Rapid preconditioning of data for accelerating convex hull algorithms

Article

Accepted Version

On the rapid preconditoning of data for accelerating convex hull computations

J. Cadenas and G. M. Megson

Given a data set of 2D points in the plane with integer coordinates, the method proposed, reduces a set of n points down to a set of s points $s \leq n$, such that the convex hull on the set of s points is the same as the convex hull of the original set of n points. The method is O(n). It helps any convex hull algorithm run faster. Empirical analysis of a practical case shows a percentage reduction in points of over 98%, that is reflected as a faster computation with a speedup factor of at least 4.

Introduction: Computing the convex hull on a set of n 2D points is a first pre-processing step to many geometric algorithms and in practical applications (e.g. computer visualisation, maps, rover path finding and home range [1]). Indeed, one can say with confidence, that finding the boundary of a given set of points is a fundamental problem in providing fast algorithms in many modern day mobile devices, games consoles, digital cameras, and client-server (web) applications that seek to reduce and create knowledge or patterns from raw data collection. Most known convex hull algorithms are of time complexity O($n$log$n$) [2]; these methods are general in the sense that they do not impose any restriction in the order of points. Linear complexity (O(n)) methods, such as the one due to Melkman [3], do exist but require a set of points that are ordered in some way, for example, [3] requires an order where the points form a simple polygonal chain. Such orderings are not always easy given the process of data collection.

Regardless of the time complexity of an algorithm, reducing the set of n points down to a set of $s \leq n$ points would result in faster computations, provided that the smaller set preserves the convex hull of the original (bigger) set. This reduction is often used as the first step in implementation of convex hull algorithms to improve their performance [4, 5]. This Letter presents a new approach, with three distinct advantages. First, we show that the method is linear and general. Second, no explicit sort of points is required; a common reduction method in existing literature requires an explicit sort of the points along a particular direction [4]. Third, by construction the reduced set of data forms a simple polygonal chain and hence straightforwardly prepares the data for linear methods such as [3]. We show through experimental evaluation that the method makes faster convex hull computations for both linear and non-linear algorithms.

Heuristic of the idea: Assume a 2D square of sides p, with integer points whose coordinates are in the range 1, …, p. As a small example consider the set of (x, y) points on the left of Fig. 1, given in any order as an array P. Assume an array L of p elements with each element L[i], i = 1, …, p, initialised to (p+1, -1), so L = [(p+1,-1), …, (p+1, -1)]. Consider the following pseudo-code:

```
1    foreach point in P do
2      x_i, y_i = point
3      y_1, y_2 = L[x_i]
4      ly = min(y_i, y_1)
5      hy = max(y_i, y_2)
6      L[x_i] = (ly, hy)
```

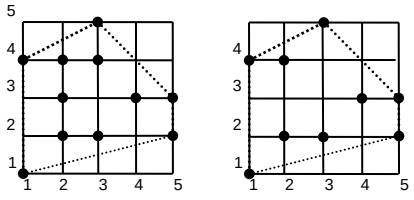

Fig. 1 *Left: (x, y) integer points on a 2D grid with p = 5. Right: Points with minimum and maximum y values for each x coordinate.*

After all n points of P have been processed by the above routine, L = [(1,4), (2, 4), (2,5), (3,3), (2,3)]. L[1] = (1, 4) since, y = 1 is the minimum point (*min*); and y = 4 is the maximum (*max*) point for column x = 1. This reduced set is shown on the right of Fig. 1; even intuition tells you the convex hull on the left is the same as the convex hull on the right. This is so, since local convexities of the boundary points are the only ones which need to be considered when deriving a convex hull [2]. Local convexities are maintained by keeping *min* and *max* for each column while removing any collinear points for each column.

The routine above visits each point of P once, therefore L is built in O(n) time. Scanning L along x builds a simple polygonal chain, since joining all points of the reduced set $s \leq n$ creates edges that do not intersect. For each valid point in L (one different to (p+1, -1)) joining *ly* to *hy* (*min* to *max*), and then from *hy* (*max*) to *ly* (*min*) of the next valid point, forms a simple polygonal chain (see 9 points of Fig. 1 on the right). Therefore, Melkman's method, for example, can be applied to the polyline built from a scan of L to build the convex hull. Scanning L to build the polygonal chain then takes O(p) time and provided that $p \leq n$ the whole method of building the polygonal chain takes O(n) time. As up to $n = p^2$ points may be in the original set, a density of points $s/n > 2/p$ makes the method O(n). Thus the method potentially reduces the percentage of original points down to $1 - 2/p$.

Table 1: Boolean array M[i] for i = 1, ..., 5 read in decimal as m and recursively iterated until m = 0 to extract positions in M that had 1's.

Iter.	*M*	*m*	m_{j-1}-m_j	*pos*	*x*
0	[1, 0, 1, 0, 1]	21			
1	[1, 0, 1, 0, 0]	20	1	0	5
2	[1, 0, 0, 0, 0]	16	4	2	3
3	[0, 0, 0, 0, 0]	0	16	4	1

Exploiting the method for building convex hulls: We have assumed a 2D region where the points lie in a square, and also that we know its size p. In general, the method can be applied to any 2D box of size p and q even without knowing the actual p, q values. In this general case, a method to reduce a set of n points down to s points, before building a convex hull is given by:

Step 1: Find the maximum and minimum x and y in the point set P (define p, q).

Step 2: Translate the point set P into a point set P' using (x', y') = (x-p+1, y-q+1).

Step 3: Build array L[i] from the set of points P' by applying the routine above.

Step 1 and 2 are both O(n). Step 3 reduces P to a set P' of $s \leq n$ and is O(n). So the whole procedure of steps 1 through 3 is O(n), provided that $p \leq n$ as explained before. The case $n < p$ will be considered later. Notice that step 3 does not require sorting the points. Also notice that the values p, q are obtained from step 1 and can be used to do a sweep of points along the *min*(p, q) for the greatest reduction. In many problems, the size of p, q is already known (detecting the boundary of binary images, collision detection, and cloud segmentation of a geographical area) and so the size of L is pre-computed without a need for step 1. Step 1, 2 and 3 can be fused into a single step, assuming that e is the bit length required to express each x, y point coordinates, and where e gets updated as points are analysed using schemes similar to compressing bit vectors [6]. As steps 1-3 are O(n), any 2D convex hull algorithm, including linear ones, can be used as a final step 4 to build a convex hull for accelerated computations.

Complexity for the case n < p: In the example above it is assumed all p elements were populated with points from the original set of n points. Now, assume there are gaps in array L. Suppose, in the small example above for p = 5, points in columns x = 2 and x = 4 are removed, so L = [(1,4), (6, -1), (2,5), (6,-1), (2,3)]; the empty entries are (6, -1) since all elements were initialised to (p+1, -1). Let an array M[i] i = 1, …, p with

Boolean entries be all initialised to False ('0'). Then, immediately after line 2 in the pseudo-code given above, insert the statement $M[x_i] = 1$. At the end of the routine, M = [1, 0, 1, 0, 1] indicating there were gaps in array L at x positions 2 and 4. The idea is now to extract from M the 1's, skipping the 0's in as many steps as 1's in M. Such extraction is accomplished by considering M as a binary number m and applying the recurrence $m_j = m_{j-1}$ AND $(m_{j-1}-1)$ [7]. The procedure is illustrated in Table 1. A '1' position in M is computed as $pos = \log_2(m_{j-1}-m_j)$ and index in L as $x = p - pos$. In three iterations, x indices 5, 3, 1 are recovered, these correspond to positions in this small example, where L has valid points, skipping empty positions 2 and 4. Note that the procedure takes O(k) time, where k is the number of 1's in M, $k \leq p$. As $n < p$, $k = n$ in this case, the method remains O(n). However, this has the cost O(p) in memory space complexity. If p size is too large the conversion of the binary string M to the decimal m is impractical. In this case, M can be blocked into strings of size r and the whole procedure remains O(n), as explained in [8]; an explicit call to $\log_2()$ function is not needed either. Block sizes of r = 32 or 64 are practical since m can be directly expressed as integers or long integers in current processors.

Fig. 2 *A 2D projection from a 3D scan of a Bison [9] of n =* 2108416 points in *a 2D grid of 5510×1366, reduced to s = 11020 points by the method.*

Experimental results: Data available for the mass estimation of mammals from convex hulls was analysed (a Bison is shown in Fig. 2) [9]. Table 2 shows the reduction of points from n down to s as a percentage and compared to previous work [4]. The speedup in execution time T(s4)/T(sh), due to this further reduction, using the Quick Hull algorithm (of O($n$log$n$) complexity) and Melkman (O(n)) algorithms was evaluated as 5.2 and 4.3 respectively. By comparison, a percentage range between 3.4% – 12.2% of the original points remained after a reduction step for their dataset for the recent work in [5].

Table 2: Mammal's data, n original points are reduced to s points, using the common method in [4] against the method here.

Mammals [9]	n	s (%) in [4] (s4)	s (%) here (sh)
Pig	535819	11.6	1.2
Polar	783025	15.4	0.9
Reindeer	845680	13.2	0.9
Bull	1411641	16.0	0.6
Bison	2108416	13.3	0.5
Camel	2120768	8.2	0.5
Elephant	7760648	16.6	0.2

Conclusion: The method presented here reduces a set of n 2D points (of integer coordinates) to a set of $s \leq n$, before applying any suitable algorithm that produces the convex hull. The method is of time O(n) for $n \geq p$, and a mechanism for the method to remain O(n) for $n < p$ is also presented, although at the cost of O(p) in storage memory. Thus, the method can be used as a first step to further improve the performance of convex algorithms of any complexity; here it was evaluated to give an extra speedup factor of at least four. The reduction method produces a simple polygonal chain expressed as an array L of p pair of points. A

percentage reduction of up to $1 - 2/p$ points can be achieved depending on the distribution of points. For a dataset of mammals, the method here resulted in bigger reductions (of over 98%) than the method presented in [4] (of around 85%). No explicit sorting, or cross-product between points, also makes the method hardware amenable.

J. Cadenas (*School of Systems Engineering, University of Reading, Reading, RG6 6AX, United Kingdom*)

E-mail: o.cadenas@reading.ac.uk

G. M. Megson (*School of Electronics and Computer Science, University of Westminster, London, W1W 6XH*)

References

1 Wan, Z., SUN M., and Jiang J.: 'Automatically fast determining of feature number for ranking-based feature selection', *Electron. Lett.*, 2012, **48**, (23), pp. 23-24

2 Preparata, F., and Shamos M.: 'Computational Geometry: An Introduction' (Chapter 3, Springer 1985)

3 Melkman A.: 'On-line construction of the convex hull of a simple polyline', *Inform. Process. Lett.*, 1987, **25**, (1), pp. 11-12

4 Akl S. G., and Toussaint G. T.: 'A fast convex hull algorithm', *Inform. Process. Lett.*, 1978, **7**, (6), pp. 219-222

5 Tang, M., Zhao, J., Tong R., and Manocha D: 'Full GPU accelerated convex hull computation', *Computer and Graphics*, 2012, **36**, (5), pp. 498-506

6 J. Teuhola: 'A compression method for clustered bit-vectors', *Inform. Process. Lett.*, 1978, **7**, (6), pp. 308-311

7 Kernigham B. W., and Ritchie D.: 'The C Programming Language' 2nd Ed, (Prentice Hall, 1978)

8 Megson G. M., and Cadenas J.: 'A rank-based convex hull method for dense data sets' (arXiv:1301.4809 [cs.CG])

9 Sellers, W. I., et al.: 'Minimum convex hull mass estimation of complete mounted skeletons', *Bio. Lett.*, June 2012, pp. 1-4