Polyline Simplification

Elmar de Koning

25 Jun 2011 MPL

A generic C++ implementation for n-dimensional Douglas-Peucker approximation

Download source - 290.7 KB

Download demo - 5.23 MB

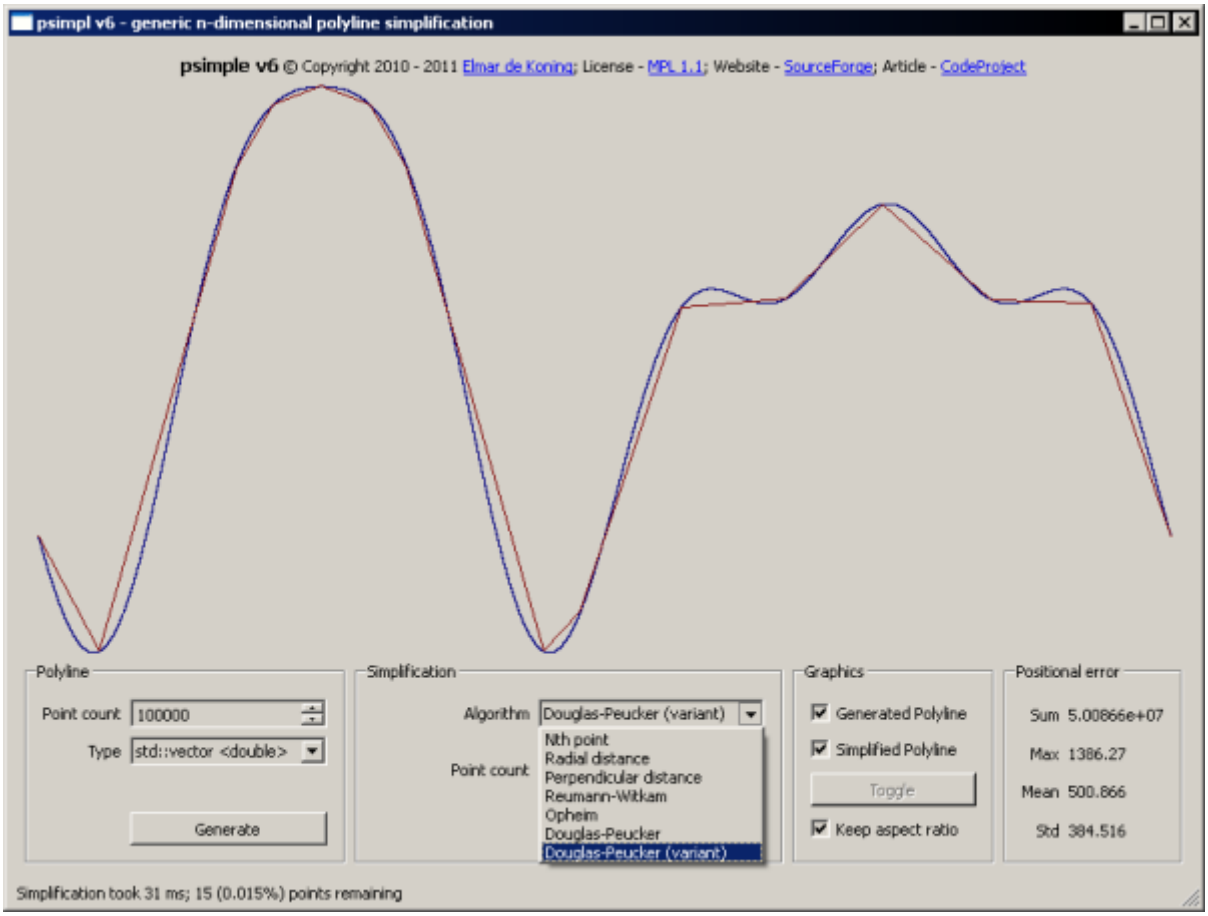


Table of Contents

Introduction

Polyline simplification is the process of reducing the resolution of a polyline. This is achieved by removing vertices and edges, while maintaining a good approximation of the original curve. One area of application for polyline simplification is in computer graphics. When the polyline resolution is higher than that of the display, multiple vertices and edges from that polyline will most likely be mapped onto a single pixel. Meaning, you are spending resources to draw something that will not be visible. This waste of resources is easily avoidable, by reducing the resolution of the polyline before drawing it.

This article presents **psimpl**, a polyline simplification library that is generic, easy to use, and supports the following algorithms:

> *Simplification algorithms*

- *N^{th} point* - A naive algorithm that keeps only each n^{th} point
- *Distance between points* - Removes successive points that are clustered together
- *Perpendicular distance* - Removes points based on their distance to the line segment defined by their left and right neighbors
- *Reumann-Witkam* - Shifts a strip along the polyline and removes points that fall outside
- *Opheim* - Similar to *Reumann-Witkam*, but constrains the search area using a minimum and maximum tolerance
- *Lang* - Similar to the *Perpendicular distance* routine, but instead of looking only at direct neighbors, an entire search region is processed
- *Douglas-Peucker* - A classic simplification algorithm that provides an excellent approximation of the original line

- A variation on the *Douglas-Peucker* algorithm - Slower, but yields better results at lower resolutions

  Error algorithms

- *Positional errors* - Distance of each point from an original polyline to its simplification

psimpl is a lightweight header-only C++ library. All the algorithms have been implemented using templates, and provide an STL-style interface that operates on input and output iterators. Polylines can be of any dimension, and defined using floating point or signed integer data types.

For more information about **psimpl**, including news and latest releases, see http://psimpl.sf.net.

If you decide to use my code for your (commercial) project, let me know! I would love to hear where my code ends up and why you chose to use it!

Similar Articles

- '*Polyline Simplification*' by Dan Sunday at softSurfer

  Forms the basis of my work. He clearly explains the principles behind *Vertex Reduction* and *Douglas-Peucker Approximation*, and provides a C++ implementation. However, his implementation is based on floats, and limited to 2D-polylines defined as arrays of `Point` objects. He also uses recursion, which can lead to stack-overflow problems.

- '*A C++ implementation of Douglas-Peucker Line Approximation Algorithm*' by Jonathan de Halleux at CodeProject

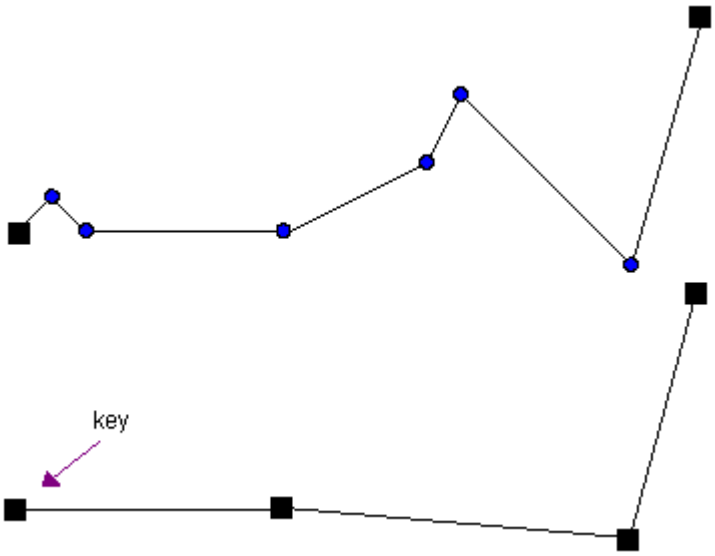
  Presents an optimized O(n log n) implementation of the *Douglas-Peucker Approximation* algorithm. Internally, a 2D-convex hull algorithm is used to achieve better performance. As a consequence, only 2D-polylines are supported. The interface itself, while flexible, is overly complex with its points, point containers, key containers, and hull templates.

- '*A C# Implementation of Douglas-Peucker Line Approximation Algorithm*' by CraigSelbert at CodeProject

  A simple port to C# of the C++ implementation of Jonathan de Halleux.

N^{th} Point

The *N^{th} point* routine is a naive O(n) algorithm polyline simplification. It keeps only the *first*, *last*, and each *n^{th}* point. All other points are removed. This process is illustrated below:



The illustration shows a polyline consisting of 8 vertices: *{v1, v2 ... v8}*. This polyline was simplified using `n` = 3. The resulting simplification consists of vertices: *{v1, v4, v7, v8}*.

The algorithm is extremely fast, but unfortunately, it not very good at preserving the geometric features of a line.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_nth_point (
    ForwardIterator first,
    ForwardIterator last,
    unsigned n,
    OutputIterator result)
```

Applies the n^{th} *point* routine to the range [`first`, `last`) using the specified value for *n*. The resulting simplified polyline is copied to the output range [`result`, `result` + *m*`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m*`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `n` is not 0

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

Algorithms don't get much simpler than this. A loop is used to copy the *first* point and each following n^{th} point of the input polyline to the simplification result. After the loop, I make sure that the *last* point is part of the simplification.

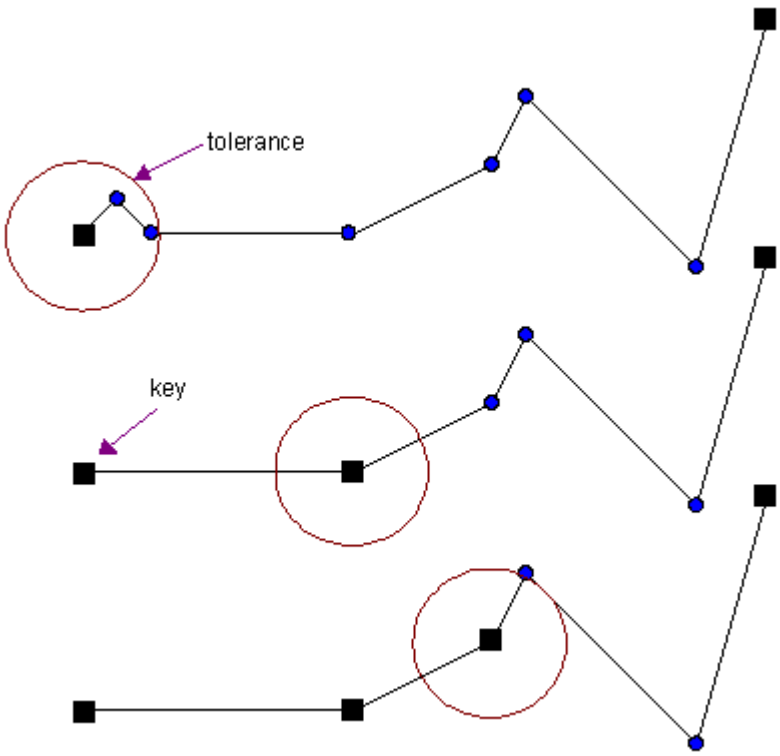
Usage

C++

```
unsigned n = 10;                  // reduce to 10%
std::vector <float> polyline;     // original polyline, assume not empty
std::vector <float> result;       // resulting simplified polyline

// simplify the 2d polyline
psimpl::simplify_nth_point <2> (polyline.begin (), polyline.end (),
                                n, std::back_inserter (result));
```

Radial Distance

Distance between points is a brute force O(n) algorithm for polyline simplification. It reduces successive vertices that are clustered too closely to a single vertex, called a key. The resulting keys form the simplified polyline. This process is illustrated below:



The first and last vertices are always part of the simplification, and are thus marked as keys. Starting at the first key (the first vertex), the algorithm walks along the polyline. All consecutive vertices that fall within a specified distance tolerance from that key are removed. The first encountered vertex that lies further away than the tolerance is marked as a key. Starting from this new key, the

algorithm will start walking again and repeat this process, until it reaches the final key (the last vertex).

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_radial_distance (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type tol,
    OutputIterator result)
```

Applies the *Distance between points* routine to the range [`first`, `last`) using the specified radial distance tolerance `tol`. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `tol` is not 0

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

Nothing special, just a single loop over all vertices that calculates point-point distances. As soon as a key is found, it is copied to the output range.

Usage

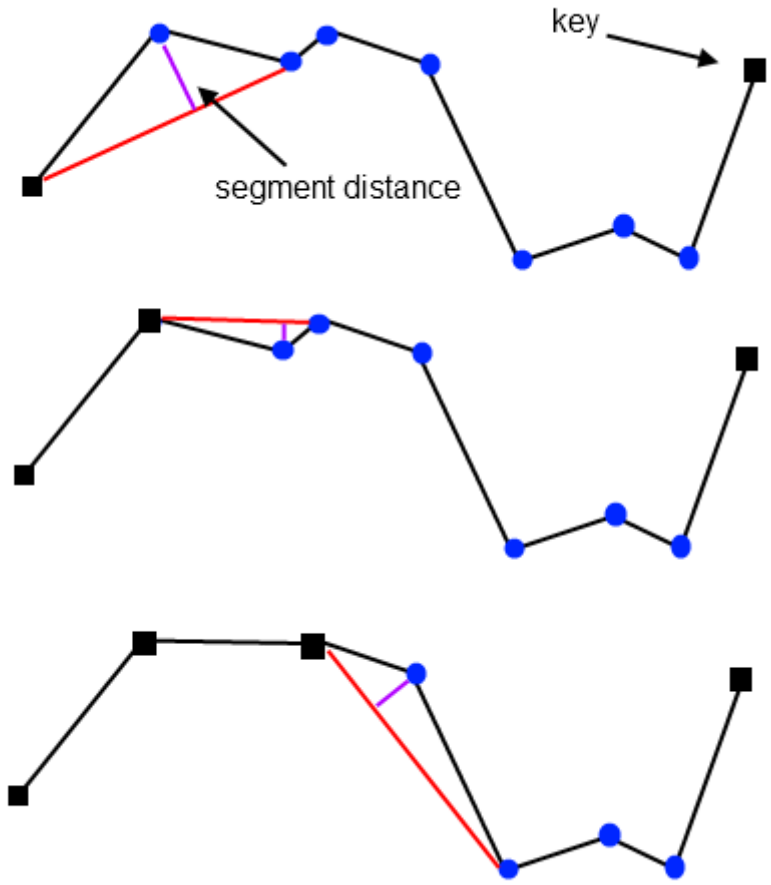
C++

```
float tolerance = 10.f;          // point-to-point distance tolerance
std::vector <float> polyline;    // original polyline, assume not empty
std::vector <float> result;      // resulting simplified polyline

// simplify the 2d polyline
psimpl::simplify_radial_distance <2> (polyline.begin (), polyline.end (),
                                      tolerance, std::back_inserter (result));
```

Note that the results container does not need to match the polyline container. You could, for instance, use a C-style `double` array.

Perpendicular Distance

Instead of using a point-to-point (radial) distance tolerance as a rejection criterion (see *Distance between points*), the O(n) *Perpendicular distance* routine uses a point-to-segment distance tolerance. For each vertex *vi*, its perpendicular distance to the line segment *S(vi-1, vi+1)* is computed. All vertices whose distance is smaller than the given tolerance will be removed. This process is illustrated below:



Initially, the first three vertices are processed, and the perpendicular distance of the second vertex is calculated. After comparing this distance against the tolerance, the second vertex is considered to be a key (part of the simplification). The algorithm then moves one vertex up the polyline and begins processing the next set of three vertices. This time, the calculated distance falls below the tolerance and thus the intermediate vertex is removed. The algorithm continues by moving two vertices up the polyline.

Note that for each vertex *vi* that is removed, the next possible candidate for removal is *vi+2*. This means that the original polyline can only be reduced by a maximum of 50%. Multiple passes are required to achieve higher vertex reduction rates.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_perpendicular_distance (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type tol,
    OutputIterator result)

template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_perpendicular_distance (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type tol,
    unsigned repeat,
    OutputIterator result)
```

Applies the *Perpendicular distance* routine (`repeat` times) to the range [`first`, `last`) using the specified perpendicular distance tolerance `tol`. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `tol` is not 0
6. `n` is not 0

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

The main function, without the `repeat` parameter, is a single loop that starts with processing the first three consecutive vertices. Depending on whether the second or third vertex is considered to be part of the simplification (called a key), the algorithm moves one or two vertices up the original polyline. As soon as a key is found, it is copied to the output iterator.

The second function, which takes a `repeat` value as input, is a wrapper around the main function, and consists of three distinct steps:

1. First iteration: Simplify from range [`first`, `last`) to a plain C-style array.
2. Intermediate iterations: Simplify from and to plain C-style arrays.
3. Last iteration: Simplify from a plain C-style array to the output iterator `result`.

After each iteration, the simplification is checked for improvement. If none was found, the current result is copied directly to the output iterator `result`. Note that up to two temporary copies may

be created: one copy of the input range, and the other of the first intermediate simplification result.

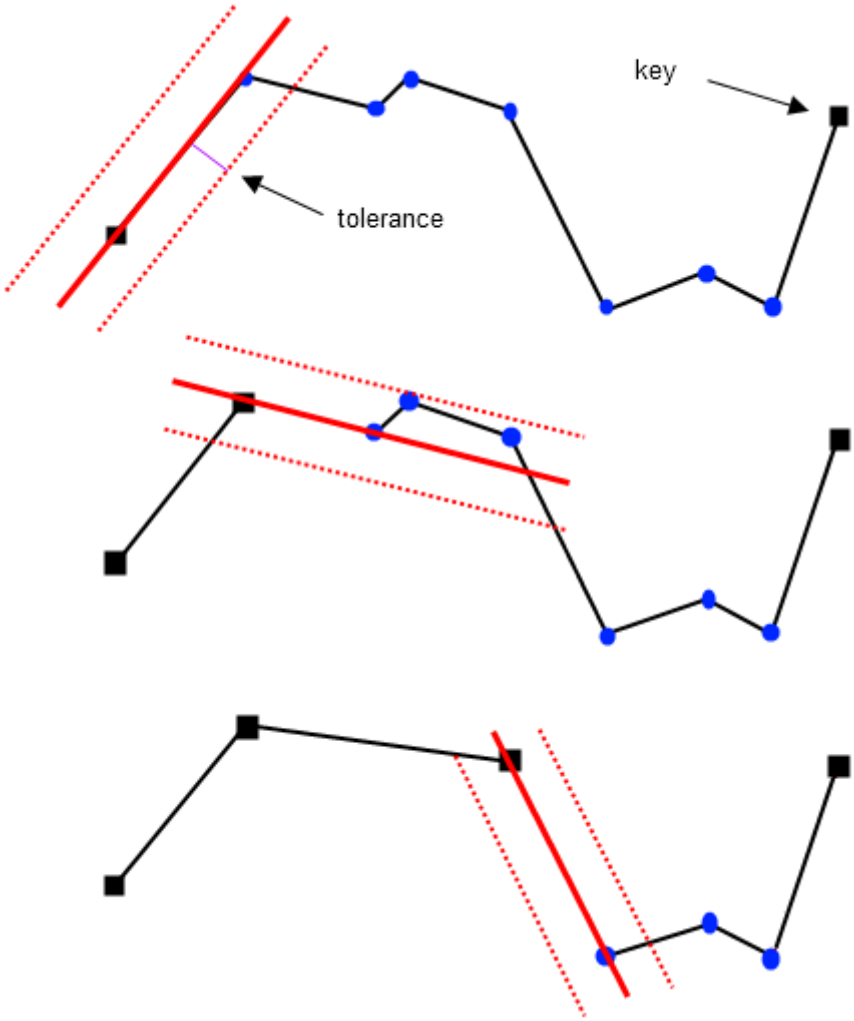
Usage

C++

```
double tolerance = 10.0;          // point-to-segment distance tolerance
std::deque <double> polyline;     // original polyline, assume not empty
std::deque <double> result;       // resulting simplified polyline

// simplify the 3d polyline - single pass
psimpl::simplify_perpendicular_distance <3> (polyline.begin (), polyline.end (),
                                             tolerance, std::back_inserter (result));

double tolerance = 10.0;          // point-to-segment distance tolerance
usinged repeat = 5;               // apply the routine 5 times
std::deque <double> polyline;     // original polyline, assume not empty
std::deque <double> result;       // resulting simplified polyline

// simplify the 3d polyline - multi pass
psimpl::simplify_perpendicular_distance <3> (polyline.begin (), polyline.end (),
tolerance,
                                             repeat, std::back_inserter (result));
```

Reumann-Witkam

Instead of using a point-to-point (radial) distance tolerance as a rejection criterion (see *Distance between points*), the O(n) *Reumann-Witkam* routine uses a point-to-line (perpendicular) distance tolerance. It defines a line through the first two vertices of the original polyline. For each successive vertex v_i, its perpendicular distance to this line is calculated. A new key is found at v_{i-1}, when this distance exceeds the specified tolerance. The vertices v_i and v_{i+1} are then used to define a new line, and the process repeats itself. The algorithm is illustrated below:



The red strip is constructed from the specified tolerance and a line through the first two vertices of the polyline. The third vertex does not lie within the strip, meaning the second vertex is a key. A new strip is defined using a line through the second and third vertices. The last vertex that is still contained within this strip is considered the next key. All other contained vertices are removed. This process is repeated until a strip is constructed that contains the last vertex of the original polyline.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_reumann_witkam (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type tol,
    OutputIterator result)
```

Applies the *Reumann-Witkam* routine to the range [`first`, `last`) using the specified perpendicular distance tolerance `tol`. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `tol` is not 0

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

Nothing special, just a single loop over all vertices that calculates their distance against the current strip. As soon as a key is found, it is copied to the output range and the current strip is updated.

Usage

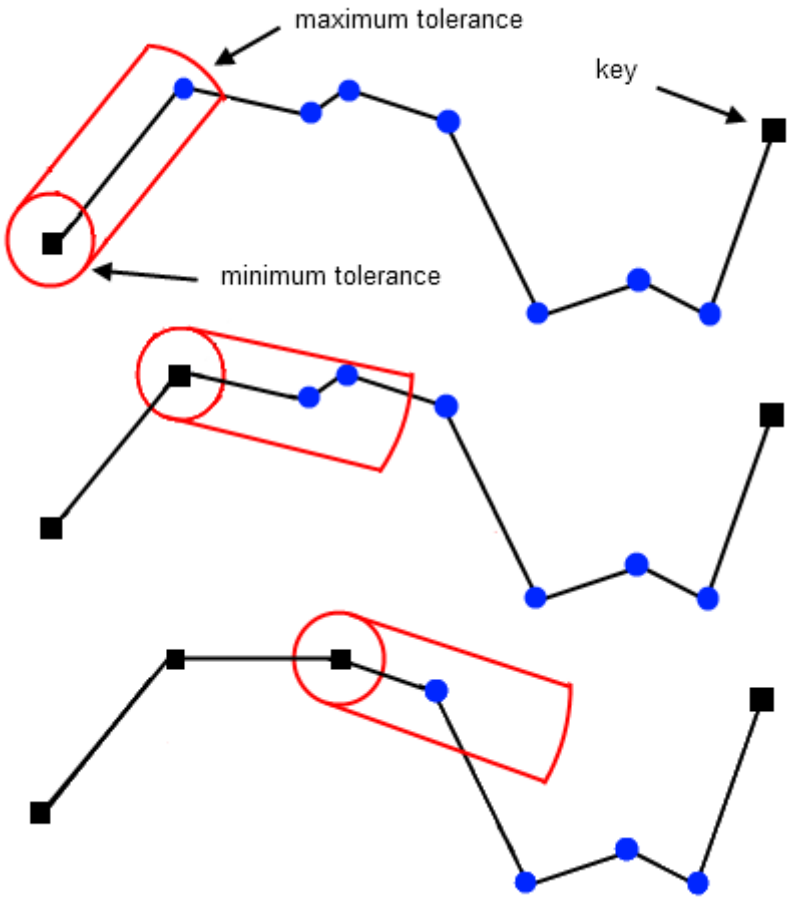
C++

```
float tolerance = 10.f;          // point-to-line perpendicular distance tolerance
std::vector <float> polyline;    // original polyline, assume not empty
std::vector <double> result;     // resulting simplified polyline

// simplify the 4d polyline
psimpl::simplify_reumann_witkam <4> (polyline.begin (), polyline.end (),
                                     tolerance, std::back_inserter (result));
```

This example demonstrates that the value type of the input and output iterators do not have to be the same.

Opheim

The O(n) *Opheim* routine is very similar to the *Reumann-Witkam* routine, and can be seen as a constrained version of that *Reumann-Witkam* routine. *Opheim* uses both a minimum and a maximum distance tolerance to constrain the search area. For each successive vertex *vi*, its radial distance to the current key *vkey* (initially *v0*) is calculated. The last point within the minimum distance tolerance is used to define a ray R (*vkey*, *vi*). If no such *vi* exists, the ray is defined as R(*vkey*, *vkey+1*). For each successive vertex *vj* beyond *vi* its perpendicular distance to the ray R is calculated. A new key is found at *vj-1*, when this distance exceeds the minimum tolerance Or when the radial distance between *vj* and the *vkey* exceeds the maximum tolerance. After a new key is found, the process repeats itself.



The *Opheim* simplification process is illustrated above. Notice how the search area is constrained by a minimum and a maximum tolerance. As a result, during the second step, only a single point is removed. The *Reumann-Witkam* routine, which uses an infinite or unconstrained search area, would have removed two points.

Interface

C++

```
template <unsigned DIM, class InputIterator, class OutputIterator>
OutputIterator simplify_opheim (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type minTol,
    typename std::iterator_traits <ForwardIterator>::value_type maxTol,
    OutputIterator result)
```

Applies the *Opheim* routine to the range [`first`, `last`) using the specified distance tolerances `minTol` and `maxTol`. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`

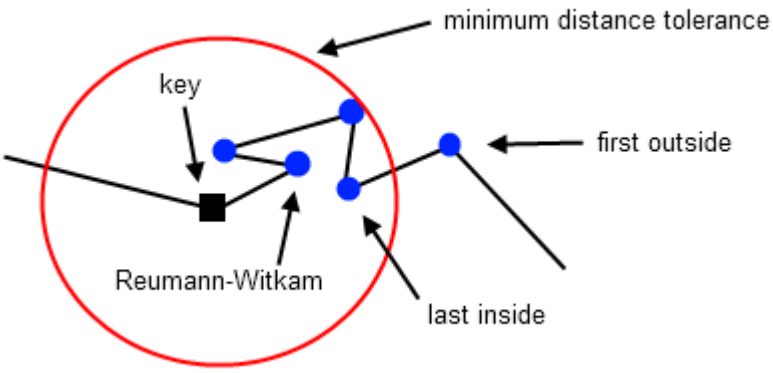
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `minTol` is not 0
6. `maxTol` is not 0

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

All the articles that I found mentioning or discussing the *Opheim* algorithm, failed to explain how to define the ray that controls the direction of the search area. As far as I can tell, there are three possible ways of determining this ray R(*vkey*, *vi*), where *vkey* is the current key.

1. The *Reumann-Witkam* way: *i = key+1*
2. The first point outside: *key < i* and *vi* is the first point that falls outside the minimum radial distance tolerance
3. The last point inside: *key < i* and *vi* is the last point that falls inside the minimum radial distance tolerance; if no such *vi* exists, fall back to the *Reumann-Witkam* way



I compared these three approaches using positional error statistics and found that '*the first point outside*' approach, most of the time, produces slightly better results than the '*Reumann-Witkam*' approach. Furthermore, there did not seem to be any real difference between the '*last point inside*' and '*the first point outside*' approaches. I ended up choosing '*last point inside*' approach, because it was a better fit for the loop that I had already implemented.

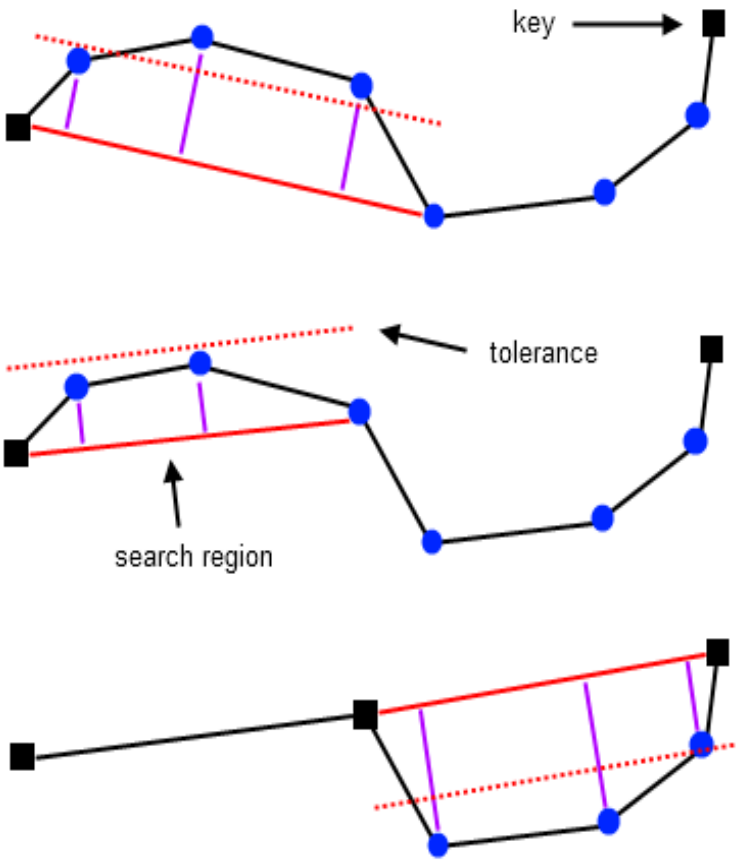
Usage

C++

```
float minimum = 10.f;            // minimum distance tolerance
float maximum = 100.f;           // maximum distance tolerance
std::vector <double> polyline;   // original polyline, assume not empty
std::vector <double> result;     // resulting simplified polyline

// simplify the 4d polyline
psimpl::simplify_opheim <4> (polyline.begin (), polyline.end (),
                             minimum, maximum, std::back_inserter (result));
```

Lang

The *Lang* simplification algorithm defines a fixed size search-region. The first and last points of that search region form a segment. This segment is used to calculate the perpendicular distance to each intermediate point. If any calculated distance is larger than the specified tolerance, the search region will be shrunk by excluding its last point. This process will continue until all calculated distances fall below the specified tolerance, or when there are no more intermediate points. All intermediate points are removed and a new search region is defined starting at the last point from old search region. This process is illustrated below:



The search region is constructed using a `look_ahead` value of 4. For each intermediate vertex, its perpendicular distance to the segment *S (v0, v4)* is calculated. Since at least one distance is greater than the tolerance, the search region is reduced by one vertex. After recalculating the distances to *S (v0, v3)*, all intermediate vertices fall within the tolerance. The last vertex of the search region *v3* defines a new key. This process repeats itself by updating the search region and defining a new segment *S (v3, v7)*.

Interface

C++

```
template <unsigned DIM, class BidirectionalIterator, class OutputIterator>
OutputIterator simplify_lang (
```

```
    BidirectionalIterator first,
    BidirectionalIterator last,
    typename std::iterator_traits <BidirectionalIterator>::value_type tol,
    unsigned look_ahead,
    OutputIterator result)
```

Applies the *Lang* simplification algorithm to the range [`first`, `last`) using the specified perpendicular distance tolerance and look ahead values. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `BidirectionalIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `tol` is not 0.
6. `look_ahead` is not 0.

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

The *Lang* simplification algorithm has the requirement that its input iterators model the concept of a bidirectional iterator. The reason for this is that a search region *S (vi, vi+n)* may have to be reduced to *S (vi, vi+(n-1))*. The easiest way to do this is by decrementing the iterator pointing to *vi+n*. Although it would be possible to just increment a copy of *vi* n-1 times, it requires extra bookkeeping. It also complicates the code somewhat, as we would only want to take this approach for forward iterators.

Usage

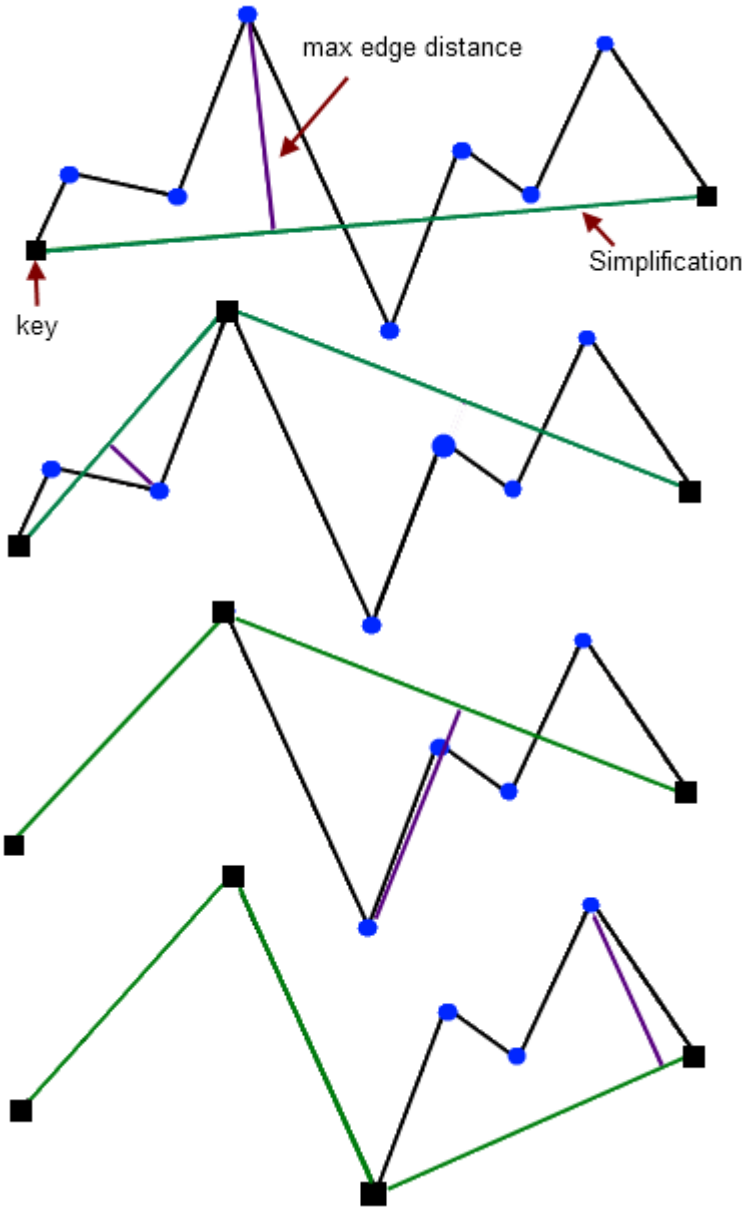
C++

```
float tolerance = 10.f;       // point-to-segment distance tolerance
unsigned look_ahead = 7;      // search region size
std::vector <float> polyline; // original polyline, assume not empty
std::vector <double> result;  // resulting simplified polyline

// simplify the 5d polyline
psimpl::simplify_lang <5> (
    polyline.begin (), polyline.end (),
    tolerance, look_ahead,
    std::back_inserter (result));
```

Using a `look_ahead` value of 7, means that the resulting simplification will always contain at least 1/7 or 14% of the original points. The `look_ahead` value constrains the simplification.

Douglas-Peucker

The *Douglas-Peucker* algorithm uses a point-to-edge distance tolerance. The algorithm starts with a crude simplification that is the single edge joining the first and last vertices of the original polyline. It then computes the distance of all intermediate vertices to that edge. The vertex that is furthest away from that edge, and that has a computed distance that is larger than a specified tolerance, will be marked as a key and added to the simplification. This process will recurse for each edge in the current simplification, until all vertices of the original polyline are within tolerance of the simplification results. This process is illustrated below:



Initially, the simplification consists of a single edge. During the first step, the fourth vertex is marked as a key and the simplification is adjusted accordingly. During the second step, the first edge of the current simplification is processed. The maximum vertex distance to that edge falls

below the tolerance threshold, and no new key is added. During the third step, a key is found for the second edge of the current simplification. This edge is split at the key and the simplification is updated. This process continues until no more keys can be found. Note that at each step, only one edge of the current simplification is processed.

This algorithm has a worst case running time of O(nm), and O(n log m) on average, where m is the size of the simplified polyline. As such, this is an output dependent algorithm, and will be very fast when m is small. To make it even faster, the *Distance between points* routine is applied as a pre-processing step.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_douglas_peucker (
    ForwardIterator first,
    ForwardIterator last,
    typename std::iterator_traits <ForwardIterator>::value_type tol,
    OutputIterator result)
```

Applies the *Distance between points* routine followed by *Douglas-Peucker* approximation to the range [`first`, `last`) using the specified tolerance `tol`. The resulting simplified polyline is copied to the output range [`result`, `result` + *m**`DIM`), where *m* is the number of vertices of the simplified polyline. The return value is the end of the output range: `result` + *m**`DIM`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains at least 2 vertices
5. `tol` is not 0.

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

Initially, my focus was on limiting the memory usage of the algorithms. So instead of using output iterators, all algorithms returned a `std::vector<bool>`. One boolean for each vertex that determined if that vertex is a key and thus part of the simplification. This list of key markers could be used as input for another algorithm, allowing different algorithms to be run in sequence. A separate function could optionally copy all keys to some output range.

This approach worked, but had some serious drawbacks:

- The code was slow, especially for non-random access iterators

- The code had become too complex with all its bookkeeping
- When using the code, I always needed a real copy of the simplification results and not a bunch of key markers

The first thing I changed was the interface of each algorithm. Instead of returning key markers, the simplification results were copied to an output range using output iterators. The second change was to store the intermediate result produced by the *Distance between points* pre-processing step in a plain C-style array. This array is then used during *Douglas-Peucker* approximation. The advantages of this approach far outweigh the increase in memory usage:

- Using the *Distance between points* routine as a pre-processing step became trivial; I only had to create an array and specify an output iterator for it
- Less code - lack of specific code for different iterator categories, and less bookkeeping
- Faster code - working with C-style arrays and value type pointers is generally faster than using iterators, especially when dealing with non-random access iterators
- Cleaner interface

The algorithm itself is a straightforward loop. The initial edge of the simplification is added to a `std::stack`. As long as the stack is not empty, an edge is popped and processed. Its key and key-edge distance are calculated. If the computed distance is larger than the tolerance, the key is added to the simplification. The edge is split and both sub-edges are added to the stack. When a vertex is added to the simplification, it is only marked as being a key. When the algorithm has finished, all marked vertices (keys) are copied to the output range.

Usage

C++

```
double tolerance = 10;                              // point-to-edge distance tolerance
std::deque <double> polyline;                       // original polyline, assume not empty
double* result = new double [polyline.size ()];  // make sure the result array
                                                    // is large enough
// simplify the 3d polyline
psimpl::simplify_douglas_peucker <3> (polyline.begin (), polyline.end (),
                                      tolerance, result);
```

This example demonstrates that input and output containers do not have to be the same.

Douglas-Peucker (Variant)

This algorithm is a variation of the original implementation. Its key differences are:

- A point count tolerance is used instead of a point-to-edge distance tolerance. This allows you to specify the exact number of vertices in the simplified polyline. With the original implementation, you can never be sure how many vertices will remain.
- Instead of processing a single edge at a time (chosen pseudo random), all edges of the current simplified polyline are considered simultaneously. Each of these edges may define a

new key. From all these possible keys, the one with the highest point-to-edge distance is chosen as the new key.

A direct result from always choosing the next key based on all possible keys at any given time, is that the simplification results are of a higher quality. This is most notable when using a very low point-count tolerance. A downside is that we cannot use the *Distance between points* routine as a pre-processing step to speed up the algorithm.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator simplify_douglas_peucker_n (
    ForwardIterator first,
    ForwardIterator last,
    unsigned count,
    OutputIterator result)
```

Applies a variant of the *Douglas-Peucker Approximation* to the range [`first`, `last`). The resulting simplified polyline consists of `count` vertices, and is copied to the output range [`result`, `result` + `count`). The return value is the end of the output range: `result` + `count`.

Input (Type) Requirements

1. `DIM` is not 0, where `DIM` represents the dimension of the polyline
2. The `ForwardIterator` value type is convertible to the value type of the `OutputIterator`
3. The range [`first`, `last`) contains vertex coordinates in multiples of `DIM`, e.g.: x, y, z, x, y, z, x, y, z when `DIM` = 3
4. The range [`first`, `last`) contains a minimum of `count` vertices
5. `count` is at least 2

In case these requirements are not met, the entire input range [`first`, `last`) is copied to the output range [`result`, `result` + (`last` - `first`)) **or** compile errors may occur.

Implementation Details

The implementation for this variant varies only slightly from the original implementation. The main differences being that there is no pre-processing step, and in the way the edges of the current simplification are processed.

For each edge that is added to the current simplification, its key is calculated. This key, alongside the edge and its distance to that edge, are stored in a `std::priority_queue`. This queue ensures that its top element contains the key with the maximum point-edge distance. As long as the simplification does not contain the desired amount of points, the top element from the queue is popped and its key is added to the simplification. The corresponding edge is split, and the two sub-edges are processed and stored in the queue.

For performance reasons, a copy is made of the input polyline in a plain C-style array. Note that for the original implementation, this copy was made automatically during the *Distance between points* pre-processing step.

Usage

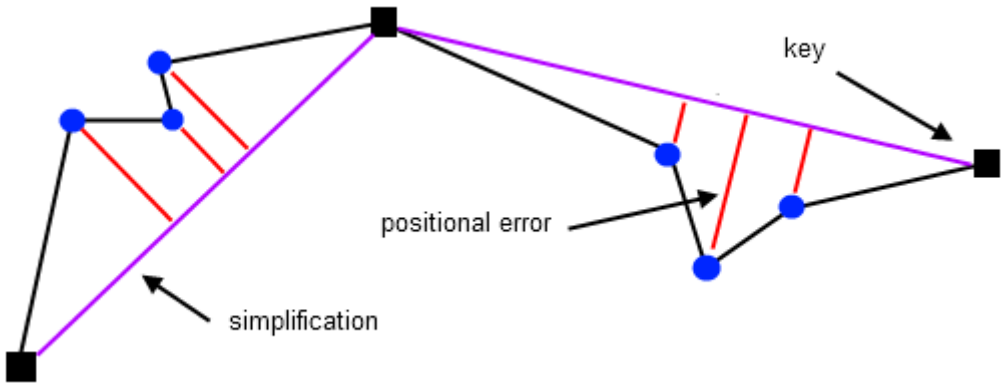
C++

```
unsigned tolerance = 25;          // point count tolerance
std::list <long long> polyline;   // original polyline, assume not empty
std::vector <double> result;      // resulting simplified polyline

// simplify the 4d polyline to 25 points
psimpl::simplify_douglas_peucker_n <4> (polyline.begin (), polyline.end (),
                                        tolerance, std::back_inserter (result));
```

This example demonstrates the use of a non-random access container with a signed integer data type.

Positional Errors

Simplifying a polyline introduces shape distortion. The higher the degree of simplification the higher the amount of distortion. One way of measuring this error induced by simplification, is by looking at the location difference between the original and the simplified line.



For each original point, the positional error is calculated as the perpendicular difference between that point and the corresponding line segment of the simplification. Better performing simplification algorithms consistently produce lower positional errors.

Interface

C++

```
template <unsigned DIM, class ForwardIterator, class OutputIterator>
OutputIterator compute_positional_errors2 (
    ForwardIterator original_first,
    ForwardIterator original_last,
    ForwardIterator simplified_first,
```

```
    ForwardIterator simplified_last,
    OutputIterator result,
    bool* valid)
```

For each point in the range [original_first, original_last) the **squared** distance to the simplification [simplified_first, simplified_last) is calculated. Each positional error is copied to the output range [result, result + (original_last - original_first)). Note that both the original and simplified polyline must be defined using the same value_type.

C++

```
template <unsigned DIM, class ForwardIterator>
math::Statistics compute_positional_error_statistics (
    ForwardIterator original_first,
    ForwardIterator original_last,
    ForwardIterator simplified_first,
    ForwardIterator simplified_last,
    bool* valid)
```

Computes statistics (*mean, max, sum, std*) for the positional errors between the range [original_first, original_last) and its simplification the range [simplified_first, simplified_last). All statistics are stored as doubles.

Input (Type) Requirements

1. DIM is not 0, where DIM represents the dimension of the polyline
2. The ForwardIterator value type is convertible to the value type of the OutputIterator (only for *compute_positional_errors2*)
3. The ForwardIterator value type is convertible to double (only for *compute_positional_error_statistics*)
4. The ranges [original_first, original_last) and [simplified_first, simplified_last) contain vertex coordinates in multiples of DIM, f.e.: x, y, z, x, y, z, x, y, z when DIM = 3
5. The ranges [original_first, original_last) and [simplified_first, simplified_last) contain a minimum of 2 vertices
6. The range [simplified_first, simplified_last) represents a simplification of the range [original_first, original_last), meaning each point in the simplification has the exact same coordinates as some point from the original polyline.

In case these requirements are not met, the valid flag is set to false **or** compile errors may occur.

Implementation Details

The algorithm is implemented using two nested loops. The outer loop processes each line segment from the simplification. The inner loop processes each point of the original polyline, computing the perpendicular distance to the current line segment. The inner loop ends when a point exactly matches the coordinates of the end point from the line segment.

When the outer loop has finished processing all line segments from the simplification, the last point from that simplified polyline should exactly match the last processed point from the original polyline. Only if this condition holds are the calculated positional errors considered valid. This

means I can only say if the results are valid after I am done computing and copying errors to the output range. So I needed some way of letting the caller know this. One option would be to throw an exception. However, I designed *psimpl* to not itself throw any exceptions (see section *About the Code*). Instead I opted for an *optional* boolean `valid`.

Usage

C++

```
std::vector <double> original;      // original polyline, assume not empty
std::vector <double> simplified;    // simplified polyline, assume not empty
std::vector <double> errors;        // calculated errors
bool valid = false;                 // indicates if the calculated errors are valid

// compute the squared positional error for each point of the original 2d polyline
psimpl::compute_positional_errors2 <2> (original.begin (), original.end (),
                                        simplified.begin (), simplified.end (),
                                        std::back_inserter (errors), &valid);

// compute positional error statistics for all points of the original 2d polyline
psimpl::math::Statistics stats =
    psimpl::compute_positional_error_statistics <2> (original.begin (), original.end (),
                                                     simplified.begin (), simplified.end
(),
                                                     &valid);
```

About the Code

As stated earlier, the implementation of all algorithms are contained within the header file *psimpl.h*. This file has the following structure:

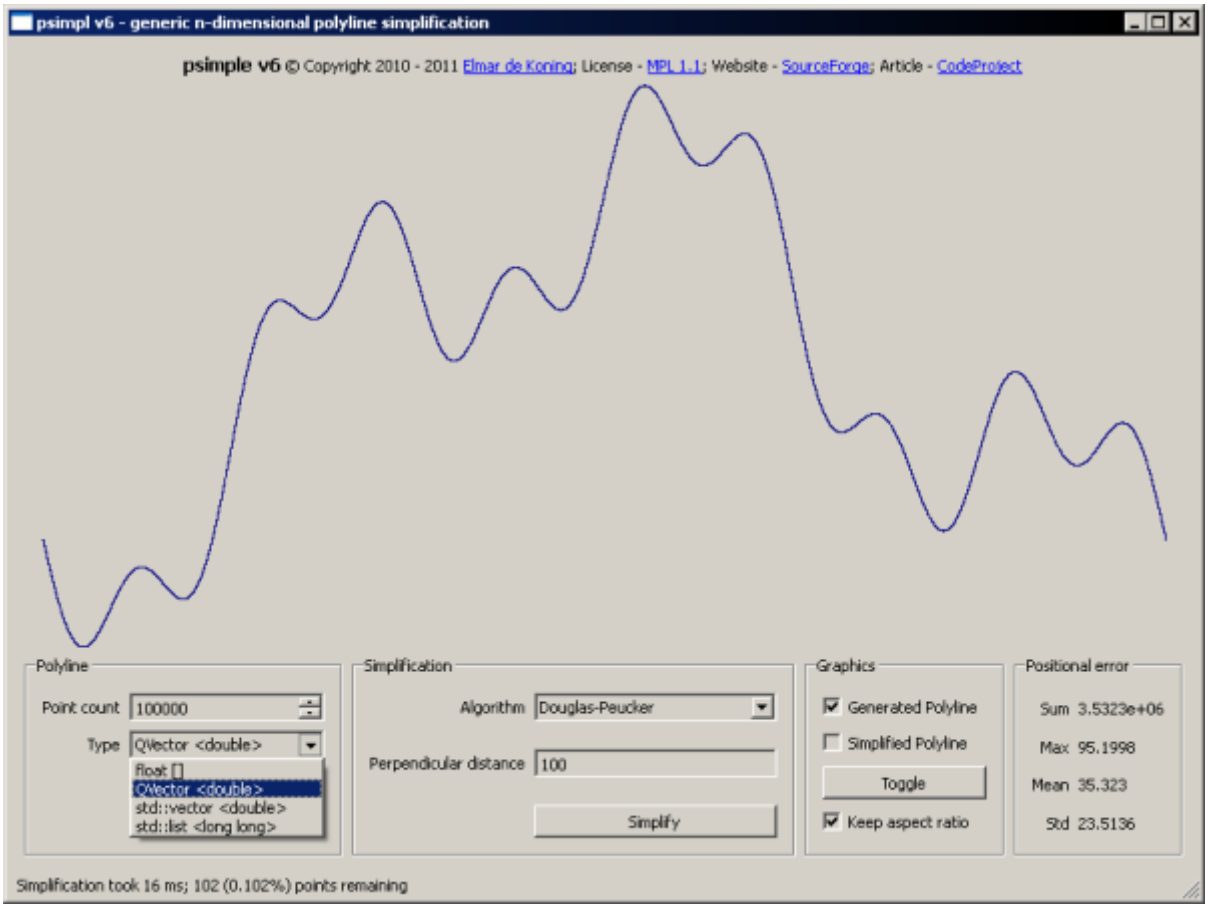
- namespace `psimpl`
  - namespace `util`
    - class `scoped_array <T>`
  - namespace `math`
    - struct `Statistics`
    - function `equal <unsigned, InputIterator>`
    - function `make_vector <unsigned, InputIterator, OutputIterator>`
    - function `dot <unsigned, InputIterator>`
    - function `interpolate <unsigned, InputIterator, OutputIterator>`
    - function `point_distance2 <unsigned, InputIterator1, InputIterator2>`
    - function `line_distance2 <unsigned, InputIterator>`
    - function `segment_distance2 <unsigned, InputIterator>`
    - function `ray_distance2 <unsigned, InputIterator>`
    - function `compute_statistics <InputIterator>`
  - class `PolylineSimplification <unsigned, InputIterator, OutputIterator>`

        - function `NthPoint`
        - function `RadialDistance`
        - function `PerpendicularDistance`
        - function `ReumannWitkam`
        - function `Opheim`
        - function `Lang`
        - function `DouglasPeucker`
        - function `DouglasPeuckerAlt`
        - function `ComputePositionalErrors2`
        - function `ComputePositionalErrorStatistics`
        - class `DPHelper`
    - function `simplify_nth_point <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_radial_distance <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_perpendicular_distance <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_reumann_witkam <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_opheim <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_lang <unsigned, BidirectionalIterator, OutputIterator>`
    - function `simplify_douglas_peucker <unsigned, ForwardIterator, OutputIterator>`
    - function `simplify_douglas_peucker_n <unsigned, ForwardIterator, OutputIterator>`
    - function `compute_positional_errors2 <unsigned, ForwardIterator, OutputIterator>`
    - function `compute_positional_error_statistics <unsigned, ForwardIterator>`

All the code is contained within the root namespace `psimpl`. The class `util::scoped_array`, similar to `boost::scoped_array`, is a smart pointer for holding a dynamically allocated array. `math` is a namespace containing all functions related to computing the squared distance between various geometric entities. The class `PolylineSimplification` provides the implementation for each simplification algorithm. It contains only member functions that operate on `InputIterator` and `OutputIterator` types. For the *Douglas-Peucker* routines, the internal class `DPHelper` is used. This helper class encapsulates all code that operates on value type pointers. The top-level functions are for convenience as they provide template type deduction for their corresponding member functions of `PolylineSimplification`.

psimpl itself does not throw exceptions. The reason for this is that I consider exception handling to be rather rare within C++ applications. Unlike the .NET world, a lot of developers just don't use it nor even think much about it.

About the Demo Application



The demo application allows you to experiment with the different simplification algorithms. It can generate pseudo random 2D-polylines of up to 10,000,000 vertices in various types of containers. The bounding-box of the generated polyline is always n x n/2, where n is the amount of vertices of the generated polyline. Use this fact to specify a proper distance tolerance. Comparing the various algorithms can be done visually and by using the computed positional error statistics. These statistics are only available when the `value_type` of the chosen container is `double`.

Internally, the generated and simplified polylines are always stored in a `QVector<double>`. Just before simplification, it is converted to the selected container type. Afterwards, this temporary container is destructed. Normally, you won't notice this, but it seems that creating and destructing a `std::list(10.000.000)` can take a rather long time. The resulting performance measurements never include these conversion steps. I chose this approach as it allowed me to quickly add new container types.

Note that the entire application is single threaded (sorry for being lazy), meaning it could go 'not responding' during a long-running simplification.

The demo application was made using *Qt 4.7.3*, *Qt Creator 2.1.0*, and *Visual Studio 2008 Express*. Complete source code is included.

Upcoming Versions

Algorithms, in no particular order, that I will add at some point in the future:

- *Random point* routine

- *Jenks* simplification
- ~~*Lang* simplification~~
- *Visvalingam-Whyatt* simplification
- *Zhao-Saalfeld* simplification
- Computing *area errors* due to simplification

Some other stuff I want to look at:

- Split *Douglas-Peucker* into two algorithms: a vanilla implementation and one using the *Distance between points* pre-processing step
- A separate *psimpl* version that operates on *point* containers instead of *coordinate* containers
- *Positional error* calculation that work for situations where the data type of a simplification differs from the original polyline

---

History

- 28-09-2010
  - Initial version
- 23-10-2010
  - Changed license from CPOL to MPL
  - Updated the source and demo packages accordingly, and added a small section about the license
- 26-10-2010
  - Clarified input (type) requirements, and changed the behavior of the algorithms under invalid input
- 01-12-2010
  - Added the N^{th} Point, Perpendicular Distance, and Reumann-Witkam routines
  - Moved all functions related to distance calculations to the `math` namespace; refactoring
- 10-12-2010
  - Fixed a bug in the Perpendicular Distance routine
- 27-02-2011
  - Added Opheim simplification, and functions for computing positional errors due to simplification
  - Renamed `simplify_douglas_peucker_alt` to `simplify_douglas_peucker_n`
- 18-06-2011
  - Added Lang simplification
  - Fixed divide by zero bug when using integers
  - Fixed a bug where incorrect output iterators were returned under invalid input
  - Fixed a bug in `douglas_peucker_n` where an incorrect number of points could be returned

- Fixed a bug in `compute_positional_errors2` that required the output and input iterator types to be the same; fixed a bug in `compute_positional_error_statistics` where invalid statistics could be returned under questionable input; documented input iterator requirements for each algorithm
- Miscellaneous refactoring of most algorithms

License

Written By

Elmar de Koning

Software Developer Optiver

Netherlands

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.

Comments and Discussions

Douglas-Peucker (Variant) logic using required number of sampled data points
Member 12006516 25-Sep-15 6:55

Douglas-Peucker (Variant) bug
guy katznelson 17-Feb-14 23:53

Possible Error in Douglas Peuker Algo
creajster123 26-Sep-13 16:15

Sorting points
Andy Bantly 22-Apr-13 11:31

My vote of 5
Alexandr Gavriluk 27-Jan-13 14:29

My vote of 5
Prasyee 1-Nov-12 22:29

My vote of 5
tfiner 1-May-12 4:15

Very fine
BillW33 19-Apr-12 4:53

How to get the 'm' value?
happy.hut 26-Mar-12 18:19

Re: How to get the 'm' value?
Elmar de Koning 28-Mar-12 10:17

Re: How to get the 'm' value?
happy.hut 29-Mar-12 19:57

Re: How to get the 'm' value?
Elmar de Koning 31-Mar-12 1:47

What is the best choice for unsigned int 16 data?
Vladimir Starostenkov 25-Nov-11 5:47

Re: What is the best choice for unsigned int 16 data?
Elmar de Koning 28-Nov-11 1:36

Visvalingham-Whyatt
Member 8053074 21-Oct-11 4:32

Re: Visvalingham-Whyatt
Elmar de Koning 21-Oct-11 9:52

My vote of 5
oeg2006 11-Jun-11 22:32

My vote of 5

dmu-its **24-Mar-11 20:00**

My vote of 5

constantin.a.cristian **11-Mar-11 12:06**

My vote of 5

Robert Sitton **8-Mar-11 7:39**

Code used in SimplexNumerica

Ralf Wirtz **8-Mar-11 5:58**

Refresh 1 2 3 Next ▷

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid

Web04 2.8:2022-12-16:1