On the Implementation of Polygonal Approximation Algorithms

Ovidiu Grigore
Remco C. Veltkamp

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-005

www.cs.uu.nl

1 Introduction

The polygonal approximation, as a simplification procedure of an input curve representation, consists the main subject of many studies, but most of them are exclusively focused only on the theoretical matters and the practical aspects about their implementation are usually ignored. From theoretical point of view the main goal is to find new algorithms for curve polygonal approximation, thus, using new ideas, new data structures or new searching techniques, always the final challenge is to improve the performances of the existing methods. To proof the superiority of a new algorithm, theoretical upper bounds of parameters like algorithm (running time) complexity and used memory are established and compared between the algorithms.

These studies, which point out only the theoretical aspects of an algorithm, do not take into account the practical behavior of a method or the effort needed to implement it and therefore, in these cases someone could say that the algorithm is not completely described. Only a theoretical approach about a method does not guaranty its right behavior in practice and more over, the most of them do not explain how to implement the algorithm in practice.

The principal aim of thinking, designing and developing a new algorithm is its practical usage. There are methods with poor theoretical performances that are easy to be implemented and, on the other side, there are algorithms with very good theoretical features for which the great effort of practical implementation makes them almost unusable. Also, there are different algorithms that have the same theoretical performances, but they can be ordered according with either the effort needed to implement them or their practical efficiency. Therefore, besides the theoretical description and evaluation, any algorithm should be analyzed also from the experimental point of view.

This study tries to cover the lack of experimental details among several theoretical polygonal approximation algorithms. Three main directions are followed in this experimental analysis:

- the possibility of an efficient implementation of the algorithms and the effort needed to do it;
- the comparison of the algorithms using experimental measurements;
- the correspondence between the theoretical feature upper bounds and the experimental results.

To fulfill this task, the first step was to design a software package that implements several algorithms used in polygonal approximation of an input curve. For this purpose there are applied optimal methods, like dynamic programming [8], and sub-optimal approaches, like recursive split [6]. Both, bounded–# and bounded–ε, approaches were studied and implemented. All these algorithms were implemented in a generic manner, trying to make the software package as general as possible. Thus, depending on the application, someone can chose between different items that are possible to be set for features like:

- the base data type (integer, double, leda_integer, Gmpz, etc.),
- the kernel representation (Cartesian, Homogeneous),
- the distance used in measuring of the approximation error.

To overcome all these requirements, the concept of traits class was used.

A library of traits classes implementing different kinds of distances between a point and a segment is available. There are implemented common distances like: the Euclidean distance, the Manhattan distance, the L_∞ distance, measured to the line support or to the segment, but also other distances, like the bounding shell distance [3]. Also, new computational strategies useful in improving the running time, like the incremental algorithm used to compute the Euclidean distance [4], or the algorithm based on the path hull structure [1], are implemented.

In the same spirit of generalization, trying to cover as many as possible concepts, the approximation error can be assessed either as the maximum value of all the distances computed between the input curve and the output polygonal approximation, or as the sum of all these distances.

Another group of algorithms, optimal polygonal approximation methods for particular distance measures, was implemented. The generic character regarding the base data type and the kernel type is still available, but these methods use graph search techniques [5] to compute the polygonal approximation with the Euclidean distance error assessment only.

After an introduction about the purpose of this study, in the second section are presented some theoretical elements about polygonal approximation methods and the optimal and sub-optimal paradigms are explained. After these, different kinds of distances used in polygonal approximation error assessment are introduced. Also, there are described specific techniques used to improve the performances of the polygonal approximation algorithms for some particular cases, like the incremental technique used for computing the sum of the squared Euclidean distances or the path hull structure used to improve maximum Euclidean distance assessment for the Recursive Split method.

The software package is described in section three. The structure and the concepts used to implement the polygonal approximation methods are discussed. Also a comparison between the generic and the dedicated implementation techniques is briefly reviewed.

In the last section the experimental results are presented. The algorithms are compared from practical measurements point of view. The steps made to improve the polygonal approximation algorithms both from theoretical and practical perspectives are presented. Conclusions that concern different correlations between the experimental results, like: comparison between the different algorithms, comparisons between different kind (generic and dedicated) of implementations and comparison between the theoretical bounds and practical results are given.

2 Theoretical Elements

2.1 Definitions

Let $\{P_1, ..., P_N\}$ be a set of successive points, defining the input curve, then, with respect to a given criterion, it can be represented simplified as a set with a reduced number of points $\{Q_1, ..., Q_M\}$, M<N.

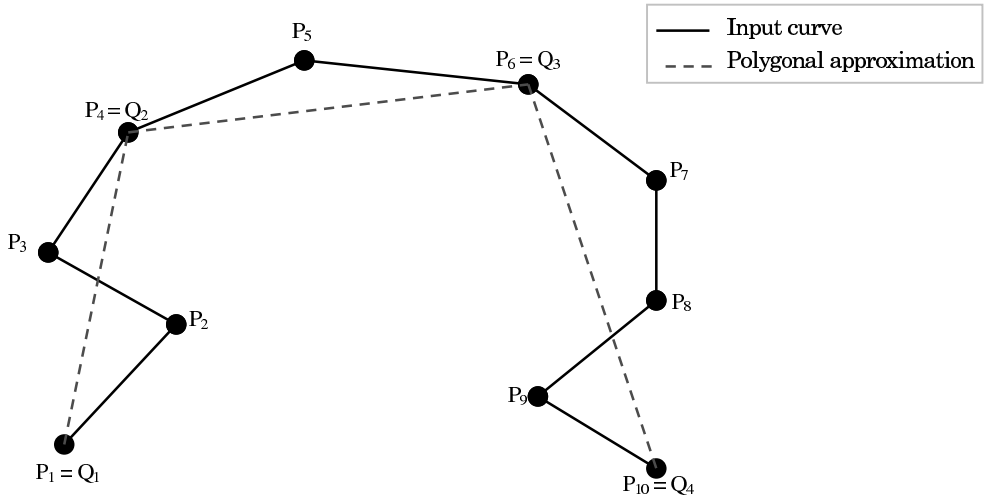


Fig. 1 Example of polygonal approximation.

To define the differences (roughness) between the input curve and its approximation (the output curve), two parameters can be used: the decrease in the number of points used in curve representation (N–M) and the error of approximation that measures the changing of the input curve. Depending on the main parameter of interest, there are two possible approaches that can to be used for solving a polygonal approximation problem:

- **min–#**, when given an error threshold that is the upper limit of the distortion accepted in the final representation, the polygonal approximation with the minimum number of points has to be computed (figure 2);
- **min–ε**, i.e. given the number of points desired to be used in the final representation, the output curve that approximates as good as possible from a given error representation criterion point of view has to be found (figure 3).

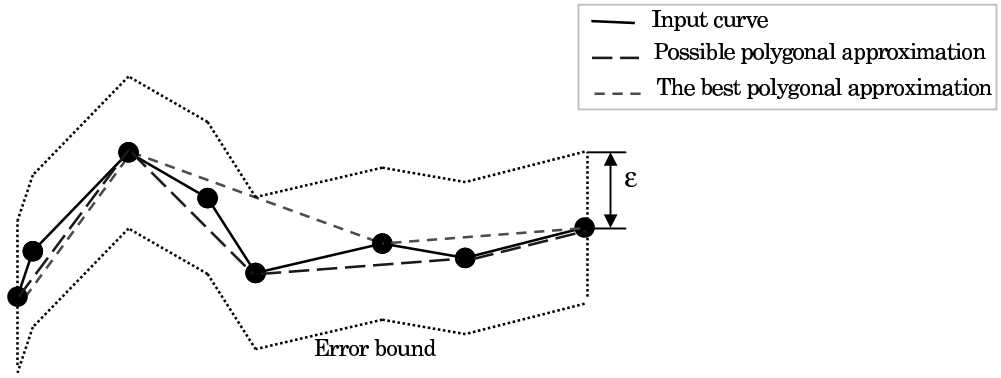


Fig. 2 Min−# polygonal approximation approach. For a given error threshold ε, the polygonal line that has minimum number of point is chosen.

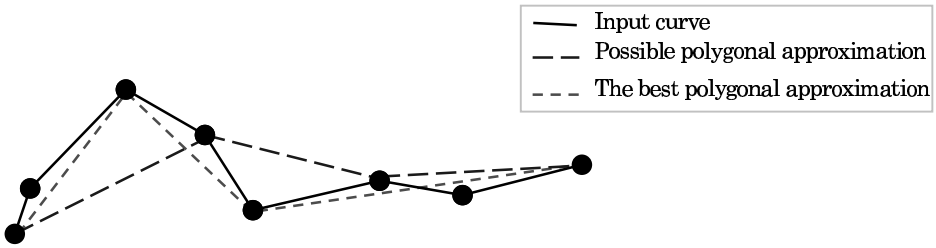


Fig. 3 Min−ε polygonal approximation approach. Given an M=4, several approximations could be done. The output one minimizes the approximation error.

2.2 Approximation Error Assessments

For both categories of algorithms an evaluation of the error appeared in approximation of the initial curve is needed. To estimate it, several distances from a point to a line can be applied, such as: the Euclidean Distance, the Manhattan Distance, the L_∞ distance, the Vertical Distance, the Bounding Shell Distance.

Also, the error obtained in approximation of a curve fragment with a line segment can be defined either as the maximum value of the distances between the segment and each point of the curve portion or as the sum of these distances.

One main group of distances consists of particular cases of the general distance L_p. For two points v

and w from a k–dimensional space it is defined as:

$$L_p(v,w) = \left(\sum_{i=1}^{k} |v_i - w_i|^p \right)^{\frac{1}{p}}$$

where v_i and w_i are the i–th coordinate of the v and w, respectively.

2.2.1 The Manhattan Distance (L_1)

For $p=1$ the distance obtained is usually called Manhattan distance or city block distance. For a (x,y) plane, it is defined as:

$$L_1(v, w) = | v_x - w_x | + | v_y - w_y |$$

This distance is equal to the length of the shortest path between the two points while going only along the axis directions. In figure 4 is presented the measure of the Manhattan distance between a point and a segment.

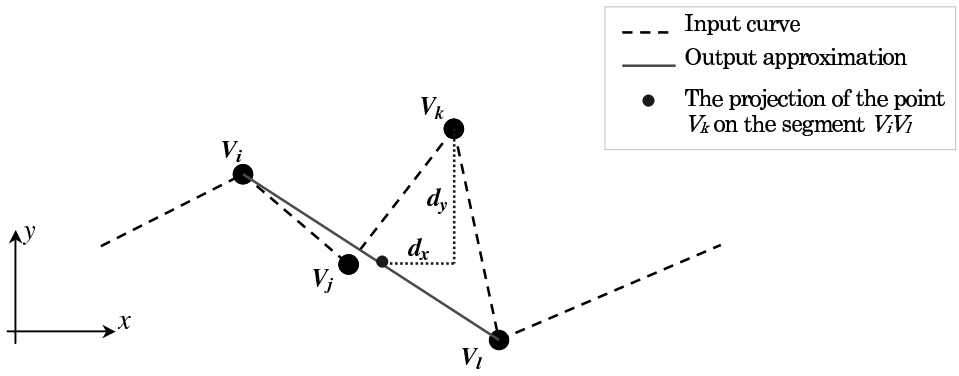


Fig. 4 Manhattan distance measurement (d_x+d_y).

2.2.2 The Euclidean Distance (L_2)

Maybe the mostly used distance is the Euclidean one, obtained from the general formula for $p=2$:

$$L_2(v, w) = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}$$

In figure 5 the distance between a point and a line is shown.

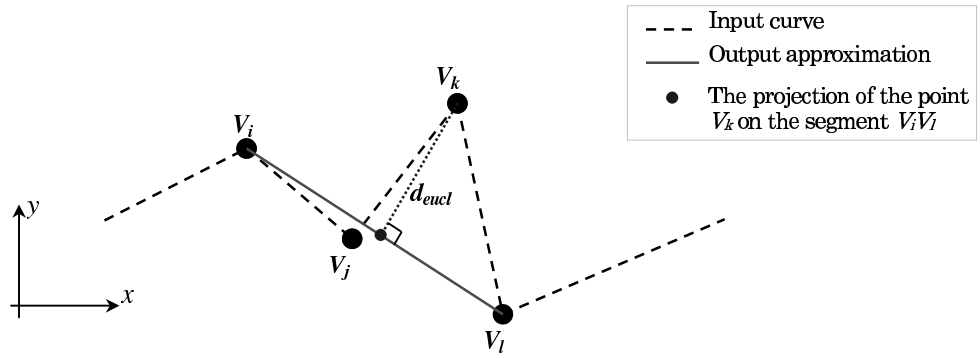


Fig. 5 Euclidean distance measurement (d_{eucl}).

2.2.3 The Max Distance (L_∞)

L_∞ is the particular case obtained for $p \to \infty$, when, for a k–dimensional space, the formula becomes:

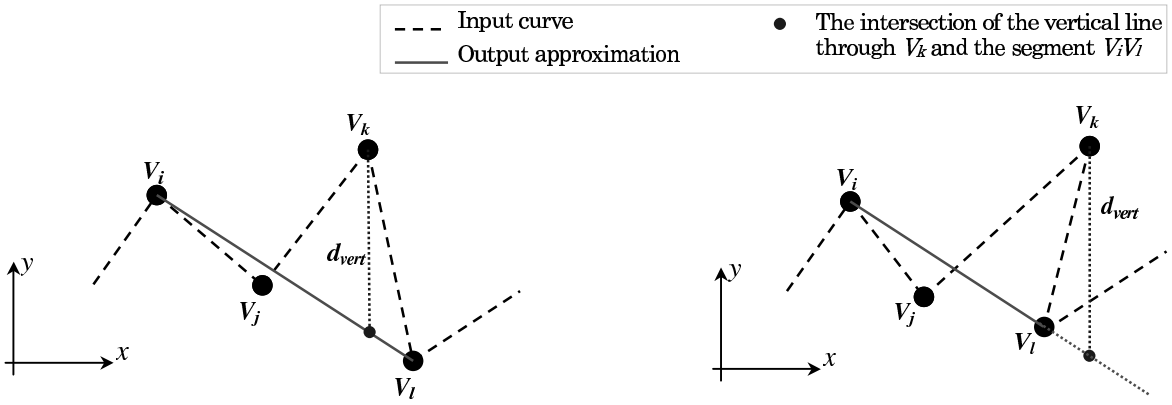
$$L_\infty(v, w) = \max_{i=1...k} \{ | v_i - w_i | \}$$

where v_i and w_i are the i–th coordinate of the points v and w, respectively.

As is easy to observe, to measure the L_∞ distance is similar to the L_1 procedure (fig. 4), but this time only the maximum value of the distances computed along the axis is taken into account.

2.2.4 The Vertical Distance

The vertical distance is equal to the difference in y–values between the current point and the intersection point of the vertical line that is passing through it and the line support of the approximation segment. Figure 6.a. shows an example of measuring vertical distance, when the intersection point is inside the segment and in figure 6.b. is presented the case when the intersection point is outside the segment.



a) the intersection point is inside the segment. **b)** the intersection point is outside the segment.

Fig. 6 Vertical distance measurement.

Please note that when this distance is used to assess curves that contain vertical parts, these cannot be practically approximated. In that case the vertical parts are returned unchanged.

2.2.5 The Bounding Shell Distance

The *shell* is a bounding volume for the curve segment that has to be approximated. It consists of the two smallest circle arcs passing through the two end points of the approximating segment that includes between them all the points of the curve portion to be approximated. If all the points are on the same side of the approximating segment then the bound on the side without points is the segment itself. The distance associated to a bounding shell is defined as the maximum value of the Euclidean distances measured between the segment and all the points of the two bounding arcs [3].

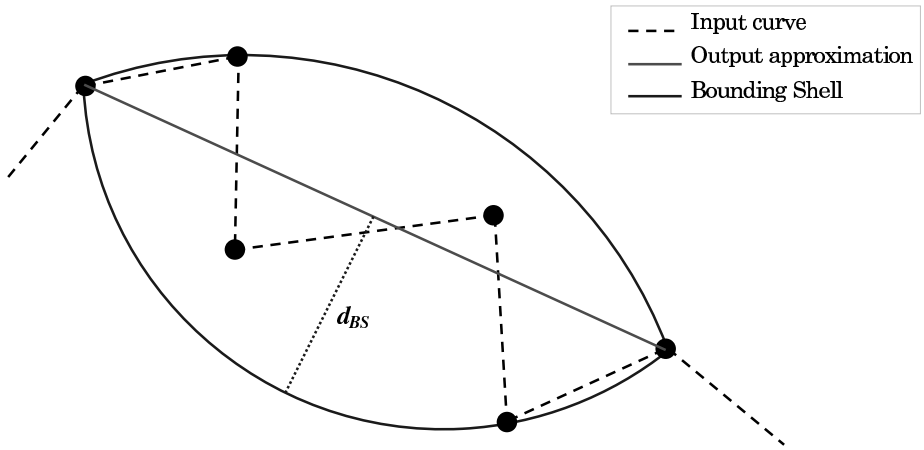


Fig. 7 Finding the bounding volume and its associated distance.

2.2.6 The Path Hull Based Distance

The *convex hull* of a set of points is the smallest convex set containing it. It can be proved that the furthest point from any segment linking two points of the initial set is always an element of the convex hull.

To build a polygonal approximation using a recursive split method based on the maximum Euclidean distance means to find in each step the furthest point from a certain segment. Therefore, only the points from the convex hull of the current curve need to be considered to find the furthest point.

On applying the recursive split method, the current curve is changing, actually in each step a sub–chain of the initial curve is processed. To deduce efficiently the convex hull of the current sub–chain using the information from the previous step of processing, an efficient data structure, called *path hull*, was developed [1].

A path hull consists of two arrays storing the vertices of the two convex hulls: the left stores and the right stores. Attached to each array is a *history stack,* used to restore the previous hull, when this operation is needed.

This method uses the *path hull* structure [1] to compute the Euclidean distance measured to the line support of the approximation segment (see chapter 1.3.6, page 8). The main idea of this algorithm is to use the convex hull of a curve to find faster the maximum distance between the curve and an approximation segment. There are two possible cases in obtaining the current convex hull: first, to *recover* it from the previous one and the second, to *build* it without using any information from the previous processing. When the *recover* procedure is chosen, the operations used to obtain the current hull, which were stored in the history stack, are applied in reversed order to restore the previous hull. To improve the practical performances, the *build* operation is implemented using the on-line convex hull algorithm [9].

2.2.7 The Incremental Technique Based Distance

The assessment performances for some particular distances can be improved in certain situations applying specific computational algorithms. The incremental technique described by Perez and Vidal in [4] is such an algorithm designed to improve the computation of the sum of the squared vertical or Euclidean distances. Let $\{ p_1, p_2, ..., p_N \}$ be a set of consecutive points representing the input curve. The incremental method can be applied to approximate subsets of consecutive points $\{ p_i, p_{i+1}, ..., p_j \}$ by the segment $\overline{p_i p_j}$ linking their ending points. The constraint is that the subsets must have the same starting point p_i and they are extended successively, point by point, in only one direction (p_j, p_{j+1}...).

For the sum of the squared vertical distances the error of approximating the curve portion $\{ p_i, p_{i+1}, ..., p_j \}$ with the segment $p_i p_j$ is computed as:

$$svd(i,j) = \sum_{k=i+1}^{j-1} ((a_{i,j} + b_{i,j} x_k) - y_k)^2 \qquad (1)$$

where:

$a_{i,j} = y - b_{i,j} x_i$ is the y-value of the line $p_i p_j$ at the y-axis.

$b_{i,j} = \dfrac{y_j - y_i}{x_j - x_i}$ is the direction coefficient of the line $p_i p_j$

If the function *svd(i, j)* needs to be computed for the same set plus a new point, as it was stated above, than the new approximation segment will share one endpoint with the last one, but its other endpoint will be different. Thus, the direction of the new segment is different from the previous one and therefore, in a classic manner of computation, the value of *svd(i, j)* can be assessed only by evaluating again all the distances from the points to the new segment. Fortunately, this expensive procedure can be avoided using the incremental technique that is based on the following formula obtained by expanding (1):

$$svd(i,j) = a_{i,j}^2 (j-i-1) + 2a_{i,j} b_{i,j} \sum_{k=i+1}^{j-1} x_k - 2a_{i,j} \sum_{k=i+1}^{j-1} y_k + b_{i,j}^2 \sum_{k=i+1}^{j-1} x_k^2 + \sum_{k=i+1}^{j-1} y_k^2 - 2b_{i,j} \sum_{k=i+1}^{j-1} x_k y_k$$

If five variables holding the sum of the x's, y's, squares of x's, squares of y's and the cross products xy are incrementally maintained, than computing the new total error *svd(i, j+1)* involves only updating these variables with the contribution of the j-th point, which takes constant time. Also, in each step the parameters $a_{i,j}$ and $b_{i,j}$ defining the new segment must be computed.

To compute the sum of the Euclidean distances *sed(i,j)*, in the same conditions, the incremental technique can be applied in a similar way according to the following formula describing its dependence on the sum of the vertical distances:

$$sed(i,j) = \frac{1}{b_{i,j}+1} svd(i,j)$$

2.2.8 Distance Measured to the Segment or to the Line Support

Whenever the procedure of measuring the distance between a point and a segment implies the projection of the point on the segment, there are two possible cases: either the projection point is inside the segment or it is outside. Therefore, two kinds of distances can be assessed:

a) the distance measured to the line support of the segment; in this case the distance is computed always between the point and its projection, even if it is outside of the segment, on its line support (fig. 8);

b) the distance measured to the segment; this time two cases can happen: first, when the projection of the point is inside the segment and the distance is computed similarly with case (a); in the second approach the projection is on the line support, outside the segment, so it is chosen the closest point from the segment, actually the closest end point of the segment, to the current point (fig. 8).

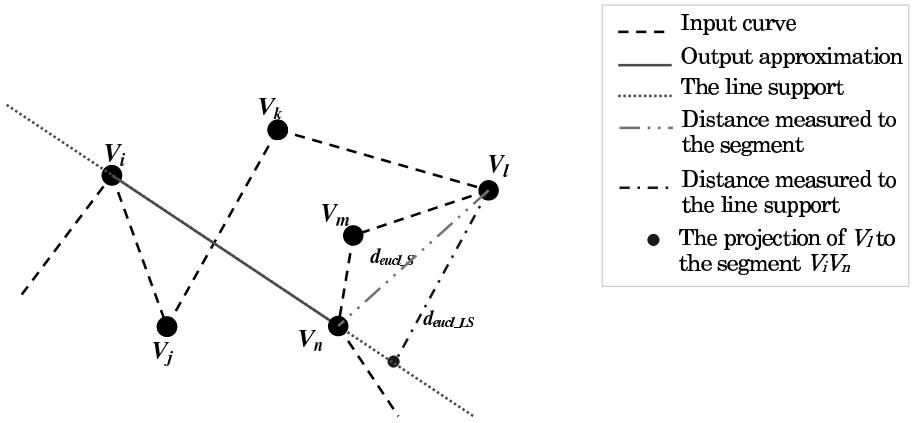


Fig. 8 Distance measured to the line support or to the segment. The difference between the two approaches is presented for the Euclidean distance.

2.3 Polygonal Approximation Algorithms

2.3.1 Optimal and Sub–Optimal Solutions

Two parameters characterize the polygonal approximation of an input curve: its number of vertices M and the error ε that approximates the input curve. Setting an upper bound for one of these parameters, the value of the other one depends on the algorithm that is applied, but the smaller the value the better the algorithm. So, from this point of view, the polygonal approximation methods can be split in two main groups:

- the ***bounded-ε*** algorithms, when the upper bound of the approximation error is given
- the ***bounded-#*** algorithms, when the upper bound for the number of vertices of the output poly-line is demanded.

According to the accuracy of their solution, we can roughly divide the algorithms used to approximate polygonal curves into two main categories: the optimal and the sub–optimal methods, respectively.

In the former case, optimization techniques, like dynamic programming, are applied to find the **best solution** of the approximation problem. Therefore, in this case, the bounded-# problem is equivalent with a min-ε approach (section 1.2, page 4) and the bounded-ε with the min-# (section 1.2, page 5). Generally, these algorithms have relative slow running time, even though there existed attempts to implement faster optimal algorithms for some specific error measurements, such as the incremental technique made by Perez and Vidal [4] for the Euclidean and vertical distances measured to the line support of the segments.

The latter group of algorithms tried to be a compromise between the loss of precision of the solution and the improvement of their running time. Usually, they are based on heuristic techniques, such as the recursive split method, that offers a faster running time, but the solution is not so good as for the optimal one.

Table 1 Time complexity of the studied polygonal approximation algorithms

Algorithm and implementation specifications	Time complexity of computing
Recursive Split, [6] General implementation	$O(N^2)$
Recursive Split, Maximum Squared Euclidean Distances, Path Hull implementation [1]	$O(NlogN)$
Dynamic Programming, [8] General implementation	$O(MN^3)$
Dynamic Programming, new implementation General implementation	$O(N^3)$
Dynamic Programming Sum of Euclidean Distances, Incremental technique [4]	$O(MN^2)$
Graph Search Method, [5] Maximum of Euclidean Distances, measured to line support	$O(N^2logN)$
Graph Search Method, Sum of Euclidean Distances, measured to line support	$O(N^3)$

To understand the difference in running time of the several kinds of algorithms, in table 1 (page 12) are presented the time complexity limits for some of them. *N* denotes the number of points of the input curve and *M* represents the number of points of the output approximation. The usual implementation cases, which don't use special structures or techniques, are named in table by *"general implementation"*

2.3.2 Dynamic Programming Method

Dynamic programming is a technique used in solving optimization problems, this time applied to finding the optimal polygonal approximation for a given input curve and the desired parameters. Generally speaking, its main idea is to build the final optimum by finding optimum solution at the partial problems. It finds first the optimum solution of the simplest possible level of the problem. Each time the next level is more complex than the previous one, but computing the optimum solution of the current level is based on the optimum solutions found in all bottom levels. In the final level, when the partial problem overlapped the initial problem, the global optimal solution is found.

In the case of polygonal approximation problems, the dynamic programming technique starts with computing all the approximations of all possible parts of the input curve using only one segment:

$$\{\Delta(1, n)\}_{n=1...N}$$

After this, step by step, the approximations of the curve using an increasing number of segments (2, 3… M) are computed using an iterative process based of the following formula:

$$D(n,m) = \begin{cases} \Delta(1,n) & n \geq m = 2 \\ \min\limits_{m-1 \leq i \leq n-1} \{D(i,m-1) + \Delta(i,n)\} & n \geq m > 2 \end{cases}$$

2.3.3 Recursive Split Method

This algorithm is a heuristic approach, finding a sub–optimal, but often good, solution of the approximation problem, the main aim being the improvement of the other features of the optimal algorithms, like running time and the memory usage.

Actually, recursive split method is based on a "divide and conquer" technique. It starts with the initial segment that links the two end points of the input curve, for which the furthest point is found. If the distance from the furthest point to the segment is greater than a given error threshold, then the problem is split into two similar sub–problems: approximating the two curve portions, one from the start point to the furthest point, the second from the furthest point to the end point. So, the algorithm is repetitively applied until all the segments obtained approximate their curve portion with an error less then the given threshold (figure 9).

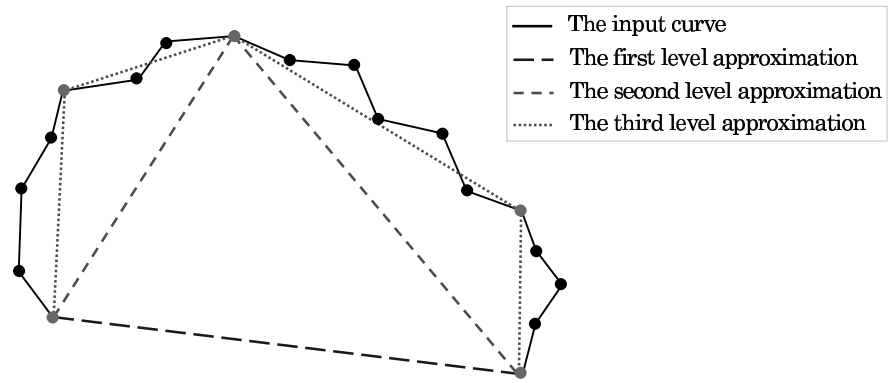


Fig. 9 Example of applying the recursive split method.

2.3.4 Graph Search Method

In this case an optimal solution of a polygonal approximation problem is obtained applying searching techniques to find the shortest path in the distances-graph associated to the input curve.

The distances-graph $G(C)$ associated to an input curve C given as an ordered set of points $\{ p_1, p_2, \ldots, p_N \}$, is that graph having a vertex v_i corresponding to each point p_i and an arc (v_i, v_j) for each pair of points (p_i, p_j) of the input curve. For a given bound error ε, the graph $G(C,\varepsilon)$ is defined as the subgraph of $G(C)$ containing only those arcs with the weight less than or equal to ε. Any path in $G(C,\varepsilon)$ that starts with p_1 and ends with p_N corresponds to a polygonal approximation of the input curve with the maximum error equal to ε.

In the bounded-ε (min-#) approach, an approximation of the input curve having the minimum number of vertices corresponds to the shortest path from v_1 to v_N in $G(C,\varepsilon)$, where the length of the path is computed as the number of its arcs. The shortest path can be found using a modified breadth-first search technique, which takes time linear in number of arcs from $G(C,\varepsilon)$. The breadth-first search technique is changed to process the points only in the forward direction, from the first point p_1 to the end point p_N, otherwise it will be possible to obtain false loops in the resulting approximation polygon (figure 10).

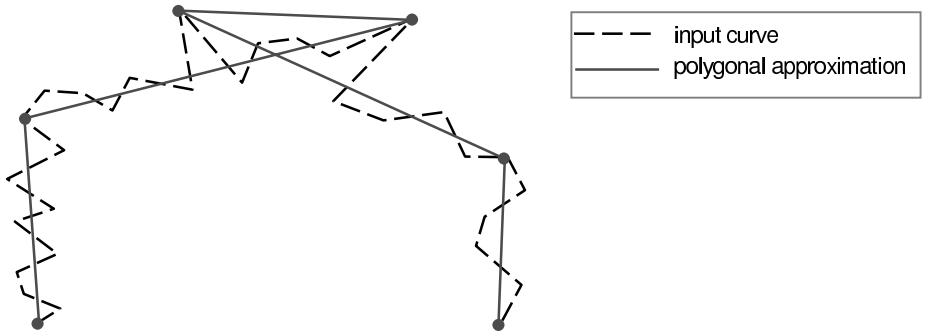


Fig. 10 Appearance of a false loop in the polygonal approximation of a curve when the classic breadth first search technique was used.

In this approach the solution of the bounded-# (min-ε) problem is to find the minimum ε^* such that its associate graph $G(C,\varepsilon)$ contains a path from v_1 to v_N with the length equal to the desired number of points for the output polygonal approximation M. For doing this, first, the set containing all the possible approximation errors of the input curve is determined. The second step is to find the minimum value ε^*. Before this, it is interesting to observe that:

- for any possible value $\varepsilon_1 < \varepsilon^*$ the minimum number of points M_1 that approximate the curve with this upper bound is greater than or equal to M:

$$M_1 \geq M$$

- for any possible value $\varepsilon_2 > \varepsilon^*$ the minimum number of points M_2 that approximate the curve with this upper bound is less than or equal to M:

$$M_2 \leq M$$

Thus, using binary search among the set of all possible errors, the minimum value ε^* is found in $N^2 \log N$ steps. In each step of the binary search, the minimum number of points necessary to approximate the curve is computed applying the min-# algorithm for the current error value.

3 Software Description

The main idea used in designing the software package is to obtain a code as general as possible, that will give it a flexibility that makes it adaptable to as many as possible user requirements. The software developed for this purpose tries to meet all these requirements, but also to combine all the theoretical concepts described before (sections 2.1-2.3).

The structure of the software package is described in figure 11. Thus, the first step is to choose the *Base data Type*. This data type is identical with the number representation and it will be used directly in computation of the polygonal approximation if in the second stage the *Cartesian Kernel* is chosen, or it is the base type, the type of the numerator and of the denominator, of the rational number representation when the *Homogeneous Kernel* will be used. After these, two main categories of functions can be applied to obtain the polygonal approximation. The first one is a more flexible designed software, in which, based on the *traits class* concept, several distances used in error assessment might be passed to the generic implemented polygonal approximation functions. The second group consists of more dedicated implementations, meaning polygonal approximation functions based on algorithms designed only for a certain kind of distance (Euclidean distance measured to the line support).

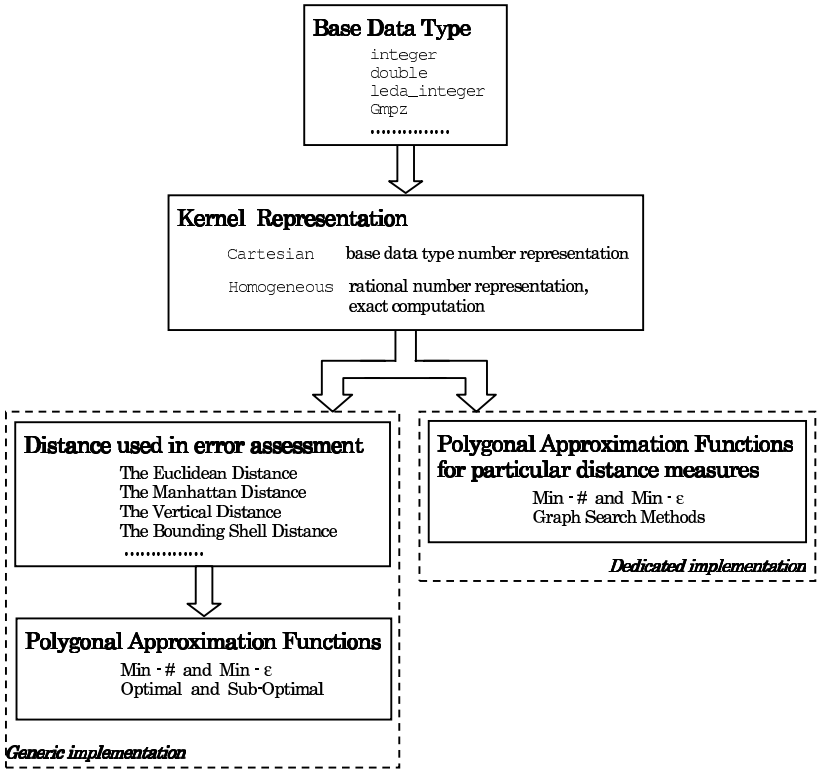


Fig. 11 The structure of the software package.

The polygonal approximation software package is structured as it follows:

- the file "*polyap_fct.h*" that contains 10 functions:
  - 6 functions from the generic implementation category used to approximate polygonal curves, covering both the bounded–# and the bounded–ε approaches and implemented in optimal (dynamic programming) and sub–optimal (recursive split) manner. Being implemented with templates and traits classes, these functions can be used with several kinds of distances.
  - 4 dedicated functions implemented algorithms based on graph search techniques, solving the min–# and the min–ε problems, using as error assessment the *sum* of or the *maximum* of the Euclidean distances measured to the line support.

  Implemented with templates, all the functions can be used with several kinds of number types and several kinds of input and output storing data structures.

- the file "*polyap_traits.h*" that contains distance traits classes. Also these functions can be used with different kinds of kernels and with different kind of number types.

3.1 Functions Description

To be as general as possible, the functions deal with templates to define the input and the output data storing structures and also a traits class to specify the distance type used in error evaluation. Thus, the general requirements for the polygonal approximation functions are:

`InputIterator`, which is actually a forward iterator corresponding to the data structure storing the input curve;

`OutputIterator`, similar to the former one, but associated to the result storing data structure;

`DistTraits`, the traits class that defines the distance method used in polygonal approximation assessment. It has also to provide several other items, as follows:

`typename DistTraits::DataT`, a number type;

3.1.1 Dynamic Programming Functions Specific Requirements

```
DataT operator()(  InputIterator begin,
                   InputIterator end )
```

an operator that returns the error obtained when the curve portion between the **`begin`** point and **`end`** point is approximated by a line segment

```
DataT cumulate(  DataT curveError,
                 DataT deltaError )
```

function that computes the error of a curve segment when two partial errors, **`curveError`** and **`deltaError`**, of its disjoint parts are known . Usually, this can be done either by taking the *maximum* value of them or by *summing* them.

3.1.2 Recursive Split Functions Specific Requirements

```
DataT operator()(  InputIterator begin,
                   InputIterator end,
                   InputIterator* p_MAX)
```

an operator that, like the previous one, returns the error obtained when the curve portion between the **begin** point and **end** point is approximated by a line segment. The difference is that this time it has to return, through the parameter **p_MAX**, the pointer to the furthest point of the curve from the segment line approximation.

3.2 Distance Traits Classes Description

As mentioned above, to apply the polygonal approximation functions from the generic implementation category, there must be specified a distance traits class. Someone can implement his own traits class or can use those from file *polyap_traits.h*, but the distance traits class has to agree with (obey to) all the requirements stated before in chapter 1.5.1 (see page 11).

All the distance traits classes that are already implemented and are accessible from the file *polyap_traits.h*, use a template class to specify the kernel type, *Cartesian* or *Homogeneous*. Also, through the kernel we induce a base number type.

4 Experimental Results and Conclusions

4.1 The Dynamic Programming Implementation

4.1.1 Original Algorithm

Dynamic programming is an optimization technique based on a progressive computation procedure that finds the best solution of a given problem by computing the optimum of its partial problems. It starts by solving the simplest possible sub-problem of the initial one and evolves step by step, each time determining the more and more complex sub-problems optimum, until it finds the final solution, which actually corresponds to the given problem.

In the case of polygonal approximation problems, the dynamic programming technique starts with computing all the approximation errors of all possible parts of the input curve using only one segment:

$$\{\Delta(1, n)\}_{n=1...N}$$

After this, step by step, the approximations of the curve using an increasing number of segments (2, 3… M) are computed using an iterative process based of the following formula:

$$D(n,m) = \begin{cases} \Delta(1,n) & n \geq m = 2 \\ \min\limits_{m-1 \leq i \leq n-1} \{D(i,m-1) + \Delta(i,n)\} & n \geq m > 2 \end{cases}$$

The initial algorithm, given in [8], that uses the function $\Delta(i,j)$, computes the error of approximating all points between two points i and j, each time being needed in partial optimum evaluation, even if this is a redundant operation. The algorithms solving the bounded-ε (min-#) and the bounded-# (min-ε) polygonal approximation paradigms are similar, therefore in the followings only the algorithm for the bounded-ε (min-#) approach is presented and analyzed, but the procedures and the conclusions can be easily extended also to the bounded-# (min-ε) case. The input curve is assumed to have N points and the desired upper bound of the error approximation is $epsi$ and the resulting number of points of the output polygonal approximation is M_out.

Dynamic Programming bounded-ε (min-#) Algorithm (DynProg-E-OLD)

```
// Compute the optimum polygonal approximation
for n:=2 to N do
    ApproxError_{n,2} = Δ(1,n)
    for m:=2 to N do
       for n:=m to N do
          ApproxError_{n,m}:=ApproxError_{n-1,m-1}
```

```
        for i:=n-2 to m-1 do
          dist:=ApproxError_{i,m-1}+Δ(i,n)
          if dist < ApproxError_{n,m} then
              ApproxError_{n,m}:=dist
              SplitPoints_{n,m}:=i
      if ApproxError_{N,m} ≤ epsi then
        M_out:=m
        break
// Save the polygonal approximation; put the points in the correct order
container.add(Point_N)
k:=N
for m:=M_out to 3 do
    k:=splitPoints_{k,m}
    container.add(Point_k)
container.add(Point_1)
```

The order of the time complexity of this algorithm is $O(MN^3)$ and the used memory is $2N^2$, equivalent with $O(N^2)$.

The last part of the algorithm is used to save and to order the points of the output polygonal approximation. To put the points in the correct order, an $N \times M$ matrix, *SplitPoints*, containing the splitting points is computed. This part of the algorithm remained unchanged in all the further versions of the algorithm.

In the first part of the algorithm, which computes the optimum approximation, an $N \times N$ matrix, *ApproxError*, is used to store all the approximation errors obtained when parts of the input curve are approximated with $m \leq M\_out$ points.

4.1.2 Improved Algorithm

The first idea for improving this algorithm was to eliminate the redundant computation of the distances $\Delta(i, j)$ in each step of the optimum searching part of the method. For developing this, the initial loop used to compute the optimum solution is split in two parts:

1. the computation of all the distances between pairs of points from the initial curve and storing them in an extra data structure, which is equivalent to construct a complete graph associated to the input curve, having the points as vertices and the distances between them as label of the its arcs;
2. the search of the optimum approximation using the distances computed in the first step, which is equivalent to find the best path in the distances graph according to a given criterion.

In such circumstances the new algorithm is:

Dynamic Programming bounded-ε (min-#) Algorithm (DynProg-E-NEW)

```
// Distances graph construction
for m:=2 to N do
    for n:=m+1 to N do
        Dist_Graph_{n,m}:= Δ(n,m)

// Searching the optimum polygonal approximation
for n:=2 to N do
    ApproxError_{n,2} = Dist_Graph_{1,N}

for m:=2 to N do
    for n:=m+1 to N do
        ApproxError_{n,m}:=ApproxError_{n-1,m-1}
        for i:=n-2 to m-1 do
            dist:=ApproxError_{i,m-1}+ Dist_Graph_{i,n}
            if dist < ApproxError_{n,m} then
                ApproxError_{n,m}:=dist
                SplitPoints_{n,m}:=i
    if ApproxError_{N,m} ≤ epsi then
        M_out:=m
        break

// Save the polygonal approximation; put the points in the correct order
container.add(Point_N)
k:=N
for m:=M_out to 3 do
    k:=splitPoints_{k,m}
    container.add(Point_k)

container.add(Point_1)
```

In this new approach, in the worst case, the used memory increases from $2N^2$ to $3N^2$, so it remains of order $O(N^2)$. The advantage of the new implementation is the decrease in time complexity, from $O(MN^3)$ to $O(N^3)$.

4.1.3 Theoretical vs. Practical Time Complexity

Experimental running time measurements made using the old and the new implementation are presented in figure 12. Data and experimental conditions are given in annex, table 1, page 56. The difference in running time between the two implementations is obvious, but moreover, the practical results obtained for the new implementation confirm the theoretical expectation, the running time in this case is constant in M.

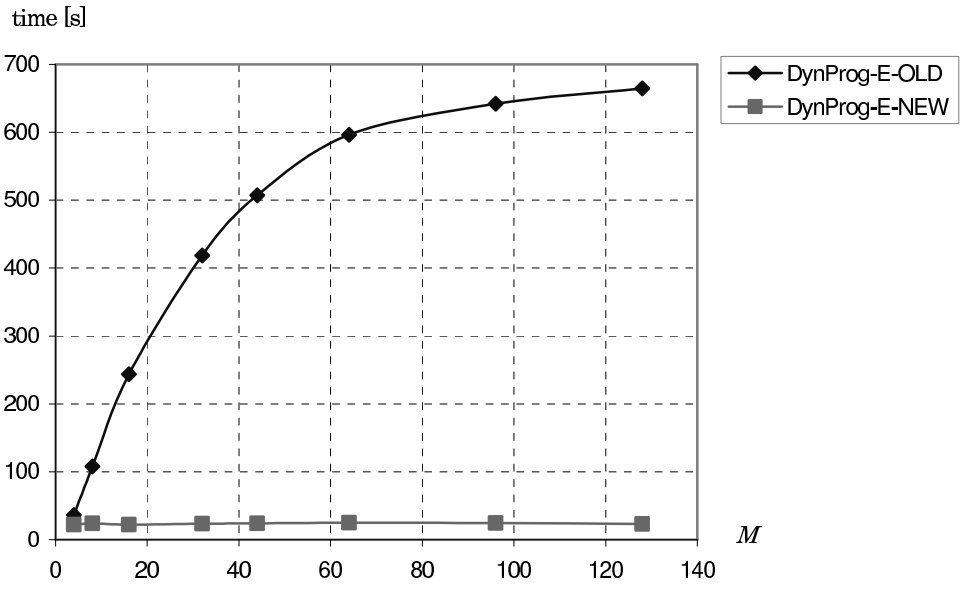


Fig.12 Running time experimental results comparison between the old and the new implementation of the Dynamic Programming method, bounded-ε approach.

In the following, a detailed analysis regarding the running time of the Dynamic Programming new implementation, both from theoretical and experimental point of view, is presented.

The theoretical running time complexity bound is obtained by finding the values for the two parts that composes the new algorithm:

1. the distances graph construction:

$$t_{dist}=\sum_{m=2}^{N-1}\sum_{n=m+1}^{N}\Delta(n,m)=\sum_{m=2}^{N-1}\sum_{n=m+1}^{N}\sum_{i=m}^{n} ct. = O(N^3)$$

2. the search of the optimum solution:

$$t_{search}=\sum_{m=2}^{M}\sum_{n=m+1}^{N}\sum_{i=n-2}^{m-1} ct. = O(MN^2)$$

3. ordering the points of the output polygonal approximation:

$$t_{order}=\sum_{m=1}^{M} ct. = O(M)$$

The total time needed to process an input curve is:

$$t_{DynProg\_NEW} = t_{dist} + t_{search} + t_{order} = O(N^3)$$

Important to remark is that the final bound of the time complexity is imposed by the first part of the algorithm in which the distances graph is computed. Therefore, as it will be further shown, any improvement made in the searching part of the algorithm won't change the theoretical bound of the time complexity as long as the distances graph computation method remains the same.

To point out the correspondence between the theoretical problem and the practical implementation, several experiments were made to estimate practical values of the running time and to compare them with the theoretical bounds computed above.

In figure 13 the influence of the output number of points M in running time complexity of the new implementation of the Dynamic Programming algorithm is presented.

The *points* are the practical assessments:

- by *squares* are represented the running time measurements for distances graph construction in the first step of the algorithm
- by *circles* the experimental values obtained in optimum searching step.

The theoretical traces are overlapped:

- the *solid* one represents the theoretical bound for the distances graph computation step, which is constant in M
- the *dotted line* corresponds to the optimum searching phase, which is linear in M, $O(M)$.

Data and experimental conditions are given in annex, table 2, page 56.

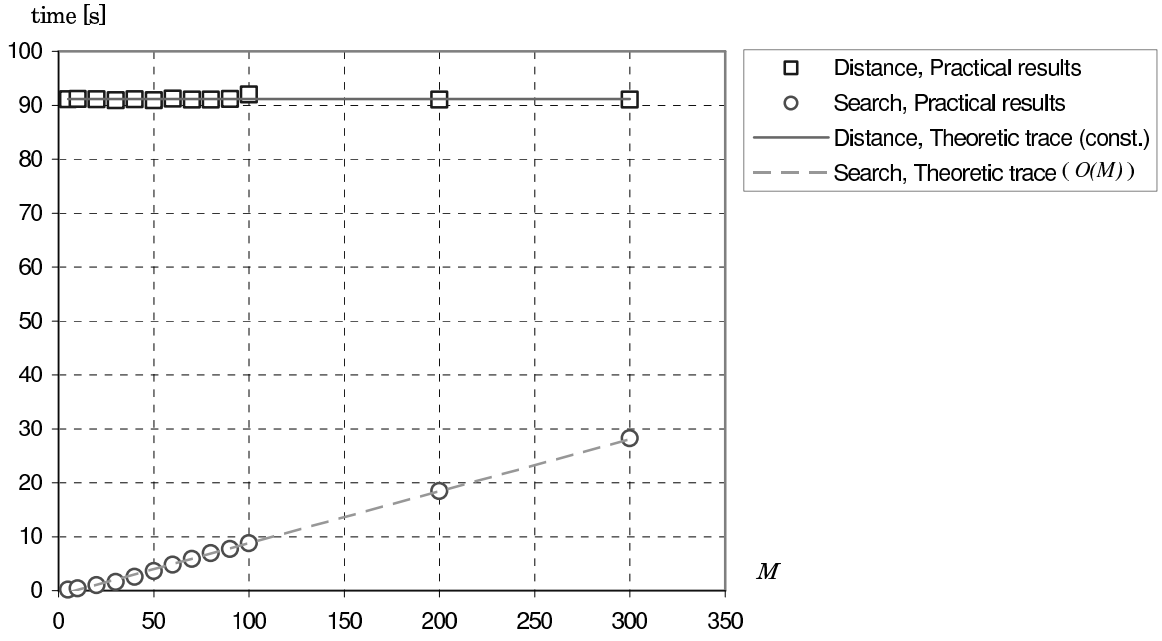


Fig.13 The influence of the output number of points M in running time of new implementation of the Dynamic Programming method, bounded-ε approach; comparison between theoretical bounds and experimental results.

The second dependence, relating the running time complexity to the input number of points N, is shown in figure 14. The experimental results are represented by squares for the distances graph computation phase and by circles for the optimum searching part of the algorithm. Corresponding to these two kinds of practical assessments, the theoretical bounds are: the continuous curve for the distances computation having a $O(N^3)$ complexity and the dotted curve for the searching step of N^2 type.
Data and experimental conditions are given in annex, table 3, page 57.

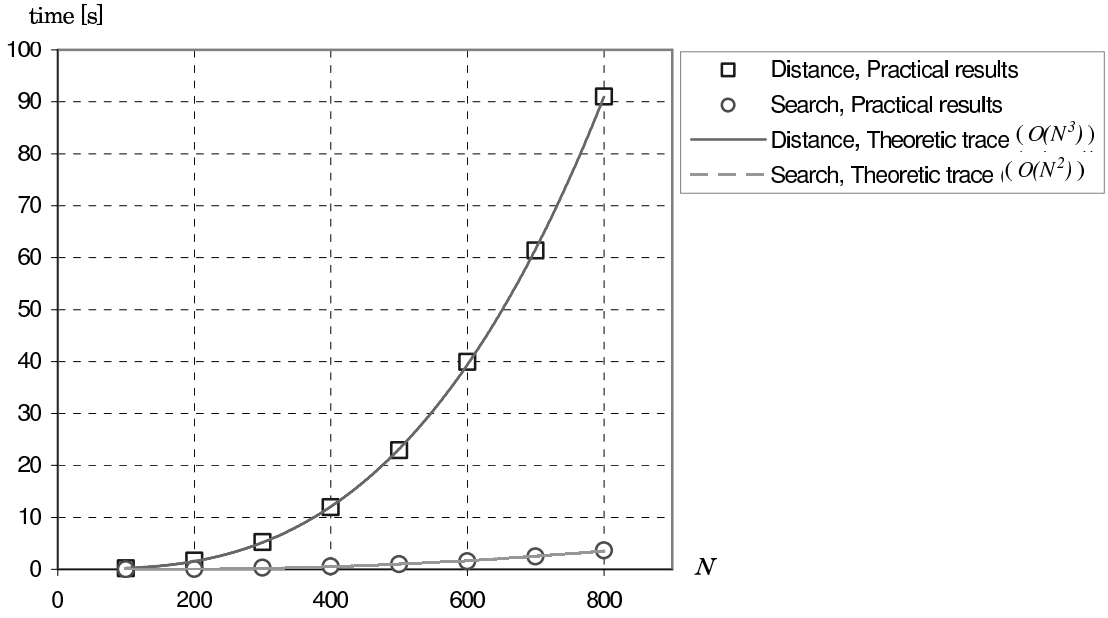


Fig.14 The influence of the input number of points N in running time of new implementation of the Dynamic Programming method, bounded-ε approach; comparison between theoretical bounds and experimental results.

The general conclusion of this part of experimental and theoretical study is the very good matching between the theoretical expectation and the practical results.

4.1.4 Improving the New Implementation of the Dynamic Programming Algorithm

Two modification of the new Dynamic Programming algorithm, are discussed in this section, the first is to improve the memory usage and the second for decreasing the running time. Even if both of them won't change the theoretical bound of the two parameters, some improvement will be obtained in the practical behavior of the algorithm.

The matrix `ApproxError` is used to store the errors of approximation of portions of the initial curve using increasing number of points. The necessity of storing all these values is due to the Dynamic Programming progressive manner of computation that starts with solving simple sub-problems and evolving in each step to a more complex one, in each stage the optimum being computed based on the

previous results. But in a more attentively analysis it will be discovered that the computation is based only on a first order relationship among the consecutives phases of it, which means that to compute the current optimum value, let's say in step m, only the values computed in the former (m-1) step are needed. So, from the beginning, the initial matrix `ApproxError` can be replaced with two N-dimensional vectors, one for storing the optimum values obtained in the last step and the other to keep the values computed in the current step. Moreover, to compute the i-th value from the current step only the component {1,... , i-1} from the former step are needed. Therefore, by changing the order of component computation, processing them from the end of the vector to the beginning of it, as it is explained in figure 15, the used memory in this stage decreases to one N-dimensional vector. The global used memory is ($2N^2+N$), instead of $3N^2$ in the old implementation, but the theoretical bound will remain the same $O(N^2)$.

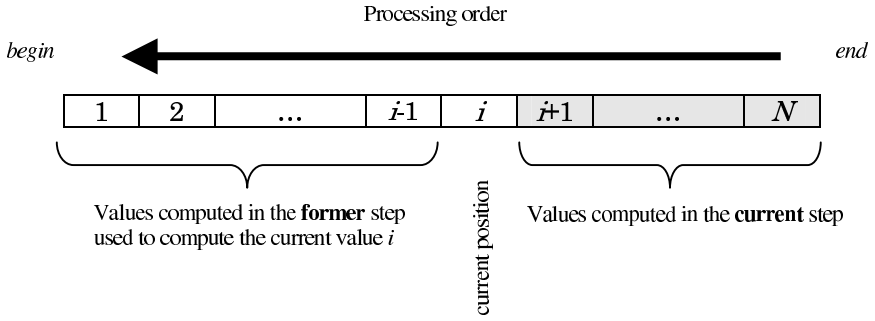


Fig.15 The structure of the vector used in the Dynamic Programming new implementation.

The second improvement concerns the running time and derived from adjusting the initial processed data structure. So, initially the entire distances graph *Dist_Graph(C)* is taken into account to find the optimum polygonal approximation. But in this data structure are stored also paths that corresponds to approximations with error greater than the allowed upper bound limit and therefore in those cases that include one of these paths the computational process is useless. The solution is to process the reduced graph *Dist_Graph(C,ε)* containing only the arcs corresponding to a polygonal approximation error grater than or equal to the upper bound ε.

So, the new and improved version of the Dynamic Programming algorithm is:

Dynamic Programming bounded-ε (min-#) Algorithm (DynProg-E-NEW_IMP)

```
// Distances graph construction
for m:=2 to N do
    for n:=m+1 to N do
        Dist_Graph[n,m]:= Δ(n,m)
// Searching the optimum polygonal approximation
for n:=2 to N do
    VectApproxError[n] = Dist_Graph[1,N]
```

```
for m:=2 to N do
    for n:= N to m+1 do
       VectApproxError_n:=VectApproxError_n-1
       for i:=n-2 to m-1 do
            if Dist_Graph_i,n ≤ epsi then
                dist:=VectApproxError_i + Dist_Graph_i,n
                if dist < VectApproxError_n then
                    ApproxError_n:=dist
                    SplitPoints_n,m:=i
       if VectApproxError_N ≤ epsi then
          M_out:=m
          break

// Save the polygonal approximation; put the points in the correct order
container.add(Point_N)
k:=N
for m:=M_out to 3 do
   k:=splitPoints_k,m
   container.add(Point_k)
container.add(Point_1)
```

The average decrease of the running time, computed over the entire set of experimental data presented in table 4 and table 5 in annex (pages 57, 58), is 10% and it depends on the complexity of the reduced graph that is established by the value of the approximation error upper bound ε. This experimental result is expected, because the algorithm modification works only over the optimum searching phase and the main part of the time is consumed in the distances graph construction step, which makes the theoretical running time complexity to remain at the same value of $O(N^3)$.

The comparison of the experimental results obtained running the new and the new improved implementations of the Dynamic Programming algorithm, for the particular case of an input error upper bound $\varepsilon=1$, is presented in figure 16. The average decrease of the running time, computed only for experimental data measured in this case ($\varepsilon=1$), is 20%. A more complete report regarding these experimental data is given in table 4 and table 5 in annex, (pages 57, 58).

Better results are obtained when special distance computation methods are applied to decrease the running time needed to construct the graph in the first stage. This is the situation when the incremental technique proposed by Perez and Vidal in [4] is used to construct the distances graph. In this case the running time complexity of the first stage become comparable with de optimum searching step, $O(MN^2)$, and, therefore, the decrease obtained by the new-improved implementation in the running time of the searching process is reflected as a better improvement of the entire algorithm.

The entire set of experimental results presented in annex, 4 and table 5 (pages 57, 58), shows that in case of using incremental technique for distance computation, the ratio of the running times measured for the new implementation and the new-improved version ranges between [1.5, 4.9], depending on the reduced-graph simplification degree.

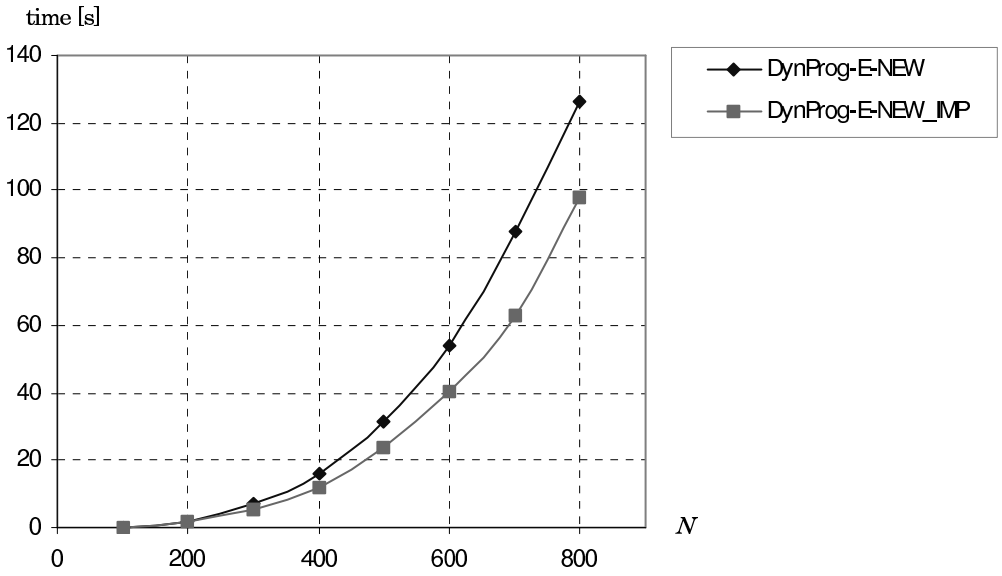


Fig.16 Running time experimental results comparison between the new (NEW) and the new-improved (NEW_IMP) implementations of the Dynamic Programming method, approach (ε=1), using the Sum of the Euclidean distances.

Figure 17 presents a comparison between the running time obtained for the new and the new-improved implementations of the Dynamic Programming algorithm, when the incremental technique is used in the distances graph assessment and for the particular case of an input error upper bound ε=1. For the data measured only in this particular case (ε =1), the average ratio representing the decrease of the running time is 4.8. The entire set of the data measured in this experiment is given in table 4 and table 5 in annex (pages 57, 58).

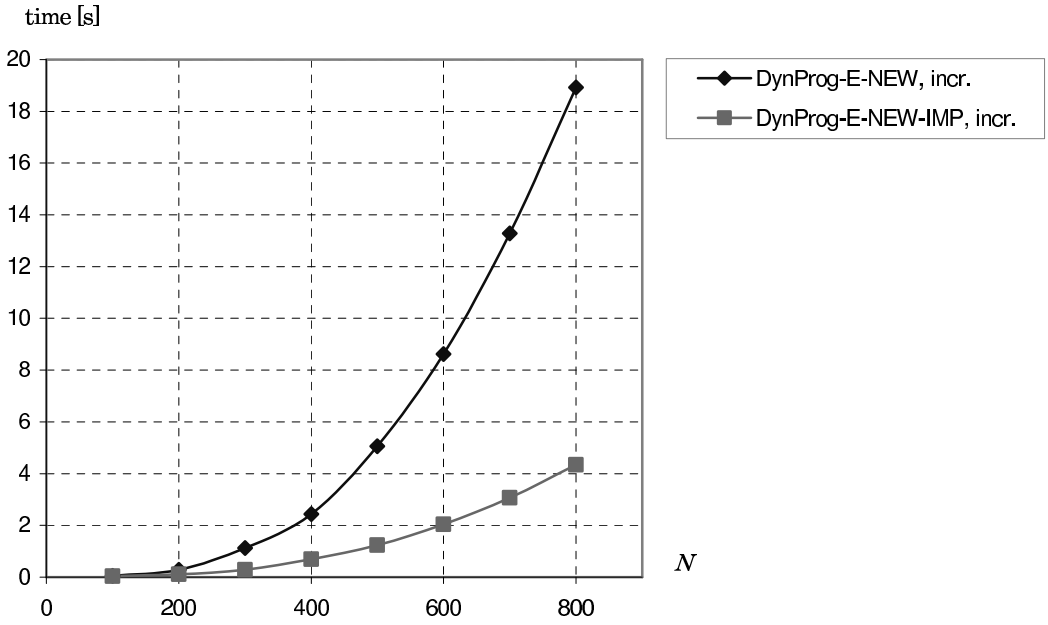


Fig.17 Running time experimental results; comparison between the new (NEW) and the new-improved (NEW_IMP) implementations of the Dynamic Programming method, bounded-ε (min-#) approach (ε =1), using the incremental technique for computing the sum of the Euclidean distances.

4.1.5 Generic Implementation vs. Dedicated Implementation

The software implementing the Dynamic Programming algorithms is designed in a generic manner, being able to deal with different kinds of distance measures, different kinds of data storing structures, different kinds of number types and kernels. The advantages of such an implementation are its generalization and flexibility, but these are paid with the increase of the running time.

To estimate this loss of performance, a dedicated version of software was implemented in such a manner to optimize the new-improved Dynamic Programming algorithm designed particularly for the sum of the Euclidean distances error assessment.

The experimental running time results measured for the generic and dedicated implementations, which are given in table 6 (annex, page 59), show that the loss in running time because of the generic implementation is ranging between 20% and 67%, depending on the values of the input and the output parameters.

A graphic representation of the difference between the running time of the generic and the dedicated implementations of the Dynamic Programming algorithm is presented in figure18.

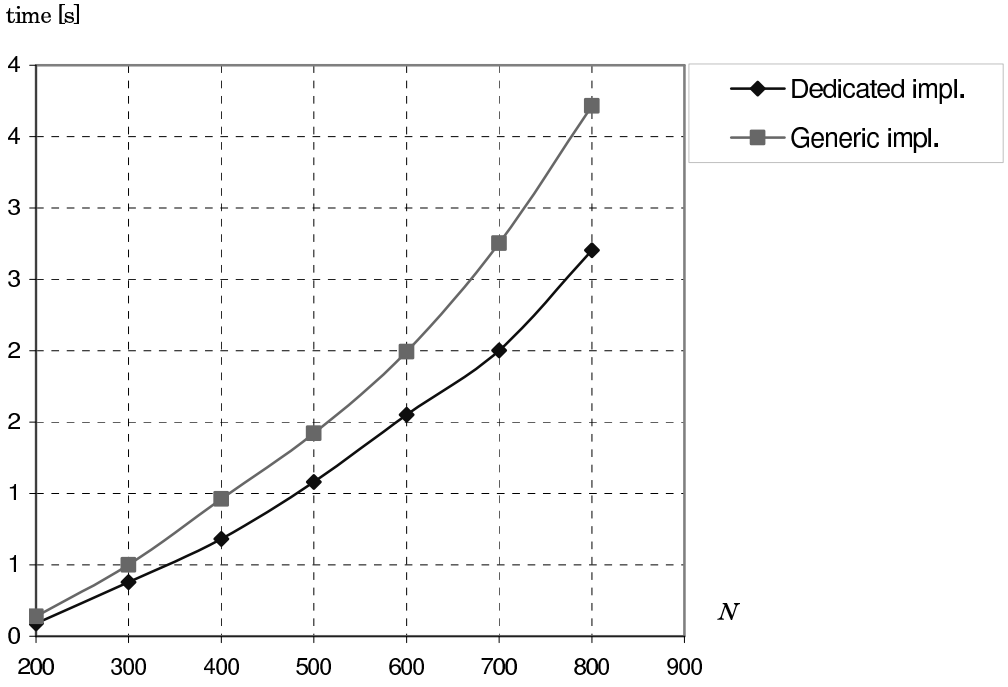


Fig.18 Running time experimental results, comparison between the generic and the dedicated implementations of the Dynamic Programming method, bounded-# (min-ε) approach (M=75), using the incremental technique for computing the sum of the Euclidean distances.

4.2 Implementations of the Recursive Split Method

The Recursive Split method is a heuristic technique used to find a sub-optimal, but often good, solution of a polygonal approximation problem. The basic idea of this algorithm consists of repetitively applying of the following steps:

- Approximate the current curve portion with the segment linking its end points. The current curve at the beginning of the algorithm is the entire input curve.
- Using a given criterion, find the splitting point of the current curve and compute the approximation error between the current curve and its approximation segment.
- If the approximation error is greater than a given threshold then repeat the above-explained steps for the two sub-curves resulted by dividing the current curve portion in the splitting point.

Due to the scheme of the Recursive Split method, the simplest way to implement it in a bounded-ε approach is to use a recursive technique, as it is explained below. The input function parameters are:

- the first point of the curve, p_1
- the last point of the curve, p_n
- the approximation error upper bound, *epsi*
- the list storing the points of the polygonal approximation, *poly_list*

Recursive Split Algorithm, bounded-ε approach,
recursive implementation (RecSplit_be_rec)

```
RecursiveSplit_be_rec(p1,pn,epsi,&poly_list)
                                          // compute the polygonal approximation
                                          // of the curve p1…pn with an error less
                                          // then or equal to the threshold epsi
{
    d:=error(p1,pn,&p_split)  // the function error returns the error of
                              // approximating the curve p1…pn with the segment p1pn
                              // and also returns the splitting point p_split

    if d > epsi  then

    add_list(p_split,&poly_list)

    RecursiveSplit_be_rec(p1,p_split,epsi)
    RecursiveSplit_be_rec(p_split,pn,epsi)
}
```

The alternative to this manner of implementation is the iterative technique that does not use the function self-calling. Designing the iterative implementation of the Recursive Split method is a more complex task that requires a supplementary *list* data structure for storing the segments of the current polygonal approximation of the input curve. In each step of the algorithm the current polygonal approximation is changed and therefore the *list* structure is dynamically implemented in such a manner to be able to cumulate all the modifications of the approximation polygon with the minimum computational effort.

The basic element of the list, which corresponds to a segment of the approximation polygon, stores the following information:

begin	corresponds to the starting point of the segment
end	represents the ending point of the segment
split_pt	is the splitting point of the current curve portion

Also, there exists the *next* field used to link the elements of the list.

With these notions already introduced, the iterative implementation of the Recursive Split method in bounded-ε approach is as follows:

```
Recursive Split Algorithm, bounded-ε approach,
                           iterative implementation (RecSplit_be_it)

     RecursiveSplit_be_it(p1,pn,epsi)   // compute the polygonal approximation
                                        // of the curve p1...pn with an error less
                                        // then or equal to the threshold epsi
     {
          // initiate the list with the first approximation segment
          // that links the two end points of the input curve
          list_elem:= new_element()  // generate a new element of the list
          list_elem.next:=NULL
          list_elem.begin:=p1
          list_elem.end:=pn

          d:=error(p1,pn,&p_split) // the function error returns the error of
                                   //approximating the curve p1...pn with the segment p1pn
                                   // and also return the splitting point p_split

          list_elem.split_pt:=p_split

          current_elem:=list_elem // store the current segment

          while d>epsi do

               elem_new:=new_element()     // generate a new element of the list
               elem_new.next:=current_elem.next
               current_elem.next:=elem_new
               new_elem.end:=current_elem.end
               new_elem.begin:=current_elem.split_pt
               current_elem.end:=current_elem.split_pt

               d:=error(current_elem.begin,current_elem.end,&p_split)

               while current_elem.next && d<=epsi do
                    current_elem:=current_elem.next
                    d:=error(current_elem.begin,current_elem.end,&p_split)

               current_elem.split_pt:=p_split
     }
```

The confidence in obtaining a better implementation was the reason of studying and testing the iterative version of the Recursive Split algorithm. Actually, a performance improvement in running time was expected after eliminating the function self-callings.

Table 2 Comparison between the experimental results obtained for the recursive and the iterative implementations of the Recursive Split polygonal approximation method in bounded-ε approach.

Number of running the algorithm	Kernel	Data type	N	error	Time [s]	
					Recursive	Iterative
1	Cartesian	double	850	0.1	0.010	0.010
1	Cartesian	double	850	1.0	0.011	0.010
1	Cartesian	double	850	10.0	0.011	0.011
1	Homogeneous	leda_integer	850	0.1	0.601	0.591
1	Homogeneous	leda_integer	850	1.0	0.550	0.480
1	Homogeneous	leda_integer	850	10.0	0.320	0.290
100,000	Cartesian	double	550	1.0	106.563	105.712
300,000	Cartesian	double	550	5.0	736.071	739.827

On the contrary, the experimental results, which are presented in table 2, show that the two implementations, recursive and iterative, are similar from the running time point of view. The explanation of this experimental result is the time needed to process the list data structure from the iterative implementation that is equivalent with the time spent in function self-callings from the recursive version.

The second reason of studying the iterative technique described above was to prepare the implementation of the Recursive Split method in bounded-# approach. In the bounded-ε all the curve portions are repetitively split until their approximation error becomes less than or equal to the input threshold. Therefore, in this approach the order of segments processing is not important, the approximation criterion giving the possibility to split them in the order of their appearance as it is happened in the recursive implementation. In the bounded-# approach this processing manner is not possible. In this case, to obtain a good approximation of the input curve, which means to have the smallest approximation error for the desired number of points, the segments must be processed in the decreasing order of their approximation error. Thus, due to this requirement and also because the recursive implementation does not offer the possibility to process the segments in the correct order, the only technique that can be applied to implement the bounded-# approach of the Recursive Split method is the iterative one.

The iterative implementation of the Recursive Split method in bounded-# approach is more complicated than the previous explained one. This is because in this new implementation the segments must be ordered regarding their approximation error. To process always the segment with the greatest approximation error a ***priority queue*** data structure was added. Thus, the correct processing order of the segments, which is necessary in obtaining the best solution of the bounded-# approach, is established by the following two queue operations:

- ***extraction*** from the queue of the first element gives in each step the segment with the greatest approximating error
- ***insertion*** in the queue of the two segments that result in a splitting operation must be done on the corresponding positions to preserve the elements decreasing order relative to the approximation error.

The iterative implementation of the bounded-# uses also a list data structure to store the segments of the approximation polygon. This list is similarly organized like that described above in the bounded-ε approach implementation.

The basic element of the priority queue stores the following information:

seg	is a link to the segment element from the polygonal approximation list
error	represents the approximation error of the current the segment
next	field used to link the elements of the priority list

The iterative implementation of the Recursive Split polygonal approximation in bounded-# approach is described below:

Recursive Split Algorithm, bounded-# approach, (RecSplit_bn)

```
RecursiveSplit_bn(p1,pn,M)   // compute the polygonal approximation
                             // of the curve p1…pn using M points
{
        // initiate the list with the first approximation segment
        // that links the two end points of the input curve
        list_elem:= new_element() // generate a new element of the list
        list_elem.next:=NULL
        list_elem.begin:=p1
        list_elem.end:=pn

        d:=error(p1,pn,&p_split) // the function error returns the error of
                                 //approximating the curve p1…pn with the segment p1pn
                                 // and also return the splitting point p_split

        list_elem.split_pt:=p_split

        //initiate the priority queue with the first segment of the list
        priorityQ.seg:=list_elem
        priorityQ.error:=d
        priorityQ.next:=NULL

        n_points:=2

        while n_points < M do
             current_elem:= extract_priorityQ() // extract the first
                                                // element of the priority queue

             elem_new:=new_element()     // generate a new element of the list
             elem_new.next:=current_elem.next
             current_elem.next:=elem_new
             new_elem.end:=current_elem.end
             new_elem.begin:=current_elem.split_pt
             current_elem.end:=current_elem.split_pt
             d:=error(current_elem.begin,current_elem.end,&p_split)
             current_elem.split_pt:=p_split
             insert_priorityQ(current_elem,d)
             d:=error(new_elem.begin, new_elem.end,&p_split)
             new_elem.split_pt:=p_split
             insert_priorityQ(new_elem,d)
             n_points ++
}
```

A common problem of all possible implementations of the Recursive Split method is the finding of the splitting point. In the classical manner of computation, the algorithm deals with the furthest curve point to the approximating segment (figure 19). In this case, finding the splitting point means to compute the maximum value of the distances between the curve points and the approximating segment.

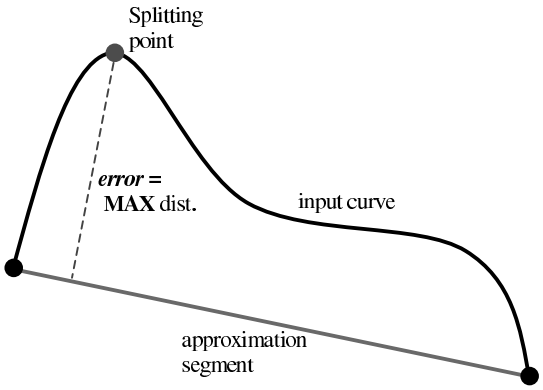


Fig. 19 Finding the splitting point as the furthest curve point to the segment

A new approach used in finding the splitting point was introduced in this study. The alternative is to detect the curve point that divides the area between the curve and the approximating segment in two equal parts (figure 20). To find this point, the sum of the distances from the curve points to the segment must be computed.

In the both approaches, different kinds of distances can be used to determine the splitting point. Therefore, the entire software implementing the Recursive Split algorithm was designed in a generic manner. Thus, at the Recursive Split function could be plugged in different kind of distance traits classes.

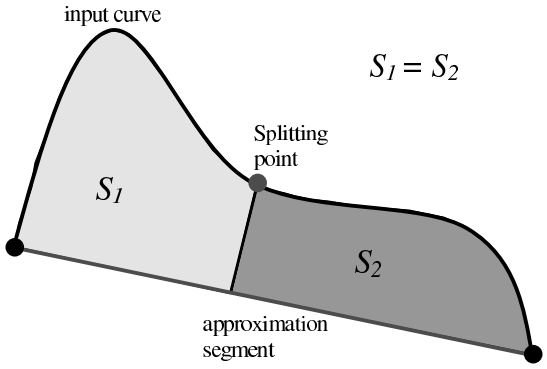


Fig. 20 The splitting point is dividing the area in two equal parts

Using the dividing area method to detect the splitting point is not equivalent to computing the polygonal approximation in bounded-ε approach, based on the sum of distances criterion. In the above-explained approach, the sum of distances is used local only to split the area between the current curve portion and its approximating segment in two equal parts, but not as a global measure over the entire curve approximation.

To obtain an implementation of the recursive Split method dealing with the sum of distances criterion, the algorithm must cumulate the local errors in such a manner to be able to asses the sum of the distances over the entire input curve. The modification of the algorithm consist of adding the following two steps used to calculate the global approximation error:

- ***subtract*** the approximation error corresponding to the current segment that disappear from the polygonal approximation after the splitting operation
- ***add*** the new approximation errors characterizing the two segments obtained in the splitting process of the current segment.

Based on these modifications, the Recursive Split method using the sum of distances for error assessment is implemented as follows:

Recursive Split Algorithm, bounded-ε approach, (RecSplit_be_sum)
the error is assessed as the Sum of distances

```
RecursiveSplit_be_sum(p1,pn,epsi) // compute the polygonal approximation
                                  //of the curve p1…pn with an error=sum of distances
                                  // less then or equal to the threshold epsi
{
        // initiate the list with the first approximation segment
        // that links the two end points of the input curve
        list_elem:= new_element() // generate a new element of the list
        list_elem.next:=NULL
        list_elem.begin:=p1
        list_elem.end:=pn

        d:=error(p1,pn,&p_split) // the function error returns the error of
                                 //approximating the curve p1…pn with the segment p1pn
                                 // and also return the splitting point p_split

        list_elem.split_pt:=p_split

        //initiate the priority queue with the first segment of the list
        priorityQ.seg:=list_elem
        priorityQ.error:=d
        priorityQ.next:=NULL
        global_error:=d

        while global_error < epsi do
                current_elem:= extract_priorityQ() // extract the first
                                                   // element of the priority queue

                global_error:= global_error - error(current_elem)

                elem_new:=new_element()     // generate a new element of the list
                elem_new.next:=current_elem.next
                current_elem.next:=elem_new
                new_elem.end:=current_elem.end
                new_elem.begin:=current_elem.split_pt
```

```
                    current_elem.end:=current_elem.split_pt

                    d:=error(current_elem.begin,current_elem.end,&p_split)

                    global_error:= global_error +d
                    current_elem.split_pt:=p_split
                    insert_priorityQ(current_elem,d)

                    d:=error(new_elem.begin, new_elem.end,&p_split)

                    global_error:= global_error +d
                    new_elem.split_pt:=p_split
                    insert_priorityQ(new_elem,d)

                    n_points ++

        }
```

4.3 Graph Search Method Implementation

To find an optimal solution to a polygonal approximation problem, the Graph Search method applies searching techniques to find the shortest path in the distances-graph associated to the input curve. Applying the breadth first search method, false loops in the resulting polygonal approximation can appear. Therefore, the breadth-first search technique was changed to process the points only in the forward direction, from the first point p_1 to the end point p_N.

To obtained better results in running time, the reduced distances-graph containing only the arcs with the weight less than or equal to the input error threshold ε is used. Also, for the same reason, in the construction of the distances-graph, the on-line convex hull based Euclidean distance is applied.

To find the shortest path in the distances-graph, the Graph Search method uses three auxiliary data structures:

- *visited* is a vector used to mark all the points that were already processed in the current step;
- *queue* is a queue structure used in ordering the points in the breadth first search manner;
- *poly_list* is a list structure storing the output approximation polygon.

The algorithm of the Graph Search method using the modified breadth search algorithm is:

```
Graph Search Algorithm, bounded-ε approach, (GraphSearch_be_max)
                        the error is assessed as the Maximum of distances

    GraphSearch_be_sum(p1,pn,epsi) // compute the polygonal approximation
                                   //of the curve p1…pn with an error=max of distances
                                   // less then or equal to the threshold epsi
    {
        // Distances graph construction
        for m:=2 to N do
            for n:=m+1 to N do
                Dist_Graphn,m:= on_line_convex_hull_distance(n,m)
```

```
// Searching the optimum polygonal approximation
// initiate the data structures
initiateQ(queue)
initiateL(poly_list)
initiateV(visited)
search_flag:=1
addQ(queue,p1)
mark_visited(visited, p1)
while search_flag=1 do
        pc:=get_topQ(queue)
        for p:=pc+1 to pn do // process the points only in the forward direction
            if Dist_Graph pc,p <epsi then    //process only the arcs from
                                             // the reduced graph
               if p=pn then    // the end of the curve is reached
                  search_flag:=0
                  break
               if not_visited(visited, p) then
                  add_list(poly_list, p)
                  mark_visited(visited, p)
                  addQ(queue,p)
}
```

4.4 Distance Traits Classes Implementation

One of the advantages of using generic programming techniques to implement polygonal approximation algorithms consists of the possibility to preserve the same function code and to plug in different kind of distance traits classes. Thus, without any change in the function implementation, several polygonal approximations with different kind of error assessment can be easily obtained.

When the software is designed in a generic manner, important features like compactness, flexibility and generality are conferred to the resulted code.

4.4.1 Implementation Solutions

In this section are pointed out the steps and the decisions made in the strategy of implementing the distance traits classes that are based on distances from L_p category. To illustrate the progress obtained in software development, the algorithms presented here refer only to the particular case of the Euclidean distance measured to the line support. Similar improvement steps were applied also for other distances from the L_i category, like: Manhattan distance and L_∞ distance measured either to the segment or to the line support. Moreover, the same strategies were used in developing the software for vertical distance. The experimental results obtained for all these distances are similar to those presented here for the Euclidean distance.

To assess the error obtained in approximation of a curve by the segment linking its end points, the distances between the points of the curve and the segment must be computed. Therefore, the initial idea used in designing the distance traits classes software was to solve the basic geometric operations by applying the corresponding functions from the CGAL library, which were already available. Thus, the first implementation was based on the following algorithm:

Euclidean Distance Algorithm using CGAL-functions (EuclDistCGAL)

```
MaxEuclideanDistance(p1,pn)      // compute the approximation error
                                 // between the curve p1...pn and the segment p1pn
{
        pc:=p1          //initialize the current point
        max:=0
        p_max:=p1
        do
            pc++        // take the next point
            d:=CGAL_EclideanDistance(pc,p1pn)      // compute the distance
                                                   //from the point pc to the segment p1pn
            if d > max then
               max:=d
               p_max:=pc
        while pc != pn
}
```

Unfortunately, poor experimental performances were obtained for this version of computing the approximation error. The first step made to improve the running time of the algorithm was to replace the calling of the CGAL functions with equivalent own-implemented *inline* code. Putting all the code inside the distance function, the running time decreases because of eliminating the time needed before to call the CGAL functions and therefore is not expected a great improvement of the performances. With these modifications, the new algorithm is:

Euclidean Distance Algorithm using inline code (EuclDistInline)

```
MaxEuclideanDistance(p1,pn)  // compute the approximation error
                             // between the curve p1...pn and the segment p1pn
{
        pc:=p1          //initialize the current point
        max:=0
        p_max:=p1
        do
            pc++                // take the next point

            // compute inline the distance from the point pc to the segment p1pn
            dx:=pn.x - p1.x
            dy:=pn.y - p1.y
            d:=(pc.y - p1.y)*dx - (pc.x-p1.x)*dy
            d:=d*d / (dx*dx + dy*dy)
```

```
            if d > max then
                max=d
                p_max:=pc
        while pc != pn
}
```

Further improvement of the algorithm used to assess the approximation error is based on an observation that will be discussed in the following. The error introduced by replacing a curve portion with the segment linking its end points is measured by computing all the distances between the points of the curve and the approximation segment. During this procedure, the parameters of the approximation segment remain unchanged and the current point is moving from the beginning to the end point of the curve. So, it is easy to observe that in computing the approximation error there are used two categories of parameters: those with constant value, which are related to the approximation segment features and the parameters with variable value that depend on the coordinates of the current point changing along the curve. According to this observation, the algorithm computing the approximation error can be improved by splitting the previous version in two parts:

- Computation of the constant parameters that are not related to the current point, which is done at the beginning of the procedure, outside the loop used to move the points along the curve;
- Computation of the parameters depending on the coordinates of the current point that is done inside the loop crossing the curve, because for each step of the loop the coordinates of the current point change and the parameters must be recomputed.

After these specifications, the new improved algorithm used to compute the approximation error is:

Euclidean Distance Algorithm Improved Version (EuclDistImp)

```
MaxEuclideanDistance(p1,pn)  // compute the approximation error
                             // between the curve p1...pn and the segment p1pn
{
        pc:=p1          //initialize the current point
        max:=0
        p_max:=p1

        // Compute the parameters depending on the segment p1pn features
        dx:=pn.x - p1.x
        dy:=pn.y - p1.y
        dt:=dx*dx + dy*dy
        v:=p1.x*pn.y + p1.y*pn.x
        do
            pc++        // take the next point

            // Compute the local parameters depending on the current point pc coordinates
            t:=v - pc.x*dy + pc.y*dx
            d:=(pc.y - p1.y)*dx - (pc.x-p1.x)*dy
            d:=t*t / d

            if d > max then
                max=d
                p_max:=pc

        while pc != pn
}
```

To compare the performances of the distance assessing algorithms discussed above, in table 3 are presented the experimental values of running time obtained for Dynamic Programming polygonal approximation method, in bounded-# (min-ε) approach. These are the experimental data obtained in the particular case of approximating the curve from file "kk1.dat" (figure 21), with N=560 input number of points and M=64 desired output number of points.

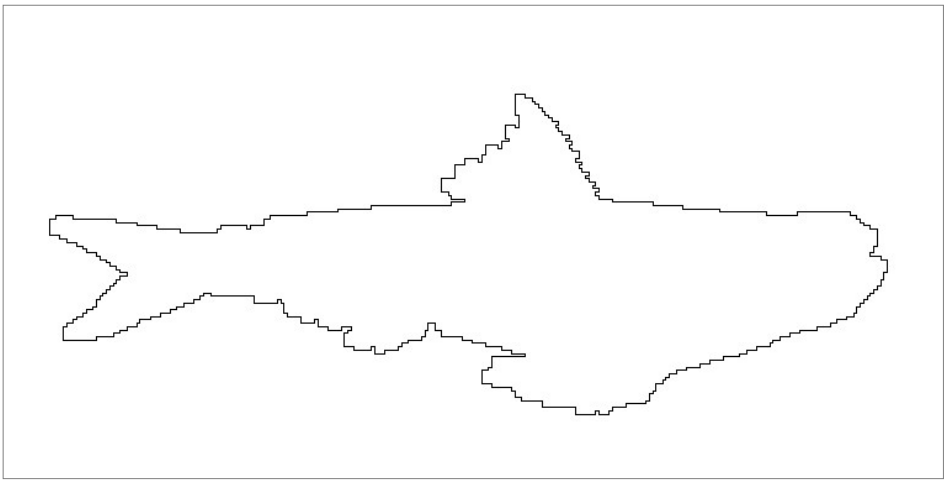


Fig. 21 Example of input curve from set "fish"; file "kk1.dat".

The experiments made for different polygonal approximation methods (Dynamic Programming, Recursive Split, in both bounded-# and bounded-ε approaches) and for different distances used to assess the approximation error (Euclidean, Manhattan, L_∞, vertical) show similar ratios between the three versions of implementation like those given in table 3.

Table 3 Comparison of the experimental running time measured for the different implementation of the Euclidean distance. The ratios are computed taking as reference the time obtained for **`EuclDistCGAL`** version.

Maximum Euclidean Distance Algorithm	Distance measured to the line support		Distance measured to the segment	
	time [s]	ratio	time [s]	ratio
`EuclDistCGAL`	374.137	1.000	997.839	1.000
`EuclDistInline`	141.814	0.379	352.649	0.357
`EuclDistImp`	36.392	0.097	57.147	0.058

4.4.2 Exact Computation

Among the advantages of using generic techniques in implementing the polygonal approximation algorithms is the possibility to plug in and, thus, to make computations in different kind of kernel representations like Cartesian and Homogeneous. The Homogeneous kernel uses rational numeric representation that offer the possibility to evaluate all the calculations in exact arithmetic and thus to obtain an exact value of the final result of the processed algorithm. Therefore, according to this

idea, one of the requirements that had to be accomplished in designing the distance assessment algorithms was to find a computation method capable to provide the exact value of the distance. To this purpose all the operations that are evaluated by the computer using approximation techniques, like: square root (SQRT) and all the trigonometric operations (sin, cos), had to be avoided. This restriction imposed to the manner of implementation imply one of the following two cases:

- There are algorithms that can be solved using either inexact computation or exact computation strategies, but usually the former method is faster that the latter one.
- There are algorithms that admit only the solution implying operations evaluated by inexact computation. In these cases, an upper bound value computed in exact arithmetic, which is a good approximation of the real solution, must be found.

4.4.2.1 The Influence in Running Time

As explained before, an implementation of the polygonal approximation algorithms using only operations involving exact computation was required. To observe the influence in running time of such a strategy of implementation, several methods, both from the exact computation and the inexact computation categories, were studied.

This section points out the difference in running time between the two categories of algorithms. The **`EuclDistImp`** algorithm, which is given in section 4.2.1, represents the best implementation of the Euclidean distance using only exact computation operations. For the second category, a good implementation of the Euclidean distance assessment using trigonometric functions, which implies inexact calculations, is presented next:

Euclidean Distance Algorithm Trigonometric Version (EuclDistTrig)

```
MaxEuclideanDistance(p1,pn)  // compute the approximation error
                             // between the curve p1...pn and the segment p1pn
{
        pc:=p1          //initialize the current point
        max:=0
        p_max:=p1

        // compute the parameters depending on the segment p1pn features
        alfa:= - atan((pn.x - p1.x) / (pn.y - p1.y))
```

```
        do
            pc++        // take the next point
            d:= pc.y * cos(alfa) + pc.x * sin(alfa)
            if d > max then
                max=d
                p_max:=pc
        while pc != pn
}
```

The comparison of the running times presented in figure 22 is obtained in an experiment of polygonal approximation based on Dynamic Programming method, bounded-# approach.

The approximation error is assessed using the maximum Euclidean distance, which is measured either to the line support or to the segment, is implemented in both manners: using exact computation operations and using trigonometric functions (inexact computation). The average value of the ratio between the running time of the inexact method and the running time of the exact method is 0.456 for the distances measured to the line support and 0.482 for the distances measured to the segment.

The entire set of the data measured in this experiment is given in table 7 in annex, page 59.

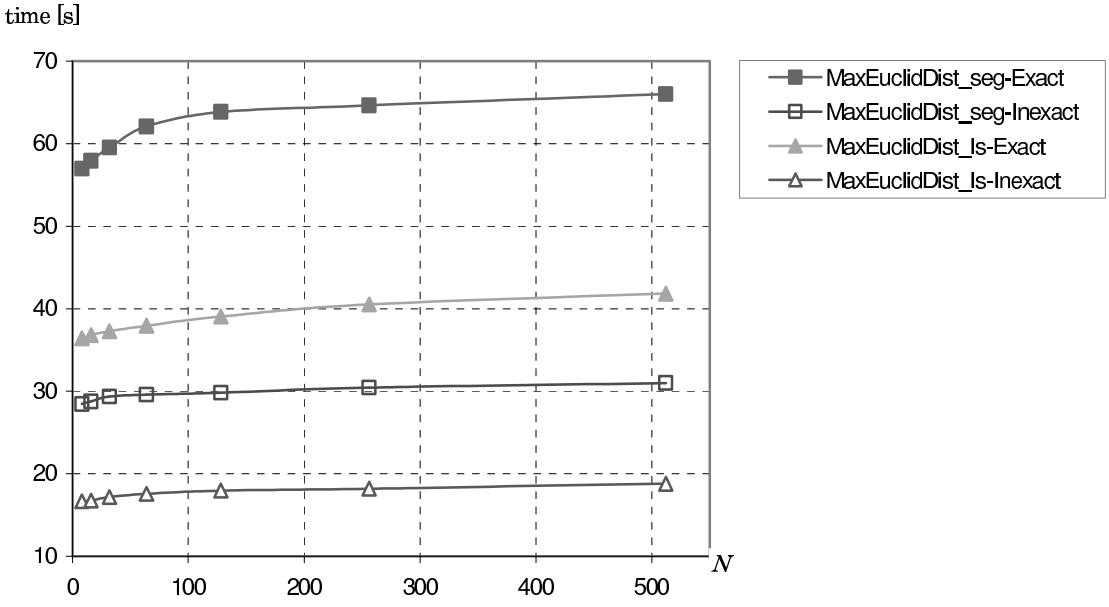


Fig.22 Running time experimental results comparison between the exact computation version (`Exact`) and the trigonometric based (`Inexact`) implementations of the Euclidean Distances assessment algorithm.

4.4.2.2 Finding Upper Bound of the Real Value

Sometimes the entities that are required to be evaluated cannot be computed avoiding operations like square root or trigonometric functions. In these cases, the only solution of the exact computation is to find an upper bound function that approximate the real value and can be assessed using only exact computation operations.

This is the situation appeared in evaluating the bounding shell distance. The bounding shell distance between a point and a segment is defined as the maximum distance between the segment and the arc passing through the end points of the segment and the input point. If the length of the segment is L and the radius of the circle containing the arc is R (figure 23), then the bounding shell distance is:

$$d_{BS} = R - \sqrt{R^2 - \frac{L^2}{4}}$$

This value cannot be computed without using the square root operation. Therefore, the only way to use this distance in exact computing applications is to find an upper bound function that is a good approximation of the real distance and that can be computed in exact arithmetic.

The best upper bound function found to approximate the bounding shell distance is:

$$d^2_{BS\_up} = \frac{\left(\frac{L}{2}\right)^4}{4R^2 - 3\left(\frac{L}{2}\right)^2}$$

To point out the approximation quality of the bounding shell distance, in figure 24 is presented the variation of the ratio between the upper bound function and the real value. As it can be observed from the figure, if the radius R is greater then $2L$, then the approximation error is less the 3%.

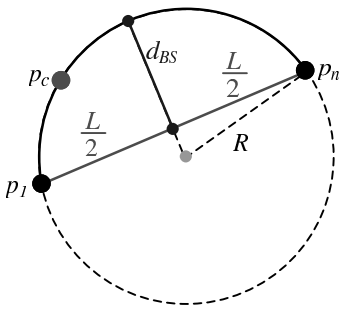


Fig.23 The elements used in computing the bounding shell distance.

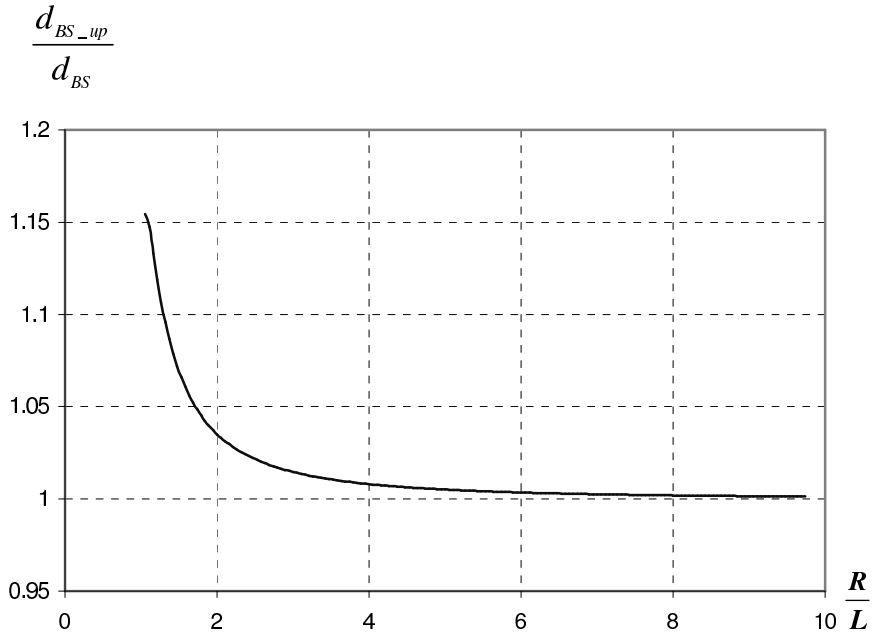


Fig.24 Variation of the upper bound function used to approximate the bounding shell distance.

4.4.2.3 Implications of the Exact Computation

Due to the generic manner of software designing, different kinds of kernels and base data types can be used to solve polygonal approximation tasks. As a direct result of this facility, there exists the possibility to calculate the correct solution, without rounding errors, of the polygonal approximation problems using only exact computation operations. To be more specific, the exact computation applications can be obtained in two ways related to the manner of combining the kernel representations and the base data types:

- Using the Cartesian kernel based on an exact real number format like the `leda_real` type;
- Using the Homogeneous kernel, which implies rational number representation, based on exact integer number format like: `leda_integer` and `GmpZ`.

Regardless the kernel type, in an exact computation task the base data type must always provide the correct (errorless) number value, without any alteration of it. This requirement is not easy to be accomplished, because of the following restrictions that characterized, at least, the standard number types:

- The rounding operations that are usually applied in the real number representations
- The bounded range of the variables

Both of these two inconveniences are the result of the fixed memory space allocation for each type of variable. For the Homogeneous kernel, when the rational number representation is used, all the operations are basically calculated by multiplication between integer values. In such a situation the

values of the numerators and the denominators are increasing exponentially. This conclusion was practically confirmed in an experiment using the Homogeneous kernel based on the `long int` base data type, in which the correct solution of the polygonal approximation can be obtain for input curves with less than 30 points. For more complex input curves the numbers became greater then the upper bound of the `long int` number type range of representation and the final result is a wrong solution of the approximation problem.

This manner of constant memory space allocation used in the standard memory management is not possible to be applied in designing the exact computation applications that don't admit rounding operations and deal with values in different ranges of variation. The solution of this problem consists of using special data types using adaptive dynamic memory space allocation, capable to store any value without loss of precision. This solution is generally valid from a theoretical point of view, because for practical reasons it can be applied only in the limit of the available computer memory. Moreover, the time needed to process such an adaptive number structure is increasing with the dimension of the memory space allocated to it.

So, two negative effects appeared in processing the exact computation tasks based on rational number representation: the exponential increase of the memory space allocated to the new data types that implies also the increase of the time needed to process them. The solution applied to improve these problems was a ***rescaling*** technique. To decrease as much as possible the memory space the rescaling method eliminates all the common factors of the numerator and the denominator for each rational number. This idea is implemented by a function computing the greatest common factor that is called after each operation involving rational numbers.

The experimental running times measured for Dynamic Programming polygonal approximation method, bounded-# approach, computed using different kinds of kernels and base data types are graphical represented in figure 25. More details about data and the measuring conditions are given in annex, table 8, page 60.

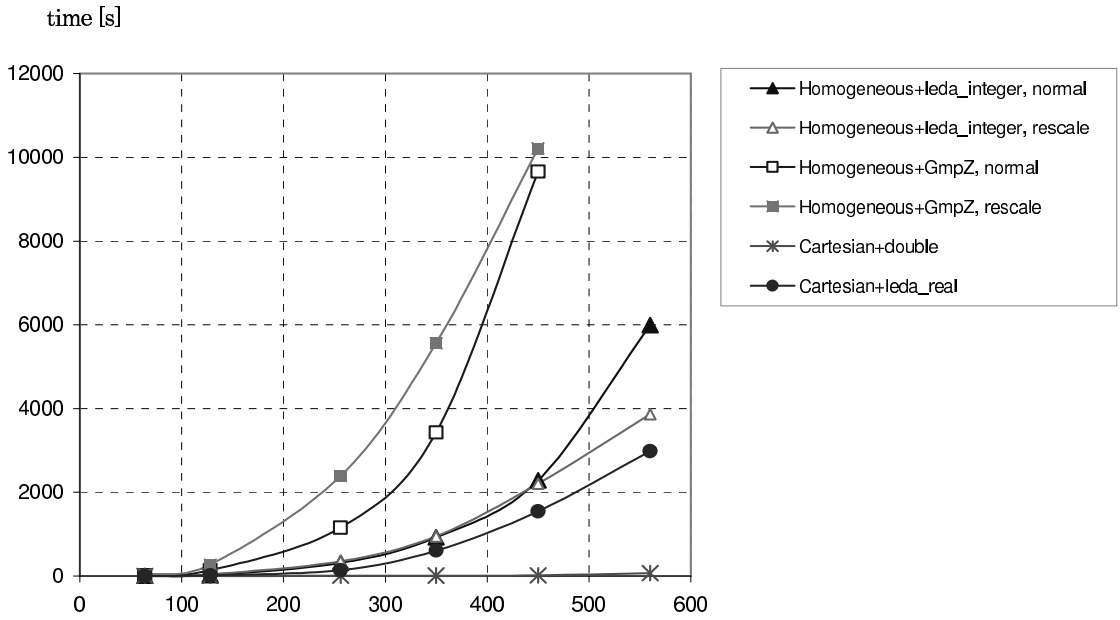


Fig. 25 The running time experimental results measured for implementations of the Dynamic Programming method using different kind of kernels and base data types.

As can be observed from the graphic, for a small number of input points, implying a reduced number of operations, the implementation using the rescaling technique is worse than the direct rational number computation. The reason of this behavior is the space used to store the rational numbers that is still small enough and therefore the time needed for adjusting operations is greater than the time used to process directly the rational numbers. But for greater numbers of input points, the number of operations increases and a larger space is allocated for the exact number representations. Thus, in this case the rescaling technique becomes efficiently, the time spent for number-adjusting operations is smaller then those used in the direct processing implementation.

Another conclusion observed from the experimental measurements is the similar behaviors of the implementation using the Cartesian kernel based on the `leda_real` number type and the case of applying the Homogeneous kernel with the `leda_integer` and rescaling technique.

4.4.3 Notes About the Path Hull Distance Implementation

An attempt to improve the computation of the maximum Euclidean distance between a curve portion and the segment linking its ends is to decrease the number of points that are considered valid in computing it. In the classical implementation all the points of the curve portion are used to find the maximum Euclidean distance, but not all of them are good candidates of the furthest point to the segment. Thus, an improvement of the algorithm can be obtained by eliminating all the points that are already known as bad solutions of the furthest point to the segment. This is possible by computing the convex hull of the curve portion, because all the points that are not part of the convex hull are inside of it and therefore they are at a smaller distance to the segment than are the points belonging to the convex hull.

The on-line method based on the ***path hull*** data storing structure is applied to compute the convex hull of the curves. When this technique is applied in a Recursive Split polygonal approximation task, in each step the furthest point divides the current curve in two sub-curves and corresponding to this operation, the current hull must be split also in two sub-hulls related to the two sub-curves. To accomplish this operation two methods were implemented.

The first method consists of introducing a history stack structure that is used to restore the previous stages of building the current hull. In such a way, one of the sub-hulls resulting from the splitting operation can be recovered from the current hull by calling back from the history stack the previous operations used to construct the current hull. The second sub-hull must be build from the beginning using the on-line convex hull algorithm.

The alternative of this strategy is to build always both of the sub-hulls. In this kind of implementation the history stack is not needed and also the function recovering the previous stages of the current hull is useless. Actually in this case only the on-line convex hull algorithm is applied twice: to the set of curve points that are before the current furthest point and to set of curve points that are after it.

Studying these two implementations from a practical point of view, interesting conclusions were found. In table 4 are presented the results obtained running the two versions of the path hull distance.

Table 4 Running time comparison between the *restore* and the *build* versions of path hull distance implementation. The results are given for two categories of curves: with large convex parts (kk1.dat) and without large convex parts (random generated).

curve	N	M	Time [s]		Restore time / Build time
			Restore	Build	
kk1.dat	560	79	0.600	0.100	6.000
kk1.dat	760	100	0.800	0.141	5.674
random	360	211	0.080	0.131	0.611
random	760	352	0.151	0.511	0.296

For the curves that contain large convex parts the recovering operation became expensive in running time. This is because the convex hull is almost similar with the processed curve portion and the operations needed to manage the history stack are time consuming. Therefore, in these cases the second solution based on building the both sub-hulls gives better results.

On the other hand, if the input curves have frequent changes of direction that interrupts its convexity then the recovering version gives better results. In these cases the building of the convex hull involves more operations than is needed to recover it from the previous one.

The experimental results given in table 4 are obtained using the curves presented in figure 21 (page 37) and figure 26. The first curve, stored in file "kk1.dat", corresponds to the first category that contains large convex parts. The second curve with small convex parts was random generated.

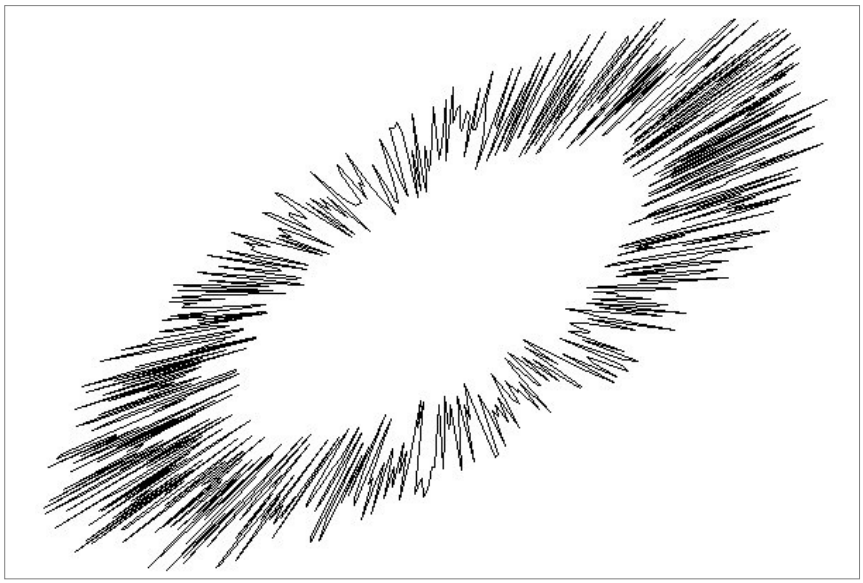


Fig. 26 Example of artificial generated curve, containing small convex parts.

4.4.4 Two-Dimensional and Three-Dimensional Distance Implementations

For more generality of the polygonal approximation software, a library containing 3-dimensional distance algorithms was added to it. Thus, a new feature is available: by using the generic implemented polygonal approximation functions together with the 3-dimensional distance traits classes, now curves from a 3-dimensional space can be processed.

The 3-dimensional distance algorithms were designed using the experience cumulated during the implementation and optimization of the corresponding 2-dimensional methods. Similar techniques with those used in 2-dimensional case, like computing the current point depending parameters inside the curve crossing loop and the others before the loop, were adapted to the 3-dimensional applications. To exemplify the correlation between the 2-dimensional and the 3-dimentional methods, the 3-dimentional version of the algorithm used to assess the maximum Euclidean distance measured to the line support that is given below can be compared with its 2-dimensional version (**`EuclDistImp`**) presented in section 4.2.1.

Euclidean Distance Algorithm 3D Version (EuclDist3D)

```
MaxEuclideanDistance_3D(p1,pn)   // compute the approximation error
                                 // between the curve p1..pn and the segment p1pn
{
        pc:=p1          //initialize the current point
        max:=0
        p_max:=p1

        // compute the parameters depending on the segment p1pn features
        dx:=pn.x - p1.x
        dy:=pn.y - p1.y
        dz:=pn.z - p1.z
        dt:=dx*dx + dy*dy + dz*dz

        do
            pc++          // take the next point

            // Compute the local parameters depending on the current point pc coordinates
            deltax:=p.x - p1.x
            deltay:=p.y - p1.y
            deltaz:=p.z - p1.z
            t:=(deltax*dx+ deltay*dy+ deltaz*dz)/dt

            cx:=deltax - dx*t
            cy:=deltay - dy*t
            cx:=deltaz - dz*t
            d:= cx*cx + cy*cy + cz*cz
            if d > max then
                max=d
                p_max:=pc
        while pc != pn
}
```

The results of running time obtained for the 2-dimentional and the 3-dimentional implementations of polygonal approximation methods based on different kind of distance assessments are given in table 5 and also some of them are graphical represented in figure 27.

More exactly, in figure 27 are represented the variations of the running time obtained for Dynamic Programming method, bounded-# approach, using different distances measured to the line support, applied for solving polygonal approximation applications in 2-dimensional and 3-dimensional spaces. The data used in this representation are given in table 9 from annex, page 60.

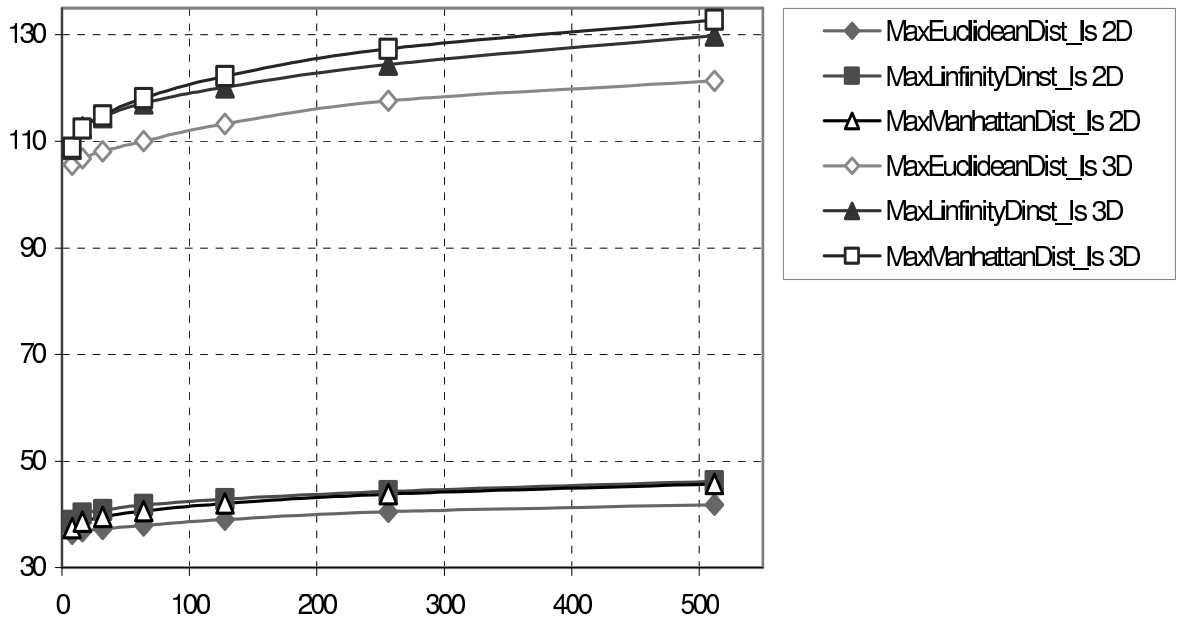


Fig. 27 The running time experimental results; comparison between different kind of distances measured for 2-dimesional and 3-dimensional implementations.

In table 5 are presented the average value of the ratios between the running times obtained for the 2-dimensional and the 3-dimensional implementations. This time are given the results of a more exhaustive study, the measurements covering almost all the possible polygonal approximation applications:

- using the Dynamic Programming and the Recursive Split methods,
- in both bonded-# and bounded-ε approaches,
- assessing the error as the maximum distance or as the sum of the distances.
- the distances are measured either to the line support or to the segment
- the input curves are in 2-dimensional or in 3-dimensional space.

As was expected the 3-dimensional applications run slower than the 2-dimensional implementations. For the distances measured to the line support they are about 3 times slower and for the distances measured to the segment they become approximately 2 times slower. This difference is because the 2-dimension distances measured to the line support are easier to compute.

Table 5 The running time experimental results measured for implementations of the Dynamic Programming and Recursive Split methods, in bounded-# and bounded-ε approaches, using different kind distances for solving 2-dimensional and 3-dimensional applications

Distance	Average ratio of 2D / 3D running times			
	Bounded-ε (ε =1)		Bounded-# (*M* =200)	
	Dyn Prog	Rec. Split	Dyn Prog	Rec. Split
MaxEuclid_ls	0.331	0.500	0.345	0.333
MaxLinfinity_ls	0.327	0.500	0.356	0.333
MaxManhattan_ls	0.324	0.333	0.344	0.333
MaxEuclid_seg	0.493	0.333	0.502	0.667
MaxLinfinity_seg	0.488	0.667	0.499	0.500
MaxManhattan_seg	0.497	0.667	0.496	0.500
SumEuclid_ls	0.343	0.500	0.356	0.667
SumLinfinity_ls	0.457	0.333	0.344	0.333
SumManhattan_ls	0.473	0.333	0.342	0.333
SumEuclid_seg	0.540	0.500	0.496	1.000
SumLinfinity_seg	0.653	0.500	0.503	0.667
SumManhattan_seg	0.595	0.500	0.499	0.500

4.5 Comparison Between Polygonal Approximation Methods

Besides the theoretical research involved in this work, there existed a practical purpose of it consisted of designing a software library containing polygonal approximation functions and their auxiliary distance traits classes. To reach this goal, a comprehensive analysis covering different categories of algorithms and implementation techniques was performed. Programming techniques like generic and dedicated implementations, recursive and iterative schemes were applied to determine their benefits in the practical behavior of the algorithms. Also, different methods from optimal and sub-optimal classes were studied and tested to find the correspondence between the loss of the solution optimality and the practical running time improvement. The results and the conclusions obtained in all these experiments are discussed below classified relative to the polygonal approximation approach and to the distance type used to assess the approximation error.

4.5.1 Bounded-ε Approach, Sum of the Euclidean Distances

In this case three algorithms were studied:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search*, in two versions:
  - sub-optimal, using the breadth first search algorithm;
  - optimal, based on finding the shortest path in a weighted graph algorithm [9].

  Both versions are implemented in a dedicated way, using only the sum of the Euclidean distances to assess the approximation error.

The running time experimental results are presented in figure 28. As was expected, the Dynamic Programming is the slowest algorithm and the Recursive Split is the fastest one. It is interesting to note the improvement of the running time performances when the Dynamic Programming implementation is running using the incremental technique to assess the sum of the Euclidean distances.

The entire set of experimental data and the measuring conditions are given in annex, table 10, page 61.

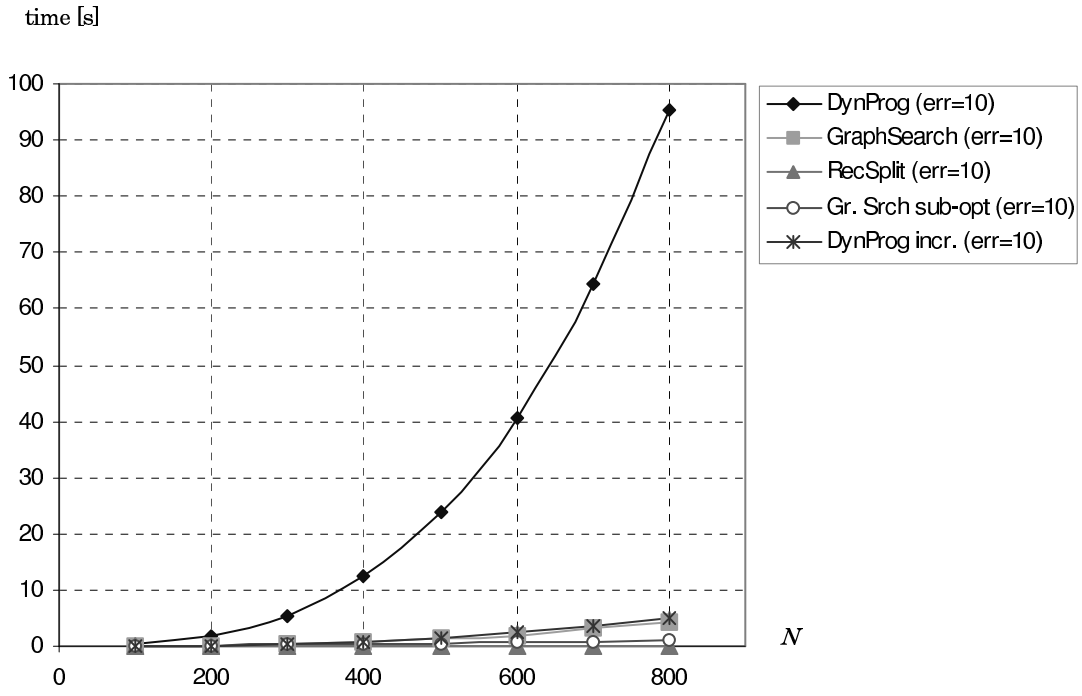


Fig. 28 Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded-ε approach, using the sum of the Euclidean distances as error assessment.

In figure 29 are graphically represented the variations of the output number of points that results applying different polygonal approximation algorithms. Compensating the gain obtained in running time, the variation of the output number of points obtained for the sub-optimum algorithms presents a deviation from the optimum values. As it can be observed, the performances in approaching the optimum values of the output number of points are in reverse order relative to the running time order: the faster is the implementation, the greater is the deviation from the optimum output results.

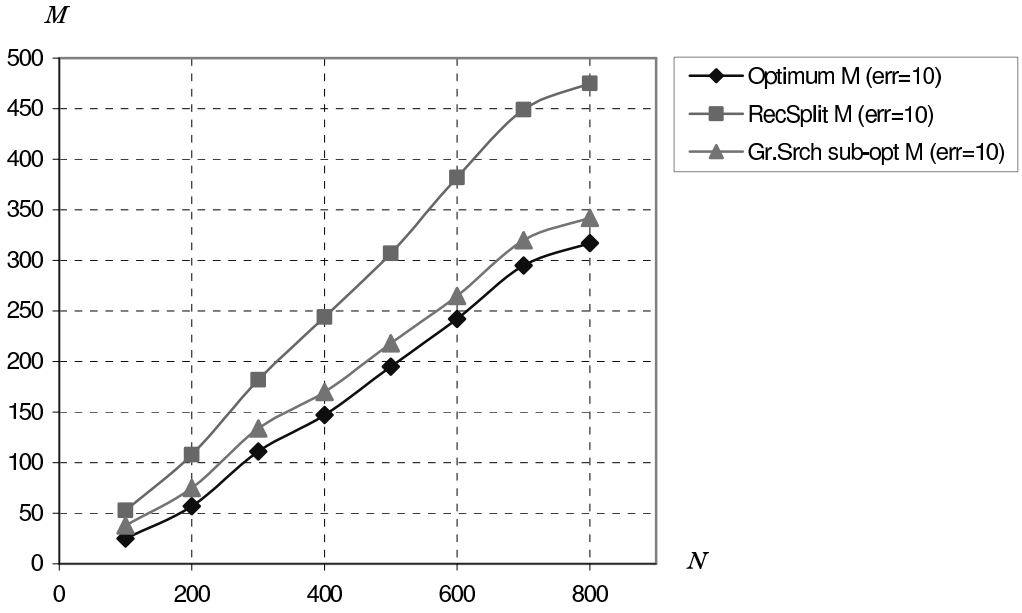


Fig. 29 Variation of the output number of points; comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithm in bounded-ε approach, using the sum of the Euclidean distances as error assessment.

Figure 30 presents the comparison between the running times obtained for Dynamic Programming using incremental distance computation and the optimal Graph Search implementation. As it was expected, the results are similar, because in the above-mentioned conditions the two algorithms become equivalent: optimal algorithms using the sum of the Euclidean distance error assessment. The small variation between the experimental running time results comes from the implementation difference that exists between the two algorithms: the Dynamic Programming uses the generic techniques and the Graph Search is implemented in a dedicated way.

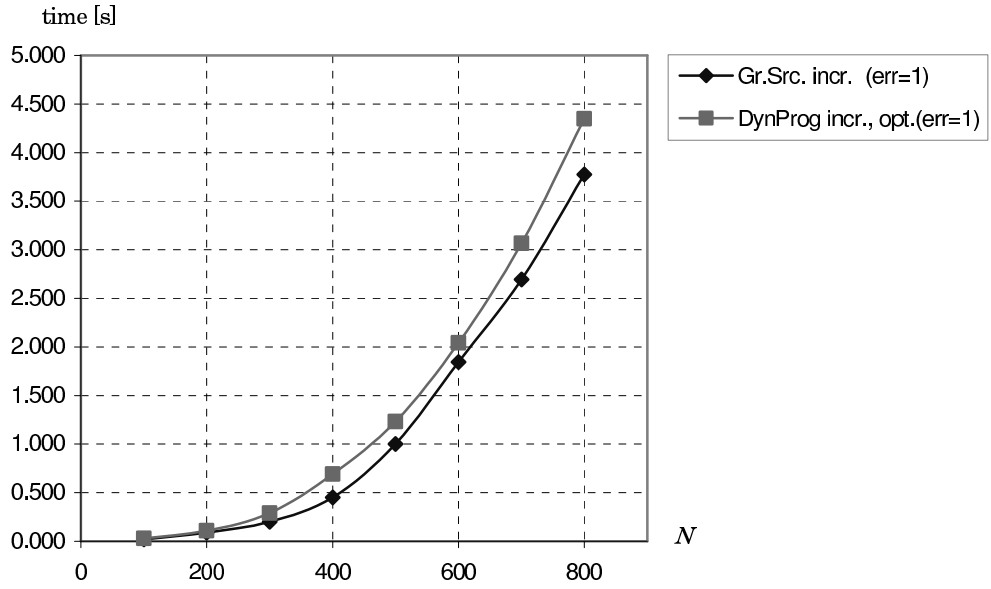


Fig. 30 Comparison of the running times obtained for the Dynamic Programming method using the incremental distance computation and the optimum Graph Search algorithm

4.5.2 Bounded-ε Approach, Maximum of the Euclidean Distances

Three algorithms were studied:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search*, which is based on the breadth first search algorithm, finds in this approach an optimal solution to a polygonal approximation problem. Applying only the maximum of the Euclidean distances to assess the approximation error, a dedicated scheme was chosen to implement this algorithm.

Two parameters were analyzed and compared: the running time and the resulting number of points obtained in the polygonal approximation.

The running time experimental results are presented in figure 31. The Dynamic Programming is the slowest algorithm, the Recursive Split is the fastest one and the Graph Search has medium results. Maybe it is interesting to point out the Graph Search running time results that are better than the Dynamic Programming even if both of them are optimal methods. But, on the other hand, this conclusion is compensated by the generality and the flexibility offered by the Dynamic Programming algorithm that can be applied using different kind of distances and different distance cumulating techniques (maximum and sum) used to assess the approximation error, relative to the rigidity of the Graph Search method that finds an optimal solution only for the maximum Euclidean distance error assessment. Moreover, as it was already presented in section 4.5.1, the Graph Search method based on the breadth first search algorithm becomes sub-optimal when the sum of the Euclidean distances is used to compute the approximation error.

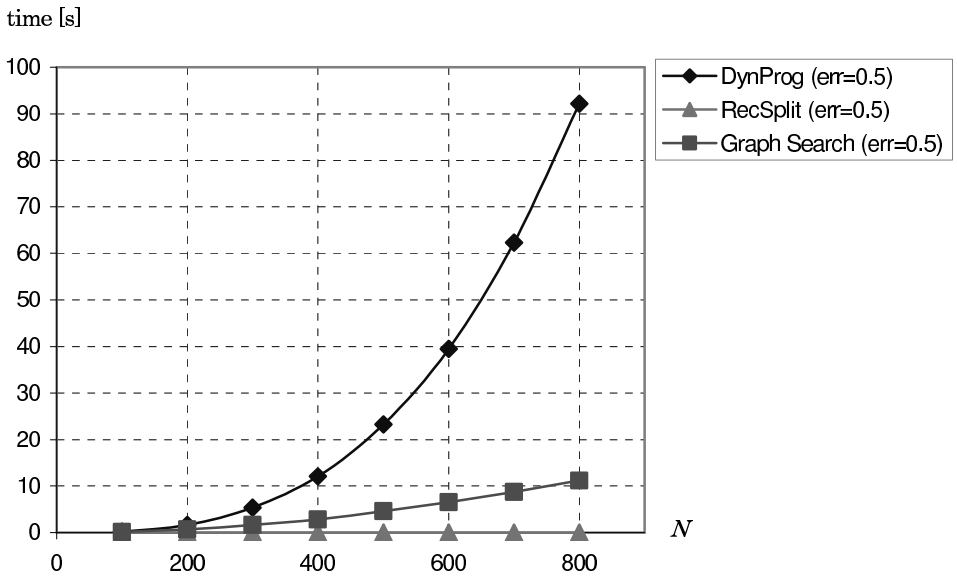


Fig. 31 Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded-ε approach, using the maximum of the Euclidean distances as error assessment

The variations of the output number of points that results applying different polygonal approximation algorithms are presented in figure 32. As it was expected, there is a deviation from the optimum solutions of the results obtained applying the sub-optimal method (Recursive Split).

More details about the experimental data and the measuring conditions are given in annex, table 11, page 62.

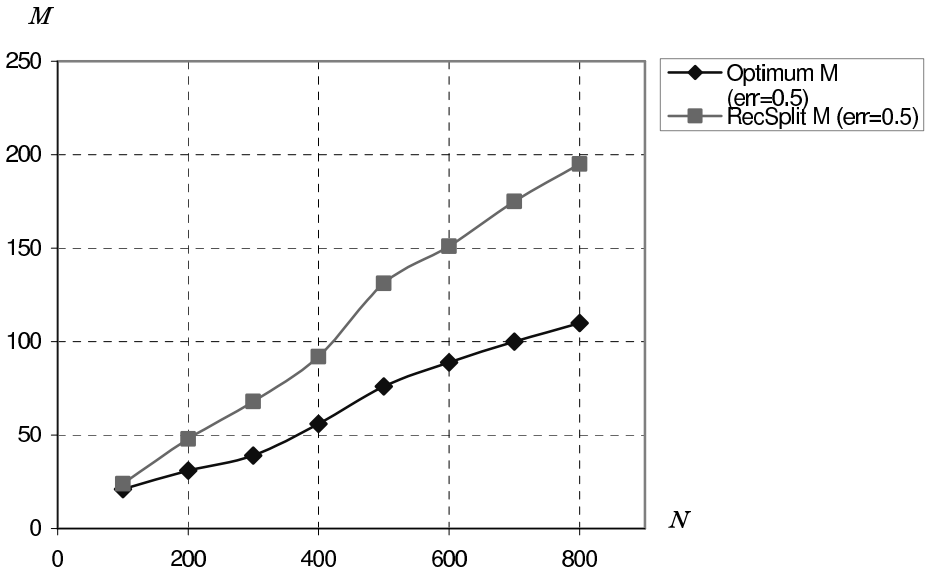


Fig. 32 Variation of the output number of points; comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithm in bounded-ε approach, using the maximum of the Euclidean distances as error assessment.

4.5.3 Bounded-# Approach, Sum of the Euclidean Distances

The algorithms studied in this case are:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search*, optimal algorithm based on finding the shortest path in a weighted graph algorithm [9]. The implementation is dedicated only to the sum of the Euclidean distances approximation error assessment.

Similar results and conclusions with those presented in section 4.4.1 are obtained also for this case.

With respect to the running time performances the algorithms are ordered from the fastest that is the Recursive Split method, followed by the Graph Search algorithm, ending with the Dynamic Programming method that gives the poorest results for the classical implementation. A good improvement of Dynamic Programming time results is obtained when in error assessment the incremental technique is applied to compute the sum of the Euclidean distances. All these results are given in graphical form in figure 33.

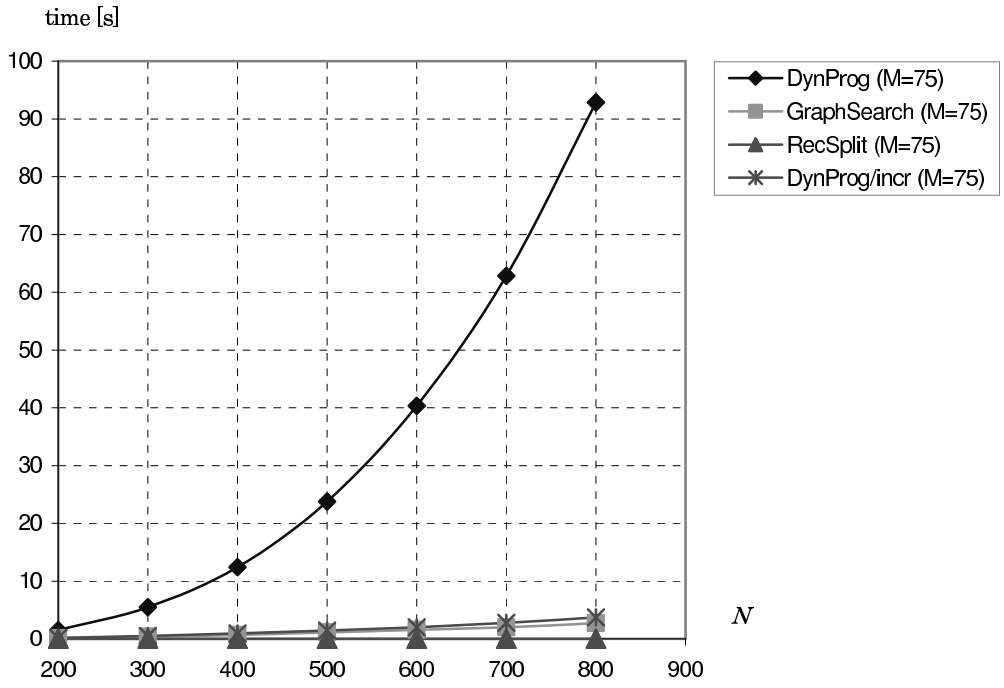


Fig. 33 Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded-# approach, using the sum of the Euclidean distances as error assessment

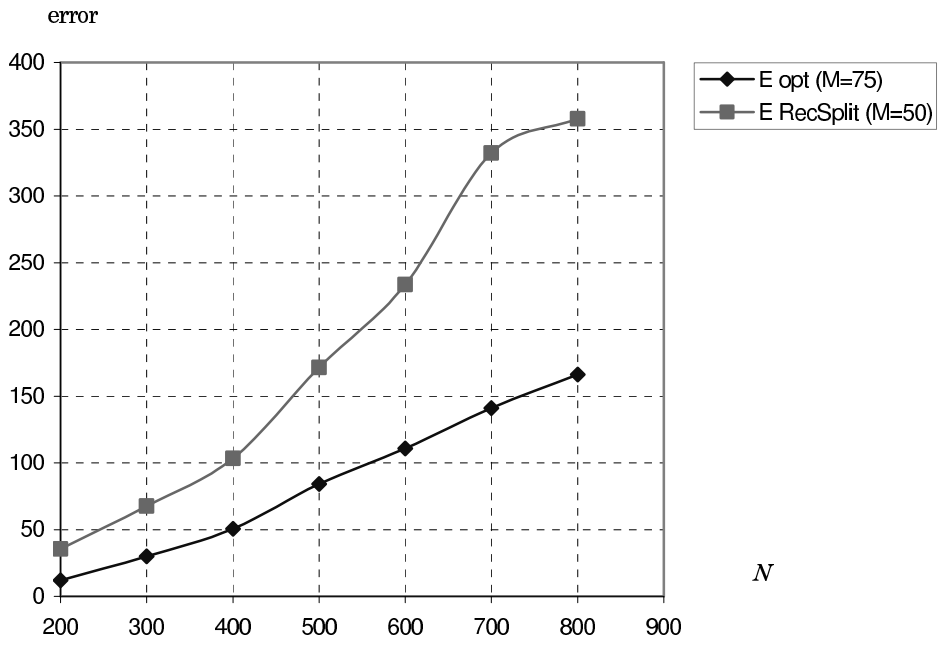


Fig. 34 Variation of the approximation error; comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithm in bounded-# approach, using the sum of the Euclidean distances as error assessment.

The advantage of the sub-optimal method is the good rate of its processing time, but this efficiency in time performance is paid by the loss of precision in finding the correct solution. In this approach the correctness of the solution is measured by the approximation error obtained for a desired output number of points. This error is minimal for the optimal methods (Dynamic Programming, Graph Search) and it has greater values for the sub-optimal methods. As a confirmation of these comments, in figure 34 are presented the variations of the approximation error obtained for optimal and sub-optimal methods.

The entire set of experimental data and the measuring conditions are given in annex, table 12, page 63.

4.5.4 Bounded-# Approach, Maximum of the Euclidean Distances

Three algorithms were studied:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search* uses the breadth first search algorithm to find an optimal solution to a polygonal approximation problem. The dedicated implementation uses only the maximum of the Euclidean distances to assess the approximation error.

The running time experimental results are presented in figure 35. Similar with the results obtained in the previous experiments, also in this one the best running time results are obtained for the Recursive Split implementation and opposite to it is the Dynamic Programming method.

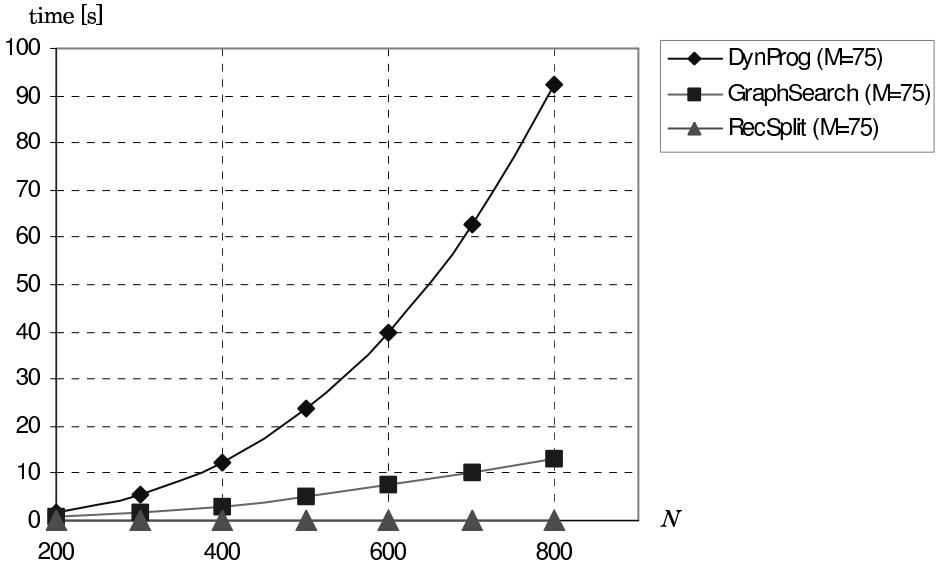


Fig. 35 Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded-# approach, using the maximum of the Euclidean distances as error assessment.

In figure 36 are given the polygonal approximation errors obtained for different algorithms. The Dynamic Programming and the Graph Search methods give for each case the minimum values of the approximation error that is the optimum solution at the bounded-# problem. On the contrary, the Recursive Split algorithm finds worse solutions, implying errors greater than the minimum possible values.

More details regarding the experimental data and the measuring conditions are given in annex, table 13, page 64.

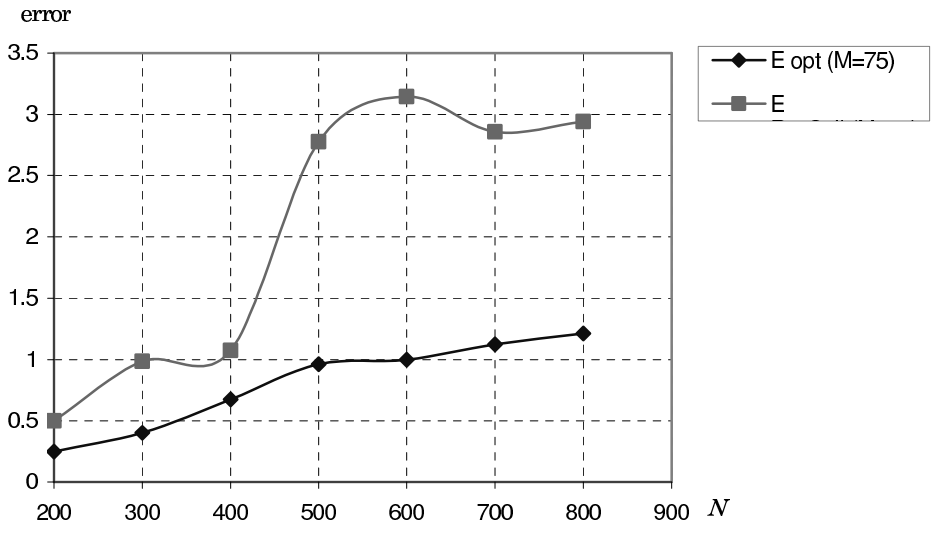


Fig. 36 Variation of the approximation error; comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithm in bounded-# approach, using the maximum of the Euclidean distances as error assessment.

Annex

All the experimental measurements presented in this study were obtained using a DELL computer with the following features:

- Intel Pentium III, 1GHz processor
- 512 kB RAM
- 20 GB HDD
- Windows 2000 operating system

For each table from this annex is given the conditions used to obtained the experimental results that are included in it. The experimental results presented here are obtained processing the input curve stored in file "kk1.dat" (fig. 21, page 37), but during the testing stage, similar results and conclusions were obtained using different input curves.

Table 1 Experimental measurements of running time obtained for the old and the new implementations of Dynamic Programming method, bounded-ε (min-#) approach

Experimental conditions: Algorithm: Dynamic Programming, bounded-ε (min-#) approach
Input curve: file "kk1.dat"
Distance: Maximum Euclidean Distance, measured to the line support
Input number of points N=560

M / Algorithm	4	8	16	32	44	64	96	128
DynProg-E-OLD	36.392	107.945	243.693	418.452	507.367	596.411	642.129	664.859
DynProg-E-NEW	22.262	23.816	22.331	23.674	24.121	25.147	24.436	23.169

Table 2 The influence of the output number of points *M* in running time of new implementation of the Dynamic Programming method, bounded-ε approach.

Experimental conditions: Algorithm: Dynamic Programming, bounded-ε (min-#) approach
Input curve: file "kk1.dat"
Distance: Maximum Euclidean Distance, measured to the line support
Input number of points N=800

M	Error	time [s]		
		Distance	Search	Total
5	469.241129	91.111	0.171	91.282
10	35.945359	91.161	0.441	91.602
20	8.513787	91.121	0.982	92.103
30	3.723077	90.921	1.622	92.543
40	1.744813	91.101	2.534	93.635
50	1.212871	90.941	3.645	94.586
60	1.000000	91.234	4.836	96.07
70	0.941176	91.061	5.859	96.92
80	0.844828	91.041	6.941	97.982
90	0.752941	91.131	7.702	98.833
100	0.634518	91.982	8.762	100.744
200	0.243243	91.051	18.453	109.504
300	0.200000	91.031	28.256	119.287

Table 3 The influence of the input number of points N in running time of new implementation of the Dynamic Programming method, bounded-ε approach.

Experimental conditions: Algorithm: Dynamic Programming, bounded-ε (min-#) approach
Input curve: file "kk1.dat"
Distance: Maximum Euclidean Distance, measured to the line support
Output number of points M=50

N	time [s]		
	Distance	Search	Total
100	0.191	0.021	0.212
200	1.633	0.081	1.714
300	5.257	0.311	5.568
400	11.937	0.581	12.518
500	22.933	0.982	23.915
600	39.916	1.533	41.449
700	61.388	2.464	63.852
800	91.011	3.655	94.666

Table 4 The running time experimental results, comparison between the new (NEW) and the new-improved (NEW_IMP) implementations of the Dynamic Programming method, bounded-# (min-ε) approach

Experimental conditions: Algorithm: Dynamic Programming, bounded-#(min-ε) approach
Input curve: file "kk1.dat"

M	N	MaxEuclid.Dist_ls			SumEuclid.Dist_ls					
		DynProg Bounded-#			DynProg Bounded-#			DynProg Bounded-# incremental implementation		
		NEW	NEW_IMP	NEW / NEW_IMP	NEW	NEW_IMP	NEW / NEW_IMP	NEW	NEW_IMP	NEW / NEW_IMP
50	200	1.743	1.583	1.1011	1.722	1.753	0.9823	0.181	0.120	1.5083
	300	5.558	5.387	1.0317	5.648	5.048	1.1189	0.571	0.381	1.4987
	400	12.718	12.388	1.0266	12.668	12.448	1.0177	1.062	0.701	1.5150
	500	24.345	23.354	1.0424	24.245	23.424	1.0350	1.723	1.152	1.4957
	600	41.039	39.647	1.0351	41.249	39.857	1.0349	2.594	1.732	1.4977
	700	64.803	62.231	1.0413	64.593	62.411	1.0350	3.906	2.344	1.6664
	800	96.001	91.938	1.0442	95.878	92.222	1.0396	5.648	2.995	1.8858
75	200	1.883	1.682	1.1195	1.722	1.523	1.1307	0.201	0.140	1.4357
	300	6.159	5.328	1.1560	5.718	5.468	1.0457	0.731	0.501	1.4591
	400	13.961	12.368	1.1288	12.918	12.388	1.0428	1.461	0.961	1.5203
	500	26.388	23.684	1.1142	24.936	23.754	1.0498	2.504	1.422	1.7609
	600	45.676	39.927	1.1440	42.589	40.338	1.0558	4.136	1.993	2.0753
	700	70.741	62.501	1.1318	66.766	62.851	1.0623	6.109	2.754	2.2182
	800	105.26	92.493	1.1380	99.192	92.864	1.0681	8.292	3.716	2.2314
100	200	1.832	1.672	1.0957	1.773	1.542	1.1498	0.231	0.150	1.5400
	300	5.668	5.448	1.0404	5.678	5.318	1.0677	0.731	0.541	1.3512
	400	13.761	12.578	1.0941	13.711	12.668	1.0823	2.013	0.951	2.1167
	500	26.187	23.754	1.1024	26.067	23.674	1.1011	3.575	1.582	2.2598
	600	44.183	40.288	1.0967	44.013	40.368	1.0903	5.819	2.294	2.5366
	700	72.938	62.881	1.1599	68.759	63.351	1.0854	8.091	3.195	2.5324
	800	106.91	92.994	1.1497	100.985	93.385	1.0814	10.785	4.266	2.5281

Table 5 The running time experimental results, comparison between the new (NEW) and the new-improved (NEW_IMP) implementations of the Dynamic Programming method, bounded-ε (min-#) approach

Experimental conditions: Algorithm: Dynamic Programming, bounded-ε (min-#) approach
Input curve: file "kk1.dat"

N	MaxEuclidDist_Is				SumEuclid.Dist_Is						
	Error	DynProg Bounded - ε			Error	DynProg Bounded - ε			DynProg Bounded - ε incremental implementation		
		NEW	NEW_IMP	NEW / NEW_IMP		NEW	NEW_IMP	NEW / NEW_IMP	NEW	NEW_IMP	NEW / NEW_IMP
100	0.1	0.211	0.210	1.0048	1	0.230	0.200	1.1500	0.050	0.031	1.6129
200		1.783	1.522	1.1715		2.003	1.632	1.2273	0.281	0.110	2.5545
300		5.918	5.289	1.1189		7.250	5.278	1.3736	1.122	0.290	3.8690
400		13.759	12.178	1.1298		15.803	12.107	1.3053	2.433	0.691	3.5210
500		26.489	23.464	1.1289		31.145	23.454	1.3279	5.057	1.231	4.1080
600		45.245	39.987	1.1315		53.984	40.138	1.3450	8.624	2.043	4.2212
700		71.137	62.951	1.1300		87.876	62.991	1.3951	13.289	3.065	4.3357
800		105.251	93.168	1.1297		126.222	97.631	1.2928	18.927	4.347	4.3540
100	0.5	0.211	0.211	1.0000	10	0.211	0.201	1.0498	0.040	0.020	2.0000
200		1.692	1.623	1.0425		1.703	1.632	1.0435	0.270	0.110	2.4545
300		6.621	5.327	1.2429		5.868	5.297	1.1078	0.831	0.361	2.3019
400		12.748	12.057	1.0573		13.561	12.428	1.0912	1.973	0.851	2.3184
500		24.665	23.284	1.0593		26.258	23.974	1.0953	3.706	1.502	2.4674
600		42.144	39.497	1.0670		44.805	40.619	1.1031	6.269	2.393	2.6197
700		66.075	62.309	1.0604		70.261	64.403	1.0910	9.884	3.595	2.7494
800		97.775	92.162	1.0609		105.47	95.307	1.1067	14.071	4.867	2.8911
100	2.5	0.291	0.251	1.1594	100	0.200	0.200	1.0000	0.030	0.030	1.0000
200		1.662	1.592	1.0440		1.642	1.592	1.0314	0.110	0.100	1.1000
300		5.358	5.258	1.0190		5.358	5.037	1.0637	0.401	0.230	1.7435
400		12.228	11.938	1.0243		12.588	12.067	1.0432	0.781	0.481	1.6237
500		23.824	23.203	1.0268		24.235	23.444	1.0337	1.502	1.012	1.4842
600		40.338	39.177	1.0296		41.299	40.088	1.0302	2.754	1.652	1.6671
700		63.542	61.719	1.0295		65.224	62.921	1.0366	4.506	2.414	1.8666
800		94.096	91.211	1.0316		96.779	95.649	1.0118	6.560	3.465	1.8932

Table 6 The running time experimental results, comparison between the generic and the dedicated implementations of the new-improved Dynamic Programming method, bounded-# (min-ε) approach,

Experimental conditions: Algorithm: Dynamic Programming, bounded-# (min-ε) approach
Distance: Sum of the Euclidean Distance, measured to the line support, computed using the incremental technique
Input curve: file "kk1.dat"

M	N	ε	Running time [s]		
			Generic	Dedicated	Generic / Dedicated
50	200	12.085813	0.120	0.080	1.5000
	300	29.987154	0.381	0.250	1.5240
	400	50.601252	0.701	0.491	1.4277
	500	84.128794	1.152	0.902	1.2772
	600	110.867001	1.732	1.201	1.4421
	700	140.939479	2.344	1.723	1.3604
	800	166.246569	2.995	2.163	1.3847
75	200	5.599638	0.140	0.090	1.5556
	300	20.198762	0.501	0.380	1.3184
	400	33.451724	0.961	0.681	1.4112
	500	57.598198	1.422	1.081	1.3154
	600	78.229302	1.993	1.552	1.2841
	700	101.005928	2.754	2.003	1.3749
	800	118.372874	3.716	2.704	1.3743
100	200	0.769231	0.150	0.090	1.6667
	300	12.584223	0.541	0.401	1.3491
	400	23.468487	0.951	0.791	1.2023
	500	41.522231	1.582	1.182	1.3384
	600	58.055911	2.294	1.752	1.3094
	700	77.919991	3.195	2.463	1.2972
	800	91.845391	4.266	3.204	1.3315

Table 7 The running time experimental results, comparison between the exact computation version and the trigonometric based implementations of the Euclidean Distances assessment algorithm

Experimental conditions: Algorithm: Dynamic Programming, bounded-# (min-ε) approach
Input curve: file "kk1.dat"
Distance: Maximum Euclidean Distance, measured to the line support (ls) and measured to the segment (seg)
Input number of points N=560

Algorithm \ M		8	16	32	64	128	256	512
Exact Computation	MaxEuclid_ls	36.407	36.814	37.262	37.931	39.034	40.529	41.832
	MaxEuclid_seg	56.975	57.921	59.532	62.082	63.823	64.623	65.981
Inexact Computation	MaxEuclid_ls	16.679	16.757	17.169	17.560	17.967	18.192	18.793
	MaxEuclid_seg	28.451	28.751	29.353	29.562	29.825	30.434	30.995

Table 8 The running time experimental results measured for implementations of the Dynamic Programming method using different kind of kernels and base data types

Experimental conditions: Algorithm: Dynamic Programming, bounded-# (min-ε) approach
Input curve: file "kk1.dat"
Distance: Sum of the Euclidean Distances
Output number of points M=64

N	M	E	Homogeneous				Cartesian	
			leda_integer		GmpZ		double	leda_real
			normal	rescale	normal	rescale		
64	64	0.000000	2.964	4.998	17.706	31.535	0.061	2.033
128	64	1.463415	27.039	43.302	136.919	272.765	0.541	19.047
256	64	16.365380	313.956	355.251	1158.004	2391.460	3.756	141.979
350	64	30.748217	922.290	949.887	3429.995	5563.649	9.524	605.844
450	64	52.350310	2285.697	2204.664	9657.349	10210.25	19.708	1543.468
560	64	80.657475	5995.215	3864.191			69.841	2986.347

Table 9 The running time experimental results; comparison between different kind of distances measured for 2-dimesional and 3-dimensional implementations

Experimental conditions: Algorithm: Dynamic Programming, bounded-# (min-ε) approach
Input curve: file "kk1.dat"
Input number of points $N = 640$

Distance \ M	8	16	32	64	128	256	512
MaxEuclid_ls 2D	36.407	36.814	37.262	37.931	39.034	40.529	41.832
MaxLinfinity_ls 2D	38.709	40.102	40.829	41.737	42.821	44.329	46.257
MaxManhattan_ls 2D	37.429	38.686	39.524	40.641	42.071	43.824	45.693
MaxEuclid_ls 3D	105.6185	106.7992	108.0989	110.0397	113.2395	117.5766	121.3566
MaxLinfinity_ls 3D	108.6593	112.5695	114.6103	117.1591	120.202	124.4351	129.8471
MaxManhattan_ls 3D	108.7654	112.4181	114.8533	118.0992	122.2546	127.3487	132.7799

Table 10 The running time experimental results; comparison between different kind of polygonal approximation algorithms for the bounded-ε approach, using the sum of the Euclidean distances as error assessment

Experimental conditions: Algorithm: bounded-ε approach
Distance: Sum of the Euclidean Distances
Input curve: file "kk1.dat"

Err	N	M_{opt}	DynProg		DynProg / incr		Graph Search			RecSplit		M_{RS}-M_{opt}
							optimal	sub-optimal				
			NEW	NEW_IMP	NEW	NEW_IMP		time	M_{gss}	time [s]	M_{RS}	
1	100	63	0.230	0.200	0.050	0.031	0.020	0.020	63	0.000	77	14
	200	100	2.003	1.632	0.281	0.110	0.090	0.070	100	0.000	133	33
	300	159	7.250	5.278	1.122	0.290	0.200	0.181	159	0.000	213	54
	400	195	15.803	12.107	2.433	0.691	0.451	0.300	195	0.000	275	80
	500	243	31.145	23.454	5.057	1.231	1.001	0.461	243	0.010	339	96
	600	290	53.984	40.138	8.624	2.043	1.843	0.751	290	0.010	421	131
	700	345	87.876	62.991	13.289	3.065	2.694	1.071	345	0.010	491	146
	800	367	126.222	97.631	18.927	4.347	3.776	1.392	367	0.020	521	154
10	100	25	0.211	0.201	0.040	0.020	0.010	0.010	38	0.000	53	28
	200	57	1.703	1.632	0.270	0.110	0.070	0.060	75	0.000	108	51
	300	111	5.868	5.297	0.831	0.361	0.281	0.180	134	0.000	182	71
	400	147	13.561	12.428	1.973	0.851	0.621	0.250	170	0.010	244	97
	500	195	26.258	23.974	3.706	1.502	1.292	0.490	218	0.010	307	112
	600	242	44.805	40.619	6.269	2.393	1.932	0.661	265	0.010	382	140
	700	295	70.261	64.403	9.884	3.595	3.054	0.881	320	0.020	449	154
	800	317	105.47	95.307	14.071	4.867	4.216	1.081	342	0.020	475	158
100	100	4	0.200	0.200	0.030	0.030	0.010	0.010	4	0.000	5	1
	200	9	1.642	1.592	0.110	0.100	0.060	0.060	9	0.000	14	5
	300	12	5.358	5.037	0.401	0.230	0.160	0.140	24	0.000	27	15
	400	21	12.588	12.067	0.781	0.481	0.430	0.250	60	0.000	52	31
	500	41	24.235	23.444	1.502	1.012	0.681	0.481	108	0.010	100	59
	600	58	41.299	40.088	2.754	1.652	1.161	0.681	155	0.010	168	110
	700	76	65.224	62.921	4.506	2.414	1.923	0.851	210	0.010	233	157
	800	91	96.779	95.649	6.560	3.465	2.704	1.051	232	0.010	259	168

Table 11 The running time experimental results; comparison between different kind of polygonal approximation algorithms for the bounded-ε approach, using the maximum of the Euclidean distances as error assessment

Experimental conditions: Algorithm: bounded-ε approach
Distance: Maximum of the Euclidean Distances
Input curve: file "kk1.dat"

Err	N	M_{opt}	DynProg		Graph Search	RecSplit		M_{RS}-M_{opt}
			NEW	NEW_IMP		time [s]	M_{RS}	
0.1	100	67	0.211	0.210	0.140	0.000	67	0
	200	104	1.783	1.522	0.701	0.010	104	0
	300	163	5.918	5.289	1.492	0.000	163	0
	400	199	13.759	12.178	2.865	0.000	199	0
	500	247	26.489	23.464	4.577	0.010	247	0
	600	294	45.245	39.987	6.399	0.010	294	0
	700	349	71.137	62.951	8.762	0.010	349	0
	800	371	105.251	93.168	11.487	0.010	371	0
0.5	100	21	0.211	0.211	0.140	0.010	24	3
	200	31	1.692	1.623	0.721	0.000	48	17
	300	39	6.621	5.327	1.622	0.000	68	29
	400	56	12.748	12.057	2.844	0.000	92	36
	500	76	24.665	23.284	4.596	0.000	131	55
	600	89	42.144	39.497	6.541	0.000	151	62
	700	100	66.075	62.309	8.693	0.010	175	75
	800	110	97.775	92.162	11.196	0.010	195	85
2.5	100	5	0.291	0.251	0.161	0.000	11	6
	200	10	1.662	1.592	0.701	0.000	17	7
	300	12	5.358	5.258	1.562	0.000	21	9
	400	18	12.228	11.938	2.864	0.000	32	14
	500	24	23.824	23.203	4.597	0.000	40	16
	600	29	40.338	39.177	6.681	0.000	47	18
	700	32	63.542	61.719	8.963	0.000	55	23
	800	34	94.096	91.211	11.597	0.000	56	22

Table 12 The running time experimental results; comparison between different kind of polygonal approximation algorithms for the bounded-# approach, using the sum of the Euclidean distances as error assessment

Experimental conditions: Algorithm: bounded-# approach
Distance: Sum of the Euclidean Distances
Input curve: file "kk1.dat"

M	N	E_{opt}	DynProg		DynProg / incr.		GrSearch	RecSplit		E_{RS}-E_{opt}
			NEW	NEW_IMP	NEW	NEW_IMP		time [s]	E_{RS}	
50	200	12.085813	1.722	1.753	0.181	0.120	0.080	0.000	35.582594	23.496781
	300	29.987154	5.648	5.048	0.571	0.381	0.250	0.010	67.550383	37.563229
	400	50.601252	12.668	12.448	1.062	0.701	0.491	0.000	103.314731	52.713479
	500	84.128794	24.245	23.424	1.723	1.152	0.902	0.010	171.431212	87.302418
	600	110.867001	41.249	39.857	2.594	1.732	1.201	0.010	233.497895	122.630894
	700	140.939479	64.593	62.411	3.906	2.344	1.723	0.000	332.158347	191.218868
	800	166.246569	95.878	92.222	5.648	2.995	2.163	0.010	357.689944	191.443375
75	200	5.599638	1.722	1.523	0.201	0.140	0.090	0.000	23.069231	17.469593
	300	20.198762	5.718	5.468	0.731	0.501	0.380	0.000	52.170747	31.971985
	400	33.451724	12.918	12.388	1.461	0.961	0.681	0.010	87.711453	54.259729
	500	57.598198	24.936	23.754	2.504	1.422	1.081	0.000	119.733025	62.134827
	600	78.229302	42.589	40.338	4.136	1.993	1.552	0.010	172.582103	94.352801
	700	101.005928	66.766	62.851	6.109	2.754	2.003	0.010	219.075711	118.069783
	800	118.372874	99.192	92.864	8.292	3.716	2.704	0.010	240.487605	122.114731
100	200	0.769231	1.773	1.542	0.231	0.150	0.090	0.000	13.700000	12.930769
	300	12.584223	5.678	5.318	0.731	0.541	0.401	0.000	41.757228	29.173005
	400	23.468487	13.711	12.668	2.013	0.951	0.791	0.010	66.930614	43.462127
	500	41.522231	26.067	23.674	3.575	1.582	1.182	0.010	99.466407	57.944176
	600	58.055911	44.013	40.368	5.819	2.294	1.752	0.000	146.738896	88.682985
	700	77.919991	68.759	63.351	8.091	3.195	2.463	0.010	167.307571	89.387580
	800	91.845391	100.985	93.385	10.785	4.266	3.204	0.010	188.124826	96.279435

Table 13 The running time experimental results; comparison between different kind of polygonal approximation algorithms for the bounded-# approach, using the maximum of the Euclidean distances as error assessment

Experimental conditions: Algorithm: bounded-# approach
Distance: Maximum of the Euclidean Distances
Input curve: file "kk1.dat"

M	N	E_{opt}	DynProg		Graph Search		RecSplit		$E_{RS}-E_{opt}$
			NEW	NEW_IMP	time [s]	M_{GS}	time [s]	E_{RS}	
50	200	0.248731	1.743	1.583	0.681	49	0.000	0.500000	0.251269
	300	0.401639	5.558	5.387	1.742	50	0.000	0.985804	0.584165
	400	0.675676	12.718	12.388	3.144	50	0.000	1.074349	0.398673
	500	0.961538	24.345	23.354	5.007	48	0.000	2.776923	1.815385
	600	1.000000	41.039	39.647	7.101	46	0.010	3.144105	2.144105
	700	1.123596	64.803	62.231	9.724	50	0.010	2.859459	1.735863
	800	1.212871	96.001	91.938	12.559	50	0.010	2.941176	1.728305
75	200	0.200000	1.883	1.682	0.771	57	0.100	0.500000	0.300000
	300	0.247525	6.159	5.328	1.712	75	0.000	0.500000	0.252475
	400	0.400000	13.961	12.368	3.164	73	0.010	0.720000	0.320000
	500	0.527835	26.388	23.684	5.218	75	0.010	1.000000	0.472165
	600	0.752941	45.676	39.927	7.561	75	0.010	1.207547	0.454606
	700	0.844828	70.741	62.501	10.014	75	0.000	3.420000	2.575172
	800	0.900000	105.261	92.493	13.139	73	0.010	1.633484	0.733484
100	200	0.200000	1.832	1.672	0.661	57	0.000	0.200000	0.000000
	300	0.200000	5.668	5.448	1.742	92	0.000	0.500000	0.300000
	400	0.248276	13.761	12.578	3.185	100	0.000	0.800000	0.551724
	500	0.400000	26.187	23.754	5.038	97	0.010	0.787692	0.387692
	600	0.500000	44.183	40.288	7.201	89	0.000	0.900000	0.400000
	700	0.500000	72.938	62.881	10.264	100	0.000	2.769231	2.269231
	800	0.634518	106.914	92.994	13.119	100	0.010	1.000000	0.365482

References

[1] J. Hershberger, J. Snoeyink, "Speeding Up the Douglas–Peucker Line–Simplification Algorithm", Poc. 5–th Int. Symp. Spatial Data Handding, IGU Commision on GIS, Charleston, South Carolina, 1992, pp. 134–143;

[2] R. Veltkamp, "Generic Geometric Programming in the Computational Geometry Algorithms Library", Computer Graphics Forum, vol.18 (1999), no.2, pp.131–137;

[3] R. Veltkamp, "Hierarchical approximation and localization", The Visual Computer, no.14, 1998, pp. 471–487;

[4] J.C. Perez, E. Vidal, "Optimum polygonal approximation of digitized curves", Pattern Recognition Letters, no.15, 1994, pp. 743–750;

[5] H. Imai, M. Iri, "Polygonal Approximation of a Curve – Formulation and Algorithms", Computational Morphology, (ed. G.T. Toussaint), pp. 71–86;

[6] D.H. Douglas, T.K. Peucker, "Algorithm for the reduction of number of points required to represent a line or its caricature", The Canadian Cartographer, 10 (2) , 1973, pp.112–122

[7] G. T. Toussaint, "On the Complexity of Approximating Polygonal–Curves in the Plane", Proceeding IASTED, International Symposium on Robotics and Automation, Lugano, Switzerland, 1985.

[8] J. Gregor, M. G. Thomason, "Dynamic Programming Alignment of Sequences representing cyclic patterns", IEEE Trans. Pattern Anal. Machine Intell. 15 (2), 1993, pp. 129–135.

[9] D. Avis, H. ElGindy, R. Seidel, "Simple On-Line Algorithm for Convex Polygons", Computational Geometry (G. T. Toussaint, ed.), North-Holland, Amsterdam, 1985, pp. 23-42.

[10] M.T.Goodrich, R. Tamassia, Data Structure and Algorithms in Java, John Wiley & Sons Ed., NY, 2001