Random integer partitions with restricted numbers of parts

Kenneth J. Locey

ABSTRACT. I present a conceptually simple and comparatively fast computing algorithm for finding uniform random integer partitions of *Q* having *N* parts. Mathematical environments implement functions to find uniform random partitions of *Q* based on well accepted algorithms, but not restricted to N parts. I present the first algorithm derived to solve this problem in a straightforward way (i.e. draw a uniform random number that codes for one and only one lexically ordered partition satisfying *Q* and *N*). The algorithm is explained conceptually, represented generically, coded in Python, and its performance is compared against feasible sets (i.e. set of all integer partitions of *Q* having *N* parts) and against the random integer partition function implemented in the Sage computer algebra system. I discuss a use of random partitioning algorithms for examining distributional patterns in the natural and social sciences.

Introduction

An integer partition is a way of writing a positive integer *Q* as a non-increasing sequence of *N* positive integers. Integer partitions are said to be unordered because sequences with the same integer values without respect to order constitute the same partition. For example [1,3,1] and [1,1,3] are differently ordered sequences of the same lexically ordered partition, i.e. [3,1,1]. Questions relating to integer partitioning (e.g. finding the number of partitions of *Q*) and identities revealing symmetrical properties of integer partitions (e.g. number of partitions of *Q* having *N* parts equals the number of partitions of *Q* having *N* as the largest part) have been intensively studied in combinatorics and number theory (Andrews and Eriksson 2004). Most mathematical environments (e.g. Sage, Sympy, Mathematica, Combinatorica, Maple, Matlab) implement some integer partitioning algorithms for finding, for example, the number of partitions for a total *Q* having *N* parts, for finding the next lexical (i.e. dictionary) ordered partition for *Q*, and for finding conjugates, Frobenius coordinates, and Ferrers diagrams.

Several mathematical environments implement widely-established computing algorithms for generating uniform random integer partitions for Q. For example, Sage and Sympy use the algorithm of Nijenhuis and Wilf (1978). In fact, this algorithm is largely the extent of current widely implemented random integer partitioning algorithms. Though many studies on the properties of random integer partitions have been made (e.g. Fristedt 1993, Goh and Schmutz 1995, Borgs et al. 2004), there are few straightforwardly implemented random partitioning algorithms (e.g. Nijenhuis and Wilf 1978, Goh and Hitczenko 2007, Arratia and DeSalvo 2011); most are limited to generating random partitions of Q.

Currently, no mathematical environment implements a function to generate uniform random integer partitions of Q having N parts. Likewise, no current algorithm generates uniform random partitions satisfying Q and N. However, these tasks can have a host of practical applications involving the study of distributional patterns in unordered numerical sequences and frequency distributions in natural systems. One recent application of integer partitioning has used uniform random samples of feasible sets (i.e. the set of integer partitions satisfying Q and N) to study and predict distributional patterns in ecological systems (Locey and White, 2013). However, use of current functions and computing algorithms necessitates potentially high rejection rates because they do not sample according to N (i.e. generate a partition of Q and reject it if it does not have N parts). Consequently, the probability of random drawing a partition satisfying Q and N can be untenable for practical computing.

Here, I present a conceptually simple and comparatively fast algorithm that generates uniform random integer partitions satisfying Q and N. I explain the algorithm in concept, generically represent the algorithm, provide specific implementation using the Python scripting language, and examine its performance compared against entire feasible sets (i.e. set of all integer partitions of Q having N parts) and against the random integer partition function implemented in the Sage mathematical environment (www.sagemath.org). Finally, I discuss the broader practical use of random integer partitioning functions outside number theory and combinatorics, that is, applied to questions and patterns in the natural and social sciences.

An algorithm to find uniform random partitions of a total Q having N parts

The number of partitions of Q can be found using well-known algorithms (Andrews and Eriksson 2004). These algorithms are implemented in several mathematical environments (e.g. Sage, Sympy) and rely on partitions of Q being lexically ordered. Consequently, each of the P(Q) partitions corresponds to a number in the range (1, 2, …, P(Q)) that can be randomly chosen and decoded into a specific partition. The following applies the same conceptual approach to partitions of Q with N parts.

Treat partition in reverse lexical order, i.e., let the number 1 designate the last lexical partition of Q having N parts, the number 2 designate the next-to-last partition, and so forth, such that the number P(Q, N) designates the first lexical partition. Now, choose a number X at random between 1 and P(Q, N) inclusive. This number will correspond to one and only one partition satisfying Q and N. Now, find the partition that corresponds to X by building the partition one summand at a time beginning with the largest summand.

For each possible maximum summand value, we can ask for the number of partitions of Q having N parts with K as the largest part. 1.) Start with the smallest possible maximum summand value K. 2.) Find the number of partitions of Q having N parts with K or less as the maximum summand value, i.e. P(Q, N, K). This can be found using the recursive functions below. 3.) ask if P(Q, N, K) is less than X. If so, then the first summand value must be larger than K. If so, increase K by 1 and find the number of partitions satisfying Q and N having K or less as the largest part. Eventually P(Q, N, K) will be larger than or equal to X, meaning that the first summand value of the partition corresponding to X must be K. 4.) Having found the first summand value, move to the next summand by decreasing X by P(Q, N, $K - 1$), decreasing Q by K, and decreasing N by 1. 5.) Return to step 1 continue finding subsequent summands until Q or $N = 0$. The key point of this process, i.e. a randomly chosen number X between 1 and P(Q, N) can be decoded to produce the first summand value of the Xth partition, can be shown graphically (Fig 1).

Generic representation of the algorithm and an associated recursive algorithm

```
P(Q, N, K): # Generic recursion function yielding the number of partitions satisfying Q
              # and N with parts not greater than K.

    if Q = 0 and N = 0 then return 1
            # by convention  P(0) = P(0, 0) = 1


    if Q <= 0 or N <= 0 or K <= 0 then return 0
            # P(0, N > 0) = 0  &  P(Q > 0, 0) = 0  &  P(Q > 0, N > 0, K <= 0) = 0


    if Q > 0 and N > 0 and K > 0 then
          sum ← 0 #
          for i in range(0, N) do sum ← sum + P(Q – i * K, N – i, K – 1)
          # Add partitions found through recursion to sum
          return _sum

random_partition(Q, N): # generate a uniform random partition for Q and N

        list partition ← []
        _min ← min_max(Q, N)     # smallest possible maximum summand value
        _max ← Q – N + 1          # largest possible maximum  summand value
       total ← number_of_partitions(Q, N)
       which ← random(1, total)    # choose an integer at random between 1 and total, inclusive

       while Q > 0 do
            for K in range(_min, _max) do count ← P(Q, N, K)
              if count >= which then count ← P(Q, N, K – 1), break out of for loop

          partition ← append(K)
          Q ← Q – K
          if Q = 0 then break out of while loop
          N ← N – 1
          which ← which – count
          _min ← min_max(Q, N)
          _max ← K

       return partition
```

Python-based implementation using Sage to find P(Q, N)

```
from sage.all import *
import random

def min_max(n,s):  # smallest possible maximum summand value
        min_max = int(floor(float(n)/float(s)))
       if int(n%s) > 0:
             min_max +=1
       return min_max

D = {}
def P(Q, N, K): # number of partitions for Q and N with K or less as the largest part
                    # uses memoization
       if Q > N*K or K <= 0: return 0
       if Q == N*K: return 1
       if (Q, N, K) not in D:
             D[(Q, N, K)] = sum(P(Q – i * K, N – i, K – 1) for i in xrange(N))
       return D[(Q, N, K)]

def random_partition(Q, N): # generate a uniform random partition of Q with N parts
        partition = []
       _min = min_max(Q, N)
       _max = Q – N + 1
       total = number_of_partitions(Q, N) # called from Sage
       which = random.randrange(1, total+1)

       while Q:
             for K in range(_min, _max + 1):
                count = P(Q, N, K)
                if count >= which:
                   count = P(Q, N, K – 1)
                   break
             partition.append(K)
             Q -= K
             if Q == 0: break
             N -= 1
             which -= count
             _min = min_max(Q, N)
             _max = K
      return partition
```

Comparison against feasible sets and the random partition function of Sage

The algorithm presented here will generate uniform random samples of integer partitions for a total Q with N parts from the set of all P(Q, N) partitions (i.e. feasible set). To show that the function accomplishes this, I compare random samples generated using the algorithm presented here against statistical features of 1) feasible sets and 2) uniform random partitions of P(Q, N) generated using the function in the Python-based computer algebra system Sage. Sage is a powerful computing environment and many of its functions are coded in Python's C interfacing language Cython, which greatly increases the speed of normal Python code. I use these comparisons against Sage to show how a fast version of an integer partitioning algorithm that generates random partitions according to Q compares against an algorithm that randomly samples according to both Q and N.

Results

Sage's algorithm (based on Nijenhuis and Wilf 1978) generates uniform random partitions with respect to Q but not N, requiring potentially high rejection rates because the probability of randomly drawing a partition satisfying Q and N can be computationally small, i.e. using Sage: Pr(Q, N) = P(Q, N)/P(Q). Comparisons reveal that distributions of statistical evenness (i.e. a standardized log transform of the variance, E_{var} of Smith and Wilson (1996)), skewness, and median summand values for random samples generated using Sage (i.e. Partitions(Q).random_element( )) are indistinguishable from those using the algorithm presented here (Fig 2). Likewise random samples generated using the algorithm presented here show no directional bias from the feasible set (Fig 2). It should be noted that N can be included into Sage's function as Partitions(Q, length=N).random_element(), but doing so does not change the sampling method, that is, it allows Sage to reject partitions not matching Q and N.

The algorithm presented here is fast compared to an algorithm that is not restricted by N (Table 1). For the present algorithm, substantial compute time could be spent finding the number of partitions of Q having N parts with K or less as the largest part, i.e. P(Q, N, K). If memoization is used, efficiency will increase with sample size, i.e. as greater numbers of random

partitions are needed (Table 1). However, this recursive function, implemented above in Python, is not the central focus of the paper, which is to present a random sampling strategy of lexically ordered restricted partitions represented as unique numbers. Actually, finding P(Q, N, K) can often be found without recursion. For example, the number of partitions of Q having N parts with exactly K as the largest part can equal the number of partitions for $Q - K$ having $N - 1$ parts, e.g. often if K >= (Q – (N – 2))/2 . For example, there are six partitions of $Q = 18$ having $N = 5$ parts and $K = 9$ as the largest part. Likewise, there are six partitions of $9 = Q - K$ having $4 = N - 1$ parts:

[9, 6, 1, 1, 1]	[6, 1, 1, 1]
[9, 5, 2, 1, 1]	[5, 2, 1, 1]
[9, 4, 3, 1, 1]	[4, 3, 1, 1]
[9, 4, 2, 2, 1]	[4, 2, 2, 1]
[9, 3, 3, 2, 1]	[3, 3, 2, 1]
[9, 3, 2, 2, 2]	[3, 2, 2, 2]

Discussion

The function presented here generates uniform random samples of integer partitions for a given total Q having N parts. Each random number (chosen from a uniform distribution using any generic random number generator) between 1 and P(Q, N) codes for one and only one lexically ordered partition satisfying Q and N. The algorithm is conceptually simple, comparatively fast, and is the first to solve this problem though the question has been asked within the computing community (e.g. stackoverflow.com/questions/2161406/how-do-i-generate-a-uniform-random-integer-partition). This algorithm makes a valuable contribution to the relatively small set of current random partitioning algorithms (e.g. Nijenhuis and Wilf 1978, Goh and Hitczenko 2007, Arratia and DeSalvo 2011). Optimization could be made for finding the number of partitions of Q having N parts with K or less as the largest part, which could greatly increase the speed of the algorithm.

The main point to the algorithm presented here is that, if restricted integer partitions can be lexically ordered and if the number of restricted partitions is known, then a random number

can be chosen that corresponds to one and only one lexically ordered restricted integer partition. Consequently, the problem addressed here (i.e. generating uniform random integer partitions for Q having N parts) could be generalized to other types of restrictions (e.g. partitions of Q having K as the largest summand, partitions of Q with N parts having K or less as the smallest summand, etc.). Challenges to generalizing this approach are 1.) finding the total number of partitions for Q and associated restrictions and 2.) understanding how the partitions are lexically ordered according to Q and the restrictions in order to decode a randomly chosen number into its corresponding integer partition.

Random integer partitioning algorithms can provide useful ways to study distributional patterns in natural systems. Examples are the distribution of abundance among species of an ecological community, economic wealth among nations of the global community, and the frequencies of amino acids in genomic sequences (Locey and White, 2013). In general, distributions of wealth and abundance (DOWs) are often examined as frequency distributions or vectors of values across populations of unlabeled and unordered entities. Consequently, if the identities of the entities are ignored, each empirical DOW is an integer partition where the individual values of N unlabeled entities sum to Q. Knowing this, ecologists, biologists, economists and other social scientists can use random integer partitioning algorithms to 1.) generate uniform random samples of feasible sets (i.e. set of all forms or shapes of the DOW), 2.) ask whether the forms or shapes of empirical distributions are similar to the majority of possible distributions, 3.) ask whether variation in the distributional properties of the feasible set explain variation in natural systems, 4.) ask whether empirical DOWs are similar to central tendencies of feasible sets, and hence, 5.) whether empirical DOWs can be predicted using integer partitioning methods (i.e. based solely on Q and N and ignoring all other details of the system), and finally, 6.) understand the sole influence of Q and N on the shape of an empirical distribution (Locey and White 2013). Consequently, random partitioning algorithms could have a profound and immediate use in addressing common questions in various sciences. These potential uses have not, to my knowledge, been previously acknowledged (e.g. Okounkov 2005).

Acknowledgments

Many thanks to ante (stackoverflow.com/users/494076/ante) for contributing a Python-based memoized recursion function for finding the number of partitions satisfying Q and N having K or less as the maximum part. Many thanks to the developers of Sage for developing a powerful computing environment and a fast function to find the number of partitions of Q with N parts. Both functions were valuable to the implementation of the random partitioning algorithm.

References

[1] Andrews, G.E. & Eriksson, K. (2004). *Integer Partitions*. Cambridge Univ. Press, New York.

[2] Arratia, R. and DeSalvo, S. (2011) Probabilistic divide-and-conquer: a new exact simulation method, with integer partitions as an example. arXiv:1110.3856v2

[3] Borgs, C, Chayes, J.T., Merten, S., and Pittel, B. (2004) Constrained integer partitions. *LATIN 2004: Theoretical informatics*, 2705-2705

[4] Fristedt, B. (1993) The structure of random partitions of large integers. *Transactions of the American Mathematical Society*. 337, 703-735.

[5] Goh, W.M.H., and Hitczenko P. (2007) Random partitions with restricted part sizes.

[6] Goh, W.M.Y. and Schmutz, E. (1995) The number of distinct part sizes in a random integer partition. *Journal of combinatorial theory*. 69, 149-158.

[7] Locey, K.J., and White, E.P. (2013) *In review.* How total abundance and species richness constrain the distribution of abundance. submitted to *Ecology Letters.*

[8] Nijenhuis, A. & Wilf, H.S. (1978). *Combinatorial Algorithms for Computers and Calculators*. Academic Press, New York.

[9] Okounkov, A. (2003) The uses of random partitions. arXiv:math-ph/0309015v1

[10] Smith, B. & Wilson, J.B. (1996). A consumer's guide to evenness indices. *Oikos*, 76, 70-82.

Figure 1. Choose a number X at random in the range (1, …, P(Q, N)) corresponding to the X^{th} reverse lexical partition of Q having N parts. Find the value of the first summand K by finding the number of partitions satisfying Q and N and having K or less as the maximum summand. Begin with the smallest maximum summand values K_{min} and proceed to K_{max}. X will occur in one of the ranges, revealing the value of K for the X^{th} partition. Having found the value of the first summand, find the value of the next summand by decreasing X by P(Q, N, $K-1$), Q by K, and N by 1. Continue the process until Q or N = 0.

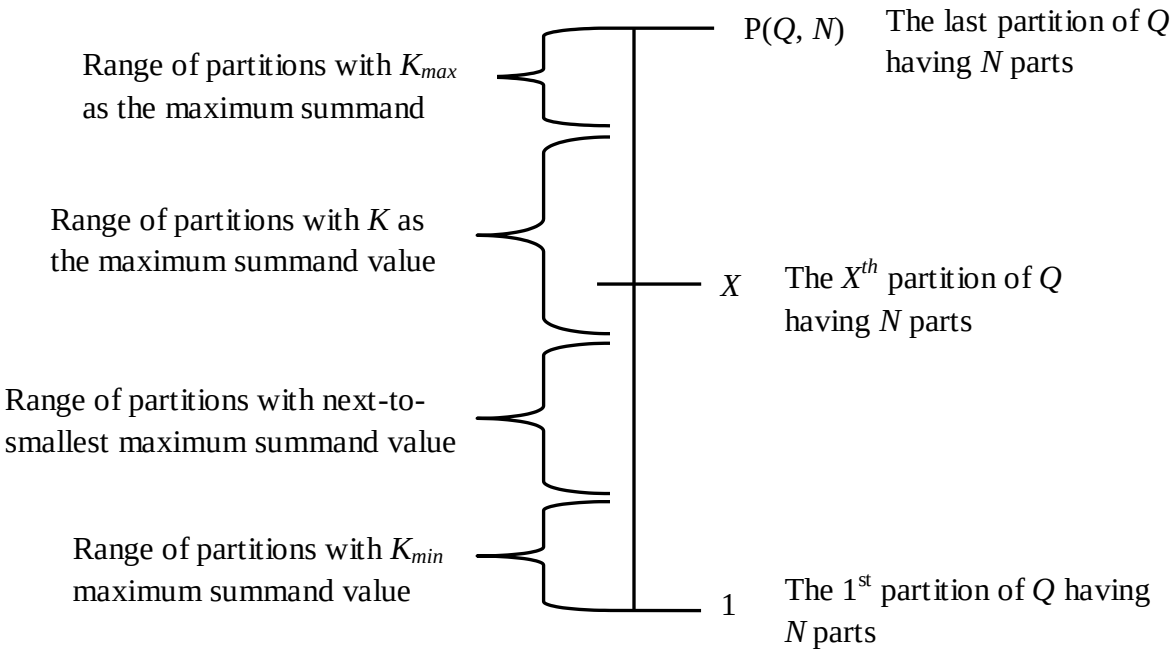
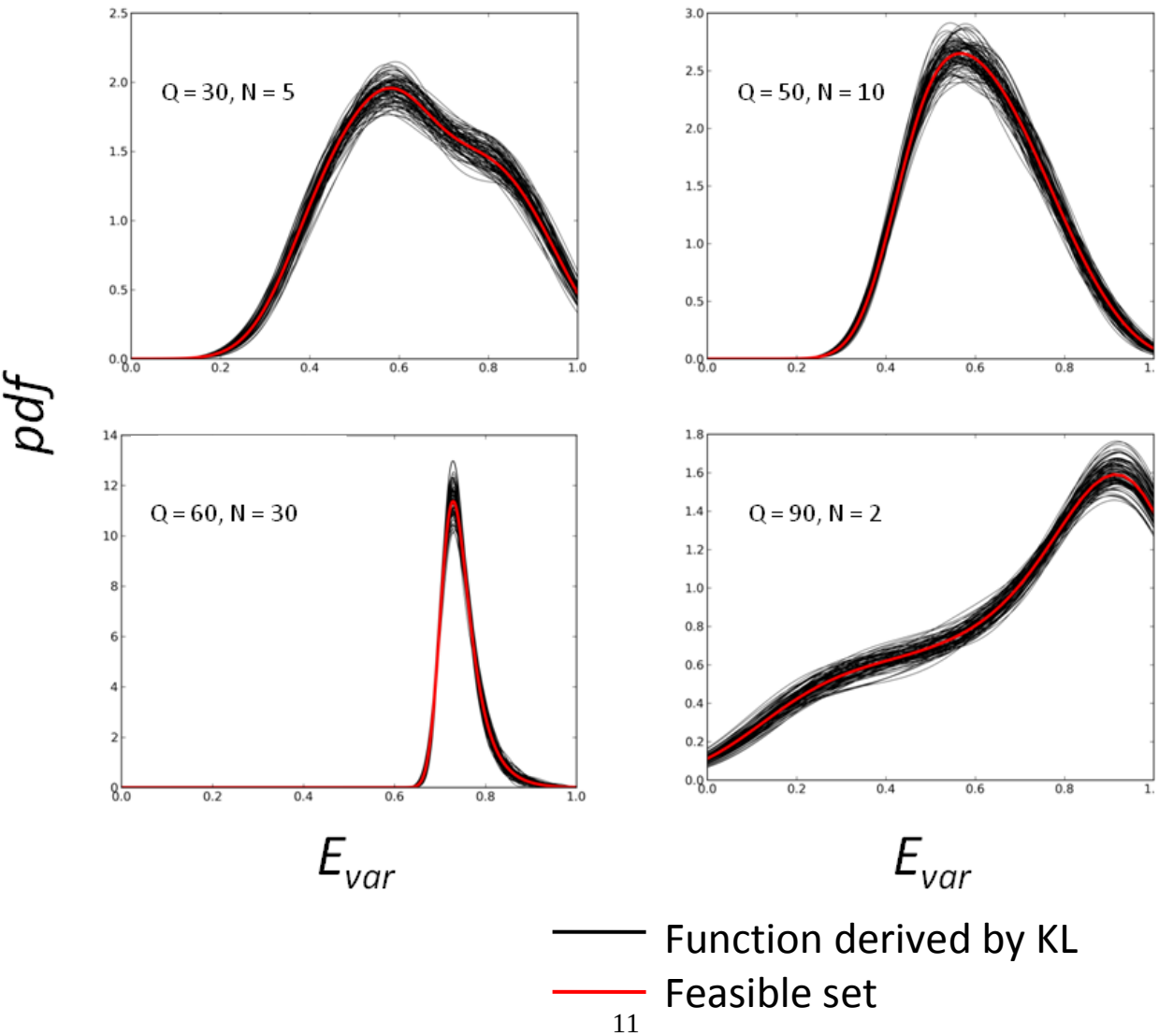
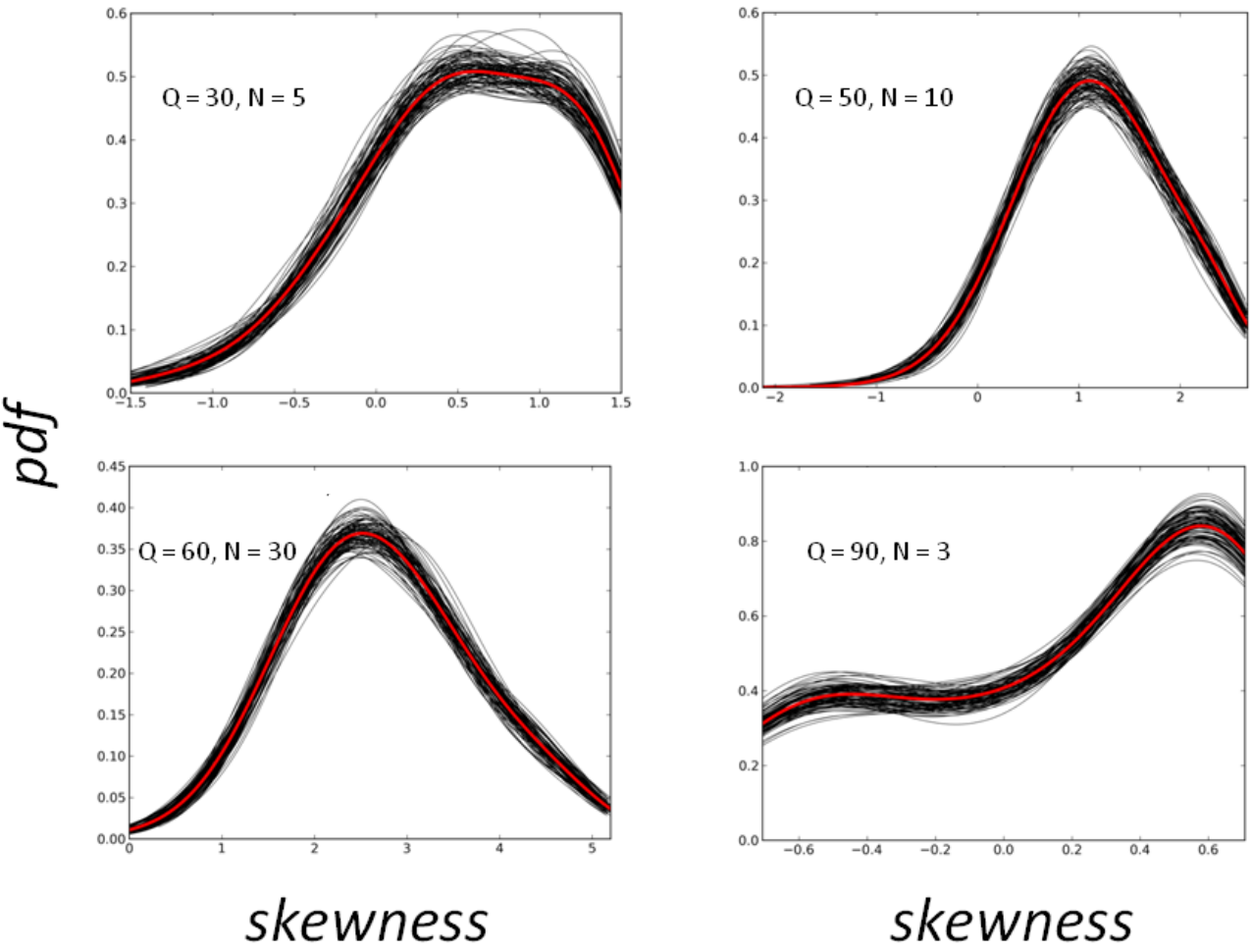
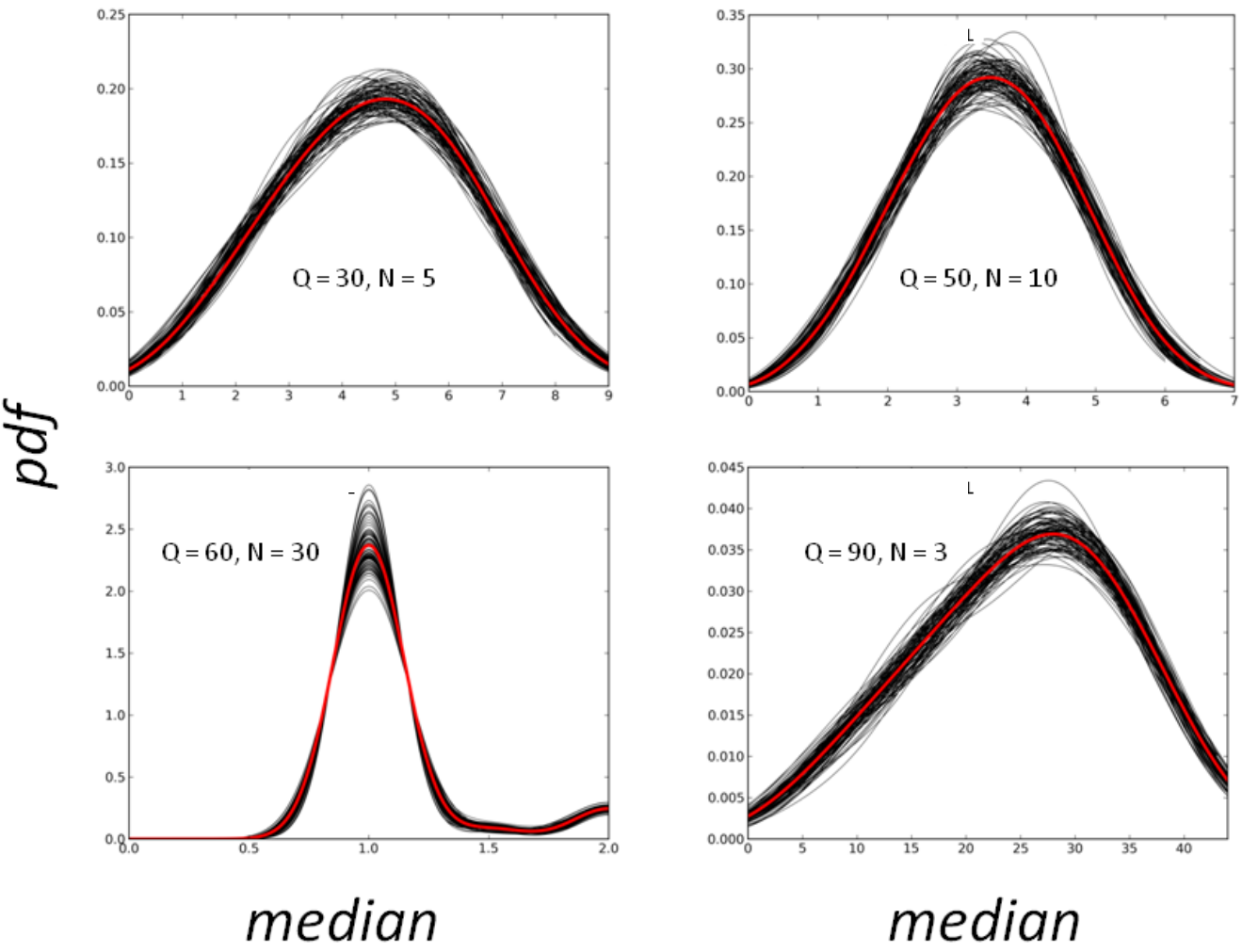
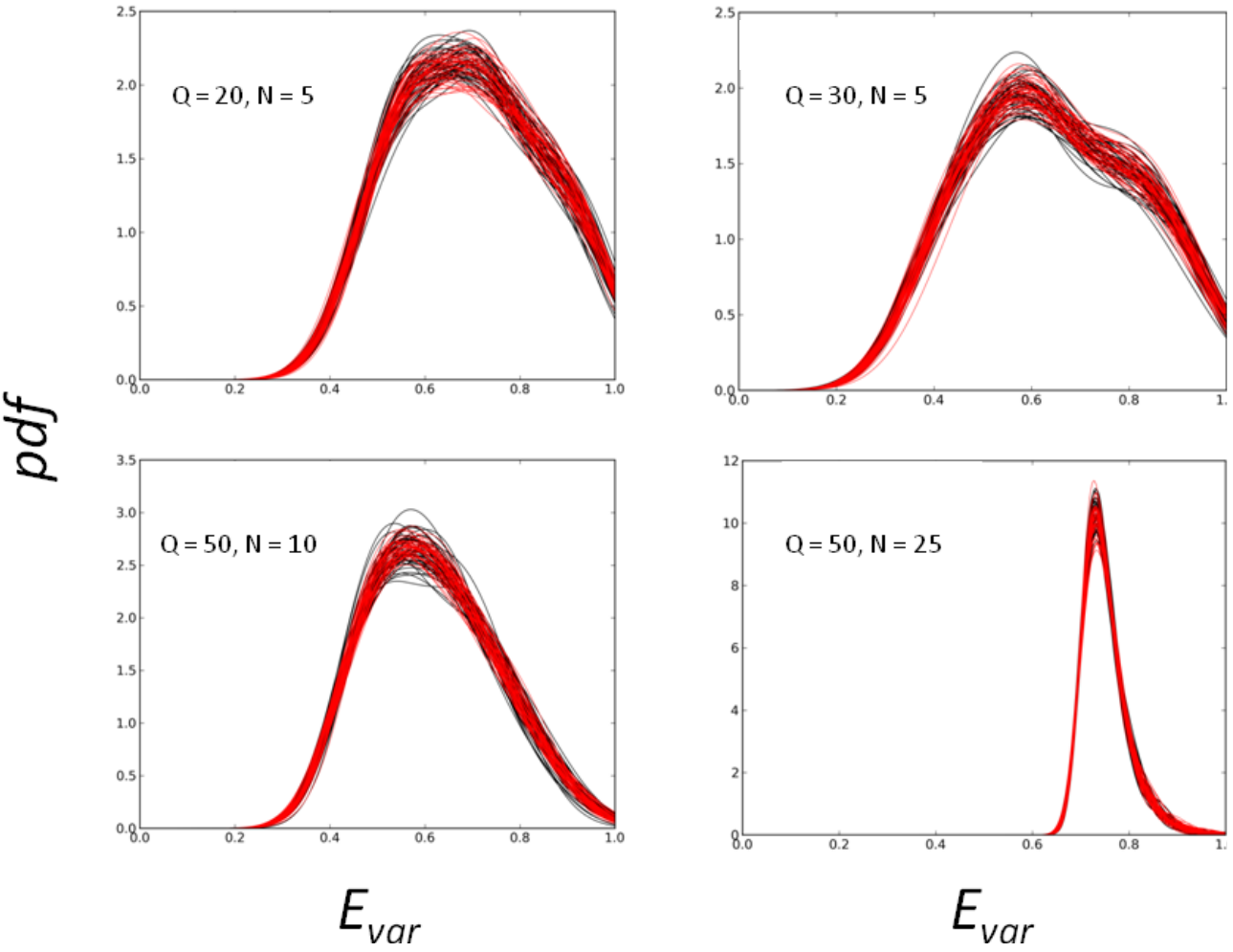


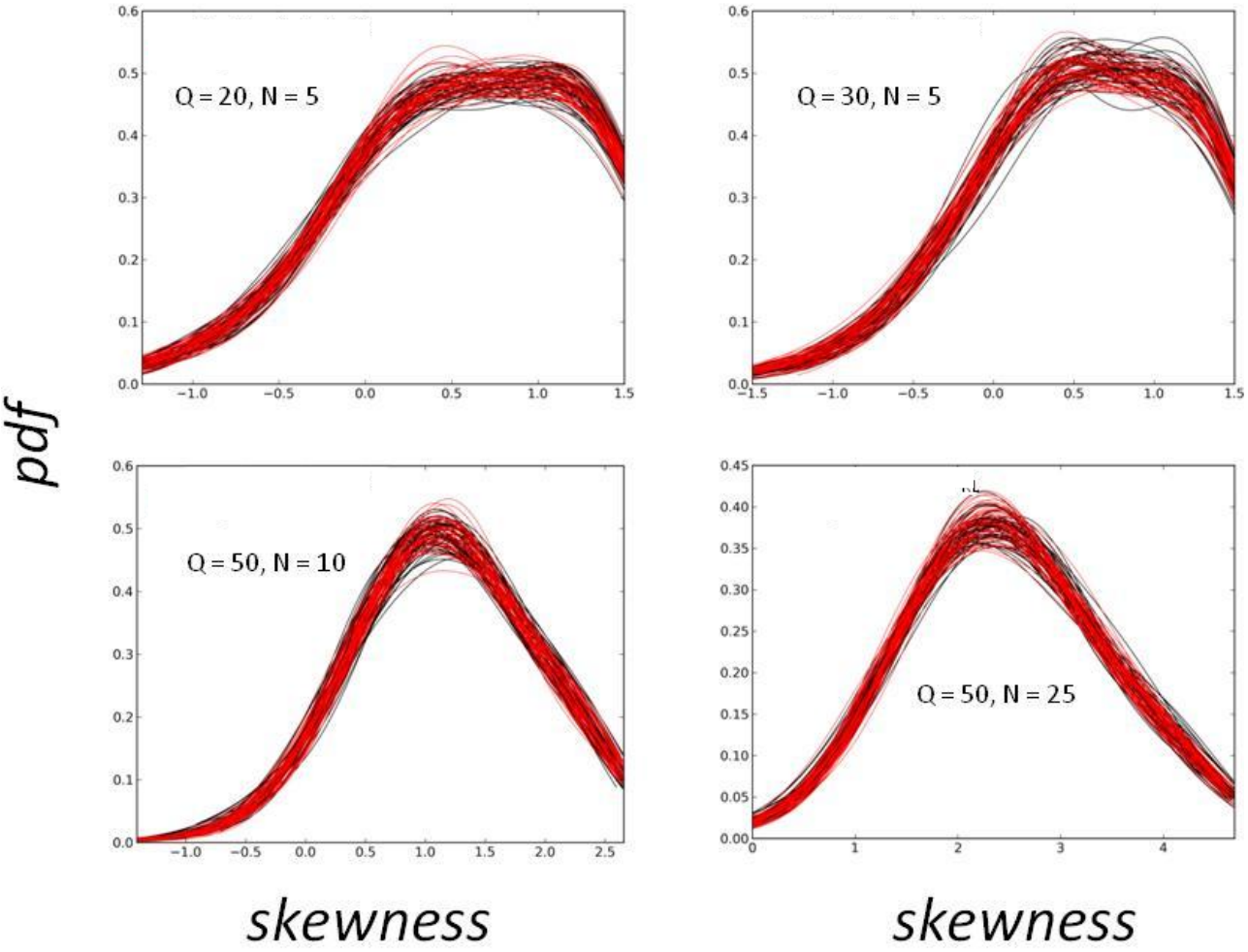
Figure 2 (6 subfigures, 4 plots each). Top 3 subfigures reveal the distribution of statistical features (evenness, skewness, median summand) across different *Q* and *N* combinations as kernel density curves across the entire set of partitions (red line) and across 100 sets of < 500 partitions generated using the algorithm developed in this paper (black lines). Bottom 3 subfigures reveal distributions of the same statistical features across different *Q* and *N* combinations for 50 sets of < 500 partitions generated with the Sage algorithm random partitioning function, (red curves) and 50 random sets of < 500 partitions generated with the algorithm presented in this paper (black curves). Red and black kernel density curves overlap with no directional bias across different *Q* and *N* combinations. Smaller combinations were chosen for these three subplots because the Sage function required impractical amounts of time.




pdf
Q = 30, N = 5
Q = 50, N = 10
Q = 60, N = 30
Q = 90, N = 3
skewness
skewness
Function derived by KL
Feasible set


pdf
Q = 30, N = 5
Q = 50, N = 10
Q = 60, N = 30
Q = 90, N = 3
median
median
Function derived by KL
Feasible set


pdf
Q = 20, N = 5
Q = 30, N = 5
Q = 50, N = 10
Q = 50, N = 25
E_{var}
E_{var}
Function derived by KL
Sage


pdf
Q = 20, N = 5
Q = 30, N = 5
Q = 50, N = 10
Q = 50, N = 25
skewness
skewness
Function derived by KL
Sage

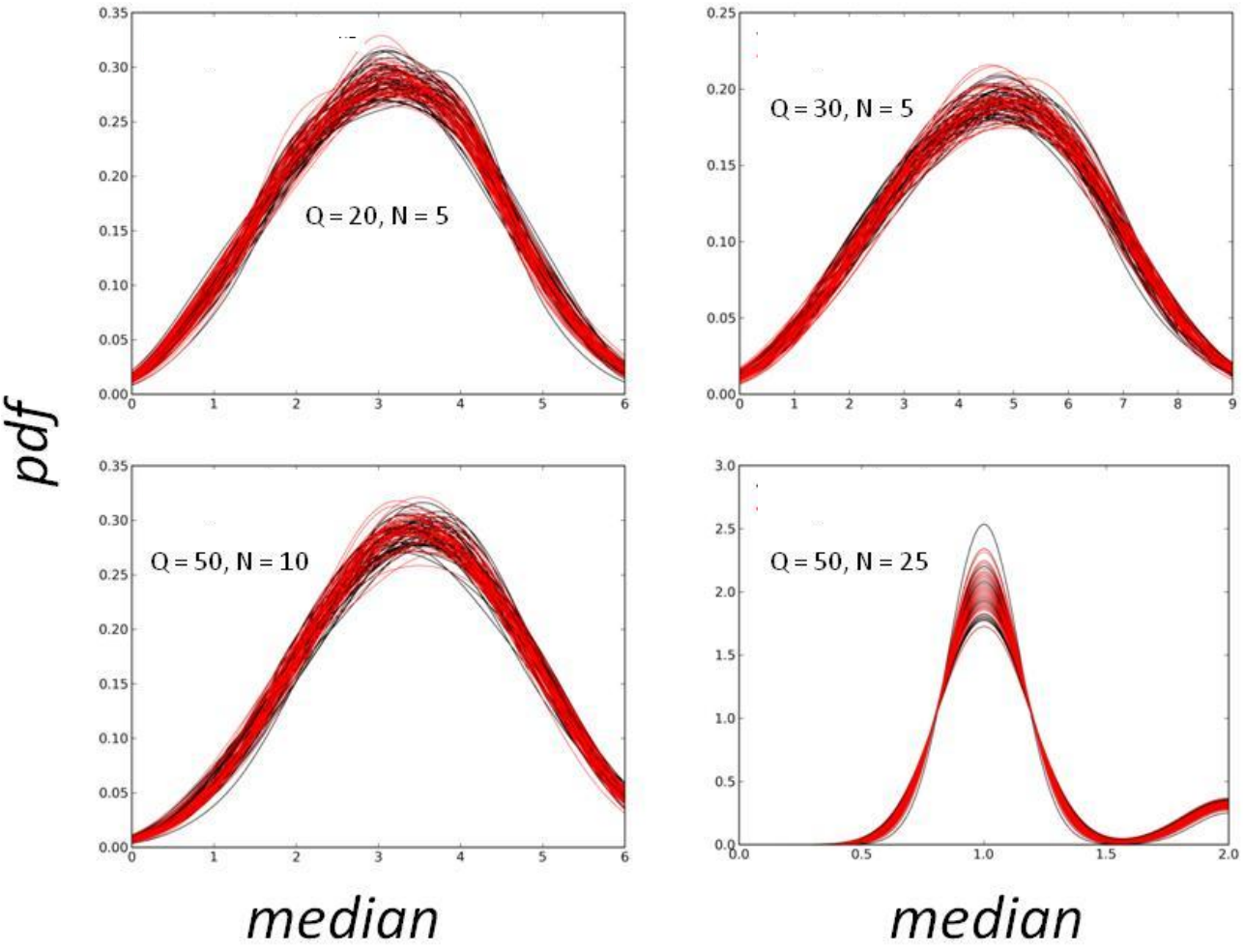

pdf
Q = 20, N = 5
Q = 30, N = 5
Q = 50, N = 10
Q = 50, N = 25
median
median
Function derived by KL
Sage

Table 1. Top: The function presented here (KL) performs as quickly as and most often more quickly than the function provided in Sage (random sampling with respect to Q only) for most combinations of Q, N, and sample size. The function implemented in Sage is slowed due to high rejection rates, and hence very small probabilities of randomly drawing a partition of Q with N parts. Question marks denote waiting periods over five minutes because, for some combinations of Q and N, no result could be expected from the Sage algorithm within reasonable time. **Bottom**: logarithmic relationships well-describe the increase in time with increasing sample size for the KL algorithm. Consequently, as sample size increases the advantage of an algorithm with no rejection rate (random sample according to Q and N) has a greatly increasing advantage over an algorithm that randomly samples according to Q only.

Q, sample size	Algorithm		N_1	N_2	N_3	N_4	N_5
			2	10	15	20	25
30, 1	KL	time(s)	0.01	0.02	0.03	0.03	0.03
	Sage	time(s)	0.01	0.63	0.34	0.13	0.03
			2	10	20	70	90
100, 1	KL	time(s)	0.02	0.28	0.7	2.54	0.09
	Sage	time(s)	0.02	**?**	**?**	43.42	0.39
			2	10	100	200	300
300, 1	KL	time(s)	0.01	0.81	32.27	206.27	0.03
	Sage	time(s)	0.05	**?**	**?**	**?**	1.3

$Q = 300$, $N = 10$	
Sample size	time(s)
1	0.81
10	2.73
100	5.37
1000	14.14
$Q = 300$, $N = 100$	
Sample size	time(s)
1	32.27
10	695.42
100	1172.37
1000	2205.56