Efficient Implementation of Polyline Simplification for Large Datasets and Usability Evaluation

Şadan Ekdemir
Abstract

**Efficient Implementation of Polyline Simplification for Large Datasets and Usability Evaluation**

Şadan Ekdemir

An in-depth analysis and survey of polyline simplification routines is performed within the project. The research is conducted using different simplification routines and performing evaluative tests on the outputs of each simplification routine. The project lies in between two major fields, namely Computer Graphics and Cartography, combining the needs of both sides and uses the algorithms that are developed for each field. After the implementation of the algorithms, a scientific survey is performed by comparing them according to the evaluation benchmarks, which are performance, reduction rate and visual similarity. Apart from the existing routines, one new simplification routine, triangular routine is developed and recursive Douglas-Peucker routine is converted into non-recursive. As a preprocessing part, Gaussian smoothing kernel is used to reduce noise and complexity of the polyline, and better performances are achieved. The end of research shows that there is no best model instead there are advantages and disadvantages of each simplification routine, depending on the prior need. It is also shown that usage of Gaussian smoothing as a filtering process improves the performance of each simplification routine.
To all, whom I met in my life that taught me how to live, how to think, how to feel and how to comprehend the world.

And to my grandfather, who has unfortunately passed away this summer.

Not of a greater debt that I have to my dear parents, to my sister, Sibel for her invaluable support, for holding my hand within each step I have taken in life, to my friends, and Susanne for giving me peace, I owe.
Acknowledgments

First and foremost, I would like to thank my parents and my sister for all their invaluable support on my survey of finding what I actually want to do in my life, and that I could have brought myself so far.

I owe a big thank to my dear teacher, Maya Neytcheva for giving me self confidence in the project work I did with her, and introducing me an amazing option for my thesis project and providing contact people from the Schlumberger Company. Special thanks to Jørn Letnes, who has been the supervisor of my thesis project and his inputs on my project progress. Also to Stefan Seipel, my reviewer, who has been overseeing my thesis project project and providing his own ideas.

Apart from all, I would like to thank Cris Luego, who has been great of help to with all his voluntarily and invaluable support and interest on my thesis project, generating new ideas and assisting me with my never ending questions. Lastly, very special thanks to Elmar de Koning for his inspirational work on polyline simplification and his personal support on my project.

Computer Graphics has been my favorite field, and I am deeply thankful to everyone who made this project possible.
1 Background

In computer graphics, polygonal models currently dominate a huge portion in interactivity. This is mainly because of their mathematical simplicity; polygonal models lend themselves to simple, regular rendering algorithms that is embedded well in hardware, which has a significant rendering acceleration in turn. However, the complexity of these models seem to grow faster than the ability of our graphics hardware to render them interactively. In other words, the number of polygons we want always has the tendency to exceed the number of polygons we can afford.

As a solution to that problem, various polygonal simplification techniques are used for grappling with complex polygonal models. The main function of these methods is to simplify the polygonal geometry of small, distant, or unimportant portions, looking for available parts to reduce the rendering cost without a significant loss in the visual content of what is seen on the scene [?]. Indeed, this is a both current and a very old idea in computer graphics, thanks to the research of James Clark, Communications of the ACM, in 1976 [?], which is called levels-of-details. At that time, computers were monolithic and rare, and work on graphics was mainly driven by researchers. The hardware itself was completely different, both architecturally and performance-wise. Therefore the simplification was a crucial factor in terms of performance. The following figure depicts one example of level-of-detail on 3-dimensional modeling 1.

![Figure 1: Representation of LOD in action on computer generated version of Mozart’s statue](image)

The original algorithm was presented as a generic approach which would be convenient for all kinds of polygonal operations with the following concern: 'Increased complexity of a scene, or increased information in the database, has less value as the resolution limits of the display are approached. It makes no sense to use 500 polygons describing an object if it only covers 20 raster units of the display.' Then comes the actual question: How can we select only the portion of the database that has meaning in the context of the resolution of the viewing device? To put in other way, we would like to present the minimal information needed to convey the meaning of what is being viewed. As an example, if we are to view a human body from a sufficiently large distance, we might only need to present the 'specks' for the eyes, and perhaps just a block for the head, instead of presenting the entire details and features of the body and face. [2]

With no doubt, there are various need for the simplification of polygonal objects, to eliminate the redundant geometry in volumetric surfaces, or reducing model size for a comprehensive and

---

1 Image is generated by using OpenSG 2 Vision (http://www.opensg.org/)
attainable size for web amplifications (to overcome the bandwidth bottleneck), or improving the performance by simplifying the polygonal scene being rendered. In this project, the last option, which is to improve the performance by simplifying the polygonal scene being rendered and an additional goal, increasing the FPS (frames-per-second) rate while being viewed on the screen. The most common use of polygonal simplification is to generate levels of detail (LODs) of the objects in the scene. By representing distant objects with a lower LOD and near objects with a larger LOD can help to accelerate rendering and increase interactivity by decreasing the workload on graphics pipeline stages, especially at vertex transformations. There are various methods and algorithm handling this issue, and in this project, several algorithms are examined and certain improvements are done. As an important remark, in this project, the scope is narrowed down into polylines from polygons and polymeshes and mainly focused on the performance results with large datasets.

### About Schlumberger

Schlumberger is the leading oilfield services provider, trusted to deliver superior results and improved E&P performance for oil and gas companies around the world. Through the well site operations and in our research and engineering facilities, they are working to develop products, services and solutions that optimize customer performance in a safe and environmentally sound manner.

One of the important goals of the company is to provide visual solutions for oil companies that are aiming to drill a landscape. The company uses a graphical software called Petrel™ which is for visualizing the datasets of polygons (that are coming from seismic images) taken from the interested geographical landlines. The main idea of the project is implementing the level of detail (LOD) approach into the software.

Petrel geophysics software provides a full spectrum of geophysical work flows, including 2D and 3D interpretation, a full set of complex volume and surface attributes including ant tracking for the identification of faults and fractures, volume interpretation (geobody detection) with seismic cross-plotting and classification, domain conversion, and the modeling-while-interpreting functionality, which enables interpreters to build a structural framework while doing their interpretation [?]. Specifically, the part that is mainly involved is called Petrel Well Design. Petrel Well Design wells interactively by digitizing the path directly in the 3D window on any type of data including raw seismic, property models, or simulation results. Figure 2 contains two screenshots from Petrel software usage in action.

### 2 Objectives

The aim of this project is to design and implement an efficient and robust algorithm for polyline simplification. Main focus is on applying algorithms on large datasets, analyzing and comparing existing methods by certain evaluation benchmarks. Polylines can be considered as a subset of polygon simplification and LOD approach, but instead working on surfaces or polymeshes, polylines are the main area of interest in this project. It would not be wrong to state that there are two major areas that are using polyline simplification, computer graphics and cartography. Computer graphics field is more interested in simplification of 3D objects and volume rendering of large datasets. Cartography field is focused on simplification of polylines (i.e. that are used in construction of maps and several GPS systems) with relatively smaller datasets. This scope of this project lies in between these two fields and aims to take the best
sides of these two fields in order to get an efficient implementation of simplification. The methods developed for polyline simplifications are used in both fields and in this report, some of those methods are discussed and several improvements are performed.

3 Related Work

As it is stated, polyline (or polygon) simplification approach is both used in Computer Graphics field (LOD) and also in Cartography field. Before going deep into the project, the reader might wonder the usage areas of the simplification approach in both fields. The major need for LOD approach in graphics field has risen when virtual reality applications exceed the capacity of modern graphical hardware. These scenes have complex structures and their display requires a huge number of polygons, even when only a little portion of the scene that is visible for the given frame. In 1976, it is suggested to use simpler versions of the geometry for objects that
had lesser visual importance, such as those would be far away from the viewer, and this simplification approach is called Levels of Details (LODs)\[7\].

There are many aspects of LODs, one of which is to filter the geometry to produce a model with fewer polygons. The aim of the polygonal simplification is to remove primitives from an original mesh in order to produce simpler models which retain the important visual characteristics of the original object. The idea, in order to maintain a constant frame rate, is to find a good balance between the richness of the models and the time it takes to display them (i.e. rendering time). In terms of describing a way to classify the kinds of hierarchies used to build a simplification algorithm, there are three main classes:

- **Discrete**: Discrete LOD hierarchy encodes a few LODs at very coarse granularity. They are simple to use, and offer the advanced that each LOD may be converted into an optimized form of rendering.

- **Continuous**: This is a very fine-grained discrete LOD hierarchy. It allows more fine-grained selection of the number of primitives to use for representation of the object. This method has many advantages as it can adjust the levels of detail in the run time. It provides, better granularity, which leads to have better fidelity and smoother transitions in different parts of the simplified object.

- **View-dependent**: This is the most complex form of LOD representation, which allows not only fine-grained selection of LODs but also permits the detail to be varied across different portions of the same object, based on viewing parameters. For example, closer portions of an object can be shown in more detail than farther ones\[7\].

In Computer Graphics, meshes are the most widely used type of model in LOD applications, whereas in Cartography, polylines and line generalization are used as the type of model. In Computer Graphics, 3-dimensional geometry plays an important role in terms of visual quality, the trade-off between retaining of the details and rendering speed and such, in Cartography, vectorial displacement measure is the key-factor \[7\]. Moreover, Cartography mainly deals with 2-dimensional (or even 1-dimensional) polylines (i.e. landlines in maps and topological representations), and Computer Graphics deal with meshes and polygonal surfaces. Hence, the algorithms for LOD varies depending on the usage of the field.

In this project, algorithms for polyline simplification are taken from the ones that are used in Cartography field, as the main concern is to perform simplification on seismic image representation, but for the evaluation of the different simplification routines, the criteria which are mainly used by Computer Graphics are used as the software generates a 3-dimensional image after the simplification.

### 4 Theory and Applications

Line simplification is an important function in cartography, for topological representations and also is widely used in commercial GIS software packages. Most line simplification algorithms require the user to supply a tolerance value, which is used to determine the extent to which simplification is to be applied. Apart from that, it is also possible and might be a good option to determine the optimal tolerance automatically, depending on the variation on the dataset. Simplification algorithms weed from the line redundant or unnecessary coordinate pairs. The majority of simplification algorithms do not, however, operate by identifying this redundancy,
but calculate and retain the prominent features of the polyline. Most simplification routines select the critical points based on the topological relationships of points and their neighbors, where the extent of this neighborhood search varies greatly between the algorithms. While such simplification routines only retain or eliminate coordinate data, smoothing algorithms displaces points in attempt to reduce angularity and give the line a more smoother appearance. 

All line simplification algorithms induce positional errors in the data set, because of the fact that they produce a discrepancy between the original line and its simplified version. The amount of this error depends both on the tolerance value and the shape of the line itself. What is usually important from the user perspective, is to maintain a specific level of quality, and not the tolerance value itself. Furthermore, line simplification involves the selective elimination of vertices along a line to remove unwanted information. This determining of the removal of unwanted information is an important process, as it directly effects the visual perceptual quality of the simplified version.

In order to clarify the need for line simplification and the main considerations for the data elimination, we can say that there are three major considerations:

- Reduced storage space: this may reduce a dataset, which will result in faster data retrieval and management, and also rendering and putting on the screen
- Faster vector processing: for example, a simplified polygon boundary would enable us to reduce the number of boundary segments to be checked for shading or point to point interaction within the polygon
- Reduced rendering time

4.1 Line simplification algorithms

For those and many other reasons which are specified above, there are several line simplification algorithms and they all have different strengths and weaknesses for specific conditions. Simplification algorithms may be clustered as follows:

- Independent point routines
- Localized processing routines
  - Unconstrained extended local processing routines
  - Constrained extended local processing routines
- Global routines

4.2 Independent point routines

Independent point algorithms are rather simple in nature and do not take into account the mathematical relationship of neighboring co-ordinating points. It is basically focused on removing a specific point which is predefined from the original polyline. An example of this method could be the $n$th point routine.
4.2.1 Nth point routine

Nth point routine is a naive $O(n)$ algorithm for polyline simplification. It basically retains only first, last and each $n$th point on the original polyline. As can be imagined, these routines are very much computationally effective, however, they are not acceptable under the consideration of accuracy, as it doesn’t check any of the curvature information along the polyline[?].

In Figure 3, an illustration of $n$th point routine can be seen.

![Figure 3: Illustration of $n$th point routine for a 10-point polyline where $n=3$](image)

The illustration above shows a polyline consisting of 8 vertices: $\{v_1, v_2, \ldots, v_8\}$ and the simplification process by using $n$th point routine while $n = 3$. The resulting simplification is including vertices: $\{v_1, v_4, v_7, v_8\}$. The algorithm is extremely fast, but unfortunately it is not very good at preserving geometric features (i.e. curvature information) of the line. For a better representation of this unwanted situation, a 100-point polyline with randomized points is generated which can be seen on Figure 4. The original line is represented in black and the simplified version is represented by the red line.

![Figure 4: Illustration of $n$th point routine for a 100-point polyline where $n=3$](image)

4.3 Local Processing Routines

This category regards a relationship between every two or three consecutive original points. Two examples for this relation can be told as;

- the distance between the two consecutive points
- the perpendicular distance from a line connecting two points to an intermediate point

These distances should not be smaller than their individual tolerance bandwidths (a user defined tolerance or angular change). Points within the bandwidth are eliminated, whereas points
exceeding the bandwidth are retained. [6] A fundamental routine for this case would be Perpendicular distance routine, which is illustrated below.

![Illustration of perpendicular distance algorithm](image)

**Figure 5: Illustration of perpendicular distance algorithm**

**Unconstrained Extended Local Processing Routines**

This category means that it is an extension to the regular local processing routines, and it does not have a constraint; it evaluates relations over sections of the line. One good example for this category is Reumann-Witkam routine.

### 4.3.1 Reumann-Witkam Routine

The method creates line strip and traverses the original polyline by moving this strip. This strip has the thickness which is the tolerance value (either user defined or calculated) and it is created by connecting the successive points in the polyline. The first strip is created by connecting the first two points of the polyline, and then the strip is shifted over the polyline into the direction of its initial tangent, until the strip hits the line. By following the strip, for each incremental vertex \(v_i\), its perpendicular distance to this line is calculated. A new key index is found at \(v_{i-1}\), when this distance exceeds the specified tolerance value. The vertices \(v_i\) and \(v_{i+1}\) are then used to define a new line and line strip. This process is done iteratively, until the last point of the polyline is reached[7].

There is one other interesting method under this category, which is a sleeve-fitting polyline simplification method, proposed by Zhao and Saafeld (1997). The algorithm is similar to the Reumann-Witkam routine as it also divides the original line into sections, but not by using line strips, by using a rectangle (with user defined width). Due to time considerations this method is not used within the context of the project, therefore will not be explained in detail.
4.3.2 Triangular Routine

This method is my own invention, idea came to my mind after checking some of the existing algorithms, especially the one called 'Perpendicular Distance' routine. Perpendicular distance routine is a method that uses a point-to-segment distance tolerance. For each vertex $v_i$, its perpendicular distance to the segment $S(v_{i-1}, v_{i+1})$ is computed. All vertices whose distance is smaller than the given tolerance are removed from the original polyline. In action, initially the first three vertices of the polyline are processed, and the perpendicular distance of the second vertex is calculated. After comparing this distance against the tolerance, the second vertex is considered to be a part of the simplification. The algorithm continues by moving one vertex up on the polyline and applies the same method for the successive set of three vertices and checks the perpendicular distances to the line segments. The calculated distance falls below the tolerance and this intermediate vertex is removed. The algorithm shifts over the polyline until the last point is reached.

What triangular routine does can be described as one step further than the perpendicular distance routine. After finding the perpendicular distance from the vertex $v_i$ to the line segment, the length of the line segment $S(v_{i-1}, v_{i+1})$ is also computed. So, a triangle $T$ with the vertices
$T(v_{i-1}, v_i, v_{i+1})$ is formed with calculated segment and height values. With this information, by applying the geometrical formula to compute the area of triangle, the area is obtained. If the area value of the specific triangle is greater than the tolerance value, the vertex $v_i$ is kept and it becomes the new first point for the next triangle. If the condition is not met, vertex $v_i$ is omitted and the new first point for the next triangle becomes $v_{i+1}$. This process continues until the last point of the polyline is reached. Figure 7 depicts the triangular routine.

The reason why I went for calculating the triangular area is mainly inspired by a mathematical measurement method for line simplification method introduced by McMaster [9]. The areal replacement between the original and simplified polyline is defined as a criterion of dissimilarity measurement for the simplification. Although it works very similar to perpendicular distance routine, it differs in the way that it considers both the perpendicular distance between the specific vertex and the line segment and the length of the line segment as well. This is thought to preserve more details in the simplified polygon after the process.

**Constrained Extended Local Processing Routines**

Constrained extended local processing routines are utilizing the criteria in the unconstrained extended local processing routines and using additional constraints to define a search region for the original line. This search region is used to divide original line into sections and make calculations. Therefore altering parameters yields different results, which gives more degrees of freedom for the simplification method. An important example for this routine is Lang simplification algorithm, which was developed by Lang in 1969.
4.3.3 Lang Routine

The search region is defined by the user and the perpendicular distance from a segment connecting two original points to the original points between them. Each search region is initialized as a region containing a fixed number of consecutive original points. The perpendicular distances from the segment to the intermediate points are calculated and if the calculated distance is larger than the user defined tolerance value, the search region is shrunk by excluding its last point and the distances are calculated again. This process will continue until all the calculated perpendicular distances from intermediate points are below the user defined tolerance value, or until there are no more intermediate points. Once all the intermediate points are moved, a new search region is defined by stating at last point of the latest (or most recent) search region. This process is repeated and shifted over the original line, until the last point of the polyline is reached.

Figure 8: Illustration of lang routine for an 10-point polyline where search region = 5
4.4 Global Routines

The methods described above are algorithms which process a line piece by piece, from the beginning to the end. This category describes a different, more holistic approach to the polyline simplification. A global routine differs from local routines by considering the line in its entirety while processing it. The only existing global simplification algorithm which is commonly used in both cartography and computer graphics is Douglas Peucker algorithm. This algorithm is not only a mathematical but also a perceptually superior, as it produces the best results in terms of both vector displacement and area displacement.

4.4.1 Douglas-Peucker Routine

Douglas Peucker algorithm is a method that tries to preserve directional trends in a line using a tolerance factor. This algorithm is also known as the iterative end-point fit algorithm or split-and-merge algorithm. In the original paper by Douglas and Peucker, 1973, the authors describe two methods for reducing the number of points required to represent a polyline. The first point on the line is defined as the ‘anchor’ and the last point as a ‘floater’. These two points are connected by a straight line segment and perpendicular distances from this segment to all intervening points are calculated. In the case that none of the perpendicular distances exceed a user specified tolerance (it is a distance value), then the straight line segment is deemed suitable to present the whole line in the simplified form. If the condition is not met, then that specific point with the greatest perpendicular distance from the straight line segment is selected as a new floating point. This process is repeated, and a new straight line segment is being defined by the anchor and the new floater. Offsets for intervening points are then recalculated as perpendicular to this new segment. This process continues; the line is being repeatedly subdivided into subpolylines with selected floating points being stored in a stack, until the tolerance criteria is met. Once the tolerance criteria is met, the anchor is moved to the most recently selected floater, and the new floating point is selected from the top of the stack of previously selected floaters. This process is repeated successively and eventually, the anchor point reaches the last point on the line, and then the simplification process is completed. At the end of this process, points previously assigned as anchors are connected by a straight line to form a simplified line. Here the important thing to keep in mind is that specifying a low tolerance value results in little line detail being removed whereas specifying a high tolerance value results in all but the most general features of the line being removed. In the Figure 9, Douglas-Peucker routine is illustrated in detail.

At this point, it can be understood that Douglas - Peucker method is a Divide-and-conquer algorithm, where a polyline is divided into smaller pieces and processed. Such algorithms are known to be recursive algorithms, as the same task is done for smaller pieces of the original dataset. Here, the reader should be warned that recursive algorithms are subject to cause stack-overflow problems with large datasets, which is the case in my project. Hence, several modifications are implemented on the original algorithm to convert it into an iterative, non-recursive algorithm to have a more robust and stable method. Further details can be found in Results section.

4.5 Line smoothing

Smoothing is a process by which data points are averaged with their neighbors in a series, such as a time series, or image. The biggest usage of smoothing is the image processing, but it is also used as a filtering method, simply because smoothing has the effect of suppressing high
frequency signal and enhancing low frequency signal. As Robert B. McMaster has stated in his article in 1992 [?], smoothing algorithms can be used as a part of line simplification algorithms, as they are capable of displacing points in an attempt to reduce angularity and give the line a more flowing appearance. Apart from that, using a smoothing kernel may help to reduce the dataset size as noise can be reduced and less points would be needed to retain the details of the original line. Since the dataset type I had in my project is very large and having a very fluctuating character, using smoothing algorithm is thought to be a doable idea for my project. There are many different methods of smoothing, but in this project, I have used and implemented smoothing with a Gaussian kernel.

Gaussian filter (or kernel) is a windowed filter of linear class, by its nature is weighted mean. Named after famous scientist Carl Gauss because weights in the filter calculated according to Gaussian distribution - the function Carl used in his works.

The reasons why line smoothing is decided to be used in the project are:

- Reducing the number of data points without losing the curvature information
• Reduce the complexity for the usage of global routines (refer to complexity analysis)

Gaussian filter

For 1-Dimensional case, the one-dimensional Gaussian filter has an impulse response given by:

\[ g(x) = \sqrt{\frac{a}{\pi}} e^{-ax^2} \]  

(1)

and when the standard deviation is added as parameter, we obtain a very general formula:

\[ g(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-a)^2}{2\sigma^2}} \]  

(2)

Here the plot of the equation is depicted below:

![Gaussian distribution illustrated](image)

In our 1-dimensional case, we can presume parameter \( a \), which is called *distribution mean* or *statistical expectation*, responsible for distribution shifting along x-axis to be zero, i.e. \( a = 0 \) and then we can obtain the simplified form given below:

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

(3)

From this, it can be seen that the function is negative exponential one of squared argument. Argument divider \( \sigma \) plays the role of scale factor. \( \sigma \) parameter has a special name, *standard deviation* and its square \( \sigma^2 \) gives the *variance*. (Practically, \( \sigma \) can be called as the 'key factor' for the simplification, and can be said that lower value of \( \sigma \) gives more accurate (or less simplified output)). Here the reader must be aware that the function is defined everywhere on the real axis \( x \in (-\infty, \infty) \), which means it spreads endlessly to the left and right hand sides.

At this point, the first thing is that we are working in a discrete realm, so our Gaussian distribution should not work on infinite values, instead it should turn into set of values at discrete points.

Secondly, our Gaussian distribution must be truncated into a frame, to be discretized. In my approach, I have selected the value of \( 2\sigma \) as the truncation values. So the yielding plot can be depicted as follows:

Thus, depending on which value is selected for \( \sigma \) (which is user-defined), the area under the truncated part can be calculated by the following formula:
Figure 11: Gaussian distribution truncated at points ± 2σ illustrated

\[
S = \frac{1}{\sqrt{2\pi\sigma}} \int_{2\sigma}^{2\sigma} e^{-\frac{x^2}{2\sigma^2}} dx = \frac{1}{\sqrt{2\pi}} \int_{2}^{2} e^{-\frac{x^2}{2}} dx
\]  

(4)

After obtaining this relationship, we can use those as window weights. That means we can calculate the values on the 1-dimensional dataset by using the integral, and using convolution to smoothen the data. Here, the key point is the accuracy; by altering the σ value, we can tune the noise in the dataset.

The convolution of the curve with the filter is computed in a discrete space, where the line has first been re-sampled. Every point in the 1-dimensional curve has a homologous point on the smoothed line so that holding the following equality:

\[
y_n = \sum_{k=2\sigma}^{2\sigma} y(i - k)g(k)
\]  

(5)

where k stands for the each discrete point within the filter. The value of σ (number of neighboring points taken into account to compute an average position) characterizes the smoothing scale. The higher the σ value, the stronger the smoothing. Thus, the choice of the σ value depends on the level of analysis. As a side note, in my implementation, I have created subsamples to skip while applying convolution, as they are calculated by the Gaussian kernel, there is no point of computing them individually. More details can be found at the Appendix part.

Here is an rough example to show the smoothing effect of the Gaussian filter.

Figure 12: Gaussian smoothing on a dataset size of 20 points, where σ = 0.75

To illustrate how σ value effects the output after the filtering process, the following figures can be observed. (Polyline with red color represents the filtered output)

As it is seen, decreasing σ value is yielding more accurate output results, with the cost of reduced simplification. In this example, when σ value of 1 is used, dataset size is reducing to 63 from 100 points. This would yield remarkable speed-up in the performance when working on
Figure 13: Effect of $\sigma$ value in the output image for dataset size of 100-points

4.6 Measurement & Error Analysis for Line Simplification

Line simplification comes with the cost of positional uncertainty in the output image. As an artifact of the simplification process, the simplified polyline has a different outline than the original polyline, hence causes a shape distortion. In this project, the error measurements are performed in order to check and conclude on the accuracy of the simplification routines. In order to calculate an acceptable surrogate for the perceptual distance, vector displacement measure is concluded to be a convenient method in terms of geometrical closeness in between the original line and simplified line (Jenks, 1985) [15]. It is also clear that larger displacement are the most perceptually significant.

There are two major types of error measurements used within the project; namely positional error sum measure, which is a distance-based measurement in which the perpendicular distances between the vertices of the original line and its simplified version are taken into account, and areal distance measure [9] which calculated the areal difference between the original line and its simplified version.

**Positional Error Sum Measure**

Positional error sum measure derives an uncertainty (or error) description for the original line and the perpendicular distances of each vertex to the simplified line. It basically looks at the locational difference between the original and its simplified version. For each original point, the positional error sum is calculated by summing all the perpendicular distances between each
original line vertex and the corresponding line segment of its simplified version. The positional error sum (PES) can be expressed as:

$$PES = \sum_{i=0}^{nv} vls(i)$$ (6)

where \(nv\) is the number of vector displacements between the original line and its simplified version, \(vls\) is the length of than individual perpendicular distance from the vertex on the original line to the line segment of the simplified line.

Figure 15 depicts the measurement method.

![Figure 14: Positional Error Sum](image)

**Areal Difference Measure**

The positional error sum measure is good at finding the local vector displacements but it does not give sufficient quantification for the global geometrical characteristics of the simplified line. While comparing different simplification methods, one method can create a greater error local error when yielding a smaller global error (i.e. geometrical likeliness of the layout). In order to capture this error measurement, areal difference method is used and areas are calculated by using Trapezoidal rule and Simpsons rule.

In order to calculate the areal difference between the original and simplified line, first the total area of the original line is calculated. After that, total area of the simplified line calculated and the absolute difference between the two areas is determined as the areal difference between the original and simplified line. Total areal difference (TAD) can be expressed as:

$$TAD = \text{Total area under the original line} - \text{Total area under the simplified line}$$
Trapezoidal Rule

Trapezoidal rule is an approximation method for calculating the definite integral

$$\int_{a}^{b} f(x)dx$$

(7)

The way the trapezoidal rule works can be described by approximation of the region under the graph of the function $f(x)$ as a trapezoid and calculating its area.

$$\int_{a}^{b} f(x)dx \approx (b - a)\frac{f(a) + f(b)}{2}$$

(8)
It is a fundamental method but since in the project polylines are taken into account, and each successive vertex can be considered to create a trapezoid with its predecessor vertex. In Figure 17, the idea behind the implemented method is depicted. Each area between the two vertices and the \textit{x-axis} forms a trapezoid and \( A_n \) is calculated by the Trapezoidal rule.

![Figure 17: Trapezoidal Rule for calculating the area under the polyline](image)

**Simpsons Rule**

Simpson’s rule is another type of approximation to evaluate the integral of a function \( f \) using quadratic polynomials (i.e. parabolic arcs instead of the straight line segments used in the Trapezoidal rule described above). Simpson’s rule can be derived by integrating a third-order Lagrange interpolating polynomial fit to the function at three equally spaced points. In a nutshell, we can assume that the function \( f \) can be tabulated at points \( x_0, x_1 \) and \( x_2 \) which are equally spaced by distance \( h \), and can be denoted \( f_n = f(x_n) \). The Simpson’s rule concisely states that

\[
\int_{x_0}^{x_2} f(x)dx = \int_{x_0}^{x_0+2h} f(x)dx
\]  

which equals to:

\[
\frac{1}{3} h(f_0 + 4f_1 + f_2)
\]

after the approximation. Here, the point \( x_1 \) is the midpoint between \( x_0 \) and \( x_2 \). We can replace the integrand \( f(x) \) by the quadratic polynomial \( P(x) \) which takes the same values as \( f(x) \) at the end points \( x_0 \) and \( x_2 \) with the midpoint \( x_1 \). Lagrange polynomial interpolation for this polynomial can be stated as
\[ P(x) = f(x_0) \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f(x_1) \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f(x_2) \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \]  \hspace{1cm} (11)

which can be simplified as

\[ \int_{x_0}^{x_2} P(x) \, dx = \frac{x_2 - x_0}{6} (f(x_0) + 4f(x_1) + f(x_2)) \]  \hspace{1cm} (12)

where \( x_1 = (x_0 + x_2)/2 \), by the Simpson’s rule[?].

In the project, Simpson’s rule is used when calculating the area under the filtered polyline. Gaussian smoothing method is used for filtering and since it alters the local minimum and maximum vertices of the polyline while filtering, a more precise areal calculation method than trapezoidal rule is needed. Therefore Simpson’s rule is used while performing areal calculations for evaluating simplification methods including the filtering as a preprocessing step.

5 Methods

In this section, reader can find detailed information about the methods and modification that are used during the project. Modifications are mainly done at the implementation parts.

5.1 Modified Algorithms

5.1.1 Converting recursive Douglas-Peucker algorithm into non-recursive

As it will be also discussed in detail in the Discussion section, Douglas-Peucker method is able to yield by far the best simplification output, as in comparison to other existing methods. As McMaster [?] stated, this method is ranked as a mathematically superior since it yields really effective results in dissimilarity measurements. Douglas-Peucker method is also considered to be best at choosing critical points and yielding the best perceptual representations of the original lines [?]. Despite having many advantages, Douglas-Peucker method is originally defined as a recursive method which leads to non-robustness and stack overflow issue in large datasets with unstable values.

*Stack overflow* occurs when too much memory is used on the call stack. The call stack is a limited amount of memory which is mainly determined at the starting of the program. Stack overflow typically causes the program to crash and it can be called as bug in the program, hence the cause for that should be avoided. One major cause of a stack overflow results from an attempt to allocate more memory on the stack than it would fit. In other words, using very large stack variables and calling them too many times by the program causes a stack overflow.

*Stack* is a region of memory on which the local automatic variables are created and function arguments are passed. The implementation allocates a default stack size per process and on modern operating systems, a typical stack has at least 1 megabyte, which can be sufficient for most purposes. However, under some special anomalous conditions, the program exceeds its stack limit and this causes stack overflow.

Douglas-Peucker method follows a heuristic improvement by stacking ’anchors’ by a recursive procedure. This routine splits the polyline into two sub-polylines and processes each sub-polyline with the same method. Here, the problem arises when a large dataset is being processed on the stack; too many recursive calls to the same stack lead to stack overflow and cause the program crash. The pseudo code for Douglas-Peucker routine is given in Table 1.
Table 1: Douglas-Peucker recursive algorithm

Suppose that $V$ is an array of vertices, any call to $Douglas\text{-}Peucker(V,i,j)$ does the function of simplifying the subchain from $V_i$ to $V_j$

Procedure $Douglas\text{-}Peucker(V,i,j)$
Find the vertex $V_f$ which is furthest from the line $V_iV_j$
Let $dist$ be the distance
if $dist > \epsilon$ then
    $Douglas\text{-}Peucker(V,i,f)$  // Split at $V_f$ and approximate recursively
    $Douglas\text{-}Peucker(V,f,j)$
else
    Output $(V_iV_j)$
end if

Throughout the work was done on the project, my first task was to implement a robust algorithm for Douglas-Peucker method, therefore I have converted the recursive algorithm into non-recursive by using an internal stack. The non-recursive routine is basically mimicking the original method, just by using internal memory stack. The new pseudo-algorithm is explained in Table 2.

Table 2: Douglas-Peucker non-recursive algorithm

Suppose that $V$ is an array of vertices, any call to $Douglas\text{-}Peucker(V,i,j)$ does the function of simplifying the subchain from $V_i$ to $V_j$

Procedure $Douglas\text{-}Peucker(V,i,j)$
Create the internal stack and add the complete polyline into the stack
while stack is not empty do
    Find the vertex $V_f$ which is furthest from the line $V_iV_j$
    Let $dist$ be the distance
    if $dist > \epsilon$ then
        Add $(V,f,j)$ into the stack  //Split at $V_f$ and add the right part on the stack
        Add $(V,i,f)$ into the stack  //Add the first part after the right part
    else
        Output $(V_iV_j)$
    end if
end while

5.1.2 Employing Gaussian filtering as part of simplification

As it is described in the theory section, Gaussian filtering is a smoothing filter which eliminates the noise from the targeted dataset. In my project I used Gaussian filtering as a preprocessing step and afterwards applied to the simplification routines and got the results. The results can
be seen at the Results section.

5.2 Implementations

For all the implementations, Microsoft Visual Studio 2008 is used. All code is developed in C++ and OpenInventor. The company provided a limited version of the source code of Petrel software and apart from that I have implemented my own framework to run experiments and perform observations by using OpenGL. Each simplification methods are implemented on my own framework first, tested and then merged into the source code of Petrel.
5.3 Algorithms

In the implementation of each simplification routine, a similar method is used:

- Generation of dataset with user defined size, create the original polyline.
- Calling one specific simplification or smoothing method and getting the simplified polyline.
- Apply the error analysis and get the dissimilarity measures.
- Splash the output on the screen by using OpenGL.

Implementation of each routine is explained below. The entire source code can be found in the Appendix section.

Nth point routine

It is a brute-force algorithm, it basically traverses along the polyline and keeps only the first, last and each nth element in the polyline.

```cpp
void nthPoint(Vector3d samples, vector<int> result, int lower_bound, int upper_bound, int n) {

    /* variables:
      * Vector3d samples — original polyline, being passed to the function,
      * not empty
      * vector<int> result — simplified polyline indices, created as empty,
      * stores the key indices
      * int lower_bound — first point index of the polyline
      * int upper_bound — last point index of the polyline
      * int n — nth point
    */

    int key = lower_bound;
    result.push_back(sample_indices[key]);
    // the first point is always part of this simplification

    int k = (upper_bound - 1)/n;
    // number of nth points after first point
    int r = upper_bound - k*n - 1;
    // number of points between the final
    // nth point and last point

    for (int i=lower_bound; i<k ; ++i){
        key += n;
        result.push_back([key]);
    }
    if (r){
        key += r;
        result.push_back([key]);
    }
}
```

Douglas Peucker routine

As described above, Douglas-Peucker routine was initially a recursive algorithm and it lead to cause stack overflows with large datasets with fluctuations. Pursuing a robust algorithm, the recursion is removed from the algorithm by employing an internal memory stack. Apart from this crucial change, the main idea is kept the same as developed by Douglas and Peucker. [7]

Here is the implementation for the recursive algorithm:
void douglasPeucker(Vector3D *samples, int lower_bound, int upper_bound, vector<int> &result, double max_error)
{
    /* variables:
    * Vector3D samples -- original polyline, being passed to the function,
    * not empty
    * vector<int> result -- simplified polyline indices, created as empty,
    * stores the key indices
    * int lower_bound -- first point index of the polyline
    * int upper_bound -- last point index of the polyline
    * double max_error -- error tolerance value defined by the user */

    int max_index = -1;
    // is a check value for determining the key
    KeyPoint key = findKey(samples, samples_indices, lower_bound, upper_bound, error_bound);
    // findKey is a function that calculates all the perpendicular distances
    // from the intermediate vertices to the line segment formed by connecting
    // lower_bound to upper_bound
    max_index = key.index; // key is found
    max_error = key.maxDist; // distance from the key to the line is stored

    // If present, the vertex with the largest error (larger than the error bound)
    // is used to split the region into two, which are then processed recursively.
    if (max_index == -1)
        return;

    vector<int> lhs;
    vector<int> rhs;
    douglasPeucker(samples, lower_bound, max_index, lhs, error_bound);
    douglasPeucker(samples, max_index, upper_bound, rhs, error_bound);

    result.insert(result.end(), lhs.begin(), lhs.end());
    result.push_back(sample_indices[max_index]);
    result.insert(result.end(), rhs.begin(), rhs.end());
}

In the project, non-recursive version is implemented as:
void douglasPeucker(Vector3D *samples, int lower_bound, int upper_bound, vector<int> &result, double max_error)
{
    /* variables:
    * Vector3D samples -- original polyline, being passed to the function,
    * not empty
    * vector<int> result -- simplified polyline indices, created as empty,
    * stores the key indices
    * int lower_bound -- first point index of the polyline
    * int upper_bound -- last point index of the polyline
    * double max_error -- error tolerance value defined by the user */

    typedef std::stack<SubPoly> Stack;
    // the job queue for the stack is created
    Stack stack; // internal stack

    SubPoly subPoly(lower_bound, upper_bound);
    stack.push(subPoly); // add the complete polyline into the stack queue

    vector<int> store(upper_bound, 0);
    store[0] = 1;
    store[upper_bound-1] = 1; // store is the array for storing the key indices

    // the recursion is mimicked here in this while loop
    while(!stack.empty())
    {
subPoly = stack.top();
// take a sub polyline from the top
stack.pop();
// find its key and remove from the job queue

KeyPoint key = findKey(samples, sample_indices, subPoly.first, subPoly.last, error_bound);
if (key.index != subPoly.last && key.index != -1){
  // store the key point if it's valid
  store[key.index] = 1;
  // split the polyline at the key and continue in two sub parts
  stack.push(SubPoly(key.index, subPoly.last));
  stack.push(SubPoly(subPoly.first, key.index));
}

for (int i = 0; i < upper_bound; i++){
  if (store[i])
    result.push_back(i);
}

Lang routine

This routine uses a fixed size search-region and traverses the original polyline with this kernel. The first and last points of the search region form a line segment and it is used to calculate the perpendicular distance to each intermediate point. The detailed explanation was already defined at the Theory section.

void langAlgorithm(Vector3D *samples, int lower_bound, int upper_bound, vector<int> &result, int search_region_size, double max_error)
{
  /* variables:
   * Vector3d samples — original polyline, being passed to the function,
   * not empty
   * vector<int> result — simplified polyline indices, created as empty,
   * stores the key indices
   * int lower_bound — first point index of the polyline
   * int upper_bound — last point index of the polyline
   * int search_region_size — size of the search region
   * double max_error — error tolerance value defined by the user */

  int current = lower_bound;  // the current key
  int next = lower_bound;     // to find the next key
  int temp, moved;

  int remaining = upper_bound - 1;
  // number of points remaining after current position
  temp = min(search_region, remaining);
  next += temp;
  remaining -= temp;
  moved = temp;

  result.push_back(current);
  // first points should always be stored as key

  while(moved){
    int d2 = 0;
    //current += 1;
    int p = current;
    p += 1;
    while (p != next){
      d2 = max(d2, minDistance(samples[current], samples[next], samples[p]));
      // minDistance is a function that calculates the minimum distance (i.e
// perpendicular distance) from a point to a line segment
if (d2 > error_bound) // if distance is greater than the tolerance
    break;
p += 1;
} else {
    // important to keep track of two test points
    int pi = pl; // previous test point
    int pj = pl; // current test point (pi+1)
    // found the next key at pi
    p0 = pi;
    // define new line L(pi, pj)
p1 = pj;
    // first points is always the part of simplification
    for (int j=2 ; j<upper_bound ; ++j){
        pi = pj;
pj += 1;
        if (minDistance(samples[p0], samples[p1], samples[pj]) < error_bound) {
            continue;
        }
        result.push_back(pi); // found the next key at pi
        p0 = pi;
        result.push_back(pj);
    }
    result.push_back(pl);
}

Reumann-Witkam routine

void reumanWitkamAlgorithm (Vector3D *samples , int lower_bound,
int upper_bound, vector<int> &result , double max_error) {
    // variables:
    * Vector3D samples — original polyline, being passed to the function,
    * not empty
    * vector<int> result — simplified polyline indices, created as empty,
    * stores the key indices
    * int lower_bound — first point index of the polyline
    * int upper_bound — last point index of the polyline
    * double max_error — error tolerance value defined by the user
    */
    // define the line L(p0, p1)
    int p0 = lower_bound;
    int pl = lower_bound;
p1 += 1;
    // important to keep track of two test points
    int pi = pl; // previous test point
    int pj = pl; // current test point (pi+1)
    result.push_back(p0);
    // first points is always the part of simplification
    for (int j=2 ; j<upper_bound ; ++j){
        pi = pj;
pj += 1;
        if (minDistance(samples[p0], samples[p1], samples[pj]) < error_bound) {
            continue;
        }
        result.push_back(pi); // found the next key at pi
        p0 = pi;
        result.push_back(pj);
    }
    result.push_back(pl);
}

Triangular routine

void triangularAlgorithm (Vector3D *samples , int lower_bound,
int upper_bound, vector<int> &result , double areal_tolerance) {

5.4 Determination of tolerance value and handling random datasets

In the project, one important point was to determine the tolerance value that is used as criteria for the simplification routine. Among the articles and journals I have been through, I did not obtain a precise information for selecting the optimal threshold and I decided to select on threshold value for all the simplification routines and make the comparisons around this tolerance value. Tolerance value is evaluated by calculating the average value of the distances from intermediate points to the line segment constructed by connecting the first and last points of the polyline.

Moreover, since the dataset generation is about generating purely random datasets, and the values vary each time they are generated, I created 3 datasets with random numbers, and performed the experiments on each dataset, later took the average for the evaluation of the routines.
5.5 Evaluation methods

Several methods and improvements are implemented, and the results are evaluated. At this section, the benchmarking system for the evaluation is described.

Performance, process time and speed

Since we are considering large datasets, performance plays an important role while evaluating a routine. The routine might be yielding a prominently good output but if it takes too long time to process, it might not be plausible to use it, as one of the main concerns to simplify the dataset is to reduce overall rendering time. Therefore, several runs are performed with varying dataset sizes and results are plotted.

Reduction Rate

As result of simplification, certain amount of reduction will be done. High amount of reduction would yield a fast rendering time, however it might cause a long process time or low-quality in the output. Another great impact of reduction rate is on the rendering time. Petrel uses OpenInventor for visualization purpose and it uses OpenGL for rendering the images. Rendering time is directly proportional with the dataset size and it linearly increases with the increasing dataset size\[\]. Therefore, it is another key factor to determine simplification quality.

Error Sum

Another important key point is to quantify the difference between the input and output polylines. In other words, the error in between the input and output should be measured in order to get a clear picture of the simplification result. In this project, the error calculation is done in 2 different ways: shape distortion, which is Positional Error Calculation, and visual perceptual difference, which is total areal displacement. Total areal displacement is calculated by using two different methods, namely Areal Calculation by using Trapezoidal Rule, and Areal Calculation by using Simpsons Method.
6 Results

In this section, the experiment results are given according to the evaluation methods described above.

6.1 Work Bench

Since real-life geographical information datasets are confidential to the Schlumberger company, I decided to work with my own generated data. In order to imitate the real life conditions, a purely randomized dataset is used. Unlike cartographic images, here we have 3-dimensional dataset vertices, one of which is the height, and the simplification is done on that. One sample output to visualize the simplification is given in Figure 18

![Figure 18: Screen shots of Petrel and simplification](image)

(a) original dataset with 1000 vertices

(b) simplified dataset with 281 vertices by using Lang method

After observing the working principle of Petrel, I started developing my own framework by trying use same way of notation as it is done in Petrel. While working on my own workbench, I used OpenGL for visualizing the simplification routines. I decided to put the original polyline and its simplified version on top of each other so that the simplification can be seen clearly
on the original polyline. Figure 19 depicts a sample output for a 150-point polyline and its simplified version by using Douglas-Peucker routine.

![Figure 19: 150-point polyline and its simplified version by using Douglas-Peucker routine](image)

## 6.2 Evaluation of the line simplification algorithms

After implementing all the simplification routines, different versions of the simplified line generated by the specific algorithm are compared according to the benchmarks that are described above. After creating the framework, several datasets are generated with varying sizes. Each dataset is used to generate the original polyline and experiments are performed by using different simplification routines. Sample outputs for 100-point polyline and the simplified versions by different methods are given in Figure 20.

### 6.2.1 Performance Analysis

In this subsection, the performances of different simplification routines are analyzed and compared. The run-time performances for varying dataset sizes from 100 vertices to 1,000,000 vertices are obtained and compared. First of all, as it is described in the previous sections, recursive Douglas-Peucker routine was not robust as it has the possibility to cause stack overflows with large datasets, therefore recursive and non-recursive Douglas-Peucker routines are tested for varying dataset and the Table 3 shows the process times according to the dataset size.

### Table 3: Process times for varying dataset sizes with recursive and non-recursive Douglas-Peucker algorithm

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Process time with Douglas-Peucker non-recursive</th>
<th>Process time with Douglas-Peucker recursive</th>
<th>difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1944</td>
<td>1969</td>
<td>1.28</td>
</tr>
<tr>
<td>10000</td>
<td>7863</td>
<td>7857</td>
<td>0.07</td>
</tr>
<tr>
<td>15000</td>
<td>19372</td>
<td>19375</td>
<td>0.01</td>
</tr>
<tr>
<td>20000</td>
<td>32459</td>
<td>32742</td>
<td>0.87</td>
</tr>
<tr>
<td>22500</td>
<td>36785</td>
<td>stack overflow N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

As it is clearly seen in Table 3, recursive Douglas-Peucker routine fails dataset sizes larger 20,000 data points, which is not convenient to use for large dataset sizes. Therefore, non-recursive Douglas-Peucker routine is used for the performance analysis.

All the results are put into the chart as given in Figure 21. What is seen on Figure 21 is that Douglas-Peucker algorithm works the slowest and after the size of 10,000 points, the
(a) original and simplified version of the polyline with 100 vertices by using Douglas-Peucker routine

(b) original and simplified version of the polyline with 100 vertices by using with nth point routine when n=3

(c) original and simplified version of the polyline with 100 vertices by using with Lang routine

(d) original and simplified version of the polyline with 100 vertices by using with Reumann-Witkam routine

(e) original and simplified version of the polyline with 100 vertices by using with triangular routine

Figure 20: A set of 100-point polyline with randomized dataset and simplified versions of it by using different simplification routines

process takes too long time to be compared with other methods, therefore process time for Douglas-Peucker after that point is disregarded. Moreover, it is also observed that nth-point routine (where n equals to 3) works fastest, compared to other tested routines.

As a different way of thinking, in order to see the comparison of the simplification methods more comprehensively, each method’s speed can be checked. Speed is a relative concept, which in this context means dataset size divided by process time in milliseconds. It can be seen in Figure 22, nth-point routine (where n equals to 3) has the highest speed and also Reumann-Witkam method and Lang method have very similar speeds.
Figure 21: Comparison of routines in terms of performance (process time in ms)

Figure 22: Comparison of routines in terms of speed (dataset size / process time (ms))
6.2.2 Reduction rate comparison

Reduction rate is another important concept while comparing the simplification routines. When simplification routines work, they reduce the number of vertices in the polyline, but at the same time shape distortion may occur as a result of reduction. Reduction rates of each individual simplification method are calculated and compared in Figure 23. There are several observations that can be done by looking at Figure 23: $n$th-point routine gives a constant reduction rate percentage with 65% (which is due to the value of $n$ that is 3), triangular routine gives the least amount of reduction rate percentage with 32%, and Douglas-Peucker routine yields the highest amount of reduction rate percentage with an average of 75% reduction rate.

Figure 23: Comparison of reduction rates of the simplification routines in percentage

6.2.3 Dissimilarity measure comparison

Once the simplification is done on the original polyline, shape distortion appears. This distortion is another important concept to analyze as it refers to visual perception of the simplification result. After obtaining the simplified polylines, dissimilarity measurements are done by measuring: a) positional error sum, b) total areal difference. Positional error sum considers the point-wise local dissimilarities between the original and simplified polyline, and it sums up the calculated perpendicular distances between the original line vertices and its simplified version. Total areal difference on the other hand, focuses on the difference between the areas under the original and simplified polylines. The idea of considering the areas under the polylines is because of the fact that some algorithms may keep the curvature or global features of the original line and this could not be visible when only calculating positional error sum.

Positional error sum

For the sake of simplicity, positional error sum calculation is divided into two parts, for large and small dataset sizes, and the results are given in Figure 24 and Figure 25.

Looking at both of the figures, what can be observed is that Reumann-Witkam routine is giving the highest value of positional error sum, which means the original polyline and its simplified version are differing the most in point-wise perspective. After that, $n$th point and Lang routines yield the highest positional error sums. On the other hand, triangular method is yielding least amount of positional error sum with a very similar error value with Douglas-Peucker routine.
Figure 24: Comparison of positional error sums of all simplification routines for large dataset size

Figure 25: Comparison of positional error sums of all simplification routines for small dataset size
Total areal difference

After comparing the local dissimilarities in between the original line and its simplified version, I have compared total areal differences in between the polylines. Main idea of this approach is to quantify the difference in between the original and simplified polylines by checking the shape similarities rather than local coordinate similarities. For specific dataset sizes, total areal differences are compared with different simplification routine outputs. At this section, in order to calculate the areas, trapezoidal rule is used and the results are shown in Figure 26.

![Areal Error Comparison (Trapezoidal)](image)

Figure 26: Comparison of total areal error sums of all simplification routines

On large dataset sizes, (i.e. as of the dataset size of 10 000 points) the total areal errors are varying depending on the size. Considering that the datasets are randomly generated, and total areal difference is not sufficient to determine which method is having the least error. However, it can be seen that $nth$-point routine is showing a very unstable trend which proves that it does not guarantee any visual quality after simplification. On the other hand, it is observed that triangular method is yield relatively less error than the other routines in overall.
6.2.4 Lang routine parameterization

At this point, there is one important thing is that Lang routine has a parametric value which is search region and is user defined. In the experiments, while comparing with other methods, this value was set to 4 for the sake of simplicity. However, it is observed that Lang routine yields different results with different search region sizes. In the following figures, this effect is shown by the three benchmarks for 4 different search region sizes, namely 3, 4, 5 and 6, respectively.

Figure 27: Impact of different $n$ values on process speed in Lang method

Figure 28: Impact of different $n$ values on positional error sum in Lang method

6.3 Gaussian smoothing

Since the dataset is purely randomized, some sort of filtering method might have an impact on the performance of the routines. As it is described in the Theory section, Gaussian smoothing is a commonly known method for filtering the dataset. In our simplified case, we have a
1-dimensional dataset with randomized values. The idea behind using the Gaussian smoothing is to have a preprocessing by filtering the dataset so that we could have a smoother line with the same characteristics as the original polyline, but having a faster and more accurate simplification process. The emerged results are thought to be interesting enough to be considered for using as a preprocessing method for simplification. Since using filtering as preprocessing method yields slightly different results, it is decided to be given at a different section than the other results.

In order to see how Gaussian smoothing is used as a preprocessing process, Figure 30 can be observed. The black line denotes the original polyline, the yellow line is the filtered version of the original polyline where $\sigma = 0.70$ and the red line represents the simplified version of filtered polyline. It can be seen that local maximum and minimum points of the original polyline are shifted upwards and downwards due to the filtering effect, however, the global curvature characteristics of the original polyline line is retained.

**Determination of $\sigma$ value**

In order to determine which $\sigma$ value to be used, I have performed an experiment to check the positional error sum and total areal error sum with different values of $\sigma$ value for fixed dataset size and also compared the time of the filtering process. The results for the distance measure
for different values of $\sigma$ are given in Figure 31. As it is seen on the figure, a $\sigma$ value around 0.7 is yielding the lowest error in the results.

![Graphs showing total areal error and positional error sum vs sigma for 1000 dataset size](image)

(a) Comparison of different $\sigma$ values on total areal error for 1000 dataset size  
(b) Comparison of different $\sigma$ values on positional error sum for 1000 dataset size

Figure 31: Determination of $\sigma$ value according to the distance measure

Apart from the error measurements, time needed for the filtering is also measured. According to the nature of Gaussian kernel filtering, for $\sigma = 1$, the dataset size is reduced by the factor of 2. (Refer to the Gaussian smoothing description, dataset size after smoothing = (original dataset size / $2\sigma$)). Therefore, there’s a drastic change for the filtration time for integer values of $2\sigma$ (i.e. when $\sigma = 1, 1.5, 2, 2.5$ etc.). However, increasing $\sigma$ value to greater value than 1 causes a drastic increase in the error measurements. For this reason, $\sigma$ value is decided to be taken as 0.7 in the experiments.

**Impact of Gaussian smoothing on performance - Process speed**

In order to have a sound comparison, preprocessing method is used to filter the dataset and the filtered dataset is simplified by using each implemented simplification routines. It is observed that simplification speed is actually increasing on when it is used with Douglas-Peucker routine. In all other routines, preprocessing yields lower speeds than the non-filtered processes. In Figure 32, the impact of filtering on the performance of Douglas-Peucker can be seen. The important outcome of this find is that Douglas-Peucker routine is yielding the most accurate simplified output compared to other existing routines and the biggest known drawback of the routine is the very low-speed. [16] Here is an opportunity area to increase the process speed by using filtration.

**Impact of Gaussian smoothing on performance - Reduction rate**

After comparing the process speed, the impact of filtering on the reduction rates are investigated. Filtering has an obvious impact on the reduction rates of all simplification routines. The experiments showed that reduction rates of all simplification routines except for $n$th-Point routine are increased, as can be seen on Figure 33. The reason of no change in $n$th-Point routine is simple; the number of points after filtration is not changed unless the $\sigma$ is not greater than 1, in the experiments, $\sigma = 0.7$.

Latter to the individual comparison, reduction rates of all the methods are compared and it is observed that Douglas-Peucker routine achieves the highest amount of reduction rate percentages in all dataset sizes, compared to the other simplification routines.
Figure 32: Comparison of simplification speed for Douglas-Peucker routine with and without filtering
Figure 33: Comparison of reduction rates for each simplification routines with and without filtering:

(a) reduction rate comparison for Douglas-Peucker routine, filtered vs non-filtered dataset

(b) reduction rate comparison for nthPoint routine, filtered vs non-filtered dataset

(c) reduction rate comparison for Lang routine, filtered vs non-filtered dataset

(d) reduction rate comparison for Reumann-Witkam routine, filtered vs non-filtered dataset

(e) reduction rate comparison for triangular routine, filtered vs non-filtered dataset
Impact of Gaussian smoothing on performance - Dissimilarity Measure

When using filtration as a preprocessing step, filtering the original dataset leaves offset which causes a high amount of positional displacement. So, looking at the positional error sum while evaluating the impact of filtering is not meaningful. For this reason, dissimilarity measure is done by calculating the total areas under the original and filtered and simplified polylines by using Simpsons rule. One crucial finding was that, total areal error was lowered in all the simplification routines when filtration is used. Figure 34 shows the total areal error of all simplification methods with and without smoothing used as a preprocessing. On the multiple plot, the effect of filtration on total areal error can be seen very easily. The number units for the total areal error are calculated by assuming all the units are in meters. More important than units, relative comparison of the total areal errors are deemed more important. The impact of filtration can be seen that all simplification routines are having less total areal error when smoothing is used as preprocessing. It is observed that triangular method with smoothing as preprocessing yields the lowest total areal error. After that, smoothing with Reumann-Witkam routine yields the lowest areal error and after that Lang routine (with search region = 5) and Douglas-Peucker routines are filling up.

![Figure 34: Comparison of total areal errors for all simplification routines with and without filtering](image)

Overall comparison

After performing all the evaluations, the performance evaluation results are observed and in order to put everything together, following table is generated. Table 4 represents the performance evaluation of the simplification methods as stars, more stars mean to have a positive effect on that side. To have the overview, total number of stars are enumerated in Total column.
Table 4: Overall comparison of the simplification algorithms according to 4 criteria, which are performance (speed), shape distortion (positional error sum), visual difference (total areal sum) and rendering speed (reduction rate)

<table>
<thead>
<tr>
<th>Simplification routine</th>
<th>Performance (speed)</th>
<th>Shape distortion (Positional Error Sum)</th>
<th>Visual Difference (Total Areal Sum)</th>
<th>Rendering Speed (Reduction rate)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Douglas-Peucker</td>
<td>*</td>
<td>***</td>
<td>***</td>
<td>****</td>
<td>11</td>
</tr>
<tr>
<td>Lang</td>
<td>**</td>
<td>***</td>
<td>*</td>
<td>***</td>
<td>8</td>
</tr>
<tr>
<td>Reumann-Witkan</td>
<td>***</td>
<td>*</td>
<td>****</td>
<td>***</td>
<td>11</td>
</tr>
<tr>
<td>nth point</td>
<td>****</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td>10</td>
</tr>
<tr>
<td>Triangular area</td>
<td>***</td>
<td>****</td>
<td>***</td>
<td>*</td>
<td>11</td>
</tr>
</tbody>
</table>
7 Discussion & Conclusions

The pursued goal in this project has been to implement and analyze the existing simplification routines on large datasets and determine the usability of each simplification method and to obtain a comprehensive performance analysis of each routine.

In order to achieve this purpose, a series of datasets are generated and impact of simplification routines on these datasets are being evaluated. While generating the datasets, the input from the company was stating that there was a lot of fluctuations in the data points they obtain from the clients, and simplification was needed to be applied on those noise-dominant regions, therefore datasets are generated as random coordinate pairs, varying between 10 and 20 meters. Datasets have varying sizes from 100 to 1,000,000 data points. After implementing the random dataset generator, instead of working on Schlumberger’s Petrel, I decided to create my own framework and perform the experiments on it. The main reason for that was that the source code Petrel was having an huge size and dependencies among the internal classes, so that a minor change in the specific parts of the code would cause unpredictable results in the overall process. Furthermore, what has been developed in my own framework are totally Petrel oriented so that the code pieces from my framework could be very easily plugged into Petrel’s framework. After achieving the performance evaluations, all the code is tested on Petrel and is proven to work smoothly.

Although the coordinates are being generated in 3D space, it is narrowed down to 2D space in order to focus only on polyline simplification. At that point, the main approach of the project stood in between two different areas; digital cartography and level of detail (LOD) approach. Polyline simplification is the major interest of digital cartography, however, in cartography the dataset sizes are relatively smaller than as of they are in LOD approach. In cartography, most of the polylines are circular and intersecting with themselves as they are representing a piece of a land or geographical structure from the top-view. However, in this project, non of the polylines are circular, nor intersecting themselves in any condition. They are laying on the x-axis of the geographical structure with incrementing values. Since the main focus of the project was on implementing an efficient algorithm on large datasets, the knowledge and experiences from both areas has been benefited in order the pursue the simplification process.

The first major objective of the project was to implement a robust, fail-free algorithm so that it would work in all different sizes of datasets. In terms of robustness, a recursive algorithm always has the potential to cause stack overflow errors with large data sizes or high amount of function calls, therefore any kind of recursion should be kept away from the routines. Only Douglas-Peucker routine had a recursive implementation and in order to obtain robustness, the recursive Douglas-Peucker routine has been converted into non-recursive routine by modifying the implementation. The performance results of each routine are checked and a very low difference percentage around 1.1% has been observed in comparison. Moreover, it is observed that recursive algorithm encountered a stack overflow error with the dataset sizes larger than 20,000 data points.

After implementing each simplification routine, benchmarks for evaluations are determined and several tests are performed. The evaluation benchmarks are divided into 3 subgroups; process time and speed comparison, reduction rate comparison, dissimilarity measure comparison, each of which signifies a different quality in terms of simplification. In terms of process time, Douglas-Peucker routine has yielded the highest process time, which in other words means has
the slowest speed in simplification process. On the other hand, \textit{nth} point routine achieved the fastest process speed among the other simplification routines. The main reason for Douglas-Peucker routine to be that slow is that it is a global simplification process which means in each step of simplification, it checks the entire piece of the sub-polyline. For that reason, the complexity of the routine is $O(n^2)$, therefore with the enlarging dataset size, the process speed vanishes. Oppositely, \textit{nth} point routine has the fastest speed and the reason is that it is a brute-force routine, which is basically a memory allocation process, retaining only the first, last and the \textit{nth} point of the polyline after simplification. It does not guarantee any kind of shape similarity after the simplification and it does not consider the curvature or the overall geometry of the polyline. On the other hand, Douglas-Peucker routine is a much more sophisticated routine. In between Douglas-Peucker and \textit{nth} point routine, there are Triangular area routine, Reumann-Witkam routine and Lang routine are laying down, respectively. Reumann-Witkam and Triangular routines have very similar process time and speed results, whereas Lang routine achieved a lower result than them.

Second important benchmark is the reduction rate, which basically means the number of the remaining points after the simplification, and has a direct relationship with the rendering speed. In terms of reduction rate, Douglas-Peucker routine has achieved the highest amount of reduction rate, whereas the triangular area routine yielded a very poor reduction rate, around 32\% of value. The other remaining routines achieved similar reduction rates to each other. Douglas-Peucker routine achieved such a high amount of reduction rate mainly due to its sophisticated searching algorithm for the crucial peak points and omitting all the other minor changes on the entire polyline set. Douglas-Peucker yields this highest amount of reduction with the cost of really long process time. It can be considered as the trade-off between the process time and reduction rate (or rendering time), we might need a high amount of accurate reduction but if takes more than 2 minutes, it might not be a feasible option to use it.

Another important quality criteria is the dissimilarity measure, which is also very important for determining the visual perceptional quality of the simplification. As McMaster stated earlier [9], there should be two different groups for quantifying the dissimilarity measure, one of which is the positional error and the other one is the areal displacement measure. Positional error sum basically means the positional difference of each vertex on the original polyline from its simplified version. It is a cumulative value, where it reflects how much of the shape is distorted or not. The overall comparison of each method showed that triangular area routine yields the least amount of positional error sum, mainly because of the fact that it performs a very detailed search considering each 3 successive vertices of the polyline while processing the simplification. On the other hand, Reumann-Witkam routine yields the highest amount of positional error sum. \textit{Nth} point follows it by the second highest amount of error summation. Douglas-Peucker routine has the second lowest cumulative positional error. Here comes an important question, does the fact Reumann-Witkam routine yields the highest positional error sum mean that it yields the worst visual perceptional quality? In order to answer this question, the total areal displacement has to be evaluated and compared. Because during the simplification method, the simplification routine might cause a lot of shape distortions but it might retain the global characteristics of the polyline, which is also an important criteria. While evaluating the total areal displacement, the areas under the original and the simplified polylines are calculated and the difference is deemed as the areal error of the simplification. It is observed that Lang method achieved the highest total areal displacement after the simplification, which means the overall shape of the simplified polyline has changed the most when Lang method is used. On the other hand, Reumann-Witkam routine, which yielded the highest positional error sum in the compar-
ison, yielded one of the least amount of total areal displacement in between the original and the simplified polylines. Reumann-Witkam, Douglas-Peucker and triangular area routine yielded very similar and relative low total areal errors, where Douglas-Peucker routine was observed to have a more robust areal error displacement, compared to the other 2 routines.

As an overall picture at this point, it has been remarked that each simplification routine has a different advantage in different aspect. Triangular areal method is achieving a really low positional and areal error sum with an acceptable process speed, whereas it yields a very low reduction rate which yields a long rendering time after simplification. On the other hand, a mathematical superior, Douglas-Peucker routine yields quite good results in dissimilarity measures and reduction rate, whereas it yields the slowest process speed for the simplification. Two methods which are having similar principles, Lang and Reumann-Witkam routines yield very close results to each other in many performance criteria, except for the total areal displacement. Reumann-Witkam routine yields a lower total areal displacement than Lang routine which means that Reumann-Witkam routine is better at preserving global characteristics of the polyline.

After comparing each simplification routines’ performances, a new idea as preprocessing step is implemented: filtering, or smoothing algorithm. The main concern for the smoothing process is to reduce the noise in the original polyline so that it can get easier and faster to perform simplification routines. As an initial trial, Gaussian smoothing kernel is used with varying $\sigma$ values. Three different $\sigma$ values are tested, from 0.5 to 1 and $\sigma$ value of 0.70 is selected to be used for smoothing processes, as it yielded a very low total areal error after smoothing process. Employing smoothing process at the beginning of the simplification, and then applying the simplification routines, indeed had an interesting impact on the performance results. First of all, the process speed of Douglas-Peucker routine has increased drastically whereas for all the other routines, using smoothing as preprocessing caused lower process speeds. This is an important finding as the main disadvantage of Douglas-Peucker routine was its slow speed. Apart from the process time comparison, reduction rates of each method is compared with and without using smoothing process, and as expected, it was observed that all the reduction rates are increased when smoothing is used. This is expected because smoothing process already reduces the noise within the original polyline therefore the reduction rate results after the simplification are lower. An important outcome of this is that triangular area routine, which yields the lowest reduction rate 32% could achieve a reduction rate of 50%.

The effect of smoothing in terms of dissimilarity measure could only be quantified for the total areal error sum. It was not feasible to check the positional error sum as smoothing process changes the coordinates of all the points, therefore very high positional error would be observed. However, this does not mean that the smoothing process causes a very different visual perception, as it retains the curvature details of the original polyline. Therefore, the total areal error results are taken into account while evaluating the effect of smoothing. When all the results are put on one plot, in Figure 34, it is observed that all the routines yield lower areal error sums when the smoothing used as a preprocessing speed. Among the ones with smoothing process, triangular area routine yielded the least areal error sum, Lang and Reumann-Witkam routines followed it by the second and third least areal error sum. Looking at the big picture, it could be said that using smoothing algorithm as a preprocessing step had a positive impact on the performance of all simplification routines.

To sum up all the evaluations, the reader should now be referred to the Table 4 to check
the overall rates of all the simplification routines that are used and tested in this project. It is clearly seen that Douglas-Peucker, Reumann-Witkam and triangular area routines have the same cumulative ratings in terms of performance criteria. Each of them have different strengths and weaknesses among each other, depending on the quality criteria. It can also be concluded that smoothing has a positive effect on the overall simplification process, except for the process time on the simplification routines apart from Douglas-Peucker.

8 Future Work

In this project, there has been 5 simplification routines under consideration and all the evaluations and comparisons are done around those routines. It is known that there are other existing simplification routines, namely Opheim algorithm, Visvalingam-Whyatt algorithm, Zhao algorithm and Saalfeld’s algorithm routines [6]. Many of those algorithms are developed for digital cartography usage and not tested on large datasets. It would be interesting to implement those algorithms on large datasets and obtain the performance results.

Another important next step would be to work on implementing and investigating the improvements on the speed of Douglas-Peucker simplification routine, as Douglas-Peucker routine yields quite good results in many performance criteria aspects. One important finding is that, applying a convex hull algorithm into the Douglas-Peucker routine is found to achieve faster results on simplification process [?]. Apart from that, it might also be an interesting approach to parallelize the simplification process, and using different processors while treating each sub-polyline. I reckon this would have a great impact on the performance of the Douglas-Peucker routine.

While determining the dissimilarity measure, only positional and areal error sums are evaluated and compared, but also angular dissimilarities in between the original and simplified polylines could be compared, to have a different perspective on quantifying the dissimilarity.

Interactivity is another important area since after all Petrel is a visualization toolkit which enables the user of the tool to interact with the dataset. In terms of interactivity, FPS (frames per second) measure is an important benchmark, but more parameters could be changed in the current status. One example would be to implement a better LOD approach, which might be considered as having 3 or 4 different visual qualities in the object shown on the screen, depending on the distance to the camera. In the furthest point from the user, the object could be simplified in the most rough but fastest way so that the visual quality would not play an important role. As the object gets closer to the camera, it can get more and more details, by using different simplification methods. Moreover, since the main focus is on large datasets, and they are viewed on the screen with perspective, the further parts of the polyline can be simplified differently than closer parts to the camera (i.e. more simplification on further parts of the polyline).
9 Appendix

Positional Error Sum calculation on C++ code script:

```cpp
Vector3D simplified_first, simplified_prev, simplified_last, original_first, original_last;
simplified_first = points_simpl[0];
simplified_last = points_simpl[num_vertices_out - 1];
original_first = points[0];
original_last = points[log_size - 1];
simplified_prev = simplified_first;
simplified_first = points_simpl[1];

int kk = 0;
int mm = 0;
while (simplified_first.operator == simplified_last == false)
{
    while (original_first.operator == original_last == false && original_first.operator == simplified_first)
    {
        int i = 1; i < log_size; i++;
        do
        {
            double result += minDistance(simplified_prev, simplified_first, original_first);
            kk++;
        }
        while (original_first == points[kk]);
        simplified_prev = simplified_first;
        mm++;
        simplified_first = points_simpl[mm + 1];
    }
}
```

Total Areal Sum calculation by Trapezoidal method in between the original and simplified polyline C++ code script:

```cpp
double sumTrapezoidal_org = 0.0;
double sumTrapezoidal_simpl = 0.0;
double sumTrapezoidal_simpl_filt = 0.0;
for (int i = 1; i < num_vertices_out; i++)
{
    sumTrapezoidal_org += (points[i].x() - points[i - 1].x()) * (points[i].y() + points[i - 1].y());
}
sumTrapezoidal_org = sumTrapezoidal_org * 0.5;

for (int i = 1; i < num_vertices_out_filt; i++)
{
    sumTrapezoidal_simpl += (points[out_indices[i]].x() - points[out_indices[i - 1]].x()) * (points[out_indices[i]].y() + points[out_indices[i - 1]].y());
}
sumTrapezoidal_simpl = sumTrapezoidal_simpl * 0.5;

for (int i = 1; i < num_vertices_out_filt; i++)
{
    sumTrapezoidal_simpl_filt += (samplesFiltered[out_indices_filtered[i]].x() - samplesFiltered[out_indices_filtered[i - 1]].x()) * (samplesFiltered[out_indices_filtered[i]].y() + samplesFiltered[out_indices_filtered[i - 1]].y());
}
sumTrapezoidal_simpl_filt = sumTrapezoidal_simpl_filt * 0.5;

double error_areal_trapezoidal = abs(sumTrapezoidal_org - sumTrapezoidal_simpl);
```

Total Areal Sum calculation by Simpson’s method in between the original and simplified polyline C++ code script:

```cpp
double sumSimpsons_org = 0.0;
double sumSimpsons_simpl = 0.0;
double sumSimpsons_filt = 0.0;
double sumSimpsons_filt_simpl = 0.0;
for (int i = 1; i < log_size; i++)
{
    sumSimpsons_org += (points[i].x() - points[i - 1].x()) * (points[i].y() + 4 * points[i].y() + points[i - 1].y());
}
sumSimpsons_org = sumSimpsons_org * 1/6;

for (int i = 1; i < num_indices; i++)
{
    sumSimpsons_simpl += (points[out_indices[i]].x() - points[out_indices[i - 1]].x()) * (points[out_indices[i]].y() + points[out_indices[i - 1]].y());
}
sumSimpsons_simpl = sumSimpsons_simpl * 1/6;

for (int i = 1; i < newSize; i++)
{
    sumSimpsons_filt += (samplesFiltered[i].x() - samplesFiltered[i - 1].x()) * (samplesFiltered[i].y() + samplesFiltered[i - 1].y());
}
sumSimpsons_filt = sumSimpsons_filt * 1/6;
```

46
sumSimpsons_filt = sumSimpsons_filt * 1/6;

for (int i = 1; i < num_vertices_out_filt; i++) {
}

sumSimpsons_filt_simpl = sumSimpsons_filt_simpl * 1/6;

double error_areal_simpsons = abs(sumSimpsons_org-sumSimpsons_simpl);
double error_areal_simpsons_filt = abs(sumSimpsons_org-sumSimpsons_filt);
double error_areal_simpsons_filt_simpl = abs(sumSimpsons_org-sumSimpsons_filt_simpl);
}

Gaussian smoothing C++ code script:

float sigma = 0.8; // this is the key value. low sigma, more accurate output (with more points)
float subsample = floor(2*sigma);
int k = ceil(2*sigma);

doubles* filter = new double[2*k+1]; // defining filter
int* k = new int[2*k+1];
for (int i=0, j=-k; i<2*k+2; i++, j++){
    filter[i] = exp(-(j)*(j)/(2*sigma*sigma));
}

double avg, wgt;
newSize = floor(log_size/subsample);
output = new double[newSize];
for (int ii=0; ii<log_size ; ii=ii+subsample){
    avg=0;
    wgt=0;
    for (int jj=-k ; jj<k ; jj++){
        if (ii+jj >= 0 && ii+jj < log_size){
            avg = avg + points[ii+jj].y()*filter[jj+k];
            wgt = wgt + filter[jj+k];
        }
    }
    output[(int)ceil(ii/subsample)] = avg/wgt;
}

Vector3D* samplesFiltered = new Vector3D[newSize];
for (int i=0 ; i<=newSize ; i++){
    samplesFiltered[i].setX((double)i);
    samplesFiltered[i].setY(output[i]);
    samplesFiltered[i].setZ(1);
}