An Improved Deterministic Algorithm for Generating Different Many-Element Random Samples

Amihood Amir[*]
Georgia Tech
and
Bar-Ilan University

Emanuel Dar[†]
Bar-Ilan University

November 19, 1996

Abstract

We consider the problem of *deterministically* selecting s uniformly random different m-element subsets of $\{1, ..., k\}$. The only known lower bound for the time to solve this problem is the trivial $\Omega(sm)$. The best two previously known solutions are of time $O(sm^3 \log m \log\log m)$ and $O(s(k+m))$, respectivly. In this paper we present an algorithm whose time comlexity is $O(s^2m^2 + sm^2 \log m \log\log m + sm \log sm)$. Thus, for $s < m \log m \log\log m$ this algorithm is the fastest known deterministic algorithm.

The main idea of the algorithm is using a uniform random number generator to efficiently construct biased random numbers.

1 Introduction

When writing heuristics or randomized algorithms one needs efficient souces of randomness. We distinguish between three possible levels:

1. Given an integer k, generate a random *integer* r, $1 \leq r \leq k$.
2. Given integers k, m, generate a *set* of m different integers chosen randomly between 1 and k.
3. Given integers k, s, m, generate a *set whose elements are s different m-element subsets of* $\{1, ..., k\}$.

Problem 1 is a clasic problem in computer science. The only use we make of it is the assumption that there exists a function `RandInt(`x`)` that returns a random integer between 1 and x. We will also assume that such a random integer is generated in unit time.

Problem 2 has proven very popular. Bentley devoted four *Programming Pearls* columns to it ([3, 6, 5, 4]), culminating with an algorithm by Floyd ([4]). Floyd's algorithm generates a random subset of $\{1, ..., k\}$

*Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat Gan, ISRAEL; (972-3)531-8770; amir@cs.biu.ac.il; Partially support by NSF grant CCR-92-23699 and Israel Ministry of Science and Arts grant 6297.
†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat Gan, ISRAEL; dar@macs.biu.ac.il;

using m calls to `RandInt`. Its deterministic time is $O(m \log m)$ but can be done in time $O(m)$ using hashing. Algorithms with the same time complexity but that produce a random sample (i.e. the elements in each set are given in random order) rather than a random set appeared independently in [1, 7]. A parallel algorithm appeared in [13].

We are interested in an efficient *deterministic* solution to problem 3. There are many applications that require an efficient generation of random sets. We were motivated by robust estimators for surface fitting (e.g. [11, 8, 14, 16]).

Our solution is based on a novel idea of using a uniform random number generator to produce biased number generation. We provide efficient algorithms for this task.

Problem Definition:

Let us formally define problem 3. Let $P_a(X)$ be the set of a-element subsets of X. and let $N_k = \{1, \ldots, k\}$. We are interested in generating a random, uniformly chosen element of $P_s(P_m(N_k))$. Our algorithm will, in fact, produce both the elements of $P_m(N_k)$ and the elements of $P_s(P_m(N_k))$ in a random order, i.e. it will produce a *random sample*.

The previously mentioned algorithms for problem 2 ([4, 1, 7, 13]) can also be used to solve a special case of the problem. By successive application of the algorithm, mutually disjoint random m-element sets can be chosen.

This paper presents a solution to the more generalized problem where the sets are different but not necessarily disjoint.

A straight-forward way of solving the problem is by repeatedly generating random elements of $P_m(N_k)$ and adding them to our sample as long as there is no conflict (*i.e.* the added sample did not appear previously). This algorithm is randomized and works well with high probabilty, especially where k is very large relative to s and m. Since sm numbers need to be generated, it is an interesting question whether such a sample can be generated *deterministically* in time $O(sm)$, assuming that `RandInt` operates in constant time. This problem is still open, but deterministic solutions exist.

All previous solutions (see [2]) involved combining efficient methods for generating random sets ([4, 1, 7]) with set *unranking* algorithms ([12, 9, 10]). A *ranking* is a bijection from the set of m-element subsets of $\{1, \ldots, k\}$ to the set of numbers $\{1, \ldots, \binom{k}{m}\}$. *Unranking* is the inverse function, i.e. recovering the set from the number.

Other than the fact that, for large k, all above algorithms are far from the trivial lower bound of $\Omega(sm)$, they use the powerful *numerical analysis* model where arithmetic operations $(+, -, \times, /, \exp, \log)$ are done in unit cost for numbers of all sizes[1].

Let us translate the complexity from the numerical analysis model to the standard *work model* where each of the four basic arithmetic operations takes unit time for words of $O(\log k)$ bits. Addition of $O(m \log k)$ bit numbers can be done in time $O(m)$ and multiplication of such numbers can be achieved in time $O(m \log m \log\log m)$ using Schönhage-Strassen techniques [15]. There are sm^2 multiplications in the algorithm of [2], thus the time that algorithm spends on multiplication alone is $O(sm^3 \log m \log\log m)$ operations. The log function is used sm times, but simulating it costs $O(m^2 \log m \log\log m)$ basic operations. Thus the total algorithm time of the algorithm in [2] is $O(sm^3 \log m \log\log m)$.

[1] It is true that if $x^{1/m}$ can be computed in time $O(m)$ the exp and log operations are not needed in [2], however, we are not aware of such a method.

We present here a simpler algorithm. Our algorithm does not require powerful unit-cost operations such as *log* or *exp*. The *main contribution* of this paper is a *novel idea* for generating random sets that does not involve unranking. We show how to efficiently generate biased random numbers using an unbiased random number generator.

Our algorithm's running time is $O(sm \log sm + s^2 m)$ on the numerical analysis model. In the work model it is $O(sm^2 \log m \log\log m + s^2 m^2)$. For cases where $s < m \log m \log\log m$ this is the *fastest known* algorithm for the generalized random sampling problem.

For simplicity of the exposition and for ease of comparison with previous algorithms, we will compute the time assuming unit cost for large number basic arithmetic operations. In section 4 we compute the time complexity in the conventional work model.

2 The Algorithm's Idea

The previous algorithms generate a number that represents a *set* at every random call, and their main task is deciphering that set from the number. To avoid unranking, we generate the *numbers* of each set sequentially. In order to assure that the generated sets are indeed uniformly randomly distributed, we need to generate the numbers within a set with a probability conditional on the previously generated numbers.

The following lemma tells us what probability each particular number should have at a given stage in the generation, in order to assure a uniformly random set distribution.

Before the lemma, let us proceed with some definitions and notations that will serve us throughout the paper.

Definitions and Notations: Denote the generated m-element sets by $S_1, ..., S_s$. We will represent the sets by tuples, ordered chronologically by the sequence of their numbers' generation. Denote the first i elements of the jth set by $S_j^i = S_j[1], ..., S_j[i]$.

Let $E^{i,j+1} = \left\{ S \middle| \; S = S_l, \text{ for some } l \in \{1, ..., j\} \text{ and } S_{j+1}^i \subset S \right\}$.

For each $c \in \{1, \ldots, k\}$, let $P_c^{i,j+1} = \{S | \; S \in E \text{ and } c \in S\}$, and let $p_c^{i,j+1} = |P_c^{i,j+1}|$.

In the future, we always deal with the case where i and j are known and fixed. To simplify notation we will henceforth drop there indices from our notation and just refer to E, P_c, or p_c.

Lemma: Suppose that $S_1, ..., S_j, S_{j+1}^i$ have been generated.

In order to guarantee a uniformly random set distribution we should assign the following probability to generating the number $c \in \{1, ..., k\}$ in $S_{j+1}[i+1]$:

If $c \in S_{j+1}^i$ then that probability is 0, else it is

$$\frac{1}{m-i} \frac{\binom{k-i-1}{m-i-1} - |P_c|}{\binom{k-i}{m-i} - |E|}$$

Proof: Since we have already generated i numbers in the current tuple, there remain $m - i$ numbers (out of a $k - i$ size universe) that still need to be generated. This means that there are $\binom{k-i}{m-i}$ such sets. Each of them should have an equal probability, provided they had not appeared as an earlier tuple. $|E|$ is the number of such sets that have already been generated in earlier tuples, so the remaining $\binom{k-i}{m-i} - |E|$ sets should have equal probability of being generated.

We now return to the probability we should assign to a specific number c. If $c = S^i_{j+1}[p]$ for some $p, 1 \le p \le i$, then obviously the probability of generating c should be 0. Consider, then, $c \neq S^i_{j+1}[p],\ p = 1, \ldots, i$.

We have already generated i numbers in the current tuple. Thus the number of sets that c can be in is: $\binom{k-i-1}{m-i-1}$. Of these sets, $|P_c|$ have already been generated, so they should not be repeated. This leaves us with $\binom{k-i-1}{m-i-1} - |P_c|$ possible tuples that $\{S_{j+1}[1], ..., S_{j+1}[i], c\}$ is a subset of.

The probability of generating c should therefore be:

$$\frac{\binom{k-i-1}{m-i-1} - |P_c|}{\binom{k-i}{m-i} - |E|}$$

.

However, we are generating tuples, and not sets. So we should multiply this probability by the probability that c will be at the $(i+1)$st place (out of the $m - i$ remaining positions). This gives us the desired value:

$$\frac{1}{m-i}\frac{\binom{k-i-1}{m-i-1} - |P_c|}{\binom{k-i}{m-i} - |E|}. \qquad \square$$

Armed with the Lemma, we now produce the following simple algorithm.

Algorithm Outline

1. Create an m-element subset of $\{1, ..., k\}$.
2. for $j = 2$ to s do
   (a) for $i = 1$ to m do
      i. Generate a number c by an appropriately weighted random function. Take $S_j[i] \leftarrow c$.
      ii. If $\{S_j[1], ..., S_j[i]\} \not\subset S_l$, for $l = 1, ..., j-1$ then generate a uniformly random $(m-i)$ element sample of $\{1, ..., k\} - \{S_j[1], ..., S_j[i]\}$ and exit loop (a).

      end

   end

End Algorithm

Correctness: Obvious from the construction. The lemma proves that there exists a weighted random function that ensures uniformly distributed m-element tuples.

Time: Step 1 can be done in time of $O(m)$.

We will show that Step 2a can be done in time of $O(sm + m \log sm)$. This step is executed $s - 1$ times.

Total Algorithm Time: $O(s^2 m + sm \log sm)$.

3 Implementation Issues

3.1 The Weighted Random Number Generator

The idea of the algorithm is to generate each tuple one number at a time in time $O(sm)$. This can only be done by a weighted random number generator. We assume that we have a uniform random number generator. Our task now is to efficiently use the uniformly random number generator for generating weighted random numbers.

Assume that we have already generated j tuples. We proved that the probability of generating the number c in the $i+1$ position of the $j+1$ tuple is: $\frac{1}{m-i}\frac{\binom{k-i-1}{m-i-1}-|P_c|}{\binom{k-i}{m-i}-|E|}$. Our strategy is to generate a random number r, $1 \leq r \leq (m-i)\left(\binom{k-i}{m-i}-|E|\right)$ and *translate* this random number to the correct number $c \in \{1,...,k\}$. The translation assumes a logical bin of size $\binom{k-i-1}{m-i-1}-|P_c|$ for each number c and returns the appropriate number c in whose "bin" r happened to fall.

It would seem like we would have too many different bin values, and then would not be able to find the bin efficiently. Fortunately, this is not the case. The bin size is dependent on $|P_c|$. By definition the range of $|P_c|$ is $\{1,\ldots,s\}$.

We implement the translation by constructing $s+1$ sets of numbers $A_0, A_1, ..., A_s$. $c \in A_{p_c}$ if $c \notin S^i_{j+1}$ and c is in p_c of the previously generated j tuples that are in E. We represent sets A_p as arrays.

Implementation of the translation

We separate the translation into two cases. In the first case, the generated number did not appear in an earlier tuple in E, and in the second case the number appears in an earlier tuple. If c has been chosen before, then, $c \in A_{p_c}$ for some $p_c \in \{1,...,j\}$. Let $P = \sum_{p=1}^{j}\left(\binom{k-i-1}{m-i-1}-p\right)|A_p|$. P is the sum of the sizes of all logical bins, thus if $r \leq P$ (r is the generated random number) then the number has been chosen before, else, the number has not appeared yet.

Implementation of the translation in case $r \leq P$

If $r \leq P$ then we have to find the number in the right A_p set.

1. Compute $N \leftarrow \binom{k-i-1}{m-i-1}$.
2. For $l = 1$ to j do
   (a) If $r \leq (N-l)|A_l|$ then
      i. $c \leftarrow A_l[\lceil r/(N-l)\rceil]$.
      ii. Exit.
   (b) Else $r \leftarrow r-(N-l)|A_l|$.

   End

Notes:

1. $\binom{k-i-1}{m-i-1} = \binom{k-i}{m-i}\frac{m-i}{k-i}$, so we pay only a constant amount of time for every generated number.
2. $|A_i|$ can be saved in the header of each set, so it does not need to be computed in every iteration.

Time: The time of generating the number for $r \leq P$ is $O(s)$.

Implementation of the translation in case $r > P$.

$r > P$ means that the generated number c has not appeared before. Therefore $c \in A_0$ and all the logical bins are of the same size.

If all the numbers previously chosen were $\left\{k - \left|\bigcup_{S\in E} S\right| + 1, k - \left|\bigcup_{S\in E} S\right| + 2 \ldots, k\right\}$ then we would be able to compute c easily. Simply take $c = \left\lceil \frac{(r-P)}{\binom{k-i-1}{m-i-1}} \right\rceil$. However, since the previously chosen numbers appeared at random in the range, we need to adjust the c computed above for the c-th *unchosen* number.

Let us assume that we maintain a bijection $Find : \{1, \ldots, k\} - \left|\bigcup_{S\in E} S\right| \to \{1, \ldots, k\}$. Then our translation will be done as follows:

1. $c' \leftarrow \left\lceil \frac{(r-P)}{\binom{k-i-1}{m-i-1}} \right\rceil$
2. $c \leftarrow Find(c')$

In the next subsection we will show how to maintain such a bijection in a manner similar to [1, 7]. The data structure to maintain it is constructed as long as $P \leq r$, and used once when $r > P$ is encountered (once that happens the sequence is easily completed by any algorithm for constructing a single random sample). The deterministic time for computing $Find(c')$ is $O(\log |\cup_{S\in E} S|) = O(\log sm)$.

We conclude:

Time: The time of generating the number for $r > P$ is $O(\log(sm))$.

Note: This type of translation can happen at most once per tuple because once we generate a number that has not appeared before we continue creating an $(m-i)$-element sample of $\{1, ..., k\} - \{S_j[1], ..., S_j[i]\}$ in a uniformly random fashion.

3.1.1 Maintaining a Dynamic Bijection

Consider the following problem. Given a universe $U = \{1, \ldots, k\}$ we would like a data structure that supports the following operations:

1. *Delete* a number i from the universe.
2. *Reinsert* a previously deleted number back into the universe.
3. *Find.* Let $B = \{b_1, \ldots, b_p\}$ be the deleted numbers. We want a bijection $Find : \{1, \ldots, k-p\} \to U - B$ that allows us to efficiently compute $Find(i)$.

A data structure that supports the *Delete* and *Find* requirements was described in [1, 7]. We briefly describe that data structure and show how it can support the *Reinsert* operation as well.

The idea in [1, 7] is to pair each deleted number with one of $\{k-p+1, \ldots, k\}$. We then define

$$Find(i) = \begin{cases} i, & \text{if } i \notin B; \\ i\text{'s pair}, & \text{if } i \in B. \end{cases}$$

The space necessary for the pairs is $O(p)$. If the pairs are kept in a balanced tree the worst case deterministic time for $Find$ is $O(\log p)$. If we use hashing we can compute $Find$ in constant time with high probability.

We define $Delete(b)$, $b \in \{1, \ldots, k-p\}$ to mean that the number $Find(b)$ is deleted from U. We proceed to show the implementation for *Delete* as described in [1, 7].

Implementation of $Delete(b)$, $b \in \{1, \ldots, k-p\}$

1. If there exists a pair $(k-p, l)$ then $pair \leftarrow l$ else $pair \leftarrow k-p$

   (If $k-p$ has been previously deleted, then b's pair will be $k-p$'s former pair. Otherwise $k-p$ itself will be b's pair.)

2. If there exists a pair (b, l') then delete it and create new pair $(l', \texttt{nil})$.

   (b had been previously deleted and its pair was l'. Thus the current delete request actually requests to delete l'. We therefore create a new pair with l' as the left element. Since $l' > k-p$ it will never appear in a $Find$ or *Delete* future request, then its pair may be chosen as empty ($\texttt{nil}$).)

3. If $pair \neq b$ then create pair $(b, pair)$ else create pair $(b, \texttt{nil})$.

4. If $pair \neq b$ and $pair = k-p$ then set a pointer from $k-p$ to the pair $(b, k-p)$.

Important Observations:

1. For any pair (x, y), if $x \in \{k-p+1, \ldots, k\}$ then y is $\texttt{nil}$.

2. For every $x \in \{k-p+1, \ldots, k\}$, either there is a pair $(x, \texttt{nil})$ or a pair (b, x), $b \leq k-p$. In the later case there is a pointer from x to the pair (b, x).

Implementation of $Reinsert(b)$, $b \in B = \{b_1, \ldots, b_p\}$

1. If there exists a pair $(k-p+1, \texttt{nil})$ then destroy pairs (b, l) and $(k-p+1, \texttt{nil})$ and create pair $(k-p+1, l)$.

2. else destroy pairs (b, l) and $(b', k-p+1)$ and create pair (b', l).

Time: The time for *Delete* and *Reinsert* is proportional to searching for a pair, which is $O(\log p)$ in a dynamic balanced tree.

3.2 Getting Ready for the Next Element

Suppose we chose element c for position $i+1$ of tuple S_{j+1}. To get ready for generating the element in position $i+2$ we need to update the set E (discard all tuples where c does not appear) and the A_i lists (to reflect the new E). The implementation below shows how to do these updates in total time $O(sm+m\log sm)$ per tuple S_{j+1}.

Finding the right A_{p_c} set

Updating the A_i sets means moving some elements from one A set to another. We need a fast connection between an element c and its A_{p_c} set. To do this we build a balanced binary search tree where a leaf is: an element c_i, its p_{c_i} number, and a pointer to the appropriate $A_{p_{c_i}}$ set. Every element in the j m-element tuples points directly to its leaf. A number that appears in more than one tuple has each of its occurrences pointing to the same leaf.

The time for finding the right A_{p_c} set for $c \in S_l$, $l = 1, ..., j$, and moving c to another set, if necessary, is $O(1)$.

The time for adding a new element c to the tree is $O(\log(sm))$, as is the time for a tree search. If c is already in the tree then we need to increment by 1 the number of times it appears. This is done by moving the pointer from A_{p_c} to A_{p_c+1}. If c is not in the tree then add a new leaf, set the pointer to A_1, and add c to A_1 set.

We only add a new element to the tree when a new number is generated. Since m numbers are generated every sequence, then the time of adding the m element to the tree is $O(mlogsm)$. For the entire algorithm (s samples) it is $O(sm\log(sm))$.

We now describe how the A_i sets are updated. Note that any changes to the tree that are necessary when a tuple is completed involve changing only values in the current tuple. Thus it no more than doubles the total time.

Updating the A_i sets and E vector.

To update the A_i and E sets effectively, we use lists of indices B_c. Each $c \in A_{p_c}$ points to a list of indices B_c. For $l = 1, ..., j$, $l \in B_c$ iff $c \notin S_l$. We also define a vector $E[1], \ldots, E[s]$ where $E[l] = 1$ iff $S_l \in E$.

Updating the A_i sets.

Suppose that the last element generated was c, and $S^i_{j+1} \cup \{c\} \subseteq S_l$, for some $l \in \{1, ..., j\}$. (If not then we continue generating the tuple randomly). We can reach A_{p_c} in constant time and remove c from A_{p_c}. For each $l \in B_c$, if $E[l] = 1$ then for each $c_i \in S_l$ move c_i from $A_{p_{c_i}}$ to $A_{p_{c_i}-1}$.

Every $c_i \in S_l$, $i = 1, .., m$, $l = 1, ..., j$ can be moved from $A_{p_{c_i}}$ to $A_{p_{c_i}-1}$ at most once in a tuple generation, making the time of updating the A_i sets in a tuple generation $O(sm)$.

Updating the E vector.

For each $l \in B_c$ that has $E[l] = 1$, $l \in B_c$ means that $c \notin S_l$ therefore $S^i_{j+1} \cup \{c\} \not\subseteq S_l$ therefore it should be deleted from E. Set $E[l] \leftarrow 0$.

Building and updating the B_c lists of indices.

The B_c lists contains information about previous tuples. Consequently, during the first tuple generation the B_c's are empty. We update at the end of each tuple generation and not after each element generation.

Consequently, chasing through every element of B_c and comparing it to E adds a total of $O(s^2m)$ to our entire algorithm running time, which is dominated by its overall time complexity.

Suppose that we finished generating the $(j+1)$-st tuple. The updating is done for each $c \notin S_{j+1}$. Simply let $B_c \leftarrow B_c \cup \{j+1\}$. This updating takes a constant amount of work for each element and is done at the end of each tuple generation, thus the total time per tuple is $O(sm)$. The following code describes the process.

1. For each $c \in S_{j+1}$ follow the pointer to its location in the A_{p_c} set and tag it.
2. For each $c \in S_l$, $l = 1, ..., j$ follow the pointer to its location in the A_{p_c}set. If the location is not tagged then update B_c. $B_c \leftarrow B_c \cup \{j+1\}$
3. For each $c \in S_{j+1}$ follow the pointer to its location in the A_{p_c} set and untag it.

We have a direct pointer from each element to its place in the A_i set, so each access is done in constant time. The time of the first step and the last step is $O(m)$. The time of step 2 is $O(jm) = O(sm)$. The total time of one updating of theB_c's is $O(sm)$. We update the B_c's after each tuple generation, so, we update the B_c's s times. Therefore the total time for updating the B_c's in the algorithm is $O(s^2m)$.

4 Complexity in the Conventional Work Model

The largest number we may handle is $O(k^m)$, requiring m computer words.

Generating a large random number is equivalent to generating m small random numbers. Thus generating $O(sm)$ large numbers will take time $O(sm^2)$.

The only large number operations we do are multiplication and addition (as opposed to previous algorithms that require large number *log* and *exp* operations). The number of multiplications is $O(sm)$, which again brings the actual cost up to $O(sm^2 \log m \log\log m)$ and $O(s^2m)$ additions which ends up as $O(s^2m^2)$ word operations.

Finally, If one is willing to sacrifice absolute randomness for faster results and if $k \geq s+m$ then our algorithm gives rise to the following very simple idea:

1. Create the first random sample.
2. Create $(s-1)$ $(m-1)$–element samples.
3. For each of the $s-1$ $(m-1)$-tuples generate the last number, giving all numbers that will complete a previously existing sample probability 0.

The above algorithm can be easily implemented in time $O(s^2m + sm\log(sm))$ without any large word operations. The only problem is that the probability of generating a sample is not equal for all samples, there are some very slight differences in the probabilities (depending on the size of k).

Acknowledgements We thank an anonymous referee for the most thorough refereeing job we have ever seen and Mike Goodrich for his suggestions and help.

References

[1] A. Amir. A pearl diver deals a poker hand. *Intl. J. of Computer Mathematics*, 48(1+2):11–14, 1993.

[2] A. Amir and D. Mintz. An efficient algorithm for generalized random sampling. *Pattern Recognition Letters*, 12:683–686, 1991.

[3] J. Bentley. Programming pearls - a little program, a lot of fun. *Comm. of ACM*, pages 1179–1182, 1984.

[4] J. Bentley and R. Floyd. Programming pearls - a sample of brilliance. *Comm. of ACM*, pages 754–757, 1987.

[5] J. Bentley and D. Gries. Programming pearls - abstract data types. *Comm. of ACM*, pages 284–290, 1987.

[6] J. Bentley and D. Knuth. Programming pearls - literate programming. *Comm. of ACM*, pages 364–369, 1986.

[7] M. Chrobak and R. Harter. A note on random sampling. *Inform. Process. Lett.*, 29:255–256, 1988.

[8] M. A. Fischler and R. C. Bolles. Random sampling consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Comm. of ACM*, June 1981.

[9] G. M. Knott. A numbering system for combinations. *Comm. of ACM*, pages 45–46, 1974.

[10] D. H. Lehmer. *Applied Combinatorial Mathematics*, chapter The Machine Tool of Combinatorics. Wiley, New York, 1964.

[11] P. Meer, D. Mintz, and A. Rozenfeld. Least median of squares based robust analysis of image structure. In *Proc. Image Understanding Workshop*, Pittsburgh, PA, Sept. 1990.

[12] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms*. Academic Press, New York, 2nd edition, 1978.

[13] V. Rajan, R. K. Ghosh, and P. Gupta. An efficient parallel algorithm for random sampling. *Inform. Process. Lett.*, 30:265–268, 1989.

[14] P. J. Rousseuw and A. M. Leroy. *Robust Regression and Outlier Detection*. Wiley, New York, 1987.

[15] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971. MR 45 # 1431.

[16] A. Tirumalai and B. G. Schunck. Robust surface approximation using least median squares regression. Technical Report CSE-TR-13-89, University of Michigan, 1989.