A PROBABILISTIC EUCLIDEAN ALGORITHM THAT YIELDS SMALL COEFFICIENTS

by

G.M. Benedek and A. Paz
Technical Report #481
December 1987
A probabilistic Euclidean Algorithm that yields small coefficients.

Gyora M. Benedek and Azaria Paz

Computer Science Dept.
Technion
Haifa, Israel

ABSTRACT.

Given \( n \) positive integers \( a_1, a_2, \ldots, a_n \), a probabilistic algorithm for the solution of the equation

\[
\sum_{i=1}^{n} a_i x_i = \text{gcd}(a_1, a_2, \ldots, a_n)
\]

is provided. The algorithm has the following properties: If the distribution of the \( a_i \)'s over an interval \([0, 2^{k+1}]\) is uniform or nonincreasing and the density of the \( a_i \)'s \( d = \frac{n}{\log(\text{max}a_i)} \) satisfies the inequality \( d \geq 1 + \log_2(k+1/\epsilon) \), then the algorithm succeeds with probability greater than \( 1 - \epsilon \) and the resulting \( x_i \)'s satisfy the inequalities: \( \max |x_i| < 0.62(\text{max}a_i)^{0.694} \) and \( \sum |x_i| < 1.004(\text{max}a_i)^{0.694} \). A more general problem is solved in a similar way.
1. INTRODUCTION

In this paper we consider the following problem: Given \( n \) different positive integers \( a_1, a_2, \ldots, a_n \) find \( n \) "small" integers \( x_1, x_2, \ldots, x_n \) s.t.

\[
\sum_{i=1}^{n} a_i x_i = \gcd(a_1, a_2, \ldots, a_n)
\]

Where the term "small" is specified in the sequel.

Finding the \( \gcd \) and the corresponding coefficients is important for many applications like the Chinese reminders theorem [BLA], calculations on polynomial rings [BLA], solving linear Diophantine equations [BRA], computing the Smith or Hermite normal form of an integer matrix [BRA], finding the inverse of an element of a Galois field [BLA], integer programming [LEN], radio frequency intermodulation problem [MOR2], etc.

Without constraints on the size of the coefficients the problem can be solved in linear time, see [KNU], [BRA]. However in these solutions the coefficients tend to grow very fast [BRA], causing several practical problems like overflow, increase in actual complexity due to calculations with big numbers and an increased need for memory. Moreover in the problem of intermodulation the coefficients must be small. Also, an algorithm resulting in small coefficients can improve the Lenstra algorithm for integer programming [LEN].

In [BEN] we show that if we demand that \( \sum_{i=1}^{n} |x_i| \) be minimal then the problem is NP-hard. Similarly, it was shown by Paz [PAZ] that the problem is NP-hard if we demand that \( \max_{i=1}^{n} |x_i| \) be minimal.

Thus we do not expect to find a polynomial algorithm that finds minimal solutions in any of the above cases and we would satisfy ourselves with the finding of a fast algorithm that gives relatively small-coefficients. There are two papers known on the subject one by Morito and Salkin [MOR1] which presents a heuristic solution without bounds on the coefficients, the other by Bradley [BRA] which gives bounds on the product of the non-zero coefficients.

In section 2 we introduce two probabilistic Blankinship [BLA] type algorithms that solve 1.1 and result in relatively small coefficients.

In sections 3, 4 we show that one of the algorithms introduced in section 2 has the following property:

If the integers at input are uniformly distributed in the interval \([0, 2^{k+1})\) and have density

\[
d = \frac{n}{\log_2(\max a_i)} > 1 + \frac{\log_2(k+1/e)}{k}
\]

then with probability greater than \( 1 - e \) the algorithm produces at output a solution of 1.1 such that

\[
\max_{i=1}^{n} |x_i| < 0.62(\max a_i)^{0.694}
\]

\[
\sum_{i=1}^{n} |x_i| < 1.004(\max a_i)^{0.694}
\]
2. TWO PROBABILISTIC ALGORITHMS.

2.1. BLANKINSHIP TYPE ALGORITHM.

For the sake of completeness we show below, with the help of an example, what we mean by a Blankinship type algorithm. The reader familiar with [BLA] can skip this section.

Let $90,20,3$ be three given integers at input. We want to find three integers $x_1,x_2,x_3$ satisfying the equation below:

$$90x_1+20x_2+3x_3=gcd(90,20,3)$$  \hspace{1cm} 2.1.1

The algorithm proceeds as follows:

a. Attach the $i$-th unit vector to the $i$-th integer at input and form the $3\times 4$ matrix as below:

$$
\begin{bmatrix}
90 & 1 & 0 & 0 \\
20 & 0 & 1 & 0 \\
3 & 0 & 0 & 1 \\
\end{bmatrix}
$$

b. Perform a given 3-dimensional Euclidean Algorithm on the first column, but execute each operation on the whole row corresponding to the respective integer in the first column.

The first iteration results in (subtract the second row from the first 4 times):

$$
\begin{bmatrix}
10 & 1 & -4 & 0 \\
20 & 0 & 1 & 0 \\
-3 & 0 & 0 & 1 \\
\end{bmatrix}
$$

Continuing this way we get the second iteration (subtract the first row from the second twice):

$$
\begin{bmatrix}
10 & 1 & -4 & 0 \\
0 & -2 & 9 & 0 \\
3 & 0 & 0 & 1 \\
\end{bmatrix}
$$

Third iteration:

$$
\begin{bmatrix}
1 & 1 & -4 & -3 \\
0 & -2 & 9 & 0 \\
3 & 0 & 0 & 1 \\
\end{bmatrix}
$$

Fourth and final iteration:

$$
\begin{bmatrix}
1 & 1 & -4 & -3 \\
0 & -2 & 9 & 0 \\
0 & -3 & 12 & 10 \\
\end{bmatrix}
$$

Denote the first integer in each row by $b_i$ denote by $v_i$ the remaining part of that row and denote by $a$ the vector of the $a_i$'s at input. It can easily be shown that $v_i a^T = b_i$ and
\[ \gcd(a_1, a_2, \cdots, a_n) = \gcd(b_1, b_2, \cdots, b_n) \]

at every iteration. Thus the \( v_j \) corresponding to the nonzero \( b_j \) at termination contains the required coefficients. In the above example the required vector is \( v_1 \) in the fourth (last) iteration with

\[ 1 = 1 \times 90 - 4 \times 20 - 3 \times 3 \]
as required.

With regard to the above example we remark the following:

a. We could have stopped at the third iteration with the first row, since \( b_1 \) after the third iteration is already equal to 1 and thus \( v_1 a^T = 1 = \gcd(a_1, a_2, \cdots, a_n) \).

b. It can be shown that the \( v_j \)'s at termination of the algorithm (iteration 4 in the example) span all the solutions of equation 2.1.1.

c. In the above example we always chose to subtract the second biggest integer from the biggest. Any other method of choice would not affect, neither the correctness nor the complexity of the algorithm, but would affect the total number of steps and, what is more important for us, the size of the resulting coefficients. It can be shown that the complexity of the above algorithm is proportional to the length of the input \([\text{BRA}]\).

d. Notice the changes in value of the greatest coefficient (in absolute value) of the \( v_j \) vectors in the above example. At the beginning that coefficient equals 1, after the first step it grows to 4 as implied by \( \lfloor \frac{90}{20} \rfloor = 4 \). We want the maximal coefficient to grow minimally. This can be achieved if the ratio between the \( b_j \)'s to be subtracted is close to 1. If the number of integers at input is large we might be able to set a strategy which, with high probability, will subtract \( b_j \)'s whose ratio is close to 1. This heuristic principle and some additional refinements are the basis of the algorithm described below.

\section*{2.2 Introduction to Algorithm B}

**Input:** a vector of \( n \) different positive integers \( a_1, a_2, \cdots, a_n \) distributed uniformly over an interval \([1, H_0] \). Without loss of generality let \( H_0 \) be a power of 2 (i.e. \( H_0 = 2^{k+1} \)).

**Output:** a vector \( (x_1, x_2, \cdots, x_n) \) with a 'small' norm (for any preassigned norm) satisfying

\[ \sum_{i=1}^{n} a_i x_i = \gcd(x_1, x_2, \cdots, x_n) \]

\( 2.2.1 \)

**Procedure:** Append to each integer \( a_i \) at input the \( i \)-th unit vector, thus yielding the \( n \times (n+1) \) matrix below

\[
\begin{pmatrix}
1 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 \\
\vdots \\
0 & 0 & \cdots & 1 \\
\end{pmatrix}
\]

and perform elementary operations on rows of the above matrix. Denote the matrix at step \( j \) by \( b_1, v_1 \)

\[ b_2, v_2 \]
(Where the $v_p$'s $1 \leq p \leq n$ are $n$-dimensional vectors.)

Then $v_p a^T = b_p$ $1 \leq p \leq n$ and $gcd(a_1, a_2, \cdots, a_n) = gcd(b_1, b_2, \cdots, b_n)$.

### 2.3 ALGORITHM B

1. Repeat
   - Split the interval of the (remaining) $b_i$'s into two equal subintervals and divide the $b_i$'s into two sets accordingly. Call the upper set $Old$.
   - Select randomly pairs of rows headed by $b_i$ and $b_j$ in $Old$, and subtract the $b_j$ row from the $b_i$ row. If the resulting $b_i$ is negative multiply the row by -1. Since both $b_i$ and $b_j$ are in the upper subinterval the resulting $b_i$ after subtraction will be in the lower subinterval. After finitely many steps all the $b_i$'s in $Old$ except one are moved to the lower set. (For the time being) discard the leftover element in $Old$.

Until the lower half of the interval has size 1.

There are two possibilities for stopping:

1. If one of the resulting $b_i$ equals 1 then that $b_i$ is the $gcd$ and its corresponding vector $v_i$ solves 1.1.

2. After a certain number of iterations all the remaining $b_i$'s equal 0. In this case we cannot continue. If the last discarded $b_i$ divides all the $a_i$'s then it is the $gcd$ otherwise the algorithm fails. As we shall show, for large number of integers at input, the probability that the $gcd$ differs from 1 is small, and accordingly the probability that the algorithm stops as result of (2) is low. In fact this probability tends to 0 when the number of integers at input tends to $\infty$.

### 2.4 A NUMERICAL EXAMPLE FOR ALGORITHM B

**Input** $n=5$ integers in the range $[1,64)$ (i.e. $k=5,H_0=64$):

$5, 9, 33, 47, 62$.

The integers are sorted only for convenience. This is not necessary for the algorithm.

To each integer we append a unit vector as below:

- $62, 1,0,0,0,0$
- $47, 0,1,0,0,0$
- $33, 0,0,1,0,0$
- $9, 0,0,0,1,0$
At each iteration we fix the upper range and work with the integers in that range.

The first range is \([32,64)\) and the upper bound is \(H_0 = \frac{H_0}{2}\):

<table>
<thead>
<tr>
<th>step</th>
<th>range</th>
<th>row vectors</th>
<th>random choice</th>
<th>result</th>
<th>discard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[32,63]</td>
<td>(33,0,0,1,0,0)</td>
<td>(47,0,1,0,0,0)</td>
<td>(14,0,1,-1,0,0)</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(47,0,1,0,0,0)</td>
<td>(33,0,0,1,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(62,1,0,0,0,0)</td>
<td>(62,1,0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(47,0,1,0,0,0)</td>
<td>(47,0,1,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[16,31]</td>
<td>(9,0,0,0,1,0)</td>
<td>(14,0,1,-1,0,0)</td>
<td>(5,0,1,-1,1,0)</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(14,0,1,-1,0,0)</td>
<td>(14,0,1,-1,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(15,1,1,0,0,0)</td>
<td>(15,1,1,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(9,0,0,0,1,0)</td>
<td>(9,0,0,0,1,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[8,15]</td>
<td>(5,0,0,0,0,1)</td>
<td>(6,1,-1,0,-1,0)</td>
<td>(1,1,-2,1,0,0)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5,0,1,-1,1,0)</td>
<td>(5,0,1,-1,1,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(6,1,-1,0,-1,0)</td>
<td>(6,1,-1,0,-1,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[4,7]</td>
<td>(9,0,0,0,0,1)</td>
<td>(9,0,0,0,0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5,0,1,-1,1,0)</td>
<td>(6,1,-1,0,-1,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[2,3]</td>
<td>(9,0,0,0,0,1)</td>
<td>(9,0,0,0,0,1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We thus get:

\[ 1 = 1\times62 - 2\times47 + 1\times33 + 0\times9 + 0\times5 \]

After every iteration, the (leftover) integer marked by an asterisk (*) is discarded.

2.5 PROPERTIES OF ALGORITHM B.

Let \(m_i\) be the upper bound on the entries of the vectors \(v_j\) after the \(i\)-th iteration, in absolute value.

Then \(m_0 = 1\) (all the \(v_j\) vectors at input are unit vectors) and \(m_1 = 1\) (all the entries of a vector resulting from the subtraction of two different unit vectors are either 0 or 1 or -1).

After the \((i+1)\)-st iteration every \(v_j\) is either carried over unchanged from the previous iteration (such a vector will be called old) or results from the subtraction of two vectors from the \(i\)-th iteration (such a vector will be called new). As is easy to see, the entries of any old vector are bounded, in absolute value, by \(m_i\) and the entries of any new vector are bounded, in absolute value, by \(2m_i\). Therefore \(m_{i+1} \leq 2m_i\).

If the range of the integers at input is \([0,2^{k+1})\) then the number of iterations is at most \(k\) and therefore the entries of the vectors \(v_j\) at output are bounded in absolute value by \(2^{k-1}\).

2.6 ALGORITHM C

Algorithm C achieves a better bound than the bound guaranteed by algorithm B due to a refined selection procedure: two integers created in the last iteration are not allowed to combine. In other words in algorithm C we subtract old vectors from new vectors until all the new vectors are
exhausted. Only then we continue with the remaining old vectors as in algorithm B. As before let \( m_l \) be the upper bound on the absolute values of the entries of the \( v_j \) vectors at the end of the i-th iteration. Then \( m_l = m_1 = 1 \) as before.

After the first iteration all the old vectors are unit vectors and all the new vectors have \( 0, +1, -1 \) entries. The columns of the nonzero entries of a new vector differ from the columns of the nonzero entries of the old vectors. This implies that \( m_2 = 1 \), since at the second iteration new vectors are combined with old (unit) vectors only.

After the \( (i + 1) \)-st iteration the entries of the old vectors (carried over unchanged from the previous iteration) are bounded, in absolute value, by \( m_i \). The entries of a new vector at the \( (i + 1) \)-st iteration are bounded, in absolute value, by \( m_i + m_{i-1} \), since any such new vector is the result of the subtraction of an \( m_{i-1} \)-bounded vector (an old vector from the previous iteration) and an \( m_i \) or \( m_{i-1} \)-bounded vector (a new or an other old vector from the previous iteration).

As \( m_i \geq m_{i-1} \) we have that \( m_{i+1} \leq m_i + m_{i-1} \). The \( m_i \) sequence is therefore bounded by the Fibonacci series i.e.

\[
m_i = \left\lfloor \frac{1+\sqrt{5}}{2} \right\rfloor^{i-1} \leq \frac{2}{1+\sqrt{5}} (\max a_i)^{0.694...} < 0.62 (\max a_i)^{0.694...}
\]

A bound \( s_k \) for any norm of the coefficient vectors can be found similarly.

In the example above the difference between the two algorithms can be seen in the fourth iteration. Consider the three rows: \( (5,0,0,0,0,1) \), \( (5,0,1,-1,-1,-1,0) \), and \( (6,1,-1,0,-1,0) \). The first is an original row and the other two were created during the third iteration. Algorithm C will not subtract the last two rows (as did algorithm B) and will subtract the old row from the other two rows instead:

<table>
<thead>
<tr>
<th>step</th>
<th>range</th>
<th>row vectors</th>
<th>random choice</th>
<th>result</th>
<th>discard</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[4,7]</td>
<td>(5,0,0,0,0,1) (5,0,1,-1,-1,0) (6,1,-1,0,0,0)</td>
<td>(5,0,1,-1,-1,0) (5,0,1,-1,-1,0)</td>
<td>(0,0,-1,1,2,0)</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>[2,3]</td>
<td>(5,0,0,0,0,1) (5,0,1,-1,-1,0) (6,1,-1,0,0,0)</td>
<td>(6,1,-1,0,-1,0)</td>
<td>(1,1,-1,0,-1,1)</td>
<td></td>
</tr>
</tbody>
</table>

Resulting in:

\[ 1 = 1 \times 62 - 1 \times 47 + 0 \times 33 - 1 \times 9 - 1 \times 5 \]

While the maximal coefficient achieved by algorithm B was 2 the maximal coefficient achieved by algorithm C is 1.

We present now a formal and detailed description of the algorithm.

**ALGORITHM C** (detailed description).

**Input**: a set of \( n \) different positive integers \( a_1, a_2, \ldots, a_n \) distributed uniformly over an interval \([1, H_0]\).

**Output**: a vector \((x_1, x_2, \ldots, x_n)\) with a 'small' norm satisfying \( \sum_{i=1}^{n} a_i x_i = \gcd(x_1, x_2, \ldots, x_n) \).

**Initialization**: as in the Blankinship algorithm append to each integer at input the i-th unit vector, thus yielding an \( n \times (n+1) \) matrix and perform elementary operations on rows of the above matrix. Denote
the matrix at step \( j \) by

\[
\begin{align*}
&b_1, v_1 \\
&b_2, v_2 \\
&\ldots \\
&b_n, v_n
\end{align*}
\]

(The \( v_p \)'s \( 1 \leq p \leq m \) are \( n \)-dimensional vectors, the \( b_i \)'s will be called the leaders.)

As in any Blankinship type algorithm we have that \( v_p a^T = b_p, 1 \leq p \leq m \) and \( \gcd(a_1,a_2,\ldots,a_n) = \gcd(b_1,b_2,\ldots,b_n) \). So far the algorithm is similar to the Blankinship type algorithm described above. We depart now from the deterministic algorithm by introducing a random decision procedure which sets the vectors to be subtracted at each step. Many such decision procedures are possible. The algorithm \( C \) described below uses a straightforward procedure.

I. Repeat

Split the interval of the (remaining) \( b_i \)'s into two equal subintervals and divide the \( b_i \)'s into two sets accordingly. Call the upper set \( U \). Divide \( U \) into two sets: \( \text{New} \) including vectors created in the previous iteration and \( \text{Old} \) including all other members of \( U \).

Select randomly a row headed by \( b_i \) from \( \text{New} \) and a row headed by \( b_j \) from \( \text{Old} \), and subtract the \( b_j \) row from the \( b_i \) row. If the resulting \( b_i \) is negative multiply the row by \( -1 \). Since both \( b_i \) and \( b_j \) are in the upper subinterval the resulting \( b_i \) will be in the lower subinterval. After finitely many steps all the \( b_i \)'s in \( \text{New} \) are moved to the lower set.

Select randomly pairs of rows headed by \( b_i \) and \( b_j \) in \( \text{Old} \), and subtract the \( b_j \) row from the \( b_i \) row. If the resulting \( b_i \) is negative multiply the row by \( -1 \). Since both \( b_i \) and \( b_j \) are in the upper subinterval the resulting \( b_i \) will be in the lower subinterval. After finitely many steps all the \( b_i \)'s in \( \text{Old} \) except one are moved to the lower set. (For the time being) Ignore the leftover element in \( \text{Old} \).

Until the lower half of the interval has size 1.

There are three possibilities for stopping:

1. If one of the resulting \( b_i \) is equals 1 then that value is the \( \gcd \) and its corresponding vector solves the given problem.

2. After a certain number of iterations all the \( b_i \)'s equal 0. In this case we cannot continue and the last discarded integer might be the \( \gcd \). We will show that the probability of this to happen tends to 0 when the number of integers at input tends to \( \infty \).

3. After a certain number of iterations \( \text{Old} \) is empty but \( \text{New} \) is not. In this case we cannot continue and the algorithm fails. As we shall show the probability that this might happen is small, and accordingly the probability that the algorithm fails as result of (3) is low. In fact this probability tends to 0 when the number of integers at input tends to \( \infty \).

If the algorithm stops as a result of (2) or (3) we can continue by an ordinary deterministic algorithm.

In order to keep the description of the algorithm simple we omitted from it the handling of the attached coefficient vectors. Notice that any operation on the \( b_i \)'s must be performed on the attached vectors \( (v_i)'s \) too. (i.e. let \( v_1 \) and \( v_2 \) be the attached vectors of \( b_1 \) and \( b_2 \) respectively, then \( b_3 = b_1 - b_2 \) in the algorithm means that the attached vector of \( b_3 \) will be \( v_1 - v_2 \). However, \( b_1 < b_2 \) holds true if the integer \( b_1 \) is smaller than \( b_2 \) regardless of \( v_1 \) and \( v_2 \).)
The detailed description of the algorithm uses the following variables:

- **k**: \( k = \log_2 \max \{a_i\} \)
- **i**: number of current iteration \((1 \leq i \leq k)\)
- **\(H_i\)**: in the \(i\)-th iteration all the \(b\)'s are between 0 and \(2H_i\) and only those between \(H_i\) and \(2H_i\) are processed
- **\(S_i\)**: the set of \(b\)'s smaller than \(H_i\) that are not processed in the \(i\)-th iteration
- **\(A_i\)**: the set of \(b\)'s after iteration \(i\)
- **\(New_i\)**: the set of \(b\)'s greater or equal \(H_i\) that were created in the \((i-1)\)-th iteration
- **\(Old_i\)**: the set of \(b\)'s greater or equal \(H_i\) that are not in \(New_i\)
- **\(NC_i\)**: the set of \(b\)'s created in iteration \(i\) by the C loop (see definition below)
- **\(ND_i\)**: the set of \(b\)'s created in iteration \(i\) by the D loop (see definition below)

**I.e.** \(New_i \subset NC_{i-1} \cup ND_{i-1}\) and \(A_i = S_i \cup NC_i \cup ND_i\).

**INITIALIZATION**

A: input \(a_1, a_2, \ldots, a_n\)

\[
k := \lceil \log_2 (\max \{a_i\}) \rceil
\]

\[
H_0 := 2^k
\]

\(i := 0\)

\[
S_0 := \{a_1, a_2, \ldots, a_n\}
\]

\(NC_0, ND_0 := \emptyset\)

B: \(i := i + 1\)

\[
H_i := H_{i-1}/2
\]

\(Old_i := \{a \in S_{i-1} \mid a \geq H_{i-1}\}\)

\(S_i := \{a \mid a \in S_{i-1} \cup NC_{i-1} \cup ND_{i-1} \text{ and } a < H_i\}\)

\(New_i := \{a \mid a \in NC_{i-1} \cup ND_{i-1} \text{ and } a \geq H_i\}\)

\(NC_i, ND_i := \emptyset\)

BF: if \(Old_i = \emptyset\) and \(|New_i| > 1\) then stop. (* Failure *)

**THE C LOOP**

C: while \(New_i \neq \emptyset\) do

begin

\[
a := \text{Random}(New_i)\quad (* \text{chose at random an element of } New_i *)
\]

\(New_i := New_i - \{a\}\quad (* \text{delete } a \text{ from } New_i *)\)

\[
b := \text{Random}(Old_i)\quad (* \text{chose at random an element of } Old_i *)
\]

if \(a - b \geq 0\) then

\[
NC_i := NC_i \cup \{a - b\}
\]

else \(NC_i := NC_i \cup \{b - a\}\)

end

**THE D LOOP**

D: while \(|Old_i| > 1\) do

begin

\[
(a,b) := \text{Random2}(Old_i)\quad (* \text{chose at random two different elements of } Old_i *)
\]

\(Old_i := Old_i - \{a\}\quad (* \text{delete } a \text{ from } Old_i *)\)

if \(a - b \geq 0\) then

\[
ND_i := ND_i \cup \{a - b\}
\]

else \(ND_i := ND_i \cup \{b - a\}\)

end

(*If Old was not empty then at this point there is exactly one element in it; this element is discarded.*)
E: if $1 \in S_i \cup NC_i \cup ND_i$ then stop  
    if $H_i = 2$ then stop  
    go to B.  

Note: the sets were subscribed by the iteration number in order to help clarify the proofs. It is not necessary to have more than 4 sets: S, Old, New, NN (where NN contains the elements of NC and ND), the other sets were introduced in order to keep the presentation clear. Similarly a single H variable is sufficient.

The complexities of algorithms B and C are identical. The number of iterations, if the deterministic stage is not invoked, is bounded by $k = \log_2 H_0 - 1$, at most the length of the longest integer at input. In each iteration there are at most $n$ ($n/2$ on the average) vector subtractions. Thus the overall vector operations are less than $nk = O(nL)$. Where $L$ is the maximal length of an integer at input. Note that there are no multiplications and no divisions in these algorithms. The total number of operations, if the deterministic stage is not invoked, is therefore $O(n^2k)$ and all the operations are additions/subtractions only. The given complexity can be achieved provided that an appropriate data-structure is used.

3. PROBABILISTIC ANALYSIS.

3.1. FINDING THE DENSITY FUNCTIONS.

In the last section we showed that algorithm C stops after at most $n^2k$ operations provided that the deterministic stage is not invoked. (In the sequel we will assume that this is indeed the case and we will consider the other case as failure). We showed also that the gcd of the b's does not change during the execution of the algorithm. We still have to show that the gcd does not change even though, at the end of loop C, we discard some of the b's. This will be shown in section 4. Algorithm C fails if at iteration $i$ Old$_i$ is empty and New$_i$ has more than one b (see BF in the algorithm). (In algorithm B this cannot happen since it does not discriminate between "old" and "new" b's.)
In the next section we show that if the number of $b$'s at input tends to $\infty$ then the probability of failure at BF tends to 0. First we show some properties of the distribution of the $b$'s during the execution of the algorithm.

Assuming that the number of $b$'s is big we will approximate the discrete density functions of the $b$'s by continuous functions. Hence we shall use the following notations: Let $f_j$ denote the density function of the $b$'s after the $j$-th iteration. Then denote:

$$\int f_j(x)dx = p_j$$

$$\int f_j(x)dx = q_j$$

$$\int f_j(x)dx = r_j$$

It follows from the above notations that:

$$p_j + q_j + r_j = 1.$$  

Let $X, Y$ be two independent random variables defined over the interval $[H_j, 2H_j]$. Let $f_X f_Y$ be their corresponding density functions over that interval and assume that the functions are derivable.

Let $Z = |X - Y|$ be a derived random variable. Then it is easy to show by textbook standard techniques that:

$$f_Z(z) = \int f_X(x) f_Y(x-z)dx + \int f_X(x) f_Y(x+z)dx$$

The above formula implies the following:

**Corollary 3.1.1:** The density functions $f_j$ satisfy the following recursion:

$$f_{j+1}(x) = f_j(x) + \frac{1}{q_{j-1}} \int f_j(x-z)f_{j-1}(z)dz + \frac{1}{q_{j-1}} \int f_j(x)f_{j-1}(z-x)dz$$

The proof of the corollary is deferred to the appendix.

Using the corollary 3.1.1 one can compute $f_1, f_2, \cdots$ etc. in sequence. E.g. if the initial distribution is uniform then
The actual computation of the functions $f_j$ will result in very complex expressions when $j$ grows large. Fortunately we do not need to perform this computations. We show in the next section that these functions satisfy some simple properties and we will be able to analyze the algorithm based on those properties only.

### 3.2. SOME PROPERTIES OF THE DENSITY FUNCTIONS.

The density functions can be normalized as follows. Define new variables $v_j$ and functions $g_j$ as below:

$$H_j f_j(x) = g_j(v_j) \quad ; \quad v_j = \frac{x}{H_j}$$  \hspace{1cm} (3.2.1)

3.1.6 reduces, by induction, under the above substitution to:

$$g_{j+1}(v_{j+1}) = \frac{1}{2} g_j(v_j) + \frac{1}{4q_{j-1}} \int_{v_j}^{v_{j+1}} g_j(w - v_j) g_{j-1}(\frac{w}{2}) dw$$  \hspace{1cm} (3.2.2)

We can prove now the following:

**Lemma 3.2.1:** If $g_0$ is nonincreasing then for every $j > 0$, $g_j$ is decreasing.

**Proof:** Follows by induction from the derivative of the $g_j$ functions as given in formula 3.2.2. \( \square \)

Lemma 3.2.1 and formula 3.2.2 imply the following properties of the $g_j$ functions:

$$g_{j+1}(1) = \frac{1}{2} g_j(\frac{1}{2})$$  \hspace{1cm} (3.2.3)

follows by substitution from 3.2.2 ($v_j = \frac{1}{2}$ implies that $v_{j+1} = 1$ by the normalization procedure);

$$g_{j+1}(0) \leq \frac{1}{2} g_j(0) + g_j(\frac{1}{2})$$  \hspace{1cm} (3.2.4)

To prove this property, substitute 0 for $v_{j+1}$ in 3.2.2 resulting in:
The monotonicity of \( g_j \) implies that \( g_j \left( \frac{1}{2} \right) \geq g_j(w) \) for any \( w, 0 \leq w \leq 1 \), and therefore

\[
g_{j+1}(0) = \frac{1}{2} g_j(0) + \frac{1}{4q_{j-1}} \int_{1/2}^1 \left[ g_j(w)g_{j-1}(\frac{w}{2}) \right] dw + \frac{1}{4} \int_{1/2}^1 g_j(w)g_{j-1}(\frac{w}{2}) dw
\]

The monotonicity of \( g_j \) implies that \( g_j \left( \frac{1}{2} \right) \geq g_j(w) \) for any \( w, 1/2 \leq w \leq 1 \), and therefore

\[
g_{j+1}(0) = \frac{1}{2} g_j(0) + \frac{1}{4q_{j-1}} g_j(\frac{1}{2}) \int_{1/2}^1 g_{j-1}(w) dw
\]

\[
= \frac{1}{2} g_j(0) + g_j(\frac{1}{2})
\]

as required.

Denote as before by \( p_j, q_j \) and \( r_j \) the area under the \( g_j \) function over the intervals \([0,1/4],[1/4,3/4]\) and \([3/4,1]\) respectively with \( p_j + q_j + r_j = 1 \). As \( g_j \) is nonincreasing we have that:

\[
p_j \geq q_j ; 2q_j \geq r_j ; p_j \leq g_j(0) \quad ; q_j \geq g_j(1/2) \quad ; r_j \geq g_j(1) / 2
\]

3.2.5

We are now ready to prove the main result of this section namely:

**Lemma 3.2.2:** For every \( j \geq 0 \) \( q_j \geq 1/9 \) provided that \( g_0 \) is nonincreasing and \( g_0(0) \leq 8/3 \).

**Proof** Consider the two statements below:

\( S1_j : q_j \geq 1/9 \)

\( S2_j : g_j(0) \leq 8/3 \)

\( S2_j \) and 3.2.5 imply that \( p_j \leq 3/4 \). Therefore by 3.2.5 we have that \( 3q_j = q_j + 2q_j \geq q_j + r_j = 1 - p_j \geq 1 - 3/4 = 1/4 \) or \( q_j \geq 1/4 \). We have thus shown that

For every \( j \) \( S2_j \) implies \( S1_j \)

3.2.6

From the condition of the lemma and from 3.2.6 we have that \( q_0 \geq 1/9 \).

We will show now that if for some \( j \) \( S2_j \) is true then either \( S2_{j+1} \) is true or \( S1_{j+1} \) and \( S2_{j+2} \) are both true. This implies, by 3.2.6, that \( S1_j \) is true for all \( j \) as required (notice that the inductive argument may pass over some \( S2_j \)'s when moving from one \( j \) to the next one, still the \( S1_j \)'s are all accounted for).
Assume that \( S_{2j} \) is true but \( S_{2j+1} \) is not. Then \( g_j(0) \leq 8/3 \) and \( g_{j+1}(0) > 8/3 \). From property 3.2.4 we get that \( g_j(1/2) \geq g_{j+1}(0) - 1/2 g_j(0) > 8/3 - 8/6 = 4/3 \) or \( g_j(1/2) = 4s/3, s > 1 \). The following derivations are implied by 3.2.3, 3.2.6 and lemma 3.2.1:

\[
\begin{align*}
g_{j+1}(1) &= \frac{1}{2} g_j\left(\frac{1}{2}\right) = \frac{2s}{3} \\
g_{j+1}\left(\frac{1}{2}\right) &\geq g_{j+1}(1) = \frac{2s}{3} \\
q_{j+1} &\geq g_{j+1}\left(\frac{1}{2}\right) = \frac{2s}{12} > \frac{1}{9}.
\end{align*}
\]

Thus \( S_{1j+1} \) holds.

But we have also the following derivations by 3.2.5 and 3.2.3:

\[
r_{j+1} \geq \frac{1}{2} g_{j+1}(1) = s/3.
\]

Therefore \( 2q_{j+1} \leq p_{j+1} + q_{j+1} = 1 - r_{j+1} \leq 1 - s/3 \) and \( g_{j+1}(1/2) \leq 4q_{j+1} \leq 2(1-s/3) \) (by 3.2.5). But, by the assumptions and 3.2.4 we have that \( g_{j+1}(0) \leq \frac{1}{2} g_j(0) + g_j(1/2) \leq 4(s+1)/3 \) resulting in \( g_{j+2}(0) \leq \frac{1}{2} g_{j+1}(0) + g_{j+1}(1/2) \leq 2(s+1)/3 + 2(1-s/3) = 8/3 \) (by 3.2.4) and this is \( S_{2j+2} \). □

3.3. THE PROBABILITY OF FAILURE.

As discussed in section 2, there are two cases in which algorithm C can "fail" (see description of the algorithm).

a. at step BF in the algorithm

b. if after a certain iteration all the \( b \)'s are equal to zero. In this case if the last discarded integer divides all input integers then it is the \( \text{gcd} \) otherwise the algorithm fails.

We will show now that when \( n \) tends to \( \infty \) then the probability of failure of either type tends to 0.

Failure of the a type happens when there are no \( b \)'s in \( \text{Old} \) and there are at least two \( b \)'s in \( \text{New} \). This can happen only if in the previous iteration the section \( [\frac{H}{4}, \frac{H}{2}] \) was empty. Let \( n_j \) be the number of \( b \)'s that participate in the \( j \)'th iteration. Failure of type b happens when all the \( b \)'s are zero and thus section \( [\frac{H}{4}, \frac{H}{2}] \) is empty. The following inequality is therefore implied by lemma 3.2.2.
3.3.1 Therefore

\[ Pr(\text{the algorithm fails at iteration } j) \leq (p_{j-1} + q_{j-1})^n \leq \left( \frac{8}{9} \right)^n. \]

Therefore

\[ Pr(\text{fail}) \leq \sum \left( \frac{8}{9} \right)^n \leq k \left( \frac{8}{9} \right)^{n'}. \]

Where \( n' \) is the number of \( b \)'s in the last iteration.

For any given \( \varepsilon > 0 \), in order to get \( k \left( \frac{8}{9} \right)^{n'} \leq \varepsilon \) we must have that \( \log k + n' \log (8/9) \leq \log (\varepsilon) \) (all logarithms are base 2 unless otherwise specified). This reduces to \( n' \log (8/9) \geq \log k + \log (1/\varepsilon) \) or \( n' \geq \log_{8/9} k + \log_{8/9} (1/\varepsilon) \). Since \( n' \geq n - k \), if \( n \geq \log_{8/9} k + \log_{8/9} (1/\varepsilon) + k = \log_{8/9} (k + 1/\varepsilon) + k \) then \( n' \) satisfies the required inequality. This proves the following:

Proposition 3.3.1: For every \( \varepsilon > 0 \), if \( n \geq \log_{8/9} (k + 1/\varepsilon) + k \) then the probability that the algorithm will fail is less than \( \varepsilon \).

Remark: The "density" of a set of numbers \( a_1, a_2, \ldots, a_n \) (see [LAG]) is defined as

\[ d = \frac{n}{\log_2 (\max a_j)} \]

We can now reformulate proposition 3.3.1 in terms of the density of the integers at input:

"Given \( \varepsilon > 0 \), if the density of the integers at input is \( d \) with \( d = n/k \geq \frac{1/\varepsilon}{k} + 1 \) then the probability that the algorithm fails is less than \( \varepsilon \)."

The parameter \( k \) in proposition 3.3.1 was defined as \( k = \log (\max a_j) \). It follows therefore from the above analysis that algorithm C succeeds with high probability when the density of the input integers is greater than 1 and the probability of success increases with the density. If the density is fixed and close to 1 then the probability of success increases with \( \max a_j \).

4. AN ALGORITHM THAT FINDS A BASIS WITH SMALL COEFFICIENTS FOR THE SET OF ALL SOLUTIONS.
Given the set of integers \((a_1, a_2, \ldots, a_n)\) at input it is required for some applications to find a basis for the set of all solutions of the equation

\[
\sum_{i=1}^{n} a_i x_i = \text{gcd}(a_1, a_2, \ldots, a_n)
\]

If this is the case then we cannot afford to discard integers as is done by Algorithm C (see stage D in its detailed description). A slight modification of Algorithm C described below as Algorithm T will take care of the discarded integers and produce a basis as above with small coefficients.

It follows from the previous section that if Algorithm C terminates successfully (which happens with high probability) then at termination, there are \(k\) discarded integers (with corresponding coefficient vectors), one in each interval \([2^j, 2^{j+1})\). Denote those integers by \(b_0, b_1, \ldots, b_k\) with \(2^{j+1} \leq b_i < 2^{j+2}\). Algorithm T processes those integers as follows:

\[
\text{TA for } m=k \text{ to } 1 \text{ by } -1 \text{ do begin}
\text{TB. while } b_m \geq 2^m \text{ do } b_m := \left| b_m - b_{m-1} \right|
\text{ for } j=m-2 \text{ to } 1 \text{ by } -1 \text{ do if } b_m \geq 2^{j-1} \text{ then } b_m := \left| b_m - b_j \right|
\text{end}
\]

Note that loop TB will be executed at most 3 times, as follows from the definitions.

We can now prove the following:

**Proposition 4.1:** By executing algorithm T after algorithm C is completed the highest coefficient grows by a factor bounded by \(\sqrt{5}\). (i.e. If \(v_i\) is a vector discarded by algorithm C and processed by algorithm T and \(v'_i\) is the resulting vector after algorithm T is completed, then the ratio between the entries of \(v'_i\) and the entries of the corresponding \(v_i\) is bounded by \(\sqrt{5}\).)

**Proof:** Note that the highest coefficient for \(b_k\) and \(b_{k-1}\) is 1 and for any \(b_j \) (for \(j \leq k-2\)) the highest coefficient is less than \(\text{fib}(k-j-2)\) where \(\text{fib}(0)=\text{fib}(1)=1\) and \(\text{fib}(i)\) is the \(i\)-th member of the Fibonacci series.

For every \(1 \leq m \leq k\) algorithm T reduces \(b \sum m\) to 0 by subtracting from it \(b_{m-1}\) at most 3 times and every other \(b_j\), \(0 \leq j \leq m-1\), at most once. Therefore the absolute value of the entries of the vector attached to \(b_m\) will grow during this process to a value bounded by 

\[
\text{fib}(k-m-2)+3\text{fib}(k-m-1)+\sum_{j=0}^{m-2} \text{fib}(k-j-2)\leq\text{fib}(k-m-2)+3\text{fib}(k-m-1)+\text{fib}(k)\text{-fib}(k-m+1)=\text{fib}(k)+\text{fib}(k-2)<\sqrt{5}\times0.62(m)
\]
Corollary 4.1: Algorithm T when executed after algorithm C is completed produces a basis \( v_1, \ldots, v_n \) for the set of solutions of the equation 1.1 such that \( \sum |x_i| < \sqrt{5} (\max a_i)^{0.694} \).

Proof: The vectors produced by algorithm C satisfy \( \sum |x_i| \leq 1.004 (\max a_i)^{0.694} \). This can be shown by using the same inductive proof used in section 2.6, except that the base of induction is now \( m_1 = 1 \) and \( m_1 = 2 \). After algorithm T is completed the resulting vectors will satisfy the inequality

\[
\sum |x_i| \leq 1.004 \sqrt{5} (\max a_i)^{0.694} < 2.25 (\max a_i)^{0.694}.
\]

Proof similar to that of proposition 4.1. □

Remark: An example where the algorithm fails can be constructed as follows: Let \( a_1, a_2, \ldots, a_n \) be subsequent elements of the Fibonacci series \( \text{fib}(1), \ldots, \text{fib}(l+n) \). It can be shown that for this sequence \( \gcd(a_1, a_2, \ldots, a_n) = 1 \), a solution with maximal coefficient \( \text{fib}(l-1) \) exists but the algorithm fails on the above sequence. Notice that the density of \( \text{fib}(l), \ldots, \text{fib}(l+n) \) is

\[
\frac{n}{(l+n) \log 2 \left[ 1 + \frac{1 + \sqrt{5}}{2} \right]} < 1.44.
\]

5. CONCLUSION.

The results obtained in this paper can be summarized as below:

Let \( \varepsilon > 0 \) and let \( a_1, a_2, \ldots, a_n \) be a sequence of integers whose distribution over the interval \([0, 2^{k+1})\) is uniform or nonincreasing.

1. If the density \( d \) of the integers \( a_1, a_2, \ldots, a_n \) at input satisfies the inequality

\[
d \geq 1 + \frac{\log_2 (k+1) - \varepsilon}{k}
\]

then the probability of success of algorithm C is greater than \( 1 - \varepsilon \).

2. The algorithm C described in section 2.6 produces at output a sequence of integers \( x_1, x_2, \ldots, x_n \) satisfying the equation 1.1.
(3) The integers $x_1, x_2, \cdots, x_n$ produced by the algorithm satisfy the inequality
$$\max_{j=1, 2, \cdots, n} |x_j| < 0.62 (\max_{j} a_j)^{0.694}.$$ 

(4) All the vectors produced by algorithm C satisfy the inequality
$$\sum |x_i| \leq 1.004 (\max_{j} a_j)^{0.694}.$$ 

(5) The algorithm C combined with the algorithm T produces a basis for the set of all the solutions of equation 1.1.

(6) All the vectors produced by algorithm C combined with algorithm T satisfy the inequality
$$\sum |x_i| \leq 2.25 (\max_{j} a_j)^{0.694}.$$
APPENDIX - PROOF OF COROLLARY 3.1.1:

In order to prove corollary 3.1.1 we use the definitions below:

- \( f_j \) denotes the density function of the \( b \)'s after iteration \( j \) (i.e. the density function of the elements of \( A_j \)).

- \( f_{NC_j} \) - the density function of the \( b \)'s in the set \( NC_j \)
- \( f_{ND_j} \) - the density function of the \( b \)'s in the set \( ND_j \)
- \( f_{New_j} \) - the density function of the \( b \)'s in the set \( New_j \)
- \( f_{Old_j} \) - the density function of the \( b \)'s in the set \( Old_j \)
- \( f_{S_j} \) - the density function of the \( b \)'s in the set \( S_j \)

The respective probability functions will be denoted by \( F \) with a corresponding subscript.

Note that \( H_j \) is the upper bound for the \( b \)'s created at the \( j \)-th iteration, thus \( H_{j+1} = \frac{H_j}{2} \) and

- \( f_j(x) = 0 \) if \( x \in [0,H_j] \)
- \( f_{New_j}(x) = 0 \) if \( x \in [H_j,2H_j] \)
- \( f_{NC_j}(x) = 0 \) if \( x \in [0,H_j] \)
- \( f_{ND_j}(x) = 0 \) if \( x \in [0,H_j] \)
- \( f_{Old_j}(x) = 0 \) if \( x \in [H_j,2H_j] \)
- \( f_{S_j}(x) = 0 \) if \( x \in [0,H_j] \)

Assuming that the number of \( b \)'s is big we will approximate the density functions by continuous functions rather then discrete ones.

Some additional notations are needed:

\[
\int_0^{H_{j+1}} f_j(x) dx = p_j
\]
It follows from the above notations that:

\[ p_j + q_j + r_j = 1. \]

Next we consider the changes induced in the functions \( f_j \) by consecutive iterations.

**Lemma A.1:** Let \( X, Y \) be two independent random variables. Let \( f_X, f_Y \) be their corresponding density functions defined over \([H_j, 2H_j]\) and assume that the functions are derivable. Then \( f_Z \), the density function of the random variable \( Z = |X - Y| \), is the function given below defined over \([0, H_j]\).

\[
f_Z(z) = \begin{cases} 
\frac{2H_j}{H_j} f_Y(x) f_X(x-z) dx + \frac{2H_j}{H_j} f_X(x) f_Y(x-z) dx 
\end{cases}
\]

A6

The proof is standard and is left to the reader.

**Lemma A.2:** The density functions \( f_j \) satisfy the recursion below:

\[
f_{j+1}(x) = f_j(x) + r_{N_j} f_{NC} + q_{j-1} f_{ND}
\]

A7

**Proof:** (see Figure A.1).
The proof is implied by the following facts:

The C loop is executed $|\text{New}|$ times and therefore $|\text{New}|$ b's are created by this loop.

The D loop is executed $|\text{Old}| - 1$ times and therefore $|\text{Old}| - 1$ b's are created by this loop. The distributions are not affected by the ignored integer if the number of integers is large.

At the end of the $(j+1)^{th}$ iteration there are three types of b's, in accordance with the three sum-mands in A.7:

1. Those that did not change during the iteration. Their density function is $f_j$.
2. Those created in loop C, whose density function is $f_{\text{NC}}$.
3. Those created in loop D, whose density function is $f_{\text{ND}}$.

Thus the density function of the b's at the end of the $(j+1)^{th}$ iteration is therefore, the weighted sum of the above three density functions.

The relative weight of $f_{\text{NC}}$ is equal to the area below $f_{\text{New}}$ over $[H_{j+1}, 2H_{j+1}]$, i.e. $r_{\text{NC}}$.

The relative weight of $f_{\text{ND}}$ is equal to the area below the density function of the old b's in the range $[H_{j+1}, 2H_{j+1}]$. Note that the density function of the old b's is $f_{j-1}$ and $[H_{j+1} - H_{j-1} - 1, H_{j-1}] = [H_{j+1}, 2H_{j+1}]$ the relative weight of $f_{\text{ND}}$ is equal to $q_{j-1}$. $\square$
Corollary A.1: The density functions $f_{j}$ satisfy the following recursion:

$$f_{j+1}(x) = f_{j}(x) + \frac{1}{q_{j-1}} \int_{z=d_{j}+\epsilon}^{2H_{j}} f_{j}(z-x) f_{j-1}(x) dz + \frac{1}{q_{j-1}} \int_{z=d_{j}+\epsilon}^{2H_{j}} f_{j}(z) f_{j-1}(z-x) dz \quad \text{A.8}$$

Proof: by lemma A.1 we get:

$$f_{NC_{m}}(x) = \int_{z=d_{j}+\epsilon}^{2H_{j}} \int_{z=d_{j}+\epsilon}^{2H_{j}} f_{Old_{m}}(z) f_{New_{m}}(z-x) dz + \int_{z=d_{j}+\epsilon}^{2H_{j}} f_{New_{m}}(z) f_{Old_{m}}(z-x) dz \quad \text{A.9}$$

and

$$f_{ND_{m}} = 2 \int_{z=d_{j}+\epsilon}^{2H_{j}} \int_{z=d_{j}+\epsilon}^{2H_{j}} f_{Old_{m}}(z) f_{Old_{m}}(z-x) dx \quad \text{A.10}$$

Notice that $Old_{j+1}$ consists of the $b$'s in $S_{j}$ that are greater than $H_{j+1}$, and $S_{j}$ consists of the $b$'s in $A_{j-1}$ which were smaller than $H_{j}$. Thus

$$f_{Old_{m}}(x) = \frac{1}{q_{j-1}} f_{j-1}(x) \quad \text{A.11}$$

Notice also that $New_{j+1} = A_{j} - A_{j-1}$ and therefore

$$f_{New_{m}}(x) = \frac{1}{r_{N_{m}}} (f_{j}(x) - f_{j-1}(x)) \quad \text{for} \quad \frac{H_{j}}{2} \leq x \leq H_{j} \quad \text{A.12}$$

Substituting A.11 into A.10 and A.9, A.10, A.12 into A.7 and simplifying the expression results in A.8 which completes the proof. $\square$
REFERENCES


