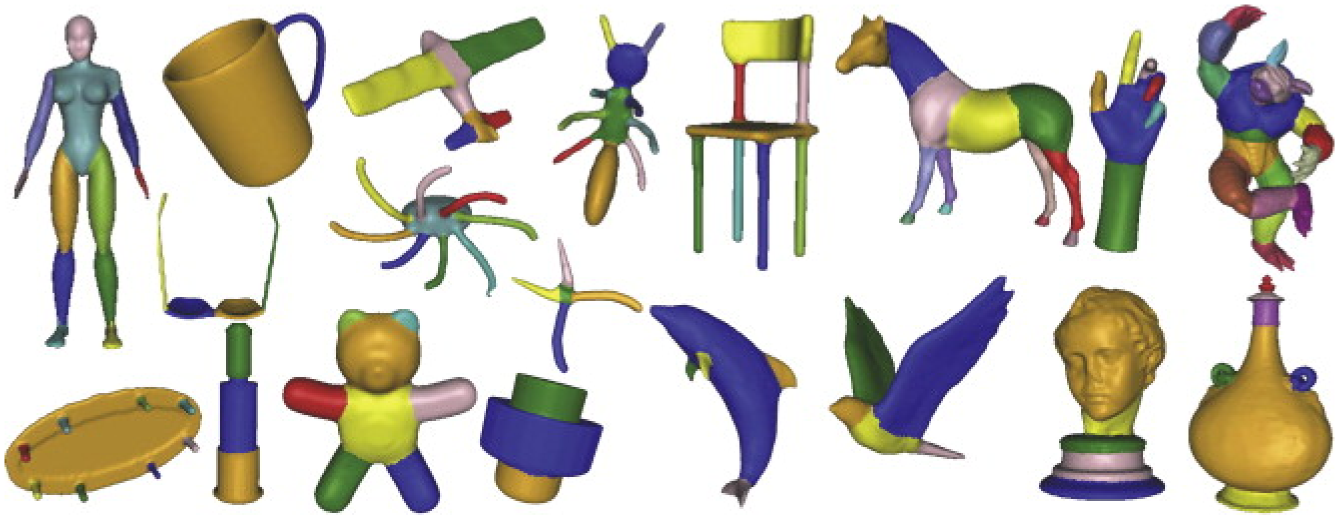
Mesh Segmentation with K-Means

Advanced Methods for Scientific Computing

Matteo Balice, Arcangelo Pisa, Riccardo Figini, Antonio Doronzo

251648, 280733, 251187, 251320

March 3, 2025



1 Introduction

K-Means is a widely used unsupervised clustering algorithm that partitions input samples into coherent groups by clustering similar points together. The number of clusters can either be predefined or determined dynamically. Once the number of centroids is set, various methods can be applied to initialize their positions. The algorithm operates iteratively: each point is assigned to the nearest centroid based on a chosen distance metric. After the assignment phase, the centroids are updated based on the values associated with them. Over successive iterations, the centroids gradually stabilize and eventually converge, meaning they no longer change position.

Our implementation allows the input of CSV files, enabling the program to read dataset points and perform clustering. However, most of our efforts have been focused on mesh segmentation, where the algorithm was adapted to handle the unique properties of geometric structures.

2 Problem Analysis

Mesh segmentation is a fundamental problem in computer graphics and geometric modeling, where a 3D mesh is divided into meaningful sub-regions. The problem is complex due to its NP-completeness in certain conditions, particularly when the segmentation aims to ensure convex patches.

The segmentation process can be viewed as a constrained graph partitioning problem, where mesh elements (vertices, edges, or faces) are grouped into subsets to minimize certain cost functions. This problem is highly dependent on the application, requiring different optimization criteria such as surface curvature, geodesic distances, or topological constraints.

Several approaches have been developed to approximate solutions, categorized into five major methods:

1. **Region Growing** – Expands from seed points based on local similarity criteria.
2. **Hierarchical Clustering** – Iteratively merges smaller clusters into larger ones.

3. **Iterative Clustering** – Uses techniques like k-means to partition the mesh.

4. **Spectral Analysis** – Utilizes eigenvalues of graph Laplacians for clustering.

5. **Implicit Methods** – Includes graph-cut and level-set methods.

The challenge in segmentation is balancing segmentation quality with computational efficiency while ensuring meaningful segmentation results tailored to specific applications. In our project, we have studied the iterative clustering with k-means, focusing on different metrics: Euclidean, geodesic with Dijkstra, and geodesic with heat method.

3 Software Design and Structure

The software is designed following a modular approach to ensure flexibility, scalability, and maintainability. The architecture separates key functionalities into independent components, facilitating extensibility. The key modules (as you can see in the source code structure) are **clustering**, **geometry**, and **mesh_segmentation**. The **clustering** module contains the KMeans class together with all the centroid initialization methods. The **geometry** module contains all the structures and operations needed for geometric processing, including spatial data structures and distance metrics. Finally, the **mesh_segmentation** module is responsible for processing and segmenting mesh data (using the KMeans class), integrating various evaluation metrics to assess segmentation quality.

Abstract interfaces are used for metric computations, ensuring that new distance measures can be integrated without modifying existing components.

The implementation leverages both CPU and GPU computation, with **CUDA** being used for performance-critical operations. This choice enables efficient handling of large-scale datasets, particularly in computationally expensive tasks like clustering and geodesic distance calculations. Additionally, the code has been parallelized using **OpenMP** to further optimize performance on multi-core processors, ensuring efficient execution of computationally intensive loops.

The evaluation framework is incorporated to assess segmentation quality systematically, aligning with standard **benchmarking** methodologies. The overall software design prioritizes efficiency while maintaining flexibility for future extensions and improvements.

To ensure reliability and maintainability, the entire software is integrated with automated testing and continuous integration workflows using **GitHub Actions** at each commit or pull request. The build process, along with *tests* and *memory leaks* detection with valgrind, is executed automatically to validate changes. Additionally, documentation is generated seamlessly using **Doxygen**, ensuring up-to-date and well-structured documentation.

Third-Party Dependencies

The software relies on several third-party libraries to enhance performance, provide utility functions, and ensure robustness. The main dependencies include:

- **csv-parser** - Provides efficient parsing of CSV files.

- **eigen** - A C++ library for linear algebra operations.

- **libigl** - A C++ geometry processing library (used for the heat equation).

- **obj-loader** - Used for loading OBJ format 3D models.

- **googletest** - A testing framework for unit tests.

- **Google benchmark** - Used for performance testing and benchmarking.

These dependencies are essential for numerical computations, debugging, testing, and overall performance optimization.

4 Numerical methodology

4.1 Metric

The **metric class** plays a fundamental role in our implementation, as it manages the functioning of *K-Means* through different strategies. This is an abstract class that needs to be implemented by a specific and real metric to calculate distances. For this reason, you can simply extend this class and

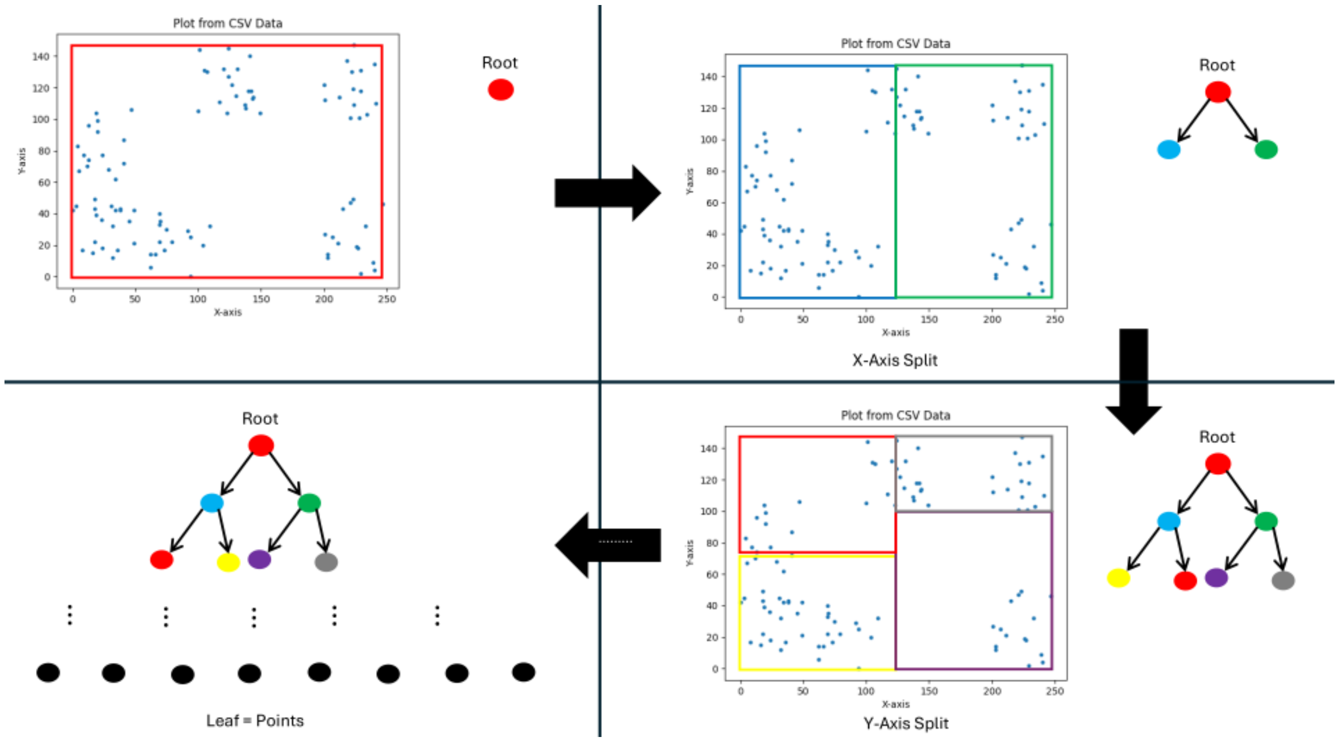


Figure 1: KD-tree Construction

implementing your specific **setup** (to be called at the start of each iteration of KMeans) **fit_cpu** and **fit_gpu** methods. Specifically, three distinct metric classes are already implemented:

- **EuclideanMetric**: The only metric capable of leveraging the KD-Tree structure. It is designed to work with data stored in CSV files, rather than mesh-based datasets, and it calculates distances using the Euclidean distance.
- **GeodesicDijkstraMetric**: Stores data in a standard vector and computes distances using Dijkstra's algorithm.
- **GeodesicHeatMetric**: Also stores data in a vector, but instead of Dijkstra's algorithm, it calculates distances based on the heat equation.

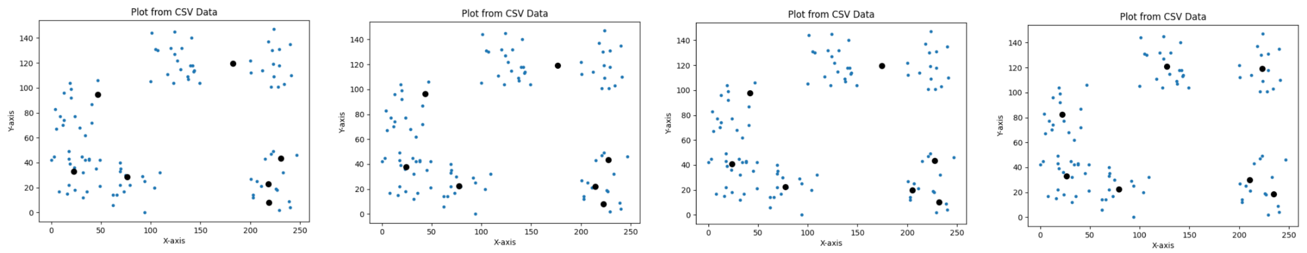
4.1.1 EuclideanMetric

As previously mentioned, the simplest implementation of the Metric class is the EuclideanMetric. Both a CPU and a GPU implementation are available.

EuclideanMetric on CPU

The EuclideanMetric on CPU has been implemented with a KD-Tree data structure. The KD-Tree is a data structure used by a portion of our program to perform clustering more efficiently. In our KD-Tree implementation, the main elements are the nodes and the centroids. The tree nodes represent a specific region in space, defined by the maximum and minimum coordinates for each dimension. Leaf nodes, in particular, contain references to the points within their respective regions. The centroids, on the other hand, are points that represent the centers of the clusters and are updated during the fitting process. At the end of the fitting procedure, each point will be associated with a centroid, which will represent its corresponding cluster. The use of the KD-Tree is divided into two phases: construction and fitting.

Construction Phase: The root node is defined with the maximum and minimum coordinates of the entire dataset. The tree is then built by recursively creating two child nodes. At each step, the data set is split into two parts based on a median point along one dimension. The splitting dimension alternates in a "round-robin" fashion. This process continues



(a) First step (b) Second step (c) Third step (d) Fourth step

Figure 2: Fitting phase of EuclideanMetric

until a subset contains only a single point, at which point a leaf node is created (Figure 1).

Fitting phase: if the node is a leaf, it means that it has a direct reference to the corresponding point, allowing the nearest centroid to be determined immediately. For a nonleaf node, on the other hand, the center point of the "box" representing the node is calculated. At this point, the nearest centroid is identified and a list of candidate centroids is constructed, excluding those that are 'too far away' from the current node. If the candidate list is reduced to a single element, it is assigned directly to the node's children. Otherwise, recursion continues by further refining the selection. The filter function takes advantage of the structure of the KD tree to partition the space efficiently, assigning points to the nearest centroids. Using recursion, candidate centroids are dynamically filtered based on distance and node position, avoiding a direct comparison of all points to all centroids. This approach optimizes the process, significantly reducing the number of comparisons needed by progressive filtering of centroids at each level of the tree. Since the KD-tree is used to optimize the assignment of points to centroids in the K-Means, the centroids must be updated after this step. This is done by keeping the sum of all points assigned to each centroid during the current iteration. Once the assignment is complete, the new position of each centroid is calculated by averaging the coordinates of the assigned points. This process is repeated until convergence, that is, until the centroids no longer change position significantly.

Both the construction and fitting phase are parallelized with **OpenMP**.

EuclideanMetric on GPU

The **CUDA**-based implementation of the EuclideanMetric iteratively assigns each data point to the closest centroid using the Euclidean distance, then computes the sum of all assigned points and the count of elements in each cluster. The centroids are subsequently updated as the average of the assigned points. This process is repeated until a predefined maximum number of iterations is reached or the algorithm converges. The implementation utilizes CUDA kernels to parallelize computations, optimizing performance by distributing tasks across multiple GPU threads. Specifically, it assigns points to clusters, accumulates sums for each centroid, and updates centroids in parallel. *Atomic* operations ensure safe concurrent updates when computing new centroids. Finally, the computed cluster assignments and centroid positions are transferred from the GPU to host memory.

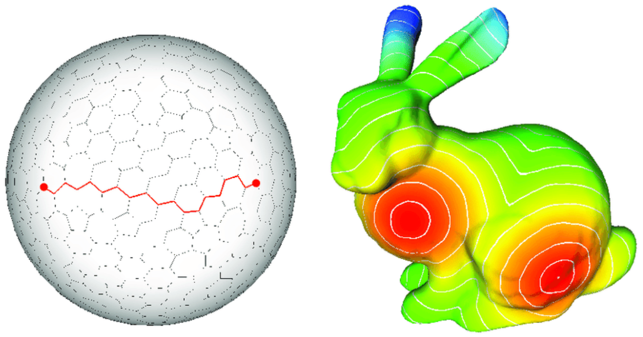


Figure 3: Geodesic distance with Dijkstra's algorithm.

4.1.2 GeodesicDijkstraMetric

The `GeodesicDijkstraMetric` class is responsible for computing geodesic distances on a 3D **mesh**. To achieve this, it clusters mesh faces using the K-Means algorithm and computes shortest paths through **Dijkstra's algorithm**. The implementation is optimized for both **CPU** and **GPU** execution, leveraging OpenMP for parallelism on the CPU and CUDA for efficient computations on the GPU. The class operates on a given `Mesh` object, performing calculations based on the barycenters of mesh faces.

A key aspect of the geodesic distance computation is the use of the **dihedral angle**, which measures the angle between two adjacent faces. This angle is determined from the normal vectors of the faces, using the dot product formula:

$$\theta = \cos^{-1}\left(\frac{\mathbf{n}_1 \cdot \mathbf{n}_2}{\|\mathbf{n}_1\|\|\mathbf{n}_2\|}\right)$$

where $\mathbf{n}_1$ and $\mathbf{n}_2$ represent the normal vectors of the two faces. The dihedral angle plays a crucial role in defining the connectivity of the mesh.

To efficiently determine the closest face to a given centroid, the class implements a `findClosestFace` function. This method computes the Euclidean distance between the centroid and the barycenters of all faces in the mesh. Given the potential computational cost of this operation, OpenMP is used to enable parallel execution, significantly accelerating the search process.

The `computeDistances` function applies **Dijkstra's algorithm** to determine the shortest paths from a given face to all others in the mesh. It maintains a **priority queue** to efficiently update distances and ensures that adjacency relationships between faces are properly taken into account.

To cluster the mesh faces into groups, the class provides two implementations of the `fit` function: `fit_cpu` and `fit_gpu`. The CPU-based implementation follows an iterative **K-Means algorithm**, where each face is assigned to the closest centroid, and new centroid locations are computed as the average of the barycenters of assigned faces. This process repeats until convergence, defined as the point when the fraction of faces changing clusters falls below a threshold. OpenMP is used to parallelize this iterative process, ensuring faster execution. Additionally, the algorithm terminates if the number of iterations exceeds a predefined limit.

For **GPU**-based execution, the `fit_gpu` function utilizes CUDA to parallelize the K-Means clustering process. It begins by transferring the necessary data—face barycenters, mesh adjacency, and initial centroids—to the **GPU**. A CUDA reduction kernel computes the nearest face for each centroid using Euclidean distances, providing a suitable starting point for geodesic calculations. This starting face is then processed by a **CPU**-based Dijkstra algorithm to compute the geodesic distances over the mesh; this step is performed sequentially due to the inherent sequential nature of Dijkstra's algorithm. Once these distances are computed, they are copied back to the **GPU** where parallel CUDA kernels assign faces to clusters and accumulate barycenter sums using atomic operations, ensuring safe concurrent updates during centroid recalculation. Finally, a specialized CUDA kernel, `kmeans_cuda_geodesic`, is invoked to perform clustering efficiently, after which the updated centroids and cluster assignments are copied back to the **CPU** for further processing.

After some evaluations, the Dijkstra's algorithm has performed better for low-density meshes.

4.1.3 GeodesicHeatMetric

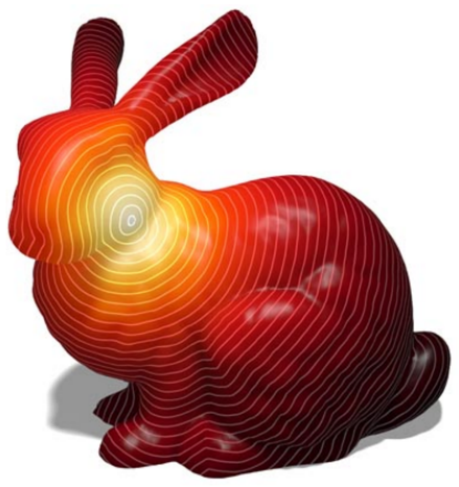


Figure 4: Geodesic distance on the Stanford Bunny. The heat method allows distance to be rapidly updated for new source points or curves

The `GeodesicHeatMetric` class provides an efficient method for computing geodesic distances on a mesh using the heat method. This approach relies on solving a heat diffusion problem and extracting distance information from the resulting temperature distribution. The class extends the `GeodesicDijkstraMetric` base class and utilizes the Eigen and libigl libraries for numerical computations.

The heat method follows a sequence of three fundamental steps. First, heat diffusion is simulated by solving the equation

$$(M - tL)u = M\delta$$

where M represents the mass matrix, L is the Laplacian operator, and δ is the heat source term. This step propagates heat across the mesh over a small time interval t, which is set based on the square of the average edge length. Next, the temperature gradient ∇u is computed using the gradient operator, capturing how the heat distribution changes over the surface. Finally, geodesic distances are obtained by solving a Poisson equation of the form

$$L\phi = \nabla \cdot \left(-\frac{\nabla u}{\|\nabla u\|} \right),$$

where the normalized negative gradient of u serves as a direction field guiding the computation of distances.

During initialization, the class extracts vertex and face data from the mesh and precomputes the necessary matrices required for solving the heat and Poisson equations. This includes constructing the cotangent Laplacian L, the mass matrix M, the gradient operator, and the divergence operator. Since the system matrix remains constant while only the source term changes for different queries, it is possible to factorize the linear system in advance using **Cholesky decomposition**. This precomputation significantly accelerates the solution process during runtime, as each new query only requires solving a triangular system rather than recomputing the full factorization.

The implementation remains entirely on the CPU, and no GPU acceleration has been incorporated. The primary reason for this choice is that Cholesky factorization involves significant data dependencies, making it difficult to parallelize efficiently on a GPU. While GPUs excel at highly parallelizable tasks, sparse direct solvers such as Cholesky decomposition often perform better on CPUs due to their reliance on hierarchical memory access patterns and recursive factorization steps. Additionally, given that the system matrix is triangular, its solution process is inherently sequential, limiting the potential for parallelization.

Algorithm 1 The Heat Method

```
I.   Integrate the heat flow $\dot{u} = \Delta u$ for some fixed time $t$.
II.  Evaluate the vector field $X = -\nabla u / |\nabla u|$.
III. Solve the Poisson equation $\Delta\phi = \nabla \cdot X$.
```

To compute geodesic distances from a given starting face, the method first selects a source vertex within the face and solves the heat diffusion equation using `heat_geodesics_solve` (solving the triangular system). Once the temperature distribution is obtained, the class extracts per-vertex distances and averages them to derive face-wise geodesic distances. This approach ensures a balance between computational efficiency and numerical accuracy, making it suitable for large-scale meshes while leveraging the advantages of precomputed factorization. The GeodesicHeatMetric class uses OpenMP for faster computations on CPU.

4.2 Centroids initialization

This section will show pictures regarding the initialization of centroids. In the images the red dots correspond to the initial position of the centroids, the blue dots to the centroids after the fit method.

Once the number of clusters has been determined (see 4.3) or the user has specified it, it is essential to proceed with a good initialization of the centroids since it directly influence the total number of iterations (speed of convergence) and the final quality results. We have developed two methods, each with its own advantages and complexities. Additionally, **random** centroid initialization is available for a faster procedure, where k points are randomly selected from the existing data.

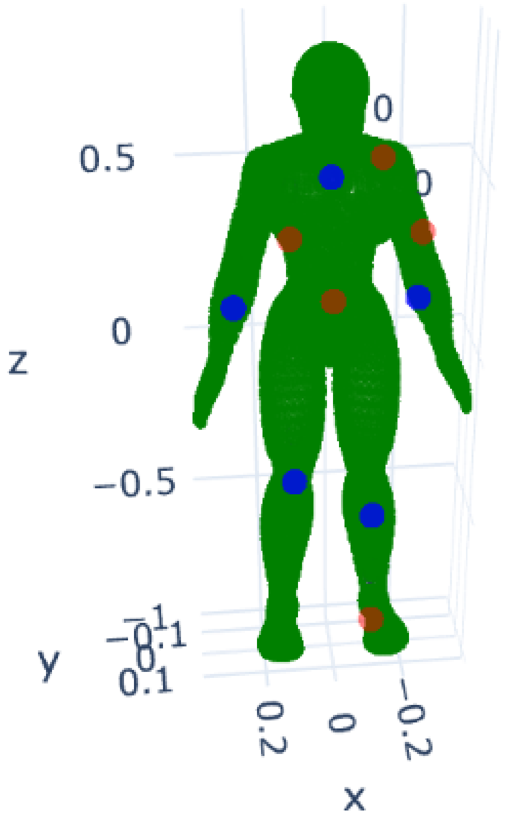


Figure 5: Random initialization

The second method implemented is called **most distant**. After randomly selecting the first centroid, the other centroids are determined by choosing points that maximize the distance from all previously fixed centroids. The selected points always come from the input data. This technique represents a middle ground between random centroid generation and the KDE-based method (third method). It is significantly faster than KDE and, in many cases, reduces the number of iterations required for the subsequent k-means computation.

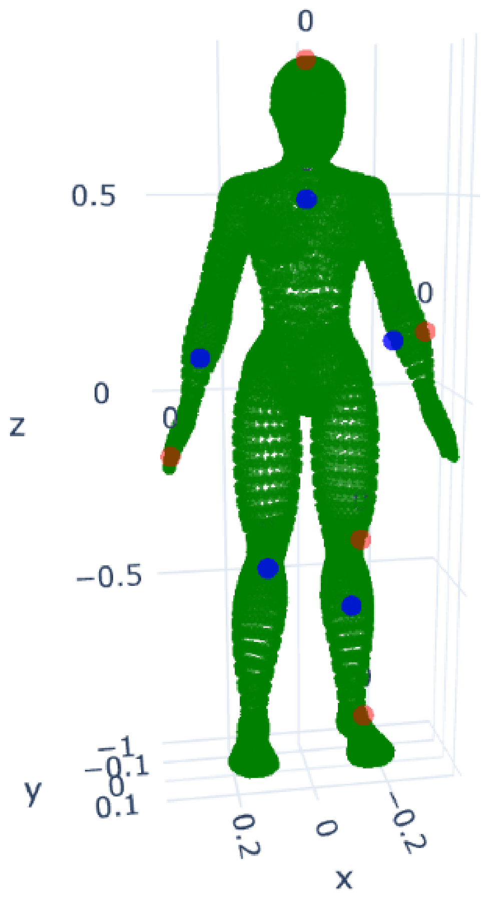


Figure 6: MostDistant initialization

The third technique uses the **Kernel Density Estimator (KDE)**.

4.2.1 Kernel Density Estimation (KDE)

Kernel Density Estimation (**KDE**) is a nonparametric method for estimating the probability density function of a distribution from a given sample. Intuitively, it constructs a smooth function that assigns larger values where data points are densely concentrated and smaller values in regions where data points are sparse. This method provides a more continuous representation of the underlying data distribution compared to histograms, making it particularly useful in various applications requiring density estimation.

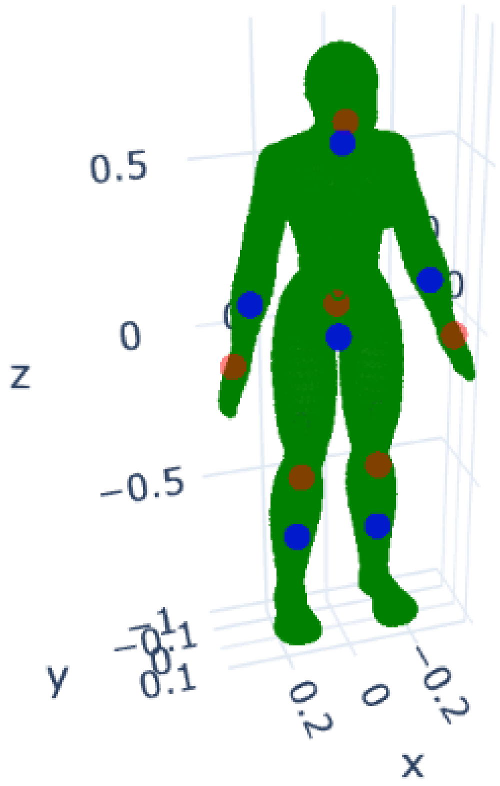


Figure 7: Kernel density estimator

A crucial parameter in KDE is the bandwidth h, which controls the degree of smoothness in the resulting estimate. A smaller bandwidth assigns greater weight to points closer to the target location, yielding a highly detailed density function. However, an excessively small h may lead to overfitting, capturing noise rather than the true underlying distribution. Conversely, a larger bandwidth produces a smoother estimate but risks obscuring important features by oversmoothing the data, leading to underfitting. In our implementation, the bandwidth is represented as a diagonal matrix, allowing for independent treatment of each dimension. Its value is determined using the rule of thumb, which suggests an optimal choice based on the standard deviation of the data:

$$h_{ii} = \text{stdDev} \cdot n^{-\frac{1}{d+4}} \cdot \left(\frac{4}{d} + 2\right)$$

where d represents the dimensionality of the sample.

Another key component of KDE is the kernel function k, which defines the shape of the density estimate. The choice of kernel affects how smooth or sharp the estimate appears. In this implementation, various kernels are available, with the default being the Gaussian kernel, defined as:

$$\text{gaussianKernel}(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{\dim(\mathbf{x})}{2}}} \cdot \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right)$$

The final density estimate is given by:

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where n is the number of points, $h > 0$ is the bandwidth, and $K(\cdot)$ represents the chosen kernel function.

The implementation of KDE is encapsulated in the `KDE` class, which provides all essential functionalities for density estimation. Additionally, a specialized version, `KDE3D`, extends this capability to three-dimensional data points. The class handles the computation of the bandwidth using the rule-of-thumb method, evaluates the density function at specific locations, and generates a grid of points for efficient evaluation. Furthermore, it includes methods for detecting local maxima in the density function, which serve as potential cluster centroids.

The centroid initialization process has the `CentroidInitMethods` class serving as its parent. The primary operations include constructing the object with problem-specific data and the number of clusters k, which can be specified at initialization or adjusted later using setter methods. This design ensures flexibility while maintaining computational efficiency in clustering tasks based on KDE.

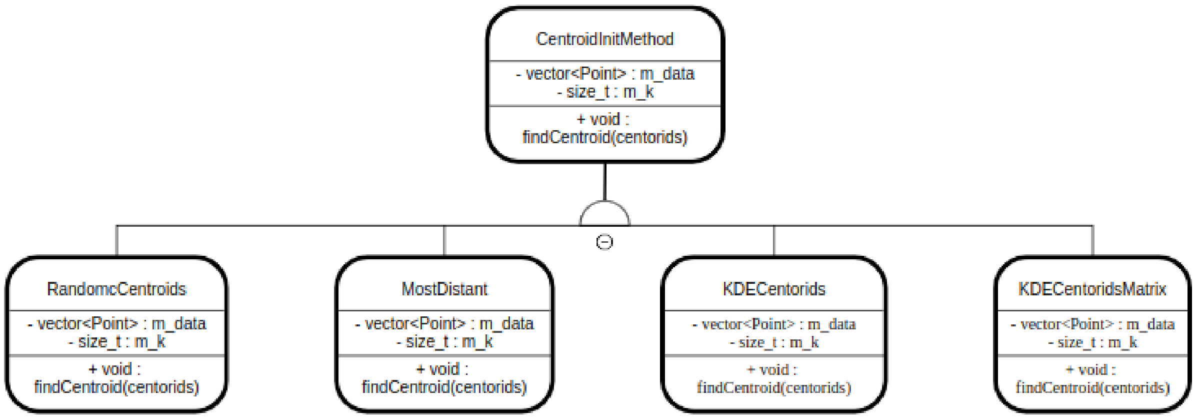


Figure 8: Centroid initialization methods

4.3 K - Initialization

Appropriate reasoning was conducted to estimate the optimal number of clusters when the user did not specify a pre-determined value. In particular, three primary methods were considered: the elbow method, the kernel density estimate (KDE) method and silhouette method.

The **Elbow Method** is one of the most widely used approaches for determining the optimal number of clusters in K-means. The main idea is to analyze the trend of the Within-Cluster Sum of Squares (WCSS), which measures the sum of the squared distances between points and their respective centroids, as the number of clusters changes. The optimal number of clusters is identified at the "elbow" point, where the WCSS starts to level off.

$$WCSS = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

The **K - Kernel Density Estimation (KDE)** method uses data density estimation to determine the optimal number of clusters. A density function is estimated using the concept of a kernel, a smooth function that counts nearby points. The peaks in this estimated function represents regions of high density and potential centroids for clusters.

Finally, the **silhouette method** adopts an approach very similar to the elbow method. As in the elbow method, we evaluate different values of k to determine the optimal one. For each point, we measure the distance to the other clusters, denoted as b(i), which is calculated based on the centroids of the clusters. From this, we subtract the proximity (or distance) to the points within the same cluster, denoted as a(i). Afterward, we compute the average silhouette score across all points and select the value of k that maximizes this average. The formulas used are as follows:

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

$$\text{SilhouetteMeanKi} = \frac{1}{n} \sum_{i=1}^{n} S(i)$$

The classes are organized hierarchically, allowing for the dynamic instantiation of different initialization methods. The main functions inherited by each class from `KInitMethods` are:

- `Constructor`: Takes a reference to a `KMeans` instance. This ensures access to the metric used for clustering and the set of points.
- `findK`: Responsible for determining and returning the optimal number of clusters (k).

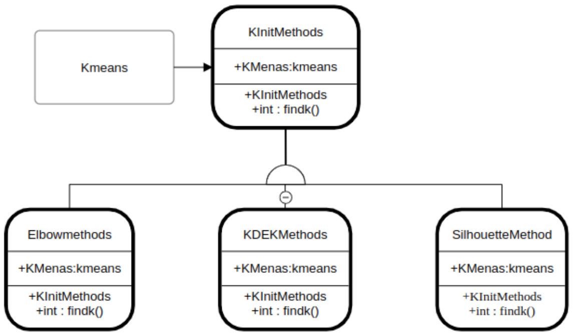


Figure 9: K - Initialization methods

5 Discussion

In this chapter, we discuss the results of our analysis on segmentation quality and algorithm scalability. We evaluated the performance using standard error metrics and examined how well the algorithms scaled with increasing data and parallel processing. At the end, we analyze the impact of different methods for initializing centroids.

System A specifications for Analysis

- CPU: 8 × 4200 MHz
- Cache Levels:
  - L1 Data: 48 KiB (×4)
  - L1 Instruction: 32 KiB (×4)
  - L2 Unified: 1280 KiB (×4)
- GPU: GTX 4050 laptop
  - Memory: 6 GB GDDR6
  - Max Block Size: 1024 threads
  - Warp: 32 threads

System B specifications for Analysis

- CPU: 96 × 2000.1 MHz
- Cache Levels:
  - L1 Data 32 KiB (x48)
  - L1 Instruction 32 KiB (x48)
  - L2 Unified 1024 KiB (x48)
  - L3 Unified 39424 KiB (x2)

5.1 Metric analysis

First, we conducted an evaluation of segmentation quality by comparing different methods using three error metrics: RandIndex, Hamming, and GCE. The results are achieved by averaging over all the dataset provided by Princeton Segmentation Benchmark [6]. The bar chart provides a comparative visualization of the error rates obtained for each method, allowing us to assess their performance.

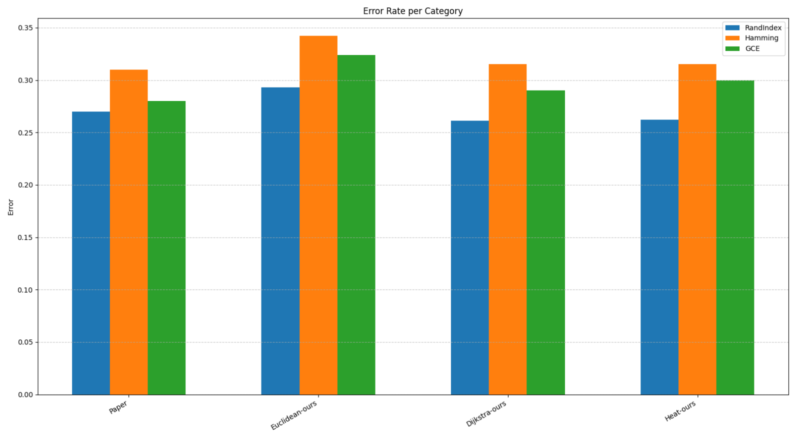


Figure 10: Quality evaluation on different metrics

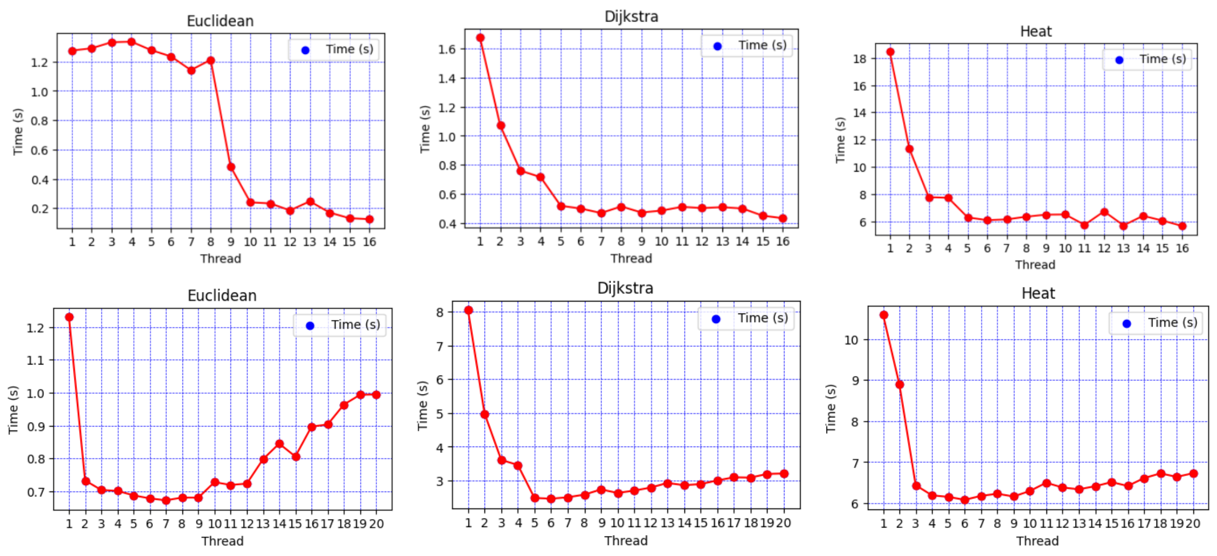


Figure 11: Strong scalability for different metrics, System A (upper row) and System B (bottom row)

The **Hamming Distance** is a region-based metric used to measure the overall difference between two segmentation results. It quantifies how much one segmentation deviates from another by comparing their respective regions. To compute this, the *Directional Hamming Distance* is first determined, which identifies the unmatched regions when comparing one segmentation to another. The *Hamming Distance* is then defined as the average of these two rates.

The **Rand Index** is a metric that evaluates the similarity between two segmentation results by measuring how consistently pairs of faces are grouped together or kept separate in both segmentations. It provides an indication of the likelihood that a pair of faces belong to the same segment in both segmentations or are placed in different segments in both cases. Given two segmentations, S_1 and S_2, the Rand Index considers all possible face pairs in the dataset and determines whether each pair is classified consistently across both segmentations. The Rand Index value ranges from 0 to 1, where a higher value indicates greater similarity between segmentations. To maintain consistency with other error-based metrics, it is common to report $1-\mathrm{RI}(S_1, S_2)$ so that lower values correspond to better segmentation results.

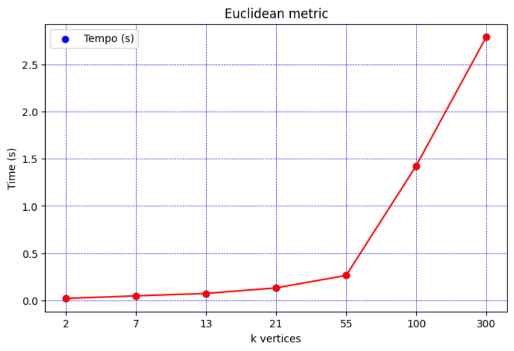
The **Consistency Error**, accounts for nested, hierarchical similarities and differences in segmentations. Based on the theory that human perceptual organization imposes a hierarchical tree structure on objects. Given two segmentations, this metric measures how much one segmentation can be considered a refinement of the other. Two main consistency error metrics are introduced: *Global Consistency Error* (**GCE**) and *Local Consistency Error* (LCE). GCE ensures that all refinements occur in the same direction, meaning that one segmentation should be a strict refinement of the other. In contrast, LCE allows for refinements in different directions for different regions of the model. In this document only the GCE has been reported but the source code is able to calculate also the LCE.

Euclidean metrics exhibits higher errors compared to the others, as expected. Dijkstra and Heat show improvements over Euclidean, with lower error rates in RandIndex and GCE, indicating that shortest-path and heat-based distances enhance segmentation quality. Notably, the trends in error values are consistent with those observed in the Paper method, confirming the reliability of the evaluation metrics.

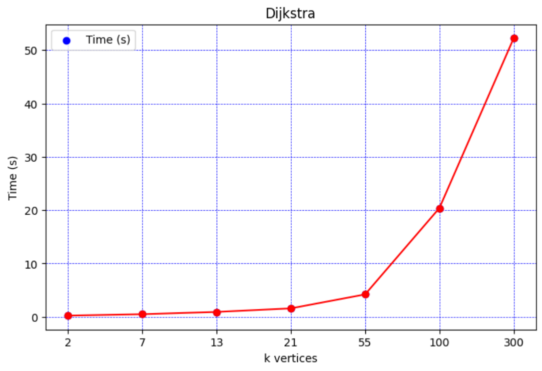
5.2 Scalability tests

Discussions on implementations continues with an analysis of **strong** and **weak scalabilities**, carried out on the main algorithms available. The **strong scalability** tests were all conducted using the same mesh, number 305, to ensure consistency in the results. The 'mostDistant' method was used for initializing centroids, as it is the only fully determin-

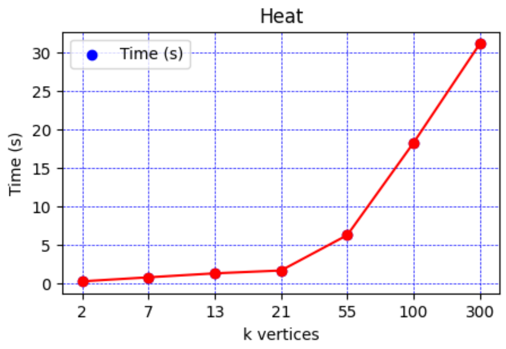
istic method, with the number of centroids set to 5. We can observe in Figure 11 that all three metrics scale very well as threads increase. In addition, since we do not have a sharing of variables across threads we do not have a strong degradation in performance as we exceed the number of threads available. We can conclude that all three metrics are well paralleled.
To measure strong scalability on **GPUs**, however, several TPB (Threads Per Block) values were considered. 13



(a) System A - Euclidean Weak Scalability



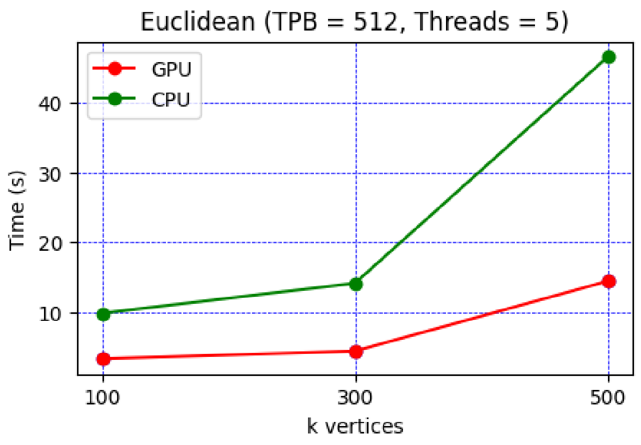
(b) System A - Dijkstra Weak Scalability



(c) System A - Heat Weak Scalability

Figure 12: Weak scalability analysis for different metrics in System A

We now turn to the analysis of **weak scalability** 12. For this study we use 7 different meshes with different dimensions. Specifically, the meshes have correspondingly 7, 13, 21, 55, 100, 300 vertices.
In these solutions we considered the times per iteration. this means that we studied scalability by considering the average times each iteration of the fitting took, rather than considering the time to conclude the algorithms. The reason is that otherwise the times would have been inconsistent with each other.
Recall that an algorithm scales correctly if, as the amount of data processed increases, the increase in processing time is not directly proportional but weaker.
For weak scalability on **GPUs**, meshes corresponding to 100, 300 and 500 vertices were used.
The scalability of the algorithms is quite good to a point where we assumed the presence of cache misses.



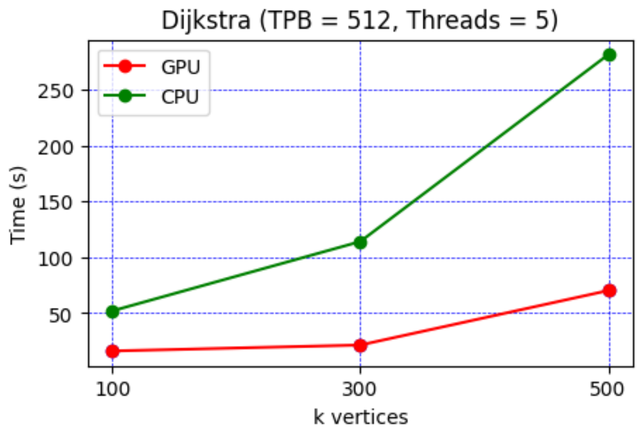
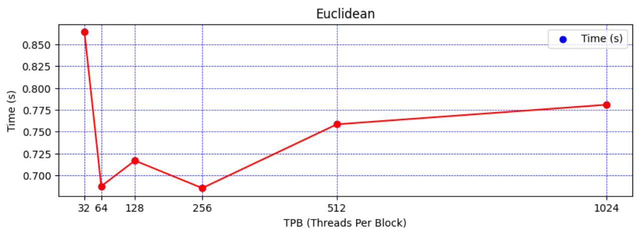


Figure 14: GPU vs CPU Weak Scalability



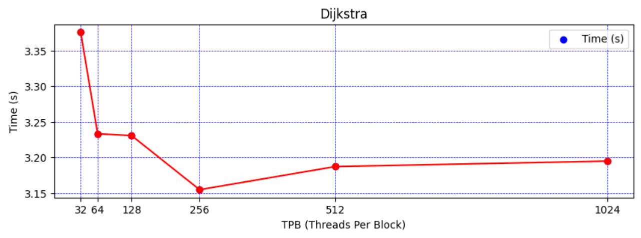
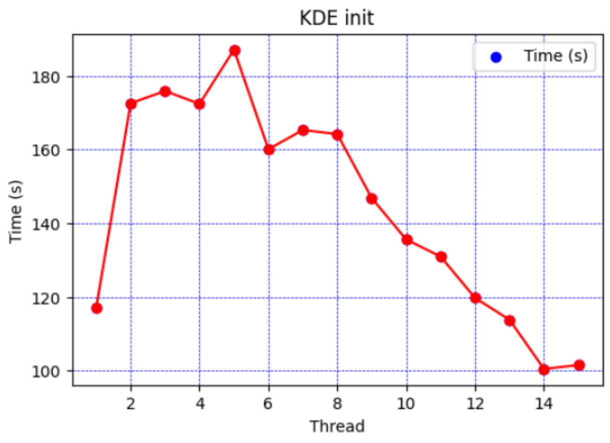


Figure 13: Strong scalability for GPU, System A

5.3 Centroids initialization analysis

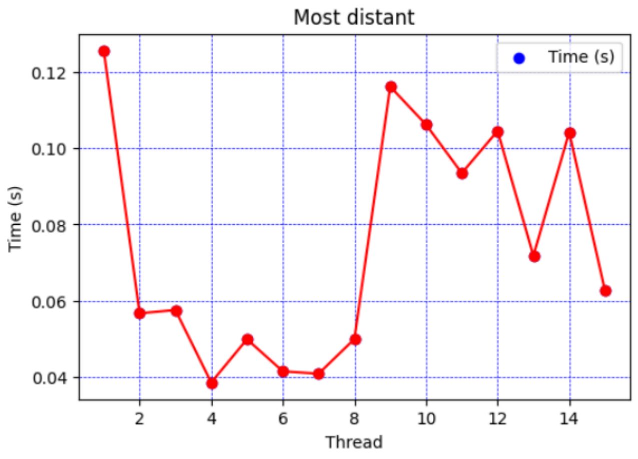
We continued our analysis by examining the scalability of algorithms for centroid initialization (excluding the random approach). In the first part, we will analyze strong scalability, while in the second, we will focus on weak scalability.

Strong scalability: for both methods available for initializing centroids, we performed the study on mesh 125 with 5 centroids. We can observe that the KDE, although a very slow algorithm, scales very well as the number of threads increases.



System A

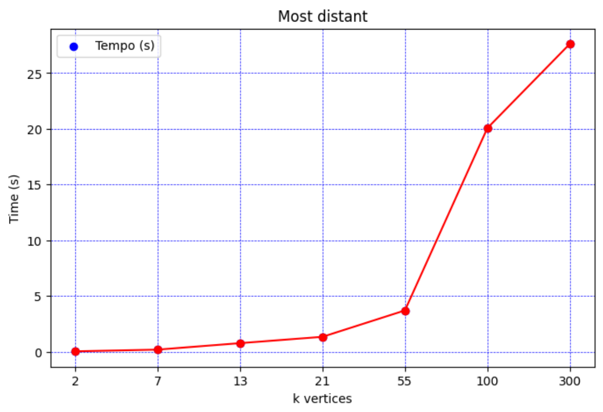
For the "Most Distant" method, the scalability is effective up to 8 threads. The reason is that the algorithm relies on shared variables, and beyond 8 threads, the overhead from value passing and synchronization slows down the execution.



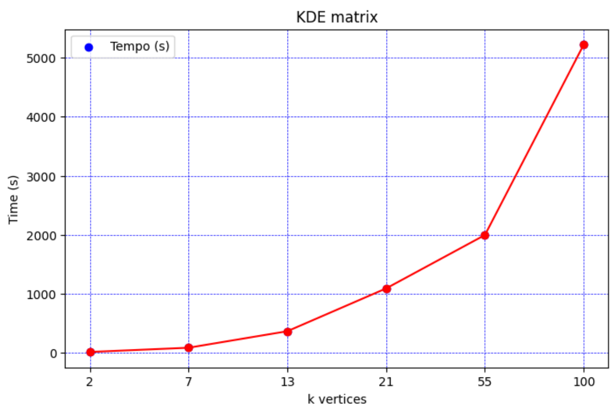
System A

Regarding **weak scalability**, we performed the tests with meshes with different amounts of faces and vertices. Specifically, the selected meshes have 2, 7, 13, 21, 55, 100 and 300 thousand points.

As we can observe on system A the scalability of Most distant and KDE algorithms is not particularly effective. Nevertheless, we can rule out a quadratic increase in the timing



System A



System A

6 Render Application

The Render App is a real-time 3D visualization and segmentation tool that enables users to load, process, and analyze mesh models interactively. It is designed using modern OpenGL alongside several external libraries to ensure efficient rendering and seamless user interaction.

6.1 Implementation Details

The application is built on a modular architecture, where different components handle mesh loading, rendering, user interaction, and segmentation processing separately. The core functionalities of the Render App are:

- **Mesh Loading:** The application supports 3D models in `.obj` format, leveraging the `tinyobjloader` library [7].
- **Rendering:** Rendering is handled using OpenGL, with `glad` [8] as the OpenGL function loader and `GLFW` [9] for window and input management.
- **Shaders:** Custom GLSL shaders define the rendering pipeline, applying Phong shading [10] for realistic lighting.
- **Camera System:** The camera is managed using the `glm` (OpenGL Mathematics) library [11], allowing for smooth object navigation and transformations.
- **User Interface:** The UI is implemented using `Dear ImGui` [12], providing an interactive panel for model selection, segmentation parameter tuning, and real-time visualization.
- **Asynchronous Processing:** To maintain UI responsiveness, segmentation runs in a separate thread using `std::async` [13], preventing the main rendering loop from blocking user input.
- **Segmentation:** Mesh segmentation is implemented using k-means clustering with various distance metrics, including Euclidean and geodesic distances [14].

6.2 Full-Screen Loader and Performance Optimization

To ensure a smooth user experience, the application incorporates a full-screen semi-transparent loading overlay while segmentation is in progress. This is achieved by dynamically updating an ImGui window that covers the entire screen, providing a visual indication of ongoing computations. The segmentation runs asynchronously, with the loader displaying until computation completes, at which point the segmented model is automatically rendered.

Performance optimizations include:

- Efficient OpenGL buffer management (`glGenBuffers`, `glBufferData`).
- Use of Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs) for reduced CPU-GPU communication overhead.
- Shader-based lighting calculations for real-time rendering without CPU-bound computations.
- Use of `std::thread` and `std::future` to manage segmentation asynchronously without affecting UI performance.

This combination of efficient rendering, asynchronous computation, and interactive UI makes the Render App a powerful tool for analyzing and visualizing 3D segmented meshes.

Third-Party Dependencies

The viewer relies on the following third-party libraries:

- **OpenGL** - A cross-platform API for rendering 2D and 3D vector graphics.

- **glad** - A loader for OpenGL functions.
- **glfw** - A library for managing OpenGL context and window creation.
- **glm** - A C++ mathematics library for graphics applications.
- **imgui** - A graphical user interface (GUI) library for rendering UI elements.
- **plog** - A lightweight logging library for debugging.
- **stb** - A collection of single-file public domain libraries for graphics and image processing.

7 Conclusion

This project provides a powerful and flexible framework for 3D mesh segmentation using **K-Means** clustering leveraging distance metrics such as *Dijkstra's* algorithm and *heat* equation. With extensive features like automatic K-detection, GPU acceleration, parallel processing, and interactive visualization tools, it enables efficient and scalable segmentation of complex 3D meshes.

The implementation supports local execution and Docker-based deployment, making it accessible to a wide range of users. Additionally, the built-in viewer enhances usability by providing an interactive visualization of segmented meshes. The project also includes quality evaluation metrics and benchmarking tools to assess segmentation performance.

The performance evaluation of the segmentation metrics highlights a clear trade-off between speed and quality. The Euclidean distance metric is the fastest but performs the worst in terms of segmentation quality, as it does not take into account the topology of the mesh. In contrast, Dijkstra's algorithm and the heat equation produce similar quality results, both better than Euclidean distance. However, their performance differs based on mesh complexity. *Dijkstra's* algorithm is more efficient for low-poly meshes, providing faster results in simpler models. On the other hand, the heat equation scales better with high-poly meshes, making it the preferred choice for complex structures where computational efficiency becomes a critical factor. These findings suggest that the choice of metric should be guided by the specific requirements of the dataset, balancing speed and segmentation accuracy.

References

[1] K-Means: Getting the Optimal Number of Clusters , (30 Dec, 2024). [Online]. Available: https://www.analyticsvidhya.com/blog/2021/05/k-mean-getting-the-optimal-number-of-clusters/#Methods_to_Find_the_Best_Value_of_K

[2] Gap statistics for optimal number of cluster , (10 Dec, 2024). [Online]. Available: https://www.analyticsvidhya.com/blog/2021/05/k-mean-getting-the-optimal-number-of-clusters/#Methods_to_Find_the_Best_Value_of_K

[3] Kernel density estimator c++ implementation. [Online]. Available: https://github.com/timnugent/kernel-density/blob/master/kde.cpp

[4] Kernel density estimator c++ implementation. [Online]. Available: https://mathisonian.github.io/kde/

[5] Kernel density estimator [Online]. Available: https://en.wikipedia.org/wiki/Kernel_density_estimation

[6] A Benchmark for 3D Mesh Segmentation: https://segeval.cs.princeton.edu/public/paper.pdf

[7] Syoyo Fujita, *TinyObjLoader: A single-file wavefront OBJ loader*, GitHub repository, 2024. Available at: https://github.com/tinyobjloader/tinyobjloader

[8] David Herberth, *GLAD: Multi-language GL/GLES/EGL/GLX/WGL Loader-Generator*, GitHub repository, 2024. Available at: https://glad.dav1d.de/

[9] Camilla Berglund, *GLFW: An OpenGL library for managing windows, input, and events*, 2024. Available at: https://www.glfw.org/

[10] Bui Tuong Phong, *Illumination for Computer Generated Pictures*, Communications of the ACM, 1975. Available at: https://dl.acm.org/doi/10.1145/360825.360839

[11] Christophe Riccio, *GLM: OpenGL Mathematics*, GitHub repository, 2024. Available at: https://github.com/g-truc/glm

[12] Omar Cornut, *Dear ImGui: A bloat-free graphical user interface library*, GitHub repository, 2024. Available at: https://github.com/ocornut/imgui

[13] Anthony Williams, *C++ Concurrency in Action*, Manning Publications, 2019. Available at: https://www.manning.com/books/c-plus-plus-concurrency-in-action

[14] Stuart P. Lloyd, *Least squares quantization in PCM*, IEEE Transactions on Information Theory, 1982. Available at: https://ieeexplore.ieee.org/document/1056489