Linearization of Graham's Scan Algorithm Complexity

Veljko Petrović[#1], Dragan Ivetić[#2]
[#]*Faculty of Technical Sciences, University of Novi Sad, Republic of Serbia*
[1]pveljko@uns.ac.rs
[2]ivetic@uns.ac.rs

Abstract **- The Graham's Scan approach to two-dimensional convex hull calculation is considered. The performance bottleneck is found in the sorting step that precedes the Graham's Scan scanning operation. Methods are considered to eliminate this bottleneck. The method chosen is replacing the O(n lg n) sorting algorithm normally used with a radix sort. To operate within Graham's scan, the radix sort algorithm must be modified. The main modification is getting it to operate on real numbers. This is achieved by using the fact that digital computers only operate on a countable and finite subset of real numbers, and using this fact to reduce the problem of sorting by real number to sorting by integer. The ramifications of this modification are taken into consideration in the light of previous theoretical work in this area. Performance as compared to other algorithms is considered. Further, the consequences of a proof that the lower bound for the temporal complexity of two-dimensional convex hull algorithms is Ω(n lg n) are considered.**

Keywords **- convex hull, algorithm, radix sort, Graham scan, computer graphics**

I. INTRODUCTION

The convex hull problem in two-dimensional space can be summarized as the search for a convex polygon that contains all the points in a certain set *Q* while being minimal in size. This is, to an extent, a simplification. In the general case finding the convex hull of a set of points Q in $\mathbb{R}^n$ is the search for a boundary of the least convex set which contains all those points. In the two-dimensional case this boundary is clearly a polygon which brings us to the initial definition [1].

One of the algorithms capable of finding the convex hull of a set of points in two dimensions is Graham's Scan. Graham's scan is a simple rotational sweep algorithm with good performance characteristics. It is O(n lg n) in the worst case. The algorithm exists in multiple variants, but the original works in three distinct phases: [2] [3]

1. Preparing the input point set.
2. Computing the initial hull.
3. Sweeping around the points removing ones which would not fit into the hull.

The input set is prepared by first picking a pivot point for the algorithm. This pivot point is usually the lowest, leftmost point in the set and is, as an extreme, always a part of the convex hull [4]. All other points are sorted by their polar angle around the pivot point. Those points which had identical polar angles are eliminated in such a manner that the only point with a given angle remaining is the one farthest from the pivot point [2].

The initial hull is computed by simply starting with the pivot point and the first two points in the sorted input points. This initial hull becomes the current hull which is updated throughout the algorithm. The sweeping phase considers points in the sorted input set one by one. For each of those points it is determined if its addition to the current hull causes a non-left turn. If it does, the latest point in the current hull is removed and the direction of the turn is tested again. This continues until a left turn is obtained, at which point the considered point is added to the current hull [2].

Listing 1 contains a pseudocode representation of the unmodified algorithm. Several external operations are used in this algorithm. They include:

- *init*(S), *push(S, E)*, *pop*(S) and *length*(S) are standard stack and/or list operations.
- *lowest(Q)* returns the lowest, leftmost point in a given set Q.
- *eliminate(Q, p0)* takes a list Q, sorted by the polar angle around p0 and returns a list where there are no points with the same angle. In case of conflict, only the point farthest away from p0 is kept. This can be made an O(n) operation.
- *nonleft(a, b, c)* determines if the angle formed by the three indicated points turns to the left or not. Easily determined by calculating the dot product of the relevant vectors.

It has previously been stated that Graham's scan has O(n lg n) complexity. It is possible, given the pseudocode representation, to see why. Lines 1, 5, 6, 7 and 8 are O(1), and need not be further discussed. The calculation of the lowest, leftmost point in line 2 is a θ(n) operation. Line 3 is a simple heap sort and has the complexity class of O(n lg n). Line 4 has the complexity of O(n). The two nested loops give the impression of great complexity but, in fact, pose little complication. The outer loop can execute, at the most n – 3

times. Discounting the inner loop, each execution of the outer loop is an O(1) *push* operation. The inner loop is less predictable. However, given that it contains only a *pop* operation, and that it is impossible to pop something from the stack which isn't there in the first place, it is clear that the while loop can only execute, *in toto*, m – 2 times. Since the *nonleft* test is O(1), and so is the *pop* operation, and since m – 2 ≤ n – 1 the total complexity of the while loop is O(n). Thus, the total complexity of the algorithm is O(1) + θ(n) + O(n lg n) + O(n) + O(1) + O(1) + O(1) + O(n) + O(n) = O(n lg n). [2].

It is interesting to note that the complexity of the algorithm does not depend on the part of the algorithm that does the calculation of the hull itself, but instead on the sorting step which is clearly the dominant member, complexity-wise. This poses the question if it is possible to decrease the complexity by trying for a more efficient sort. It is commonly known that there exists a lower bound of Ω(n lg n) for any sorting algorithms based on arbitrary key comparisons [5]. However, there exist more specialized algorithms that exhibit better performance in certain cases.

```
GRAHAM-SCAN(Q):
01.    if(length(Q) <= 3) return Q
02.    p0 := lowest(Q)
03.    (t_p1...t_pn) := hsort(Q, p0)
04.    (p1...pm) := eliminate((t_p1...t_pn), p0)
05.    init(S)
06.    push(p0, S)
07.    push(p1, S)
08.    push(p2, S)
09.    for i := 3 to m do
10.       while(nonleft(peek(S), top(S), pi) do
11.          pop(S)
12.       push(pi, S)
13.    return S
```

Listing 1: Unmodified Graham's Scan

This paper is divided into four separate sections. The first is the introduction, which introduces the concepts used in the paper, chiefly, Graham's Scan algorithm. The second section outlines radix sort and how it may be modified and incorporated into Graham's Scan. The third section deals with the implication of this modification and the linearization of Graham's Scan temporal complexity. It deals with the existing proof of a lower bound for two-dimensional convex hull calculation, and compares the modified Graham's Scan to other algorithms. The fourth section concludes the paper, briefly describes what has been accomplished and outlines potential avenues for further research.

II. The Radix Sort Modification

The core concept of this paper is to adapt the radix sort algorithm to Graham's scan in order to reduce the complexity class of the resulting algorithm to O(kn). Radix sort is a variant of non-comparison based sorting generally meant for integer keys. Briefly, it operates by sorting a given list of integers by sorting them sequentially by their digits in a certain radix. The specific version used here starts with the least significant digit (LSD) and uses iterated counting sort applied to bytes. This means that it will sort a, say, four-byte value in four passes. The total performance of a radix sort is O(kn) [5].

As described, radix sort will only work with integers. Further, it will only work on integers that can be expressed in a certain, fixed, number of radices, though this number can be arbitrarily large. To be used in Graham's scan, radix sort will need to sort by polar angle. This requires sorting points by real keys, which is quite different than sorting a single list of integers. The problems which need to be solved are:

- Sorting real numbers with radix sort.
- Sorting points by key, and not only the keys themselves.
- Maintaining the performance gain.
- Assuring unchanged precision.

It should be stated immediately that radix sort cannot sort true real numbers. However, since no digital computer works with true real numbers, but an approximation thereof, this shouldn't pose any problem. The numbers that radix sort needs to sort in this instance are limited to the range [0, 2π]. Any implemented approximation used on an actual computer will have limited precision. Given the limited precision, and the limited range of possible values it is clear that it is possible to construct an O(1) function such that equation 1 holds.

$$f : [0,2\pi] \cap RF \rightarrow N_0$$
$$a \leq b \Leftrightarrow f(a) \leq f(b) \wedge a,b \in [0,2\pi] \cap RF$$

Equation 1: Mapping represented real numbers to integers

In Equation 1 RF is the set of real numbers which can be represented in the system of approximation in question. Given that a fixed-size system of representation can only

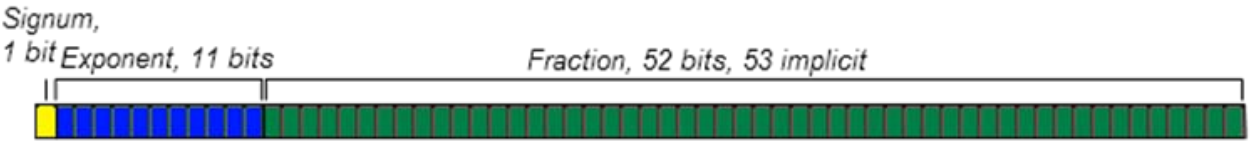

Figure 1: Double precision floating point number

represent a fixed number of distinct numbers, it is possible to construct *f* by ordering all possible real representations using the index of a given input parameter in the resulting list as the result of *f*. With this in mind, it is clear that radix sort can be used to sort by polar angle, provided an appropriate *f* is used.

Most modern computers represent real numbers using floating point, specifically, the IEEE 754 standard. The implementation used in this paper is based on the double precision IEEE 754 standard as seen on Figure 1. Double precision floating point values in this standard can be, provided they are positive, compared as integers [6].

$$V = (-1)^S \times 2^{E-EB} \times 1.M$$

Equation 2: Calculating the value of a double precision IEEE 754 floating point representation

```
RADIX-SORT(L):
01.     len := length(L)
02.     for i := 0 to len-1 do
03.       input[i] := raw(L[i])
04.       mIndices[i] := i
05.       mIndices2[i] := i
06.     h0 := 0; h1 := 256; h2 := 512; h3 := 768;
07.   h4 := 1024; h5 := 1280;
08.   h6 := 1536; h7 := 1792
09.     for i := 0 to len-1 do
10.       counters[h0 + getbyte(input[i], 0)]++
11.       counters[h1 + getbyte(input[i], 1)]++
12.       counters[h2 + getbyte(input[i], 2)]++
13.       counters[h3 + getbyte(input[i], 3)]++
14.       counters[h4 + getbyte(input[i], 4)]++
15.       counters[h5 + getbyte(input[i], 5)]++
16.       counters[h6 + getbyte(input[i], 6)]++
17.       counters[h7 + getbyte(input[i], 7)]++
18.     for pass := 0 to 7 do
19.       offsetTable[0] := 0
20.       for i := 1 to 255 do
21.         offsetTable[i] := offsetTable[i - 1]
            + counters[pass * 256 + i – 1]
22.       for i := 0 to len – 1 do
23.         id := mIndices[i]
24.         byt := getbyte(input[id], pass)
25.         mIndices2[offsetTable[byt]] := id
26.         offsetTable[byt]++
27.       tmp := mIndices
28.       mIndices := mIndices2
29.       mIndices2 := tmp
30.     return mIndices
```

Listing 2: Modified and adapted radix sort

In equation 2, *V* is the value of the number, *S* is the signum, *E* is the exponent, *EB* is the exponent bias, which is equal to 1023 and *M* is the mantissa. The exponent bias makes sure that the value actually encoded in the bit representation isn't negative. Because the implicit 53 bit of the mantissa being always considered one, as long as the signum bit is always 0 the double precision representation can be interpreted as an integer and still compared with same results [6].

Since polar angles are in the interval [0, 2π], double precision floating point values can be treated as integers for the purposes of sorting. Radix sort is normally an in-place sort stable sort [5], but for the purposes of this paper it was necessary to maintain association between the sorted polar angle values and the points they referred to. There is no immediatelly convenient way of doing so. It is possible to use a hashtable, but this would lead to performance issues and greater memory consumption. Thus, the radix sort used was modified to create a permutation which, when applied to the input list of points would create the sorted version, as can be seen in listing 2. The input to the RADIX-SORT algorithm is a list of floats which corresponds to the calculated polar angles of the input list of points.

The precision of this approach is not in question, as long as computers unable to handle true reals are used. The imprecision of the described approach is exactly equal to the imprecision of the method of depicting real numbers in fixed space. As a result of this, the precision of the modified algorithm is no better, and no worse, than any other comparable algorithm. RADIX-SORT uses several external operations. The *raw(x)* operation turns a double precision IEEE float into an integer based on their binary representation. This doesn't require any substantial operations, and doesn't influence performance. The *getbyte(x, i)* operation extracts the i-th byte from x. This can be accomplished with a couple of bitwise operations.

Graham's scan is relatively easy to adapt to include RADIX-SORT. The modification is localised to lines 3-6 of listing 3. First, slist is created as a separate list of polar angles around p0. To do this, the *getPolarAngle(p1, p2)* external operation is employed. Slist is then sorted creating a permutation perm. The external operation *eliminate(L, p, p0)* works as before, but generates a new list and, instead of expecting a sorted L, it uses the permutation p to sort L.

As far as performance is concerned, the only substantive difference between this implementation and the original is in the complexity of the sorting step which is O(kn). The copying of the polar angle is an O(n) operation. Thus the temporal complexity of the entire algorithm is
O(1) + θ(n) + O(n) + O(kn) + O(n) + O(1) + O(1) + O(1) + O(n) + O(n) = O(kn).

III. Implications and Comparisons

There exists a proof which states that the lower bound for convex hull algorithms in two dimensions is Ω(n lg n). Given n real numbers $x_{1...n}$, a set P is constructed as in Equation 3.

```
GRAHAM-SCAN-RADIX(Q):
01.    if(length(Q) <= 3) return Q
02.    p0 := lowest(Q)
03.    for i := 0 to len – 1 do
04.       slist[i] := getPolarAngle(p0, Q[i])
05.    perm := radix-sort(slist)
06.    (p1...pm) := eliminate(Q, perm, p0)
07.    init(S)
08.    push(p0, S)
09.    push(p1, S)
10.    push(p2, S)
11.    for i := 3 to m do
12.       while(nonleft(peek(S),top(S),pi) do
13.          pop(S)
14.       push(pi, S)
15.    return S
```

Listing 3: Graham's Scan modified to include RADIX-SORT

$$P = \{ p_i \mid 1 \le i \le n \}$$
$$p_i = (x_i, x_i^2)$$

Equation 3: Constructing a set for the proof

Then, the convex hull of P is computed. The order in which the points $p_{1...n}$ appear on the lower half-hull of convP is the order in which $x_{1...n}$ should be sorted. Thus, if the convex hull can be computed in o(n lg n) time, points can be sorted in o(n lg n) time [7]. This conflicts with the known lower bound for general sorts [5]. Despite appearances, the described modification to Graham's Scan does not conflict with this proof. The proof rests on the theoretical framework of algebraic trees and assumes the coordinates of the points are actual real values. The modified Graham's scan does not work on actual real values, and as such the proof does not apply.

It is of some considerable interest to compare this modification against other algorithms for two-dimensional convex hulls. Table I shows the names of the more common algorithms and their expected performance characteristics in an average case and in the worst-possible case.

TABLE I
PERFORMANCES OF VARIOUS ALGORITHMS FOR CALCULATING TWO-DIMENSIONAL CONVEX HULLS

Name	**Complexity, average case**	**Complexity, worse case**	**Reference**
Gift Wrapping Algorithm	O(nh)	O(nh)	[2][4][8]
Graham's Scan	O(n lg n)	O(n lg n)	[2][3]
Quickhull	O(n lg n)	$O(n^2)$	[9]
Incremental with Edelsbrunner modification	O(n lg n)	O(n lg n)	[10]
Preparata – Hong	O(n lg n)	O(n lg n)	[11]
Chan's Algorithm	O(n lg h)	O(n lg h)	[12]
Graham's Scan, modfied	O(kn)	O(kn)	/

Some convex hull algorithms belong to a class of algorithms known as *output-sensitive*. That means that they express their complexity as a function of not only n, but also the size of the output set – h. To compare algorithms easily, it is necessary to estimate a value for h. This is a non-trivial problem of stochastic geometry, but there exist certain solutions in the literature as seen in Table II.

TABLE II
STOCHASTIC ESTIMATIONS FOR HULL SIZE

Distribution	Estimate	Reference
Circular uniform	$n^{1/3}$	[3][4]
Square uniform	$n^{1/3}$	[4]
Normal planar distribution	$(\lg n)^{1/2}$	[3]
Uniform within convex polygon	lg n	[3]

For purposes of easy and intuitive comparison, for an average case the circular/square uniform distribution used which means that $h = n^{1/3}$. In the worst case, the input set of points is on the border of a circle, which means that h = n. The only remaining parameter is a value for k. An illustrative estimate for k is 3. Of course this is only good for simple comparisons.

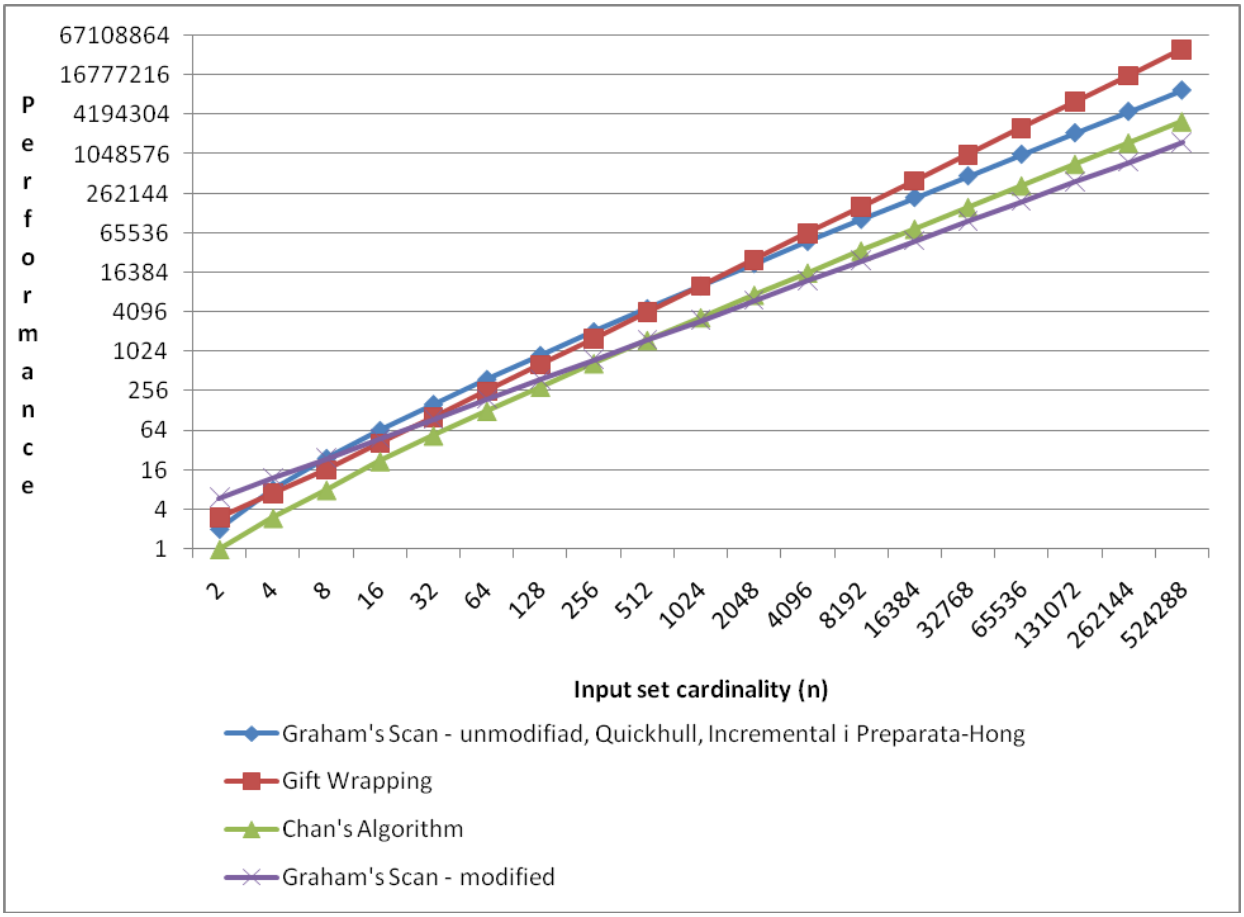

Figure 2: Average case complexity graph

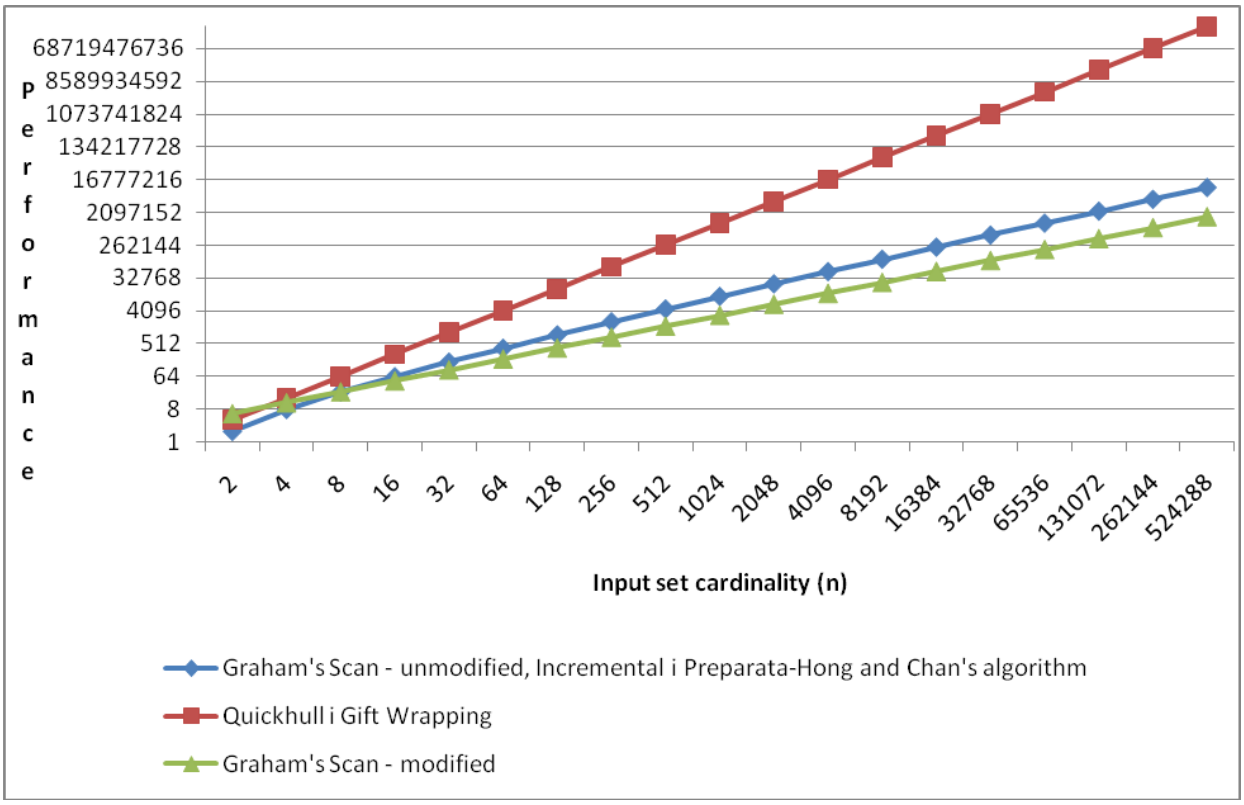

Figure 3: Worse case complexity graph

The estimate of k used is chosen primarily to illustrate the behavior of the complexity as n increases. A true value for k is best determined via experiment.

A graphical representation of the comparison of these algorithms can be seen on Figure 2 and Figure 3. Figure 2 is the comparison between algorithms in an average case, and figure 3 is the comparison between algorithms in the worst possible case. As can easily be seen, the modified Graham's scan is the fastest algorithm in the long run. This is to be expected as a constant multiplier, no matter how large, can always be surpassed by a function of n, as n increases.
However, the weakness of modified Graham's scan is that it takes a trully large input set before its superiority sets in. How practical this is, can only be determined by experimenting.

IV. CONCLUSION

This paper has outlined how the temporal complexity of Graham's Scan can be linearized provided it operates on a finite, countable subset of reals that can be represented on some digital computer. It provides the framework to create such an algorithm independantly of the system a given computer uses to represent real numbers.

This principle is illustrated on the example of the floating point representation of reals, specifically one described in IEEE's 754 standard. A concrete implementation of the idea, in pseudocode, allows for discussion of implementation detail and a more nuanced analysis of expected performance.

Further possible avenues of research include an analysis of potential applications, an experimental determination of the performanse characteristics of the algorithm and an experimental comparison between this algorithm and reference implementations of already well known algorithms.

References:

[1] M. De Berg, O. Cheong, M. Van Kreveld, M. Overmars, *Computational Geometry Algorithms and Applications*, Berlin, Germany, Springer-Verlag, 2008.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, Cambridge, USA, MIT Press, 2001.
[3] V. Bayer, „Survey of Algorithms for the Convex Hull Problem", preprint, 1999.
[4] R. Sedgewick, *Algorithms*, Reading, USA, Addison-Wesley, 1983.
[5] D.E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, Reading, USA, Addison-Wesley, 1998.
[6] *IEEE Standard for Floating-Point Arithmetic*, IEEE 754, 2008.
[7] M. Ben-Or, „Lower bounds for algebraic computation trees," in Proc. 15th Annu.ACM Sympos. Theory Comput., pp. 80-86.
[8] R. A. Jarvis, „On the identification of the convex hull of a finite set of points in the plane", Information Porcessing Letters 2, pp. 18-21.
[9] J. O'Rourke, *Computational Geometry in C*, Cambridge, UK, Cambridge University Press, 1998.
[10] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Berlin, Germany, Springer-Verlag, 1987.
[11] F. P. Preparata, S.J. Hong, „Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Communications of the ACM, Vol. 20, No 2, pp. 87-93.
[12] T. M. Chan, „Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions", Dicreete & Computational Geometry vol. 16, pp. 361-368.