Computing 37, 71 – 84 (1986)

Computing
© by Springer-Verlag 1986

Grid Methods in Simulation and Random Variate Generation

L. Devroye, Montreal

Received July 1, 1985

Abstract — Zusammenfassung

Grid Methods in Simulation and Random Variate Generation. Points can be generated uniformly in a compact set A of R^d by constructing a fine rectangular grid covering A, selecting a grid rectangle, and performing an acceptance test if the rectangle in question is not entirely contained in A. For very fine grids, the acceptance test is needed with very small probability. We look at the storage requirements and expected time performance of this method, and apply it in avoidance problems and in the design of fast generators for random variates with a bounded density on $[0, 1]$.

Gittermethoden für Simulation und Erzeugung von Zufallsveränderlichen. Punkte, die in einer komplexen Teilmenge A von R^d gleich verteilt sind, können auf folgende Weise erzeugt werden: Wir überdecken A mit einem engmaschigen Rechteckgitter und führen einen Annahmetest aus, wenn das betrachtete Rechteck nicht zur Gänze in A enthalten ist. Für sehr engmaschige Gitter wird dieser Annahmetest mit sehr kleiner Wahrscheinlichkeit aufgerufen. Wir untersuchen den Speicherbedarf und die mittlere Zeitkomplexität dieser Methode und wenden das Verfahren auf Probleme mit verbotenen Teilmengen und auf schnelle Generatoren für Zufallsveränderliche mit beschränkter Dichte in $[0, 1]$ an.

1. Introduction

Grids of equi-sized rectangles have a variety of uses as a data structure. They are a valuable tool in data manipulation (bucket sorting and searching), in large volume storage problems (storage of large data bases, storage of satellite picture data), and in operations research (approximate solutions of difficult problems such as the travelling salesman problem can be obtained by a grid method). In this paper, we would like to indicate the value of the grid structure in random variate generation and simulation. Some of the material is not new. What is important is the connection between the performance of a certain algorithm in terms of time and space requirements versus the size of the grid. We will give valuable rules for selecting the grid size as a function of certain performance characteristics. In some cases, such as simulation of the car parking problem, it is shown that the grid method is simply not a good choice. In general however, grid methods are very fast, and the performance improves with increasing size of the grid. We will consider three prototype problems:

1. Generating a point uniformly in a compact set A of R^d.
2. Avoidance problems.
3. Non-uniform random variate generation.

It is perhaps helpful to collect most of the important symbols in a short list for easy reference:

A	compact set of R^d
H	hyperrectangle of R^d
C	grid
C_n	grid of size n
D	directory
k	number of good rectangles in directory
l	number of bad rectangles in directory
N_i	number of grid intervals for i-th coordinate axis
f	density
M	bound on density f
Z	random integer (usually on $1, \ldots, k+l$)
X	random variable (usually possessing a density)

2. Generating a Point Uniformly in a Compact Set

Let us enclose the compact set A of R^d by a hyperrectangle H with sides $h_1, h_2, \ldots, h_d$. Divide each side up into equal intervals, N_i intervals of length $\frac{h_i}{N_i}$ for side h_i. Now, there are three types of grid rectangles, the good rectangles (entirely contained in A), the bad rectangles (those partially overlapping with A), and the useless rectangles (those entirely outside A). Before we start generating, we need to set up an array of addresses of rectangles, which we shall call a directory. For the time being, we can think of an address of a rectangle as the coordinates of its leftmost vertex (in all directions). The directory (called D) is such that in positions 1 through k we have good rectangles, and in positions $k+1$ through $k+l$, we have bad rectangles. Useless rectangles are not represented in the array. Then, the informal algorithm for generating a uniformly distributed point in A is as follows:

```
REPEAT
    Generate an integer $Z$ uniformly distributed in $1, 2, \ldots, k+l$.
    Generate $X$ uniformly in rectangle $D[Z]$ ($D[Z]$ contains the address of
    rectangle $Z$).
    Accept $\leftarrow [Z \leq k]$ (Accept is a boolean variable.)
    IF NOT Accept THEN Accept $\leftarrow [X \in A]$.
UNTIL Accept
RETURN $X$
```

See Fig. 1 for an example of a grid and corresponding directory in R^2.

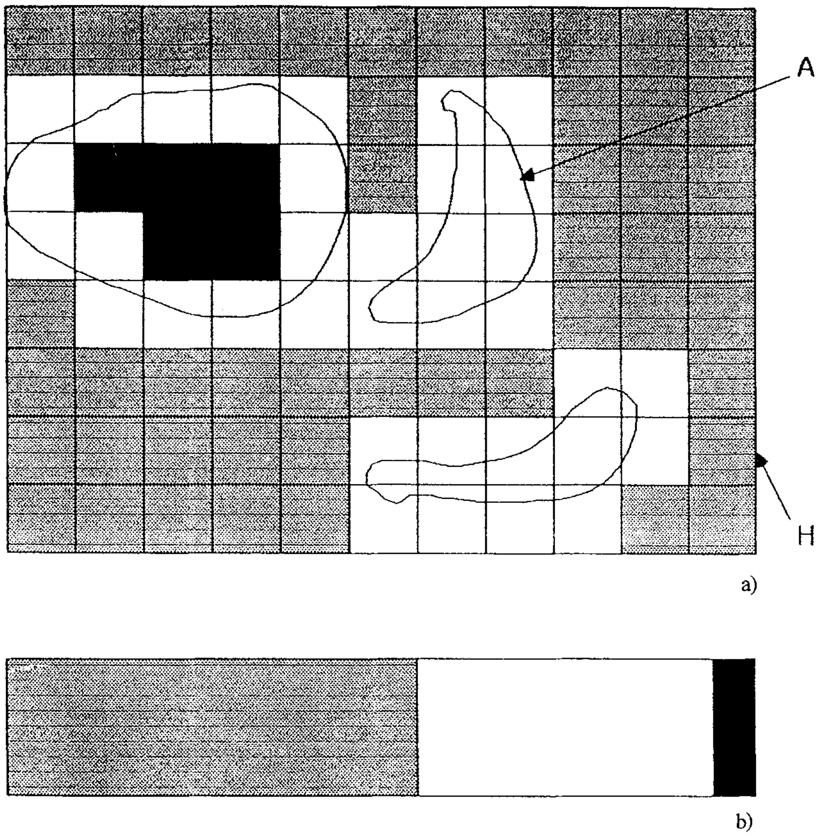

Fig. 1. a) Good rectangles in black; bad rectangles in white; useless rectangles (not stored) in gray. b) Directory of Addresses: 5 good rectangles (black); 35 bad rectangles (white); 48 useless rectangles shown in gray need not be stored

The expected number of iterations is equal to

$$\frac{\text{area}(C)}{\text{area}(A)}$$

where C is the union of the good and bad rectangles (if the useless rectangles are not discarded, then $C = H$). If the area of one rectangle is a, then area $(C) = a(k+l)$. For most bounded sets A, this can be made to go to 1 as the grid becomes finer. That this is not always the case follows from this simple example: let A be $[0,1]^d$ union all the rational vectors in $[1,2]^d$. Since the rationals are dense in the real line, any grid cover of A must necessarily cover $[0,1]^d$ and $[0,2]^d$, so that the ratio of the areas is always at least 2^d. Fortunately, for all compact (i.e., closed and bounded) sets A, the given ratio of areas tends to one as the grid becomes finer (see Theorem 1).

The speed of the algorithm follows from the fact that when a good rectangle is chosen, no boundary checking needs to be done. Also, there are many more good rectangles than bad rectangles, so that the contribution to the expected time from boundary checking is small. Of course, we must in any case look up an entry in a directory. This is reminiscent of the urn or table look-up method and its modifications (such as the alias method (Walker, 1977) and the alias-urn method (Peterson and Kronmal, 1982)). Finer grids give faster generators but require also more space.

One of the measures of the efficiency of the algorithm is the expected number of iterations. We must make sure that as we let the grid become finer and finer, this expected number tends to one.

Theorem 1:

Let A be a compact set of nonzero area (Lebesgue measure), and let us consider a sequence of grids $G_1, G_2, \ldots$ which is such that as $n \to \infty$, the diameter of the prototype grid rectangle tends to 0. If C_n is the grid cover of A defined by G_n, then the ratio

$$\frac{\operatorname{area}(C_n)}{\operatorname{area}(A)}$$

tends to 1 as $n \to \infty$.

Proof of Theorem 1:

Let H be an open rectangle covering A, and let B be the intersection of H with the complement of A. Then, B is open. Thus, for every $x \in B$, we know that the grid rectangle in G_n to which it belongs is entirely contained in B for all n large enough. Thus, by the Lebesgue dominated convergence theorem, the Lebesgue measure of the "useless" rectangles tends to the Lebesgue measure of B. But then, the Lebesgue measure of C_n must tend to the Lebesgue measure of A.

The directory itself can be constructed as follows: define a large enough array (of size $n = N_1 N_2 \ldots N_d$), initially unused, and keep two stack pointers, one for a top stack growing from position 1 down, and a bottom stack pointer growing from the last position up. Thus, the two stacks are tied down at the ends of the array and grow towards each other. Travel from grid rectangle to grid rectangle, identify the type of rectangle, and push the address onto the top stack when it corresponds to a good rectangle, and onto the bottom stack when we have a bad rectangle. Useless rectangles are ignored. After this, the array is partially full, and we can move the bottom stack up to fill positions $k+1$ through $k+l$. If the number of useless rectangles is expected to be unreasonably large, then the stacks should first be implemented as linked lists and at the end copied to the directory of size $k+l$. In any case, the preprocessing step takes time equal to n, the cardinality of the grid.

It is important to obtain a good estimate of the size of the directory. We must have

$$k+l \geq \frac{\operatorname{area}(A)}{a} = \frac{\operatorname{area}(A)}{\operatorname{area}(H)}\, n.$$

We know from Theorem 1 and the fact that area $(C_n)=(k+l)\,a$, that

$$\lim_{n\to\infty} \frac{k+l}{n} = \frac{\text{area}\,(A)}{\text{area}\,(H)},$$

provided that as $n\to\infty$, we make sure that $\inf_i N_i \to \infty$ (this will insure that the diameter of the prototype rectangle tends to 0). Upper bounds on the size of the directory are harder to come by in general. Let us consider a few special cases in the plane, to illustrate some arguments. If A is a convex set for example, then we can look at all N_1 columns and N_2 rows in the grid, and mark the extremal bad rectangles on either side, together with their immediate neighbors on the inside. Thus, in each row and column, we are putting at most 4 marks. Our claim is that unmarked rectangles are either useless or good. For if a bad rectangle is not marked, then it has at least two neighbors due north, south, east and west that are marked. By the convexity of A, it is physically impossible that this rectangle is not completely contained in A. Thus, the number of bad rectangles is at most $4\,(N_1+N_2)$. Therefore,

$$k+l \le n\,\frac{\text{area}\,(A)}{\text{area}\,(H)} + 4\,(N_1+N_2).$$

If A consists of a union of K convex sets, then a very crude bound for $k+l$ could be obtained by replacing 4 by $4\,K$ (just repeat the marking procedure for each convex set). We summarize:

Theorem 2:
The size of the directory is $k+l$, where

$$\frac{\text{area}\,(A)}{\text{area}\,(H)} \le \frac{k+l}{n} = (1+o(1))\,\frac{\text{area}\,(A)}{\text{area}\,(H)}.$$

The asymptotic result is valid whenever the diameter of the grid rectangle tends to 0. *For convex sets A on R^2, we also have the upper bound*

$$\frac{k+l}{n} \le \frac{\text{area}\,(A)}{\text{area}\,(H)} + 4\,\frac{N_1+N_2}{N_1\,N_2}.$$

We are left now with the choice of the N_i's. In the example of a convex set in the plane, the expected number of iterations is

$$\frac{(k+l)\,a}{\text{area}\,(A)} \le 1 + \frac{\text{area}\,(H)}{\text{area}\,(A)}\,\frac{4}{n}\,(N_1+N_2).$$

The upper bound is minimal for $N_1=N_2=\sqrt{n}$ (assume for the sake of convenience that n is a perfect square). Thus, the expected number of iterations does not exceed

$$1+\frac{\text{area}\,(H)}{\text{area}\,(A)}\,\frac{8}{\sqrt{n}}.$$

This is of the form

$$1+\frac{\text{constant}}{\sqrt{n}}$$

where n is the cardinality of the enclosing grid. By controlling n, we can now control the expected time taken by the algorithm. Note that the algorithm is fast if we can avoid the bad rectangles very often. It is easy to see that the expected number of inspections of bad rectangles before halting is the expected number of iterations times

$$\frac{l}{k+l},$$

which is equal to

$$\frac{l\,\text{area}(H)}{n\,\text{area}(A)}=o(1) \text{ since } \frac{l}{n}\to 0$$

(as a consequence of Theorem 1). Thus, asymptotically, we spend a negligible fraction of time inspecting bad rectangles. In fact, using the special example of a convex set in the plane with $N_1=N_2=\sqrt{n}$, we see that the expected number of bad rectangle inspections is not greater than

$$\frac{\text{area}(H)}{\text{area}(A)}\frac{8}{\sqrt{n}}.$$

3. Avoidance Problems

In some simulations, usually with geometric implications, one is asked to generate points uniformly in a set A but not in $\cup A_i$ where the A_i's are given sets of R^d. For example, when one simulates the random parking process (cars of length one park at random in a street of length $L+1$ but should avoid each other), it is important to generate points uniformly in $[0,L]$ minus the union of some intervals of the same length.

Fig. 2. Car parking problem: there is no room left on this street

Towards the end of one simulation run, when the street fills up, it is not feasible to keep generating new points until one falls in a good spot. Here a grid structure will be useful. In two dimensions, similar problems occur: for example, the circle avoidance problem is concerned with the generation of uniform points in a circle given that the point cannot belong to any of a given number of circles (usually, but not necessarily, having the same radius). See Fig. 3.

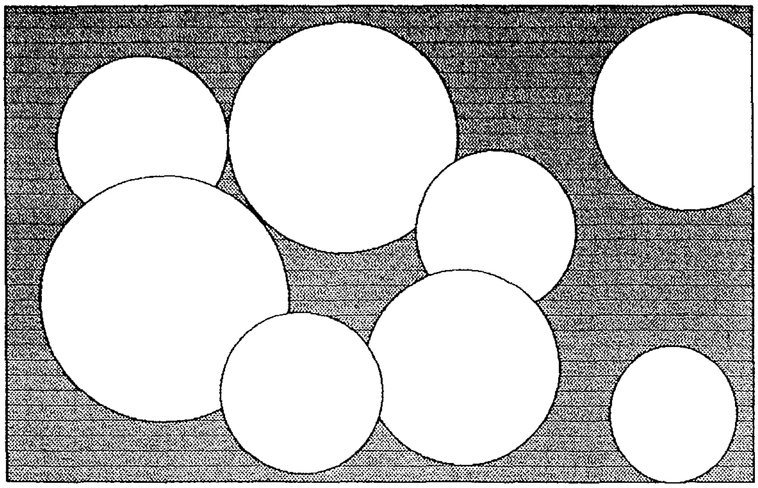

Fig. 3. Circle avoidance problem: generate a point uniformly in the gray area

We could use the grid method in all these cases, but there is an additional ingredient because our problems are dynamic, and we cannot afford to recompute the directory each time. Thus, we also need a fast method for updating the directory. For this, we will employ a dual data structure (see e.g. Aho, Hopcroft and Ullman, 1983). The operations that we are interested in are "Select a random rectangle from among the good and bad rectangles", and "Update the directory" (which involves changing the status of good or bad rectangles to bad or useless rectangles, because the avoidance region grows continuously). Also, for reasons explained above, we would like to keep the good rectangles together. Assume that we have a d-dimensional table for the rectangles containing three pieces of information:

(i) The coordinates of the rectangle (usually of vector of integers, one per coordinate).
(ii) The status of the rectangle (good, bad or useless).
(iii) The position of the rectangle in the directory (this is called a pointer to the directory).

The directory is as before, except that it will with time shrink in size as more and more rectangles are declared useless. The update operation involves changing the status of a number of rectangles (for example, if a new circle to be avoided is added, then all the rectangles entirely within that circle are declared useless, and those that straddle the boundary are declared bad). Since we would like to keep the time of the update down to the number of cells involved times a constant, it is obvious that we will have to reorganize the directory. Let us use two lists again, a list of good rectangles tied down at 1 and with top at k, and a list of bad rectangles tied down at n and with top at $n-l+1$ (it has l elements). There are three situations:

(A) A good rectangle become bad: transfer from one list to the ohter. Fill the hole in the good list by filling it with the top element. Update k and l.

(B) A good or bad rectangle becomes useless: remove the element from the appropriate list, and fill the hole as in case (A). Update k or l.

(C) A bad rectangle remains bad: ignore this case.

For generation, there is only a problem when $Z > k$: when this happens, replace Z by $Z+n-l-k$, and proceed as before. This replacement makes us jump to the end of the directory.

Let us turn now to the car parking problem, to see why the grid structure is to be used with care, if at all, in avoidance problems. At first, one might be tempted to think that for fine enough grids, the performance is excellent. Also, the number of cars (N) that are eventually parked on the street cannot exceed L, the length of the street. In fact, $E(N) \sim \lambda L$ as $L \to \infty$ where

$$\lambda = \int_0^\infty e^{-2\int_0^t (1-e^{-u})/u\, du}\, dt = 0.748\ldots$$

(see e.g. Renyi (1958), Dvoretzky and Robbins (1964) or Mannion (1964)). What determines the time of the simulation run is of course the number of uniform $[0,1]$ random variates needed in the process. Let E be the event

$$[\text{Car 1 does not intersect } [0,1]].$$

Let T be the time (number of uniforms) needed before we can park a car to the left of the first car. This is infinite on the complement of E, so we will only consider E. The expected time of the entire simulation is at least equal to $P(E)E(T|E)$. Clearly, $P(E)=(L-1)/L$ is positive for all $L>1$. We will show that $E(T|E)=\infty$, which leads us to the conclusion that for all $L>1$, and for all grid sizes n, the expected number of uniform random variates needed is ∞. Recall however that the actual simulation time is finite with probability one.

Let W be the position of the leftmost end of the first car. Then

$$\begin{aligned} E(T|E) &= \frac{L}{L-1} \int_1^L E(T|W=t)\frac{dt}{L} \\ &\geq \frac{L}{L-1} \int_1^{1+\frac{1}{n}} E(T|W=t)\frac{dt}{L} \\ &\geq \frac{1}{L-1} \int_1^{1+\frac{1}{n}} \frac{1}{t-1}\, dt = \infty . \end{aligned}$$

Similar distressing results are true for d-dimensional generalizations of the car parking problem, such as the hyperrectangle parking problem, or the problem of parking circles in the plane (Lotwick, 1984) (the circle avoidance problem of Fig. 3 is that of parking circles with centers in uncovered areas until the unit square is covered, and is closely related to the circle parking problem). Thus, the rejection method of Ripley (1979) for the circle parking problem, which is nothing but the grid method with one giant grid rectangle, suffers from the same drawbacks as the grid method in the car parking problem. There are several possible cures. Green and Sibson (1978) and Lotwick (1984) for example zoom in on the good areas in parking problems by using Dirichlet tessellations. Another possibility is to use a search tree. In the car parking problem, the search tree can be defined very simply as follows: the tree is binary; every internal node corresponds to a parked car, and every terminal node corresponds to a free interval, i.e. an interval in which we are allowed to park. Some parked cars may not be represented at all. The information in one internal node consists of:

p_l: the total amount of free space to the left of the car for that node;
p_r: the total amount of free space to the right of the car.

For a terminal node, we store the endpoints of the interval for that node. To park a car, no rejection is used at all. Just travel down the tree taking left turns with probability equal to $p_l/(p_l+p_r)$, and right turns otherwise, until a terminal node is reached. This can be done by using one uniform random variate for each internal node, or by reusing (milking) one uniform random variate time and again. When a terminal node is reached, a car is parked, i.e. the midpoint of the car is put uniformly on the interval in question. This car will cause one of three situations to occur:

1. The interval of length 2 centered at the midpoint of the car covers the entire original interval.
2. The interval of length 2 centered at the midpoint of the car forces the original interval to shrink.
3. The interval of length 2 centered at the midpoint of the car splits the original interval in two intervals, separated by the parked car.

In case 1, the terminal node is deleted, and the sibling terminal node is deleted too by moving it up to its parent node. In case 2, the structure of the tree is unaltered. In case 3, the terminal node becomes an internal node, and two new terminal nodes are added. In all cases, the internal nodes on the path from the root to the terminal node in question need to be updated. It can be shown that the expected time needed in the simulation is $O(L\log(L))$ as $L\to\infty$. Intuitively, this can be seen as follows: the tree has initially one node, the root. At the end, it has no nodes. In between, the tree grows and shrinks, but can never have more than L internal nodes. It is known that the random binary search tree has expected depth $O(\log(L))$ when there are L nodes, so that, even though our tree is not distributed as a random binary search tree, it comes as no surprise that the expected time per car parked is bounded from above by a constant time $\log(L)$. We will report on the properties of the car parking tree elsewhere.

4. Fast Random Variate Generators

It is known that when (X, U) is uniformly distributed under the curve of a density f, then X has density f. This could be a density in R^d, but we will only consider $d=1$ here. All of our presentation can be extended to R^d. Assume that f is a density on $[0,1]$, bounded by M. The interval $[0,1]$ is divided into N_1 equal intervals, and the interval $[0,M]$ for the y-direction is divided into N_2 equal intervals. Then, a directory is set up with k good rectangles (those completely under the curve of f), and l bad rectangles. For all rectangles, we store an integer i which indicaters that the rectangle has x-coordinates

$$\left[\frac{i}{N_1}, \frac{i+1}{N_1}\right).$$

Thus, i ranges from 0 to N_1-1. In addition, for the bad rectangles, we need to store a second integer j indicating that the y coordinates are

$$\left[M\frac{j}{N_2}, M\frac{j+1}{N_2}\right).$$

Thus, $0 \le j < N_2$. It is worth repeating the algorithm now, because we can re-use some uniform random variates.

Generator for Density f on $[0,1]$ Bounded by M

```
(NOTE: $D[1], \ldots, D[k+l]$ is a directory of integer-valued $x$-coordinates, and $Y[k+1], \ldots, Y[k+l]$ is a directory of integer-valued $y$-coordinates for the bad rectangles.)
REPEAT
    Generate a uniform [0,1] random variate $U$.
    $Z \leftarrow \lfloor (k+l)U \rfloor$ ($Z$ chooses a random element in $D$)
    $\Delta \leftarrow (k+l)U - Z$ ($\Delta$ is again uniform [0,1])
    $X \leftarrow \frac{D[Z]}{N_1} + \Delta$
    Accept $\leftarrow [Z \le k]$
    IF NOT Accept THEN
        Generate a uniform [0,1] random variate $V$.
        Accept $\leftarrow [M(Y[Z]+V) \le f(X) N_2]$
UNTIL Accept
RETURN $X$
```

This algorithm uses only one table-look-up and one uniform random variate most of the time. It should be obvious that more can be gained if we replace the $D[i]$entries by

$$\frac{D[i]}{N_1},$$

and that in most high level languages we should just return from inside the loop. The awkward structured exit was added for readability. Note further that in the algorithm, it is unimportant whether f is used or cf where c is a convenient constant. Usually, one might want to choose c in such a way that an annoying normalization constant cancels out.

In Fig. 4, a 31 by 19 grid is used for a jagged density: 377 cells are stored, of which more than 78% are good cells, i.e. cells for which one table look-up is all that is needed.

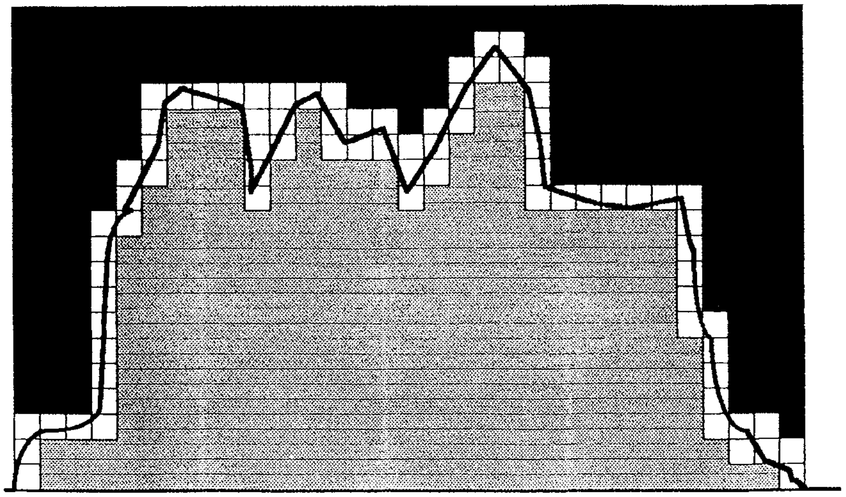

Fig. 4. 31 by 19 grid for density. 81 bad rectangles (white); 296 good rectangles (gray); 212 useless rectangles (black); probability of immediate acceptance is 296/377 = 0.7851 ...; storage: 377 cells

When f is nonincreasing (an important special case), the set-up is facilitated. It becomes trivial to decide quickly whether a rectangle is good, bad or useless. Notice that when f is in a black box, we will not be able to declare a particular rectangle good or useless in our lifetime, and thus all rectangles must be classified as bad. This will of course slow down the expected time quite a bit. Still, for nonincreasing f, the number of bad rectangles cannot exceed $N_1 + N_2$. Thus, noting that the area of a grid rectangle is $\frac{M}{n}$, we observe that the expected number of iterations does not exceed

$$1 + M\frac{N_1 + N_2}{n}.$$

Taking $N_1 = N_2 = \sqrt{n}$, we note that the bound is

$$1 + O\left(\frac{1}{\sqrt{n}}\right).$$

We can adjust n to off-set large values of M, the bound on f. The expected number of computations of f for monotone densities does not exceed

$$\frac{l}{k+l}\left(\frac{(k+l)M}{n}\right)=\frac{lM}{n}\leq\frac{M(N_1+N_2)}{n}.$$

For unimodal densities, a similar discussion can be given. Note that in the case of a monotone or unimodal density, the set-up of the directory can be automated.

It is also important to prove that as the grid becomes finer, the expected number of iterations tends to 1. This is done below.

Theorem 3:

For all Riemann integrable densities f on $[0,1]$ bounded by M, we have, as $\inf(N_1,N_2)\to\infty$, the expected number of iterations,

$$(k+l)\frac{M}{n}$$

tends to 1. The expected number of evaluations of f is $o(1)$.

Proof of Theorem 3:

Given an n-grid, we can construct two estimates of $\int f$,

$$\sum_{i=0}^{N_1-1}\frac{1}{N_1}\sup_{\frac{i}{N_1}\leq x\leq\frac{i+1}{N_1}} f(x),$$

and

$$\sum_{i=0}^{N_1-1}\frac{1}{N_1}\inf_{\frac{i}{N_1}\leq x\leq\frac{i+1}{N_1}} f(x).$$

By the definition of Riemann integrability (Whittaker and Watson, 1927, p. 63), these tend to $\int f$ as $N_1\to\infty$. Thus, the difference between the estimates tends to 0. But, by a simple geometrical argument, it is seen that the area taken by the bad rectangles is at most this difference plus $2N_1$ times the area of one grid rectangle, that is,

$$o(1)+\frac{2M}{N_2}=o(1).$$

Densities that are bounded and not Riemann integrable are somehow peculiar, and less interesting in practice. Let us close this section by noting that extra savings in space can be obtained by grouping rectangles in groups of size m, and putting the groups in an auxiliary directory. If we can do this in such a way that many groups are homogeneous (all rectangles in it have the same value for $D[i]$ and are all good), then the corresponding rectangles in the directory can be discarded. This, of course, is the sort of savings advocated in the multiple table look-up method of Marsaglia (1963). The price paid for this is an extra comparison needed to examine the auxiliary directory first.

A final remark is in order about the space-time trade-off. The storage requirements are for at most $N_1 + N_2$ bad rectangles and $\frac{n}{M}$ good rectangles when f is monotone. The bound on the expected number of iterations on the other hand is

$$1 + \frac{M}{n}(N_1 + N_2).$$

If $N_1 = N_2 = \sqrt{n}$, then keeping the storage fixed shows that the expected time increases in proportion to M. The same rate of increase, albeit with a different constant, can be observed for the ordinary rejection method with a rectangular dominating curve. If we keep the expected time fixed, then the storage increases in proportion to M. The product of storage $(1 + 2M/\sqrt{n})$ and expected time $(2\sqrt{n} + n/M)$ is $4\sqrt{n} + n/M + 4M$. This product is minimal for $n = 1, M = \sqrt{n}/2$, and the minimal value is 8. Also, the fact that storage times expected time is at least $4M$ shows that there is no hope of obtaining a cheap generator when M is large. This is not unexpected since no conditions on f besides the monotonicity are imposed. It is well-known for example that for specific classes of monotone or unimodal densities (such as all beta or gamma densities), algorithms exist which have uniformly bounded (in M) expected time and storage. On the other hand, table look-up is so fast that grid methods may well outperform standard rejection methods for many well known densities.

References

Aho, A. V., Hopcroft, J. E., Ullman, J. D.: Data Structures and Algorithms. Reading, Mass.: Addison-Wesley 1983.

Chen, H. C., Asau, Y.: On generating random variates from an empirical distribution. AIIE Transactions *6*, 163–166 (1974).

Dvoretzky, A., Robbins, H.: On the parking problem. Publications of the Mathematical Institute of the Hungarian Academy of Sciences *9*, 209–225 (1964).

Lotwick, H. W.: Simulation of some spatial hard core models and the complete packing problem. Journal of Statistical Computation and Simulation *15*, 295–314 (1982).

Mannion, D.: Random space-filling in one dimension. Publications of the Mathematical Institute of the Hungarian Academy of Sciences *9*, 143–153 (1964).

Marsaglia, G.: Generating discrete random variables in a computer. Communications of the ACM *6*, 37–38 (1963).

Neumann, J. von: Various techniques used in connection with random digits. Collected Works, vol. 5, pp. 768–770, Pergamon Press, 1963. Also in: Monte Carlo Method, National Bureau of Standards Series *12*, 36–38 (1951).

Norman, J. E., Cannon, L. E.: A computer program for the generation of random variables from any discrete distribution. Journal of Statistical Computation and Simulation *1*, 331–348 (1972).

Peterson, A. V., Kronmal, R. A.: On mixture methods for the computer generation of random variables. The American Statistician *36*, 184–191 (1982).

Renyi, A.: On a one-dimensional problem concerning random space-filling. Publications of the Mathematical Institute of the Hungarian Academy of Sciences *3*, 109–127 (1958).

Ripley, B. D.: Simulating spatial patterns: dependent samples from a multivariate density. Journal of the Royal statistical Society, series C *28*, 109–112 (1979).

Walker, A. J.: An efficient method for generating discrete random variables with general distributions. ACM Transactions on Mathematical Software *3*, 253–256 (1977).
Whittaker, E. T., Watson, G. N.: A Course of Modern Analysis. Cambridge: Cambridge University Press 1963.

L. Devroye
McGill University
School of Computer Science
Burnside Hall
805 Sherbrooke Street West
Montreal, PQ, Canada H3A 2K6