Journal of Statistical Software

July 2004, Volume 11, Issue 3. *http://www.jstatsoft.org/*

Fast Generation of Discrete Random Variables

George Marsaglia
Florida State University

Wai Wan Tsang
University of Hong Kong

Jingbo Wang
University of Hong Kong

Abstract

We describe two methods—and provide C programs—for generating discrete random variables with functions that are simple and fast, averaging ten times as fast as published methods and more than five times as fast as the fastest of those. We provide general procedures for implementing the two methods, as well as specific procedures for three of the most important discrete distributions: Poisson, binomial and hypergeometric.

Keywords: random numbers, discrete variates, Poisson, binomial, hypergeometric.

1. Introduction

The methods we describe here were developed in the early 1960's (Marsaglia 1963) with the second method having a variation based on a device from the 70's (Walker 1974). In the intervening years we have found few methods that compare favorably with these. The methods were developed when a few hundred memory locations could be a burden, and seem even more pertinent today, when thousands of locations are readily available. The generating functions take only a few lines of C code. Listings for the two functions are included in the **browse files** section, as is code for setting up parameters that the generating functions require.

2. Method I: Condensed table-lookup.

Consider a simple example of the method described in Marsaglia (1963) and advocated in the 1964 NBS handbook (Abramowitz and Stegun 1964).
We have a random variable X with distribution given by a table:

value	prob
a	.2245
b	.1271
c	.3452
d	.3032

Probably the fastest way to produce a realization of X in a computer is to create a table $T[10000]$ with ten thousand entries made up of 2245 a's, 1271 b's, 3452 c's and 3032 d's. Then if i is a random integer in $0 \leq i < 10000$, set $x = T[i]$. That requires a 10K table—not very large by modern standards, but if our probabilities were expressed to several more base-10 digits, the resulting table is likely to be too large. Let's see if we can get by with a smaller table, but with nearly as much speed.

Since we are choosing uniformly from the 10000 elements in the T array, we are free to place elements in the table in any way we want. Using the old Fortran convention in which `4*a,2*b,5*c` means fill with 4 a's, 2 b's, 5 c's, etc., suppose we fill the table as follows:

```
2000*a,1000*b,3000*c,3000*d,200*a,200*b,400*c,40*a,70*b,50*c,30*d,5*a,b,c,c,d,d
```

The groupings are determined by the first, then second, third,... digits of the probabilities for each value.

We still have 10000 elements, with a total of 2245 a's, 1271 b's, etc., but now the first 9000 elements of the T table are just 9 blocks of 1000 identical elements. We could do as well if we ignore those 9000 elements in T and instead, with probability 9000/10000, choose one element from {a,a,b,c,c,c,d,d,d}.

Similarly, the 800 elements from $T[9000]$ to $T[9799]$ are just 8 blocks of 100 identical elements, so, with probability 800/10000 we can choose one element from the small array {a,a,b,b,c,c,c,c}.

Finally, the 190 elements in $T[9800]$ to $T[9989]$ are just 19 blocks of 10 identical elements, so with probability 190/10000 we can select an element from {a,a,a,a,b,b,b,b,b,b,b,c,c,c,c,c,d,d,d}.

Thus we have a considerable saving in space, (from 10000 to 46), and still good speed, (about 90% as fast), if we generate our X as follows, given a random integer i in $0 \leq i < 10000$:

```
Setup:
  A[9] ={a,a,b,c,c,c,d,d,d};
  B[8] ={a,a,b,b,c,c,c,c};
  C[19]={a,a,a,a,b,b,b,b,b,b,b,c,c,c,c,c,d,d,d};
  D[10]={a,a,a,a,a,b,c,c,d,d};
Generating procedure, given random i in 0<=i<=9999:
  if(i<9000) return A[i/1000];
  if(i<9800) return B[(i-9000)/100];
  if(i<9990) return C[(i-9800)/10];
             return D[i-9990];
```

Rather than using the digits for base 10, we might form tables from the digits of the probabilities to base 100. We then have only two tables, A and B, using the Fortran convention for multiple table entries:

```
A[98] ={22*a,12*b,34*c,30*d}
B[200]={45*a,71*b,52*c,32*d}
```

Total table space is 298, greater than the 46 for base 10, but still much smaller than the 10000 for base 10000. The generating procedure will be faster—almost as fast as that for the table of 10000:

```
    if(i<9800) return A[i/100];
               return B[i-9800];
```

Of course a table lookup method may be overkill for such a simple discrete distribution. But the above method can be applied very effectively to discrete distributions with hundreds of values and with probabilities to eight or more places. We will, in fact, provide implementations that express the probabilities as rationals with denominator 2^{30}. Then if we use base 64, each probability will have five base-64 'digits' that provide five small tables similar to, but perhaps many times as large, as those in the example above. Furthermore, analogs to the above integer divisions: $i/1000$, $(i-9000)/100$, etc. will be effected by shifts of our six-bit 'digits', resulting in very fast generating procedures.

Given the underlying discrete distribution, the generating function looks like this:

```
   int Dran() /* Uses 5 compact tables; jxr is global static Xorshift RNG */
    {unsigned long j;
    jxr^=jxr<<13; jxr^=jxr>>17; jxr^=jxr<<5; j=(jxr>>2);
    if(j<t1) return AA[j>>24];
    if(j<t2) return BB[(j-t1)>>18];
    if(j<t3) return CC[(j-t2)>>12];
    if(j<t4) return DD[(j-t3)>>6];
    return EE[j-t4];
    }
```

The initialization procedure uses the probabilities to set up static global AA,BB,CC,DD,EE arrays and parameters t1,t2,t3,t4 determined by the table sizes. We use a Xorshift RNG described in Marsaglia (2003); any of the several hundred given there could be used, as they do very well in tests of randomness and are very fast.

Here are some specific comparisons for the compact-table method:
If we express the probabilites for the Poisson distribution, $\lambda = 100$, to the nearest rational with denominator 2^{30}, (keeping only the numerators), then use the base-64 digits to form five tables, the table entries will total 10202 and the resulting C function will produce Poisson-100 variates at over 60 million/second (Intel 1800MHz CPU).
For similarly expressed binomial probabilities, $n = 100$, $p = .345$, the five tables will have total 5102 entries, with similar speed—over 60 million/second.

If we use base 1024, the three tables will total about 90020 elements for the Poisson-100 distribution and 47057 for the binomial, $n = 100$, $p = .345$. Speed in both cases will be faster than the 5-table version, around 70 million/second.

A version based on two tables from the digits for base 2^{15} would be only a little faster than the three-table version, but would require ten times as many table entries.

For Poisson and other discrete variates with an infinite number of probabilities, we select only those for which, for a sample of size $2^{31}(10^{9.33})$, the expected number of occurences exceeds 0.5. The other probabilities are assumed zero. For those unusual situations where occurences with probability less than 5×10^{-10} must be accounted for, special tail-handling procedures should be used.

The setup and generating procedures are much the same, whatever the choice of the base digits. The accompanying C programs are five-table versions based on expressing the probabilities to five base-64 digits.

Note that the actual memory space required for the compacted tables depends on n, the number of possible values of the discrete random variable. If $n \leq 256$ than the table entries can be bytes, `char` in C, while $n > 256$ requires 16 bits for each table entry, `short int` in C. Should you encounter discrete random variables taking more than 65536 values, your table entries will require 32 bits each.

3. Method II: Table + square histogram.

This second method is likely to require fewer tabled values. It may be summarized as follows:

Use one byte from a 32-bit random integer to get an entry in a table of size 256.
If that tabled value is positive, return it.
Else use a slower method to produce a variate among values not covered by the single table.

To set up the procedure for a discrete random variable taking values $0, 1, \ldots, n-1$ with probabilities $p_0, p_1, \ldots, p_{n-1}$, express each p_i as

$$p_i = \frac{k_i + \theta_i}{256} \text{ with } k_i \text{ an integer and } 0 \leq \theta_i < 1.$$

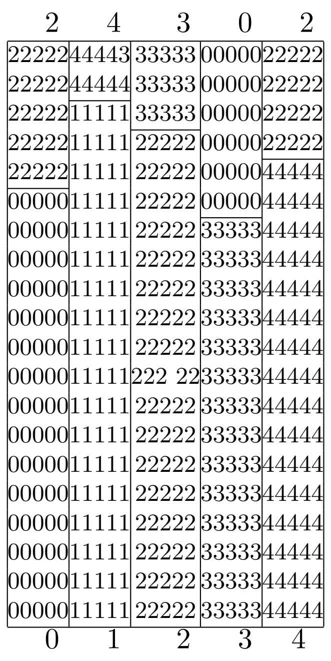
(The k's are the first digits in the expansion of the p's to base 256.)
Now fill a table, J[256], with k_0 0's, k_1 1's,...,k_{n-1} $(n-1)$'s. If the table is not full, (it might typically have 2-15 unassigned cells), fill the remaining J[ ] locations with -1's.

Then to generate the required variate D: Let i be a random 32-bit integer, with j formed from the rightmost 8-bits of i.(In C, `j=i&255;`). If J[j]≥ 0 return J[j], else return a D by the square histogram method described below, with probabilities $\theta_0, \theta_1, \ldots, \theta_{n-1}$ normalized to sum to 1. The single uniform [0,1) variate that the square histogram requires may be formed as the floating point version of $i/2^{32}$, with i the random 32-bit integer whose rightmost eight bits determined the table-lookup index.

The result is a fast and simple procedure. It depends on a preliminary setup for the square histogram, with required tables K[ ] and V[ ] of size n. The method produces the specified discrete variates at about 40 million/second, not as fast as the compact-table method, but still far faster than most available methods.

4. The square histogram method.

A **square histogram** is depicted in the following figure:



Every square histogram contains n equal-width columns ($n = 5$ in this example), and each column has a bottom-part and a top-part. The bottom-part of each column "belongs" to the index $0, 1, \ldots, n-1$ of that column, the top-part of each column "belongs" to the index label at the top of that column, stored in an array K[n]. (The top part may be empty.) Each column has a division point, contained in an array V[n], indicating where ownership of the column content changes. For this example, in C notation, `K[5]={2,4,3,0,2}` and `V[5]={.15,.38,.57,.74,.96}`.

The height of the square histogram is .20 in this example. Note that the division points in the V array are not $\{.15, .18, .17, .14, .16\}$ as might be expected, but rather $.15, .20+.18, .40+ .17, .60+.14, .80+.16$. That is because if we know the index of a column, we already know the range of the uniform number that determined it, reducing the arithmetic necessary for deciding whether we are below or above the division point.

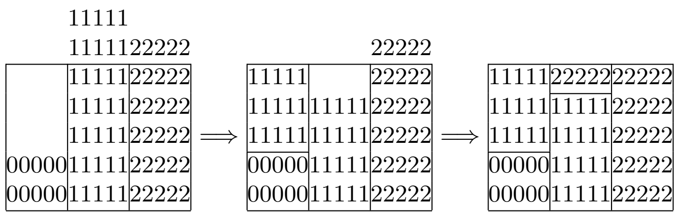
Now to generate our discrete random variable, we choose a point uniformly from the square histogram and return the label of the area in which that point falls. Instead of the usual two uniform variates to get a random point in the square histogram we can, by means of the arrays $K[n]$ and $V[n]$, get our result from a single uniform [0,1) variate U:

$$j = \lfloor 5U \rfloor;\ \text{if } U < V[j] \text{ return } j;\ \text{else return } K[j].$$

This rule provides a fast and easy way to generate a random variable J taking values $0, 1, 2, 3, 4$ with probabilities .21, .18, .26, .17, .18. (the sums of the areas belonging to each index). And such a simple rule will work for any discrete random variable taking values $0, 1, 2, \ldots, n-1$ with probabilities $p_0, p_1, \ldots, p_{n-1}$, $\sum p_i = 1$.

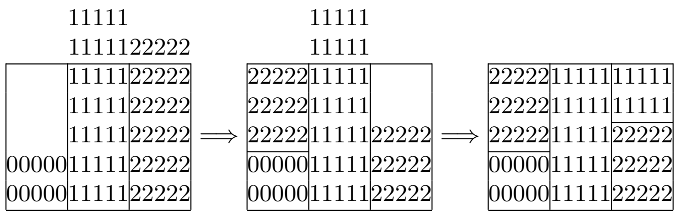
All one need to do is form a regular histogram for $0, 1, \ldots, n{-}1$ with bars of height $p_0, p_1, \ldots, p_{n-1}$, then use the Robin Hood rule—take from the richest to bring the poorest up to average—bringing one, then another and another until all are average and a square histogram has been formed.

We illustrate with simple square-histogram-forming examples. Suppose we have a random variable J that takes values $0, 1, 2$ with probabilities $\frac{2}{15}, \frac{7}{15}, \frac{6}{15}$. The average height of the square histogram will be 5/15, so we form a histogram with heights given by those probabilities, drawn within and above the anticipated final form of the histogram. We iterate with the Robin Hood method. First, we bring the poorest, column 0, up to average, reducing the 'wealth' of column 1, then take from the newly richest, column 2, to bring the diminished column 1 up to average.



We now have a square histogram with arrays $K[3] = \{1, 2, 2\}$ and $V[3] = \{\frac{2}{15}, \frac{9}{15}, \frac{15}{15}\}$ and the generating rule, using a uniform [0,1) variate U: $j = \lfloor 3U \rfloor$ if $U < V[j]$ return j, else return $K[j]$. Note how the values in the V array are determined. If $j = 0$, then $0 \leq U < 5/15$ and testing $U < 2/15$ provides the division of the '0' column into the two parts belonging to '0' or the three parts belonging to '1'. Similarly, the range $5/15 \leq U < 10/15$ will lead to selection of the '1' column, and testing $U < 9/15$ will provide the division of that column into the 4 parts belonging to '1' and the single part belonging to '2'.

But there are other ways to square the histogram. Starting with the same initial column values, $\frac{2}{15}, \frac{7}{15}, \frac{6}{15}$ as above, suppose for the first step we take from the second richest, '2', to bring the poorest, '0', up to average, then finish the squaring process:



We end with arrays $K[3] = \{2, 1, 1\}$ and $V[3] = \{\frac{2}{15}, \frac{10}{15}, \frac{13}{15}\}$ and the same rule for generation: $j = \lfloor 3U \rfloor$; if $U < V[j]$ return j, else return $K[j]$. But note that this time we will do the 'else' part with total probability 5/15, in contrast to the first, Robin Hood, method where the 'else' part was required with smaller total probability, 4/15.

The frequency of requiring the 'else' part of the generating procedure

$$j = \lfloor nU \rfloor;\ \text{if } U < V[j] \text{ return } j;\ \text{else return } K[j];$$

is proportional to the "over-area": that part of the final squared histogram that lies above the division points. An optimal squaring provides the least possible over-area. The Robin Hood method will not usually be optimal, but experience suggests that it will be close. Finding an optimal squaring of a given histogram turns out to be NP hard. It is discussed in references Marsaglia and Tsang (1987) and Tsang (1981). Our square histogram method was developed to provide a more efficient and transparent implementation of an approach suggested in Walker (1974).

Assuming initial probabilities $p_0, p_1, \ldots, p_{n-1}$ that sum to 1 and average $a = 1/n$ are given, an algorithm for the Robin Hood method of squaring the histogram goes as follows:

```
First, initialize: For $i$ from 0 to $n-1$ do: $K[i] = i$ and $V[i] = (i+1)*a$.
Then do these two steps $n-1$ times:

    1. Find the smallest probability, $p_i$, and the largest, $p_j$.
    2. Set $K[i] = j$; $V[i] = (i-1)*a + p_i$; replace $p_j$ with $p_j - (a - p_i)$; replace $p_i$ with $a$;
```

This will provide the generating procedure:

$$j = \lfloor nU \rfloor;\ \text{if } U < V[j] \text{ return } j,\ \text{else return } K[j].$$

5. Comparisons

We compare our two methods with some fast methods in the literature for generating Poisson, binomial and hypergeometric variates.

The ratio of uniforms (Stadlober and Zechner 1999) is a rejection algorithm which has simple calculation and reasonable rejection rate.

The patchwork rejection (Stadlober,E. 1989) applies (without mention) the ideas of the Monty Python method from Marsaglia and Tsang (1984) and Marsaglia and Tsang (1998). It uses a simple hat function and rearranges the area under the frequency function *f(x)* to fill as much of the area under the hat function as possible. As a result, it has a very low rejection rate. These two methods are both applicable for any distribution so they are included in all three comparisons: binomial, Poisson and hypergeometric.

We also compare our methods with two methods designed for specific distributions:

- PD (acceptance complement) (Ahrens and Dieter 1982) which was believed to be one of the fastest methods designed for Poisson variates, and
- BTPE (triangle-parallelogram-exponential rejection) (Kachitvichyanukul and Schmeiser 1988) which uses combined majorizing and minorizing functions to sample from binomial distributions.

The C codes for comparisons were compiled and linked in Microsoft Visual C++ 6.0 under Windows XP on a Pentium IV 2.26GHz as well as with gcc in a DOS window, but similar results were obtained when the code was compiled on a Unix system. To make comparisons fair, uniform random numbers were produced by the fast Xorshift method of Marsaglia (2003).

The execution times for generating 100 million variates from binomial, Poisson and hypergeometric distribution are given in the tables below. The results show our Method I (Condensed Table-Lookup) has the best performance and almost invariant speed under different parameters. Method II (Table+Square Histogram) is often nearly as fast as Method I, but it slows down for parameters which provide a lower success rate in phase one, requiring more frequent need for the square-histogram. The Stadlober-Zechner patchwork rejection method seems best among the other four methods, but its fastest is only 1/5 as fast as our Method I.

Our methods run from 5 to 15 times faster than competing methods, averaging around 10. You may wish to extract them from 'browse files' and try for yourself, comparing with your own or other methods for not only speed but complexity, program size and (your view of) elegance.

For certain values of the parameters, the setup time and memory costs for our Method I or Method II may make simple, direct methods more suitable. See Tables 1, 2, 3.

6. Attachments

We have provided `C` versions of the two methods described here, for inclusion in the "Browse files" section of the journal. We suggest that you first compile and run `5tbl.c`, then `TplusSQ.c`. Each will ask for your choice of parameters for Poisson, then binomial, then hypergeometric distributions. Each will apply its method to that distribution as well as do chisquare tests on the output. You may then want to examine the components of the two files, for illumination or for extracting portions that might be usefully applied to your discrete distributions.

The `5tbls.c` file contains `void get5tbls(void)` that creates the five tables for the compact-table method, based on 30-bit probabilities expressed as five base-64 digits.

Those probabilites are created by `PoissP()` or `BinomP()` or `HyperGeometricP()`. Given the parameter lambda, `PoissP()` creates the table `P[ ]` of Poisson probabilities as numerators of rationals of the form $j/2^{30}$. Those p's for which $2^{31}p < 1$ are assumed zero.

`void BinomP(int n, double p)` creates the static int array `P[ ]` for the binomial distribution, as numerators of rationals $j/2^{30}$. Those p's for which $2^{31}p < 1$ are assumed zero, as are the p's created by the function `HyperGeometricP()` for the hypergeometric distribution.

`int Dran()` is the function that uses a 32-bit xorshift integer to return a discrete random value from the appropriate one of the five tables.

`void Dtest(int n)` generates a sample of n values using `Dran()` and does a goodness-of-fit test, grouping cell counts so expected numbers are > 20. It uses included `Phi()` and `Chisquare()` functions.

The file `TplusSQ.c` also tests the output of Poisson, binomial and hypergeometric generators coming from the function `Dran()`, except that the `Dran()` function this time is based on Method II: Table plus Square Histogram. The parameters and arrays are set up by the function `DSQset( )`, which relies on the the same routines, `PoissP()` or `BinomP()` or `HyperGeometicP()` to create the integer array `P[ ]`. `DSQset()` converts each integer probability `P[i]` to double `p[i]`. Then `p[i]` is expressed as `p[i]`$=\frac{k_i+\theta_i}{256}$ in order to create the `J[256]` fast table-lookup and arrays `K[]` and `V[]` for the square histogram. Then the discrete variate generator `Dran()` takes the form required for Method II.

Those wishing to apply one or the other of these two methods for other kinds of discrete distributions will have to create their own routines to replace the `PoissP()` or `BinomP()` or `HyperGeometricP()` functions that create the table `P[]` of (integer) probabilities, the 30-bit numerators j of the distribution's probabilities expressed as $j/2^{30}$. Then the routine `get5tbls()` will create the five tables that the `Dran()` generator requires for Method I, or `DSQset()` will create the `J,K` and `V` arrays for the Table+Square Histogram version of the

n	p	Method I	Method II	Patchwork	RatioUniform	BTPE
20	0.1	1.703	1.718	19.687	16.750	9.563
20	0.4	1.813	1.828	15.640	16.563	22.203
100	0.1	1.813	1.875	13.219	16.547	26.093
100	0.4	1.985	2.078	12.891	16.463	600.27
1000	0.1	2.188	2.359	12.672	16.547	73.156
1000	0.4	2.266	2.796	12.610	16.578	31.422
10000	0.1	1.625	3.593	12.485	22.875	14.578
10000	0.4	1.657	4.546	12.563	28.375	8.547
100000	0.1	1.703	5.906	12.469	28.406	7.719
100000	0.4	1.985	2.078	13.750	15.859	17.047

Table 1: Time, in seconds, to generate 10^8 binomial variates

lambda	Method I	Method II	Patchwork	RatioUniform	PD
1	1.719	1.703	18.59	18.92	6.015
10	1.844	1.922	14.55	18.30	18.22
25	1.985	2.094	13.66	15.84	21.44
100	2.203	2.422	13.44	15.86	20.83
250	2.265	2.829	13.31	15.86	16.81
1000	1.625	3.704	13.26	18.58	16.75

Table 2: Time, in seconds, to generate 10^8 Poisson variates

N1	N2	K	Method I	Method II	Patchwork	RatioUniform
20	20	20	1.750	1.781	21.062	13.563
100	100	20	1.844	1.765	13.438	13.797
100	100	100	1.885	1.917	10.373	13.560
100	1000	100	1.860	1.869	8.203	15.656
1000	1000	100	1.984	2.088	9.674	13.578
1000	1000	1000	2.231	2.530	7.906	13.625
1000	10000	100	1.849	1.898	8.359	20.391
1000	10000	1000	2.219	2.375	7.765	23.094
10000	10000	1000	2.265	2.907	7.937	26.376
10000	10000	10000	1.665	4.202	7.828	37.969

Table 3: Time, in seconds, to generate 10^8 hypergeometric variates

`Dran()` generator.

Then that appropriate `Dran()` function can be used to provide extremely fast and simple generation of random variables from the specified discrete distribution, perhaps after first testing for any mishaps by applying the `Dtest(100000000)` function for a sample of 10^8.

For some choices of parameters, or for certain distributions, other methods may be easier to apply or require less memory. But the two methods advocated here may be worth considering for a wide variety of discrete distributions.

References

Abramowitz M, Stegun IA (eds.) (1964). *Handbook of Mathematical Functions.* U.S. Government Printing Office, Washington D.C.

Ahrens JH, Dieter U (1982). "Computer Generation of Poisson Deviates from Modified Normal Distributions." *ACM Transaction on Mathematical Software(TOMS)*, **8**.

Kachitvichyanukul V, Schmeiser BW (1988). "Binomial Random Variate Generation." *Communications of the ACM*, **31**(2).

Marsaglia G (1963). "Generating Discrete Random Variables in a Computer." *Communications of the ACM*, **6**, 37–38.

Marsaglia G (2003). "Xorshift RNGs." *Journal of Statistical Software*, **8**(14).

Marsaglia G, Tsang WW (1984). "A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions." *SIAM Journal Scientific and Statistical Computing*, **5**, 349–359.

Marsaglia G, Tsang WW (1987). "A Decision Tree Algorithm for Squaring the Histogram in Random Number Generation." *Ars Combinatoria*, **23A**, 291–301.

Marsaglia G, Tsang WW (1998). "The Monty Python Method for Generating Random Variables." *ACM Transactions on Mathematical Software*, **24**(3), 341–350.

Stadlober E, Zechner H (1999). "The patchwork Rejection Technique for Sampling from Unimodal Distributions." *ACM Transaction on Modeling and Computer Simulation (TOMACS)*, **9**.

Stadlober,E (1989). "Ratio of Uniforms as a Convenient Method for Sampling from Classical Discrete Distributions." *Proceedings of the 21st ACM conference on Winter simulation.*

Tsang WW (1981). *Analysis of the Square-the-Histogram Method for Generating Discrete Random Variables.* Master's thesis, Computer Science, Washington State University.

Walker AJ (1974). "Fast Generation of Uniformly Distributed Pseudorandom Numbers with Floating Point Representation." *Electronics Letters*, **10**, 553–554.

Affiliation:

George Marsaglia
Profesor Emeritus, Florida State University
Home Address: 1616 Golf Terrace Drive
Tallahassee FL 32301, United States of America
E-mail: `geo@stat.fsu.edu`

Wai Wan Tsang
Computer Science Department
The University of Hong Kong
Pokfulam Road, Hong Kong
E-mail: `tsang@cs.hku.hk`

Jingbo Wang
Computer Science Department
The University of Hong Kong
Pokfulam Road, Hong Kong
E-mail: `jbwang@cs.hku.hk`

Journal of Statistical Software
July 2004, Volume 11, Issue 3.
`http://www.jstatsoft.org/`

Submitted: 2004-06-05
Accepted: 2004-07-12