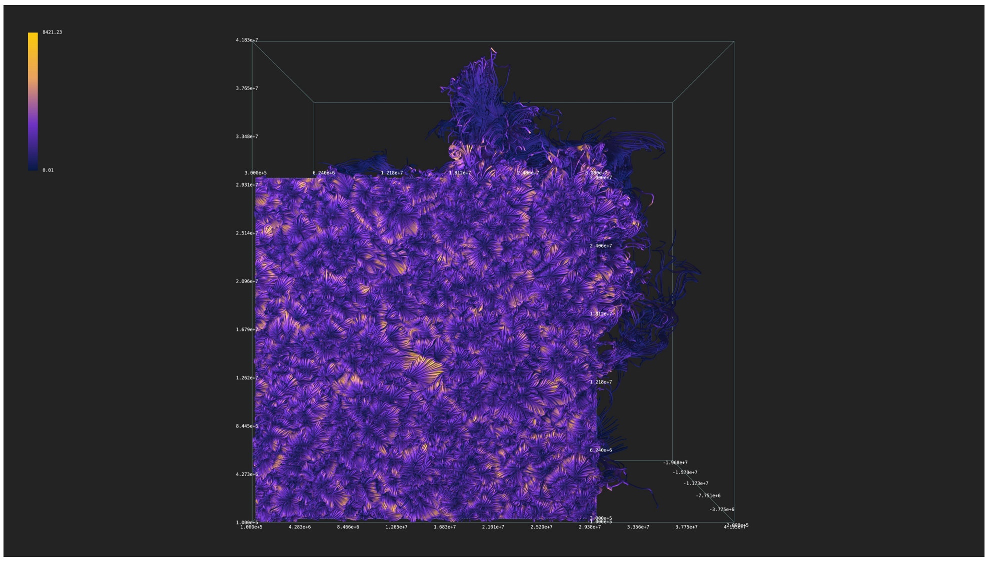


Figure B7: Quarter dataset, top view, visualized with streamlines and layers

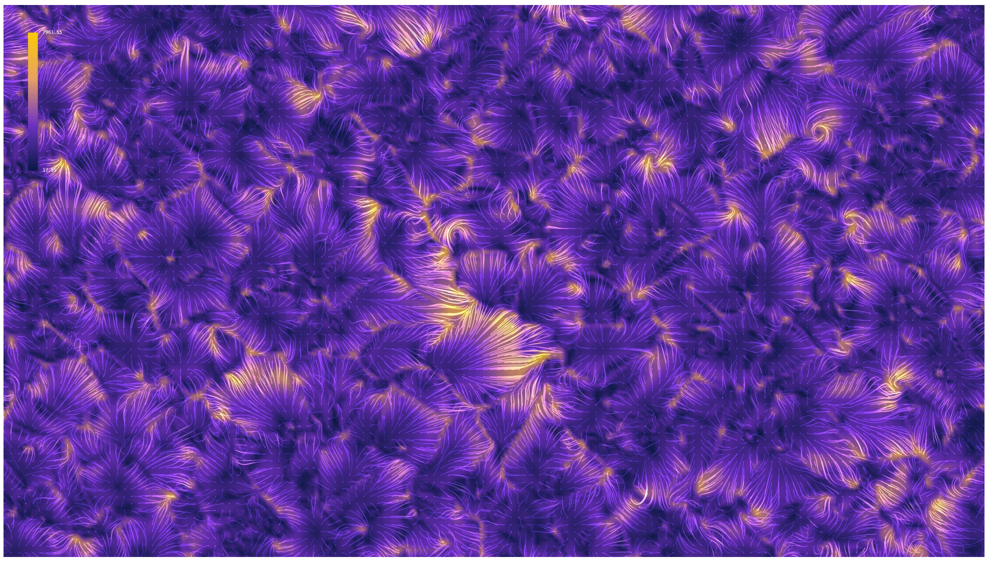


Figure B8: Quarter dataset, top detail view, visualized with streamlines and layers

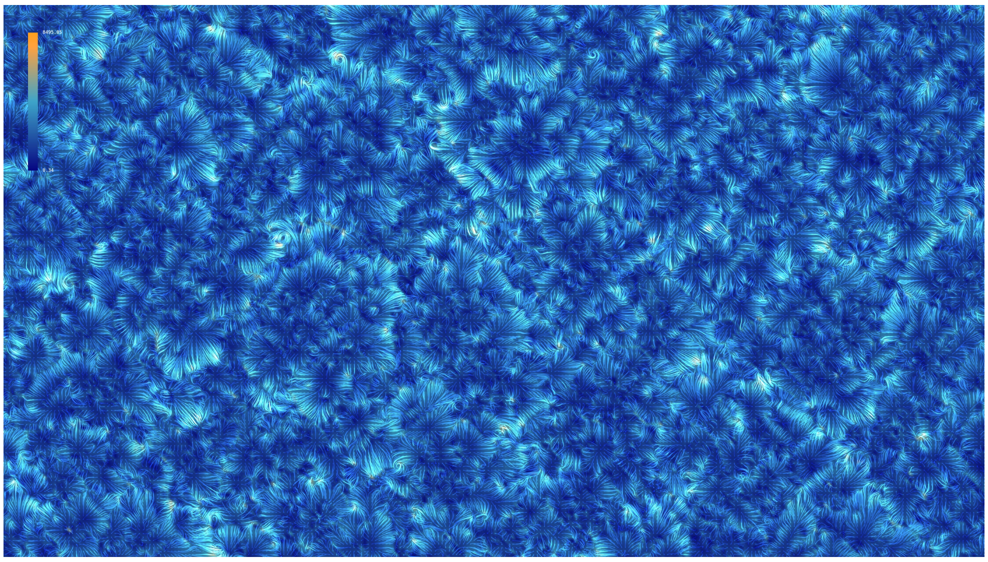


Figure B9: Whole dataset, top detail view, visualized with streamlines and layers

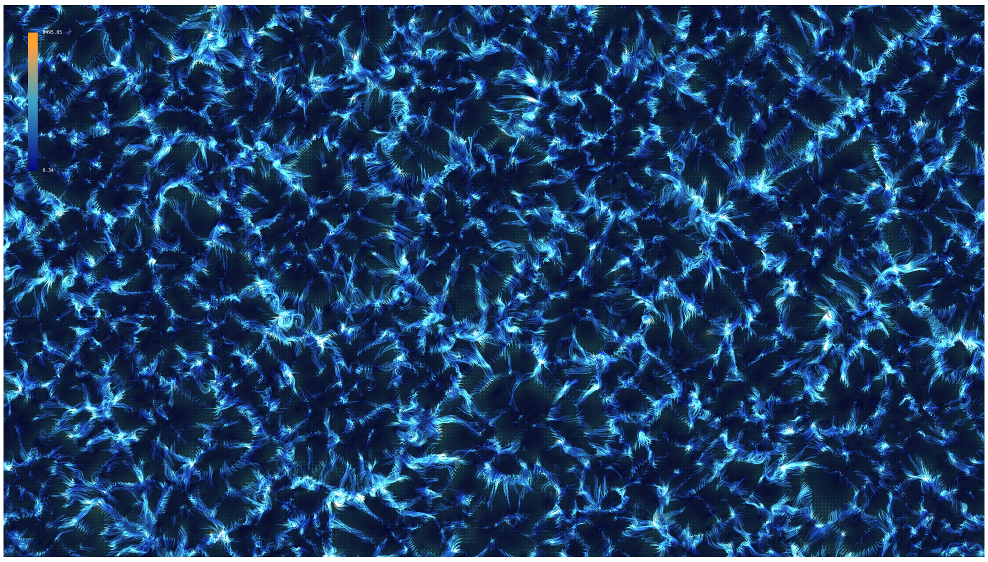


Figure B10: Whole dataset, bottom detail view, visualized with streamlines and layers

Appendix C

Acronyms

HSV	HSV (hue, saturation, value) color model.
ODE	Ordinary differential equation.
RK	Runge-Kutta.
SAGE2	Scalable Amplified Group Environment.
UI	User interface.
VTK	The Visualization Toolkit.

Appendix D

Contents of Enclosed DVD

```
readme.txt ........................ description of DVD contents
usermanual.pdf ........................ application user manual
tests ........ the directory containing test forms and scenarios
src
  flow ........ the directory containing source code of web app
  sage ....... the directory containing source code of SAGE2 app
  thesis ... the directory containing LaTeX source codes of the thesis
text
  thesis.pdf ........................... thesis in PDF format
  thesis.ps ............................. thesis in PS format
```