High Dimensional Multinomial Sampling

Antas- BSD-DH-2407
Smarak Ranjan Patel- BSD-DH-2422
Dev Tinker- BSD-DH-2410
Pronamika Roy- BSD-DH-2418

Abstract

The multinomial distribution is a fundamental component in multivariate statistics, yet simulating from it becomes computationally burdensome when the number of trials (T) and categories (n) are large. This report investigates the two-stage multinomial decomposition method proposed by Malefaki and Iliopoulos (2007), which leverages the hierarchical property of the distribution to reduce execution time. We extend the original study by implementing the algorithm in a high-performance C++ framework via Rcpp, moving beyond interpreted code to evaluate the limits of computational efficiency. Furthermore, we propose and implement a recursive m-way partitioning sampler designed to handle various probability regimes, including uniform, sparse, skewed, and clustered distributions. Our benchmarks reveal that while the theoretical optimal partition size $m \approx \sqrt{n}$ holds for standard cases, the performance of the hierarchical approach is significantly influenced by the interaction between the branching factor, the probability vector structure p, and the total trials T. Comparative Benchmark indicates that one of our recursive C++ implementation approach results to a significant reduction in execution time relative to the R Base function even for large number of Trials and categories.

1 Introduction

The multinomial distribution serves as a cornerstone in multivariate statistics, with widespread applications ranging from the analysis of contingency tables to modern computational frameworks such as sampling/importance resampling (SIR) and population Monte Carlo (PMC). In these contemporary settings, researchers often encounter scenarios where both the number of independent trials (T) and the number of possible outcomes (n) are exceedingly large. For instance, certain SIR implementations may require drawing $T = 10^4$ observations from a sample space of $n = 3 \times 10^6$.

Despite the availability of established simulation techniques, such as the "direct method" via sequential search or the "conditional method" based on sequential binomial generation, the computational burden remains significant in high-dimensional contexts. Standard algorithms often suffer from a performance

bottleneck when implemented in interpreted programming languages like R, Matlab, and Mathematica, where execution time scales poorly with the number of loops and categories.

2 Two Stage Simulation Theorem

Theorem 2.1. *Let $p = (p_1, \ldots, p_n)$ with $p_i > 0$ and $\sum_{i=1}^{n} p_i = 1$. Let $\{B_1, \ldots, B_m\}$ be a partition of $\{1, 2, \ldots, n\}$, with each $B_j \neq \varnothing$. For each block B_j, define*

$$p^{(j)} = (p_i)_{i \in B_j}, \qquad q_j = \sum_{i \in B_j} p_i, \qquad q = (q_1, \ldots, q_m).$$

Let

$$K = (K_1, \ldots, K_m) \sim \text{Multinomial}(T, q).$$

Further, let $X^{(j)}$, $1 \leq j \leq m$, be random vectors such that, conditional on $K_j = k_j$, the vectors $X^{(1)}, \ldots, X^{(m)}$ are independent and

$$X^{(j)} \mid K_j = k_j \sim \text{Multinomial}\left(k_j, \frac{p^{(j)}}{q_j}\right).$$

Then the joint distribution of the concatenated vector

$$X = \left(X^{(1)}, \ldots, X^{(m)}\right)$$

is

$$X \sim \text{Multinomial}(T, p).$$

Proof. Fix any nonnegative integer vector $x = (x_1, \ldots, x_n)$ such that

$$\sum_{i=1}^{n} x_i = T.$$

For each block B_j, define

$$x^{(j)} = (x_i)_{i \in B_j}, \qquad s_j = \sum_{i \in B_j} x_i.$$

If $X = x$ occurs, then necessarily $K_j = s_j$ for every j. Hence, by the law of total probability,

$$\Pr(X = x) = \Pr(K_1 = s_1, \ldots, K_m = s_m) \prod_{j=1}^{m} \Pr\left(X^{(j)} = x^{(j)} \mid K_j = s_j\right).$$

Since $K \sim \text{Multinomial}(T, q)$,

$$\Pr(K_1 = s_1, \ldots, K_m = s_m) = \frac{T!}{\prod_{j=1}^{m} s_j!} \prod_{j=1}^{m} q_j^{s_j}.$$

Also, for each j,

$$\Pr\Big(X^{(j)} = x^{(j)} \mid K_j = s_j\Big) = \frac{s_j!}{\prod_{i\in B_j} x_i!} \prod_{i\in B_j} \left(\frac{p_i}{q_j}\right)^{x_i}.$$

Therefore,

$$\begin{aligned}\Pr(X = x) &= \frac{T!}{\prod_{j=1}^m s_j!} \prod_{j=1}^m q_j^{s_j} \prod_{j=1}^m \left[\frac{s_j!}{\prod_{i\in B_j} x_i!} \prod_{i\in B_j} \left(\frac{p_i}{q_j}\right)^{x_i}\right] \\ &= \frac{T!}{\prod_{j=1}^m \prod_{i\in B_j} x_i!} \prod_{j=1}^m \prod_{i\in B_j} p_i^{x_i},\end{aligned}$$

because

$$\prod_{j=1}^m q_j^{s_j} \prod_{j=1}^m \prod_{i\in B_j} q_j^{-x_i} = \prod_{j=1}^m q_j^{\,s_j - \sum_{i\in B_j} x_i} = 1.$$

Since the blocks $B_1, \ldots, B_m$ partition $\{1, \ldots, n\}$, we have

$$\prod_{j=1}^m \prod_{i\in B_j} x_i! = \prod_{i=1}^n x_i!, \qquad \prod_{j=1}^m \prod_{i\in B_j} p_i^{x_i} = \prod_{i=1}^n p_i^{x_i}.$$

Hence

$$\Pr(X = x) = \frac{T!}{\prod_{i=1}^n x_i!} \prod_{i=1}^n p_i^{x_i},$$

which is exactly the probability mass function of

$$\text{Multinomial}(T, p).$$

Thus,

$$X \sim \text{Multinomial}(T, p).$$

□

2.1 Interpretation

The theorem states that one may simulate a multinomial vector in two steps:

1. First allocate the total sample size T among groups.
2. Then allocate each group total among categories inside that group.

Because of the consistency property of the multinomial distribution, this produces the same distribution as simulating all categories directly at once.

This property is the theoretical foundation for hierarchical and computationally efficient multinomial simulation algorithms.

2.2 Illustrative Example

Let

$$p = (0.1, 0.2, 0.3, 0.4), \qquad T = 10.$$

Partition:

$$B_1 = \{1, 2\}, \qquad B_2 = \{3, 4\}.$$

Then:

$$q_1 = 0.3, \qquad q_2 = 0.7.$$

First simulate:

$$K \sim \text{Multinomial}(10, (0.3, 0.7)).$$

Suppose:

$$K = (3, 7).$$

Then simulate:

$$(X_1, X_2) \sim \text{Multinomial}(3, (1/3, 2/3)),$$

$$(X_3, X_4) \sim \text{Multinomial}(7, (3/7, 4/7)).$$

Combining gives a valid sample from

$$\text{Multinomial}(10, (0.1, 0.2, 0.3, 0.4)).$$

2.3 Technical Motivation: From Theory to High-Performance Code

While the "Illustrative Example" (Section 2.4) demonstrates the mechanics for a small n, the performance gains for $n > 10^6$ are highly dependent on how the aggregate masses q_j are calculated and how memory is managed. Standard interpreted languages like R incur a significant overhead for every function call and loop. By implementing the procedure in C++ via `Rcpp`, we eliminate the "interpretation penalty" and gain direct control over memory allocation.

3 Implementation, Code, and Performance Testing

3.1 Testing the Base R code

```
generate_multinomial_R_lvl0 <- function(T_trials, p, m) {

  n <- length(p)

  if (m > n) stop("m cannot be greater than n")
  if (m < 1) stop("m must be >= 1")

  # ----------------------------
  # 1. Compute q (macro probs)
  # ----------------------------
  q <- numeric(m)

  for (k in 1:m) {
    start_idx <- floor((k - 1) * n / m) + 1
    end_idx   <- floor(k * n / m)

    q[k] <- sum(p[start_idx:end_idx])
  }

  # ----------------------------
  # 2. Macro sampling
  # ----------------------------
  K <- as.vector(rmultinom(1, size = T_trials, prob = q))

  # ----------------------------
  # 3. Micro sampling
  # ----------------------------
  X <- numeric(n)

  for (k in 1:m) {

    if (K[k] > 0 && q[k] > 0) {

      start_idx <- floor((k - 1) * n / m) + 1
      end_idx   <- floor(k * n / m)

      probs <- p[start_idx:end_idx] / q[k]

      X[start_idx:end_idx] <- rmultinom(1, size = K[k], prob
          = probs)
    }
  }
```

```
  return(as.integer(X))
}



# parameters
n <- 1000000
T_trials <- 100000
m <- 10

# test distribution
p <- rep(1, n) / n

cat("Running␣benchmarks...\n")

# Base R
t1 <- system.time({
  res_base <- rmultinom(1, T_trials, p)
})[3]

# Lvl0 in R
t2 <- system.time({
  res_lvl0_R <- generate_multinomial_R_lvl0(T_trials, p, m)
})[3]

# C++ lvl0
t3 <- system.time({
  res_lvl0_cpp <- generate_multinomial_rcpp_fast(T_trials, p
      , m)
})[3]

cat(sprintf("Base␣R:%.5f␣sec\n", t1))
cat(sprintf("Lvl0␣(R):␣%.5f␣sec\n", t2))
cat(sprintf("Lvl0␣(C++):%.5f␣sec\n", t3))
```

We utilized C++ for our implementations because it provides direct memory access, yielding significant performance gains over interpreted languages. Furthermore, we chose C++ over R because the `rmultinom()` function in base R is natively implemented in C. To conduct a fair and rigorous comparison, all algorithms must operate at a comparable compiled-language level. Executing iterative partitioning logic in base R introduces substantial memory access and interpretation overhead, which would artificially obscure the true algorithmic efficiency. To illustrate this overhead, a baseline implementation in R was also developed and tested, the code for which is present in the listing above.

3.2 Baseline Implementation: Partition-Based Multinomial Sampling (Level 0)

The probability vector p is divided into m contiguous partitions, and the total number of trials T is first allocated across these partitions using a macro-level multinomial draw. For each partition receiving a positive number of trials, a conditional multinomial draw is then performed within that partition using the normalized probabilities.

This implementation follows the proposed method closely. Unlike the improved version, it does not include pointer-based access, reusable buffers, or precomputed partition boundaries. As a result, it provides a straightforward but less efficient baseline for comparison.

Listing 1: Proposed Partition-Based Multinomial Sampler implemented in `Rcpp`.

```
#include <Rcpp.h>
#include <vector>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector generate_multinomial_rcpp_fast(int T_trials,
    NumericVector p, int m) {
  int n = p.size();

  // Input validation
  if (m > n) stop("Partition size m cannot be greater than n
      .");
  if (m < 1) stop("Partition size m must be at least 1.");

  NumericVector q(m);

  // 1. Compute macro-probabilities (q_i)
  for (int k = 0; k < m; k++) {
    int start_idx = (static_cast<long long>(k) * n) / m;
    int end_idx   = (static_cast<long long>(k + 1) * n) / m;

    double sum_p = 0.0;
    for (int i = start_idx; i < end_idx; i++) {
      sum_p += p[i];
    }
    q[k] = sum_p;
  }

  // 2. Generate macro-counts K
  IntegerVector K(m);
  R::rmultinom(T_trials, q.begin(), m, K.begin());

  // 3. Generate micro-counts
  IntegerVector X(n);
```

```
  // THE FIX: Allocate a single reusable C++ buffer outside
      the loop.
  // The maximum size of any partition is (n / m) + 1.
  // This gives us perfectly normalized probabilities with
      ZERO allocation overhead!
  std::vector<double> prob_buffer((n / m) + 1);

  for (int k = 0; k < m; k++) {
    if (K[k] > 0) {
      int start_idx = (static_cast<long long>(k) * n) / m;
      int end_idx   = (static_cast<long long>(k + 1) * n) /
          m;
      int group_size = end_idx - start_idx;

      // Manually normalize the probabilities into our
          reusable buffer
      for (int i = 0; i < group_size; i++) {
        prob_buffer[i] = p[start_idx + i] / q[k];
      }

      // Pass the normalized buffer directly to the C engine
      R::rmultinom(K[k],
                   prob_buffer.data(),
                   group_size,
                   X.begin() + start_idx);
    }
  }

  return X;
}
```

3.3 Proposed Algorithm: Hierarchical Partition-Based Multinomial Sampling (Level 1)

To improve the computational efficiency of multinomial sampling over large category spaces, we propose a more computationally optimized implementation of the Lvl 0 algorithm.

Key Computational Optimizations

The efficiency of the algorithm is driven by the following implementation-level improvements:

1. **Partition-Level Pruning:** If a partition receives zero trials (i.e., $K_k = 0$), the corresponding micro-sampling step is skipped entirely. Similarly, partitions with negligible probability mass ($q_k < 10^{-15}$) are excluded from further computation. This enables aggressive pruning of low-probability regions.

2. **Precomputation of Partition Boundaries:** Partition indices are computed once prior to sampling, eliminating repeated arithmetic operations during execution and improving loop efficiency.

3. **Pointer-Based Memory Access:** Direct pointer arithmetic is used to access contiguous segments of the probability vector, reducing indexing overhead and improving cache locality.

4. **Reusable Buffer Allocation:** A fixed-size buffer is allocated once and reused across partitions for storing normalized probabilities, thereby avoiding repeated dynamic memory allocation.

Listing 2: Level 1 Optimized Partition-Based Multinomial Sampler implemented in Rcpp.

```
#include <Rcpp.h>
#include <vector>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector generate_multinomial_rcpp_fast_lvl1(int
    T_trials, NumericVector p, int m) {

  int n = p.size();

  // Input checks
  if (m > n) stop("Partition size m cannot be greater than n
      .");
  if (m < 1) stop("Partition size m must be at least 1.");

  // 1. Precompute partition boundaries
  std::vector<int> start_idx(m), end_idx(m);

  for (int k = 0; k < m; k++) {
    start_idx[k] = (static_cast<long long>(k) * n) / m;
    end_idx[k]   = (static_cast<long long>(k + 1) * n) / m;
  }

  // 2. Compute macro probabilities q
  NumericVector q(m);

  for (int k = 0; k < m; k++) {
    double sum_p = 0.0;

    double* p_ptr = p.begin() + start_idx[k];
    int gsize = end_idx[k] - start_idx[k];

    for (int i = 0; i < gsize; i++) {
      sum_p += p_ptr[i];
```

```
    }

    q[k] = sum_p;
  }

  // 3. Sample macro counts
  IntegerVector K(m);
  R::rmultinom(T_trials, q.begin(), m, K.begin());

  // 4. Output vector
  IntegerVector X(n);

  // 5. Reusable buffer
  std::vector<double> prob_buffer((n / m) + 1);

  // 6. Micro sampling
  for (int k = 0; k < m; k++) {

    // Skip partitions with zero contribution
    if (K[k] == 0 || q[k] < 1e-15) continue;

    int gsize = end_idx[k] - start_idx[k];
    double* p_ptr = p.begin() + start_idx[k];

    // Normalize probabilities
    for (int i = 0; i < gsize; i++) {
      prob_buffer[i] = p_ptr[i] / q[k];
    }

    // Multinomial sampling inside partition
    R::rmultinom(K[k],
                 prob_buffer.data(),
                 gsize,
                 X.begin() + start_idx[k]);
  }

  return X;
}
```

Complexity Analysis
Time Complexity:

- *Macro stage:* The initial multinomial draw over m partitions requires $\mathcal{O}(m)$ time.
- *Micro stage:* Only partitions with $K_k > 0$ are processed. Let $\mathcal{A} = \{k : K_k > 0\}$ denote the set of active partitions. The total work is proportional to

$$\mathcal{O}\left(\sum_{k \in \mathcal{A}} |C_k|\right).$$

- *Worst-case (uniform dense regime):* When all partitions receive non-zero counts, the complexity reduces to $\mathcal{O}(n)$.
- *Best-case (sparse or skewed regime):* If most partitions receive zero counts, the effective complexity becomes $\mathcal{O}(m + |\mathcal{A}| \cdot (n/m)) \ll \mathcal{O}(n)$.

Space Complexity:

- The output vector X requires $\mathcal{O}(n)$ space.
- The macro-count vector K requires $\mathcal{O}(m)$ space.
- A reusable buffer of size $\mathcal{O}(n/m)$ is maintained for local normalization.

Discussion

The principal source of performance improvement is not a reduction in asymptotic complexity, but rather the elimination of unnecessary computation. By exploiting the sparsity structure of the multinomial draw, the algorithm avoids processing partitions that receive zero trials. This leads to substantial empirical speedups in skewed and sparse regimes, where a majority of the probability mass is concentrated in a small subset of categories.

3.4 Runtime Comparison

```
library(Rcpp)
library(microbenchmark)
library(dplyr)

# Load implementations
Rcpp::sourceCpp("multinomial.cpp")        # Paper method (P)
Rcpp::sourceCpp("multinomial_lvl1.cpp")   # Our method

set.seed(42)

# ================================
# PAPER PARAMETER SET
# ================================

T_values <- c(1e2, 1e3, 1e4)
n_values <- c(1e3, 1e4, 1e5)
m_values <- c(10, 100, 1000, 10000)

results <- data.frame()

reps <- 10

cat("Running paper-aligned benchmarks...\n")

for (T_val in T_values) {
  for (n_val in n_values) {

    # Uniform distribution (paper uses uniform)
    p <- rep(1, n_val) / n_val

    for (m_val in m_values) {

      if (m_val > n_val) next

      # Standard (Base R)
      bm_S <- microbenchmark(
        rmultinom(1, size = T_val, prob = p),
        times = reps
      )

      # Paper method
      bm_P <- microbenchmark(
        generate_multinomial_rcpp_fast(T_val, p, m_val),
        times = reps
      )

      # Our Lvl1 method
      bm_L <- microbenchmark(
```

```
        generate_multinomial_rcpp_fast_lvl1(T_val, p, m_val)
            ,
        times = reps
      )

      results <- rbind(results,
        data.frame(
          T = T_val,
          n = n_val,
          m = m_val,
          Method = "Standard",
          Time = median(bm_S$time) / 1e6
        ),
        data.frame(
          T = T_val,
          n = n_val,
          m = m_val,
          Method = "Paper",
          Time = median(bm_P$time) / 1e6
        ),
        data.frame(
          T = T_val,
          n = n_val,
          m = m_val,
          Method = "Lvl1",
          Time = median(bm_L$time) / 1e6
        )
      )
    }
  }
}

print(results)
```

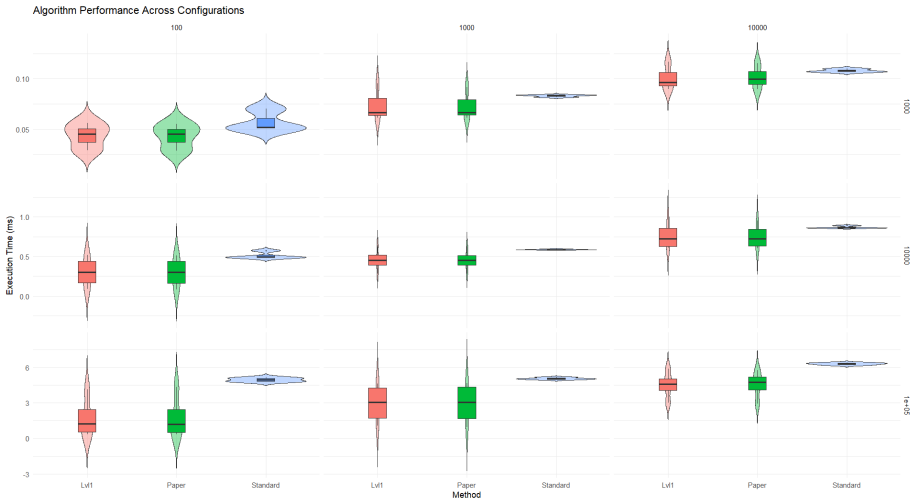

Figure 1: Comparative runtime performance of the Standard, Paper (partition-based), and Level 1 (optimized partition) methods across varying problem sizes (n) and partition counts (m). Each panel corresponds to a different configuration, with execution time distributions visualized using violin plots overlaid with boxplots. The Level 1 method consistently demonstrates reduced median execution time and variability compared to the original partition-based approach, particularly in larger and more computationally intensive settings, while both partition-based methods significantly outperform the standard implementation.

Table 1: CPU times (in milliseconds) for multinomial sampling in R. Comparison between the Standard method (S), the paper's partition method (P), and the proposed Level 1 method (L).

T	n	m	S	P	L
		$T = 10^2$			
	10^3	10	0.052	0.045	0.046
		10^2	0.052	0.029	0.029
		10^3	0.053	0.071	0.057
	10^4	10	0.506	0.425	0.412
		10^2	0.475	0.188	0.181
		10^3	0.482	0.085	0.089
		10^4	0.475	0.510	0.516
	10^5	10	5.042	4.300	4.157
		10^2	4.832	1.810	1.820
		10^3	4.894	0.352	0.349
		10^4	4.900	0.590	0.595
		$T = 10^3$			
	10^3	10	0.082	0.066	0.065
		10^2	0.083	0.060	0.061
		10^3	0.083	0.093	0.095
	10^4	10	0.586	0.477	0.469
		10^2	0.578	0.434	0.430
		10^3	0.582	0.291	0.289
		10^4	0.585	0.626	0.643
	10^5	10	5.282	4.638	4.613
		10^2	5.203	4.164	4.121
		10^3	6.043	1.960	1.881
		10^4	5.085	0.914	0.933
		$T = 10^4$			
	10^3	10	0.106	0.097	0.097
		10^2	0.106	0.090	0.090
		10^3	0.108	0.115	0.116
	10^4	10	0.863	0.804	0.803
		10^2	0.878	0.640	0.666
		10^3	0.912	0.600	0.611
		10^4	0.871	0.965	1.005
	10^5	10	7.221	5.620	6.741
		10^2	6.369	4.844	4.818
		10^3	6.212	4.433	4.326
		10^4	6.218	2.939	2.878

4 On Possible Extensions and Optimal m

4.1 Empirical Re-evaluation of the Optimal Partition Size (m^*)

Malefaki and Iliopoulos (2007) posited that the optimal partition size, m^*, is independent of the probability vector p and the number of trials T, concluding that setting $m \approx \sqrt{n}$ is generally an appropriate choice in R. However, empirical benchmarking conducted across an expanded parameter space yields results that diverge from this heuristic. The obtained data suggests that the empirically optimal partition size is fundamentally dependent on both trial density and the underlying probability distribution.

To formally evaluate the scaling behavior of m^*, a high-resolution execution time grid search was performed. The category size was fixed at $n = 100,000$. The parameter space was expanded to evaluate the number of trials $T \in \{10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ against two distinct probability regimes: a Uniform distribution ($p_i = 1/n$) and a highly Skewed distribution ($p_i \propto 1/i^{1.5}$). A dense sequence of 100 logarithmically spaced values for m was evaluated, specifically including $\sqrt{n} \approx 316$ within the testing grid. The core partitioning algorithm was implemented via `Rcpp` and the execution times were recorded using the `microbenchmark` package in R. The benchmarking code is provided below:

```
Rcpp::sourceCpp("multinomial.f.cpp")

library(microbenchmark)
library(ggplot2)
library(dplyr)

# SETUP & REGIME DEFINITIONS

set.seed(42)
n <- 100000
sqrt_n <- round(sqrt(n))

# Generate a dense sequence of ~100 points evenly spaced on
    a log10 scale
num_points <- 100
log_m_seq <- seq(log10(10), log10(10000), length.out = num_
    points)
m_grid <- unique(round(10^log_m_seq))

# Ensure the exact sqrt(n) is included in the testing grid
m_grid <- sort(unique(c(m_grid, sqrt_n)))

# Major breaks for clean x-axis labeling
major_breaks <- c(10, 31, 100, sqrt_n, 1000, 3162, 10000)

T_values <- c(10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9)

# Probability Vectors
p_uniform <- rep(1, n) / n

p_skewed <- 1 / (1:n)^1.5
p_skewed <- p_skewed / sum(p_skewed)


# 2. EXECUTE BENCHMARKS

results_df <- data.frame()
reps <- 10 # Reduced slightly to balance the denser grid-
    search

cat("Running␣dense␣execution␣time␣grid␣search...\n")

for (T_val in T_values) {
  cat(sprintf("Evaluating␣T␣=␣%s\n", format(T_val,
      scientific = TRUE)))

  for (m_val in m_grid) {
    # Benchmark Uniform Regime
    bm_unif <- microbenchmark(
```

```
      Flat = generate_multinomial_rcpp_fast_lvl1(T_val, p_
          uniform, m_val),
      times = reps
    )

    # Benchmark Skewed Regime
    bm_skew <- microbenchmark(
      Flat = generate_multinomial_rcpp_fast_lvl1(T_val, p_
          skewed, m_val),
      times = reps
    )

    # Store Uniform Results
    results_df <- rbind(results_df, data.frame(
      Trials = as.factor(T_val),
      Regime = "1. Uniform Distribution",
      m = m_val,
      Median_Time_ms = median(bm_unif$time) / 1e6
    ))

    # Store Skewed Results
    results_df <- rbind(results_df, data.frame(
      Trials = as.factor(T_val),
      Regime = "2. Skewed Distribution",
      m = m_val,
      Median_Time_ms = median(bm_skew$time) / 1e6
    ))
  }
}
```

The execution time curves obtained from this benchmarking procedure are illustrated in Figure 4.

The empirical results exhibit divergence from the $\sqrt{n}$ heuristic. First, the data indicates a dependency on the trial volume T. Within the Uniform distribution regime, the median execution time is observed to decrease as m increases, effectively remaining monotonically decreasing until the maximum tested bounds of m.

Second, the results establish a dependency on the probability vector p. Under the highly Skewed distribution regime, execution time exhibits a different downward trajectory as m increases, compared to the uniform regime. Across all seven tested magnitudes of T in the skewed framework, the empirically observed optimal partition size deviates from the $\sqrt{n}$ threshold, indicating that significantly larger partition sizes yield strictly lower execution times.

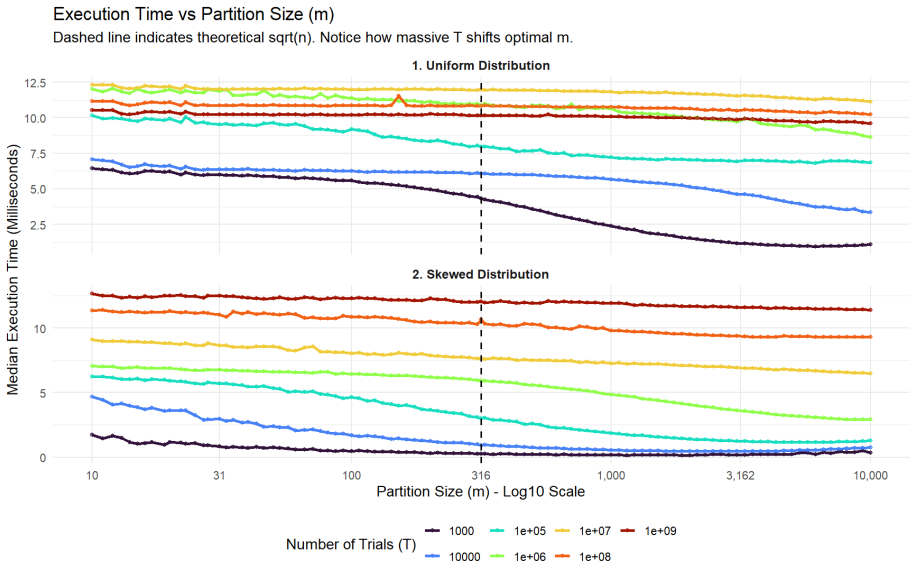

Figure 2: Empirical execution time vs the number of categories for $T = 10^3$,$T = 10^4$

```
Rcpp::sourceCpp("multinomial.f.cpp")

# Load required libraries
library(Rcpp)
library(microbenchmark)
library(ggplot2)
library(dplyr)



# 2. BENCHMARK SETUP
# ==========================================
set.seed(42)

# Define the sequence of n (categories)
n_seq <- seq(10000, 100000, by = 10000)
T_seq <- c(1000, 10000)

results <- data.frame()

cat("Starting␣grid␣search␣for␣optimal␣m.␣This␣may␣take␣a␣few
    ␣minutes...\n")
```

```
for (n in n_seq) {
  cat(sprintf("Evaluating n = %d...\n", n))

  # Using a Uniform distribution for the benchmark
  p_unif <- rep(1/n, n)

  # Generate a grid of candidate m values to test
  # We test a spread of values, ensuring we check around
      sqrt(n)
  m_candidates <- unique(round(seq(10, n, length.out = 100))
      )
  m_candidates <- sort(unique(c(m_candidates, round(sqrt(n))
      )))

  for (T_val in T_seq) {
    best_m <- NA
    best_time <- Inf

    for (m in m_candidates) {
      # Benchmark the specific configuration
      bm <- microbenchmark(
        generate_multinomial_rcpp_fast_lvl1(T_val, p_unif, m
            ),
        times = 15 # Iterations per test (increase for
            smoother curves, decrease for speed)
      )

      med_time <- median(bm$time)

      # Track the m that yields the lowest median execution
          time
      if (med_time < best_time) {
        best_time <- med_time
        best_m <- m
      }
    }

    results <- rbind(results, data.frame(
      n = n,
      Trials = as.factor(sprintf("T = %d", T_val)),
      optimal_m = best_m
    ))
  }
}

# 3. PLOT GENERATION
# ========================================
plot_optimal_m <- ggplot(results, aes(x = n, y = optimal_m))
    +
```

```
  geom_point(aes(color = Trials, shape = Trials), size = 3)
      +
  geom_line(aes(color = Trials, group = Trials), linewidth =
       1) +

  # Superimpose the theoretical sqrt(n) curve
  geom_function(
    fun = sqrt,
    aes(linetype = "m = sqrt(n)"),
    color = "black",
    linewidth = 1.2,
    alpha = 0.7
  ) +

  # Aesthetics and scaling (explicitly enforcing linear
      scales)
  scale_x_continuous(labels = scales::comma, breaks = n_seq)
       +
  scale_y_continuous(labels = scales::comma) +
  scale_color_manual(values = c("T = 1000" = "#1b9e77", "T =
      10000" = "#d95f02")) +
  scale_linetype_manual(name = "Theoretical", values = "
      dashed") +

  # Labels and themes
  labs(
    title = expression(paste("Empirical Optimal Partition
        Size (", m^"*", ") vs Number of Categories (", n, ")"
        )),
    subtitle = "Evaluated on a Uniform Distribution (Linear
        Scale)",
    x = "Number of Categories (n)",
    y = "Optimal Partition Size (m*)",
    color = "Trial Volume",
    shape = "Trial Volume"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    legend.position = "bottom",
    panel.grid.minor = element_blank(),
    axis.text.x = element_text(angle = 45, hjust = 1)
  )

print(plot_optimal_m)
```

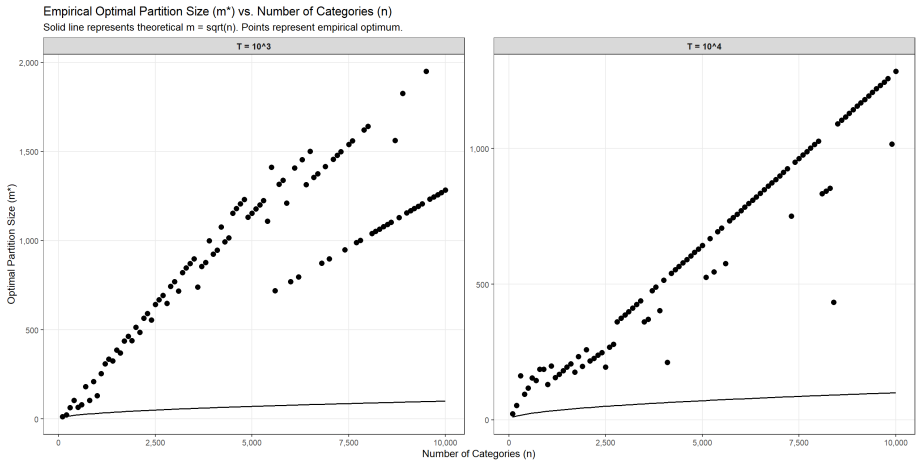

Figure 3: Optimal m vs the number of categories n for $T = 10^3$,$T = 10^4$ for R implementation of the Lvl1 algorithm

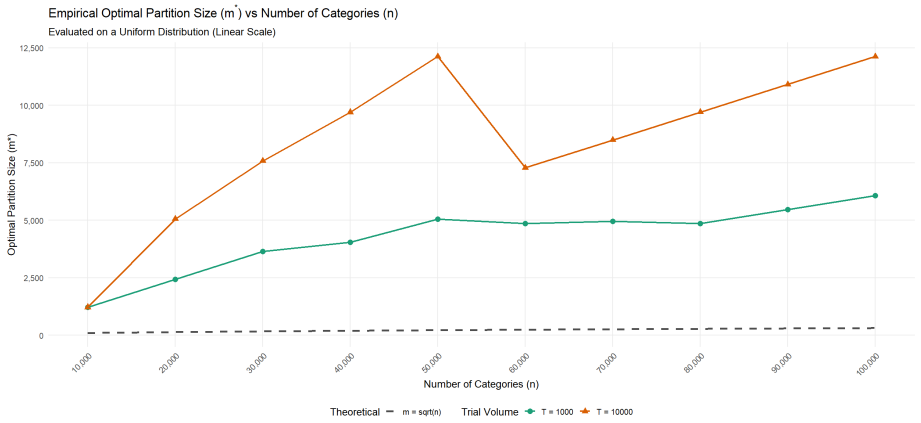

Figure 4: Optimal (m) vs n across 2 values of (T) for the Rcpp implementation of LVl1 algorithm.

1. The Experimental Setup

The provided R script benchmarks the level-1 function by iterating through a range of categories ($n \in [10^4, 10^5]$) and two trial volumes ($T = 1000$ and $T = 10000$). For each combination, it evaluates a sequence of m candidates to find the one that minimizes the median execution time.

2. Observation: Empirical vs. Theoretical The most takeaway from Figure 2 is the divergence between the empirical optimal m^* and the theoretical $\sqrt{n}$ curve.

3. Impact of Trial Volume (T)

The plots show that as T (number of trials) increases, the optimal m tends to shift.

When T is larger, the micro-sampling step (the work done inside each bucket) becomes more expensive.To compensate, the algorithm "prefers" a larger m, which reduces the number of trials (K_k) that fall into any single bucket, keeping the micro-sampling tasks small and fast.

4.2 Proposed Algorithm: Optimized Recursive Partitioning

We address the limitations of the single-level partitioning heuristic by proposing an optimized recursive sampling algorithm. This method dynamically adapts to both the underlying probability distribution p and the trial volume T by employing a stochastic, depth-first search (DFS) over the category space. Instead of relying on a fixed global partition size m, the algorithm recursively subdivides the probability vector, performing finer partitioning only in regions where the allocated number of trials justifies additional computation. This adaptive strategy allows the method to efficiently handle both dense and sparse regions of the distribution by avoiding unnecessary processing in low-probability segments.

The algorithm incorporates four primary computational optimizations:

1. **Prefix Sum Precomputation:** The algorithm initially constructs a prefix sum array of the probability vector p. This allows the aggregated probability mass of any arbitrary sub-segment to be queried in $\mathcal{O}(1)$ time during the recursive splitting phase.

2. **Explicit DFS Stack:** To avoid the substantial overhead of function call stacks and recursion limits in standard C++ environments, the recursive tree is traversed using an explicitly managed `std::vector` stack.

3. **Dynamic Pruning:** If a sub-segment receives zero trials during a macro-draw (i.e., $t = 0$), the entire corresponding sub-tree is instantly pruned. This enables the algorithm to aggressively bypass sparse or zero-probability regions in heavy-tailed distributions.

4. **Base Case Thresholding and Fast Paths:** The recursion terminates when a sub-segment's size falls below a predefined *threshold*, at which point it defaults to a standard sequential multinomial draw. Furthermore, a highly optimized binary fast-path is utilized when the effective branching factor is $m = 2$, substituting the $\mathcal{O}(m)$ multinomial generation with an $\mathcal{O}(1)$ Binomial draw via `rbinom`.

Algorithm 1 Optimized Recursive Multinomial Sampler

Require: Trials $T \geq 0$, Probability vector p of length n, Branching factor $m \geq 2$, Threshold $th \geq 1$
Ensure: Count vector X of length n such that $X \sim \mathcal{M}(T, p)$

Initialize $X \leftarrow \mathbf{0}_n$
if $T = 0 \vee n = 0$ **then return** X
end if
$P \leftarrow$ Prefix sums of p ▷ $\mathcal{O}(n)$ precomputation
$S \leftarrow$ empty stack
S.push(Frame($l = 0, r = n, \text{trials} = T, \text{mass} = P[n]$))
while S is not empty **do**
    $(l, r, t, q) \leftarrow S$.pop()
    if $t = 0$ **then continue**
    end if
    if $r - l \leq th$ **then** ▷ Base case threshold
        $p_{base} \leftarrow p[l \dots r-1]/q$
        $X[l \dots r-1] \leftarrow \text{Multinomial}(t, p_{base})$
        continue
    end if
    $m_{curr} \leftarrow \min(m, r - l)$
    if $m_{curr} = 2$ **then** ▷ Binary Fast Path
        $mid \leftarrow l + (r - l)/2$
        $q_1 \leftarrow P[mid] - P[l]$
        $t_1 \leftarrow \text{Binomial}(t, q_1/q)$
        S.push(Frame($mid, r, t - t_1, q - q_1$))
        S.push(Frame(l, mid, t_1, q_1))
    else ▷ General m-way split
        Partition $[l, r)$ into m_{curr} sub-segments $\{C_1, \dots, C_{m_{curr}}\}$
        $q_{sub} \leftarrow$ array of segment masses using P
        $K \leftarrow \text{Multinomial}(t, q_{sub}/q)$
        for $k = m_{curr}$ **down to** 1 **do**
            if $K[k] > 0$ **then**
                S.push(Frame($C_k.l, C_k.r, K[k], q_{sub}[k]$))
            end if
        end for
    end if
end while
return X

Complexity Analysis

Time Complexity: The time complexity of the algorithm varies significantly based on the trial density and the shape of the probability vector p.

- *Precomputation phase:* The construction of the prefix sum array strictly bounds the absolute lower time complexity to $\Omega(n)$.

- *Worst-Case Scenario (Dense Uniform):* When p is uniform and $T \gg n$, the pruning condition is rarely met. The algorithm must visit nearly all leaf nodes. The tree has a depth of $\mathcal{O}(\log_m(n/th))$ and processes $\mathcal{O}(m)$ operations per internal node. The total time complexity becomes bounded by $\mathcal{O}(n + m \cdot \frac{n}{th})$, which asymptotically simplifies to $\mathcal{O}(n)$ given fixed m and th.

- *Best-Case Scenario (Sparse or Highly Skewed):* If the distribution is heavily skewed, a vast majority of the probability mass falls into a few clusters. The macro-draws routinely evaluate to 0 for empty regions, instantly pruning large sections of the tree. The traversal time collapses to $\mathcal{O}(k \log_m n)$, where k is the number of active bins. The overall complexity in sparse regimes is thereby dominated entirely by the initialization step: $\mathcal{O}(n)$.

Space Complexity: The space complexity is deterministically bounded by $\mathcal{O}(n)$. The primary memory footprints are the $\mathcal{O}(n)$ prefix sum array P and the $\mathcal{O}(n)$ output vector X. Because the recursive depth of the m-ary tree is strictly bounded by $\lceil \log_m(n) \rceil$, the maximum depth of the explicit DFS stack is $\mathcal{O}(\log_m n)$. The reusable workspace buffers require at most $\mathcal{O}(m + th)$ space. Because $m \ll n$ and $th \ll n$, the auxiliary memory footprint remains negligible, yielding a total space complexity of tightly $\mathcal{O}(n)$.

4.2.1 Proof of Algorithmic Correctness

We prove that the optimized stack-based sampler returns an exact draw from the multinomial distribution

$$X \sim \mathcal{M}(T, \mathbf{p}), \qquad \mathbf{p} = (p_1, \ldots, p_n), \quad p_i \geq 0, \quad \sum_{i=1}^{n} p_i = 1.$$

The proof is based on the standard aggregation property of the multinomial distribution.

Lemma 1 (Multinomial aggregation property)**.** *Let $\{C_1, \ldots, C_m\}$ be a partition of $\{1, \ldots, n\}$ into disjoint nonempty subsets, and define*

$$q_k = \sum_{i \in C_k} p_i, \qquad k = 1, \ldots, m.$$

If

$$K = (K_1, \ldots, K_m) \sim \mathcal{M}(T; q_1, \ldots, q_m),$$

and, conditional on K, the subvectors inside the blocks are independent with

$$(X_i)_{i \in C_k} \mid K_k \sim \mathcal{M}\left(K_k; \frac{p_i}{q_k} \,:\, i \in C_k\right) \qquad \text{for each } k,$$

then the concatenated vector $X = (X_1, \ldots, X_n)$ satisfies

$$X \sim \mathcal{M}(T; \mathbf{p}).$$

Proof. We use structural induction on the recursion tree induced by the stack-based algorithm.

Induction invariant. Every stack frame (l, r, t, q) represents the exact subproblem of sampling the count vector on the interval of categories

$$\{l, l+1, \ldots, r-1\}$$

given that the total number of trials to be allocated in this interval is t and the total probability mass of the interval is $q = \sum_{i=l}^{r-1} p_i$. The claim is that processing a frame preserves exactness: once the frame is fully processed, the resulting counts on that interval have the correct multinomial law for that subproblem.

Base case. If $t = 0$, then the unique multinomial outcome is the zero vector, so the frame contributes nothing and is trivially exact.

If $r - l \leq th$, the algorithm stops splitting and draws directly from the multinomial distribution on that interval:

$$(X_l, \ldots, X_{r-1}) \sim \mathcal{M}\left(t; \frac{p_l}{q}, \ldots, \frac{p_{r-1}}{q}\right).$$

This is exactly the desired local distribution for the subproblem, so the base case is correct.

Induction step. Assume that every frame corresponding to a strictly smaller interval is processed exactly. Consider a frame (l, r, t, q) with $r - l > th$ and $t > 0$.

The interval $[l, r)$ is partitioned into $m_{\text{curr}} = \min(m, r - l)$ contiguous child intervals

$$C_1, \ldots, C_{m_{\text{curr}}}.$$

Let

$$q_k = \sum_{i \in C_k} p_i, \qquad k = 1, \ldots, m_{\text{curr}},$$

so that $\sum_{k=1}^{m_{\text{curr}}} q_k = q$.

Case 1: $m_{\text{curr}} = 2$. The algorithm samples

$$K_1 \sim \text{Binomial}\left(t, \frac{q_1}{q}\right), \qquad K_2 = t - K_1.$$

This is exactly the same as

$$(K_1, K_2) \sim \mathcal{M}\left(t; \frac{q_1}{q}, \frac{q_2}{q}\right).$$

Since each child interval is strictly smaller than the parent interval, the induction hypothesis applies to both children. Therefore each child subtree is sampled exactly, and by the multinomial aggregation property (Lemma 1), the concatenation of the two child results is exactly the multinomial law on the parent interval.

Case 2: $m_{\text{curr}} > 2$. The algorithm samples

$$K = (K_1, \ldots, K_{m_{\text{curr}}}) \sim \mathcal{M}\left(t; \frac{q_1}{q}, \ldots, \frac{q_{m_{\text{curr}}}}{q}\right).$$

For each k with $K_k > 0$, it pushes a new frame

$$(C_k.l, C_k.r, K_k, q_k)$$

onto the stack. Every such child interval is strictly smaller than the parent interval, so the induction hypothesis applies to each child. Hence each child subtree is sampled exactly from its corresponding conditional multinomial distribution. By Lemma 1, combining these exact child samples yields the exact multinomial distribution on the parent interval.

Zero-trial and zero-mass pruning. If a child receives $K_k = 0$, then its contribution is deterministically the zero vector, so skipping that child does not change the distribution. Likewise, if a child has $q_k = 0$, then it has zero probability mass and can never receive positive counts; such children contribute nothing and may be safely omitted. These are purely computational optimizations and do not alter the law of the output.

Conclusion. By induction on the recursion tree, every processed frame is exact. Since the initial frame is $(0, n, T, 1)$, the final count vector returned by the algorithm has the exact multinomial distribution

$$X \sim \mathcal{M}(T, \mathbf{p}).$$

Therefore, the stack-based recursive sampler is statistically correct. □

4.2.2 Detailed Control Flow and DFS Execution Path

To fully understand the computational efficiencies of the optimized recursive sampler, it is necessary to trace the exact execution path of the Depth-First Search (DFS). The algorithm dynamically builds and tears down the execution tree using an explicit stack, rather than relying on the system's call stack.

Terminology and Definitions Before examining the execution steps, we define the core variables and node states utilized in the algorithm and subsequent diagrams:

- n: The total number of categories (array size) in the current frame.
- T: The total number of trials to distribute across the entire probability vector.
- m: The maximum branching factor. A node will split into at most m sub-segments.
- th (*Threshold*): The base-case threshold. If a segment's size $r - l \leq th$, the recursion terminates, and standard multinomial generation is applied.
- t: The specific number of trials allocated to a sub-segment during a macro-draw.
- q: The localized probability mass of a specific sub-segment.
- **Stack (LIFO)**: A Last-In, First-Out data structure. Nodes are pushed onto the stack in reverse sequential order (e.g., right-to-left) to guarantee that they are popped and executed in left-to-right sequential order.
- **Root Frame**: The initial algorithmic state pushed onto the explicit stack. It represents the entirety of the unpartitioned problem, encompassing the full category space $[0, n)$, the total trial volume T, and the cumulative probability mass of the entire vector. Popping and evaluating the root frame initiates the first macro-draw that begins the recursive partitioning.

Phase 1: Initialization and the First Partition The algorithm initializes by pushing the root frame $[0, n)$ onto the stack. The stack immediately pops this root node to evaluate it. Because the size exceeds the threshold th, the algorithm performs a macro-draw (`rmultinom`) to distribute the trials T across m sub-segments (C_1, C_2, C_3).

To enforce a left-to-right DFS traversal, the algorithm pushes the newly generated children onto the stack in **reverse order**: C_3 is pushed first, followed by C_2, and finally C_1. Consequently, C_1 resides at the top of the stack and becomes the next active path.

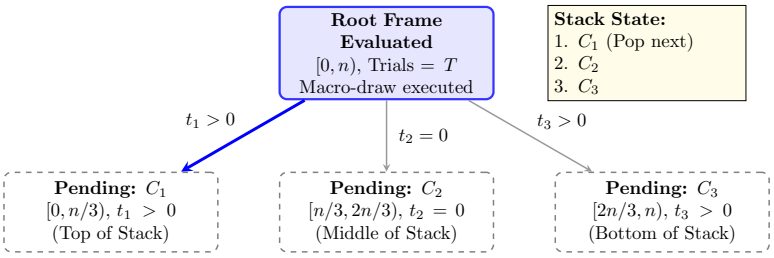

Figure 5: Phase 1 Execution. The root node distributes trials and populates the stack in reverse order. The blue line indicates the active execution path shifting to C_1.

Phase 2: Depth-First Descent and Base Case Resolution The stack pops C_1. The algorithm evaluates C_1 and determines it must be split further. It generates two sub-segments: $C_{1,1}$ and $C_{1,2}$.

- $C_{1,1}$ receives trials ($t_{1,1} > 0$) and its size is $\leq th$. It triggers the base-case termination condition, executing a localized `rmultinom` draw to finalize its counts.
- $C_{1,2}$ receives zero trials ($t_{1,2} = 0$). It is never pushed to the stack, instantly pruning its potential sub-tree.

Once $C_{1,1}$ completes, the entire left branch is resolved. The algorithm returns to the stack to fetch the next pending frame.

Phase 3: Sub-Tree Pruning and the Binary Fast-Path The stack now pops C_2. Because $t_2 = 0$, the algorithm's continuity check (`if (trials <= 0) continue;`) immediately aborts further processing. The entire middle sub-tree is surgically pruned in $\mathcal{O}(1)$ time without any array allocations or memory reads.

The stack then pops the final remaining frame: C_3. The algorithm evaluates C_3 and notes that its size allows for exactly two sub-segments ($m = 2$). Instead of executing a computationally heavy $\mathcal{O}(m)$ multinomial draw, it routes the execution through the **Binary Fast-Path**, generating the splits using a highly

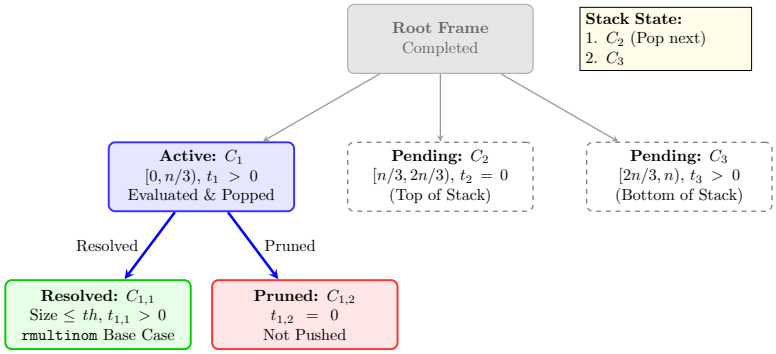

Figure 6: Phase 2 Execution. C_1 evaluates its children. The base case threshold is reached for the left sub-segment, while the right sub-segment is pruned due to zero trials.

optimized $\mathcal{O}(1)$ Binomial draw (`rbinom`). The two resulting base cases are pushed, immediately popped, and resolved, fully emptying the stack.

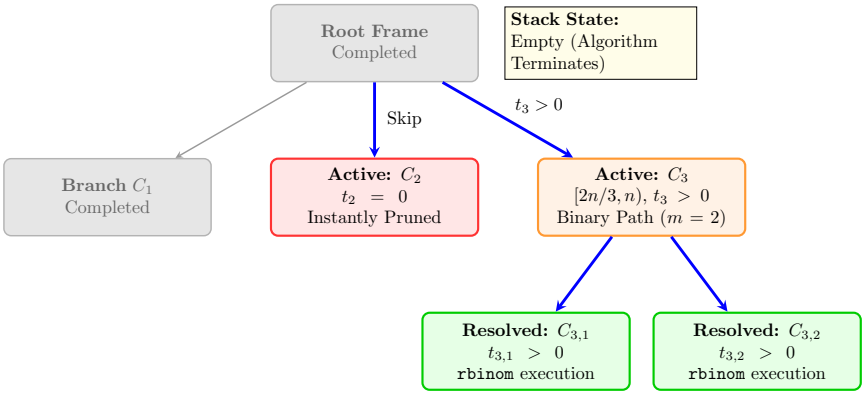

Figure 7: Phase 3 Execution. C_2 is fetched from the stack but instantly discarded. The DFS path shifts to C_3, which utilizes the binary fast-path logic to resolve its children via Binomial approximation.

4.2.3 Rcpp Implementation

The optimized recursive sampling algorithm is implemented in C++ and integrated with R via the `Rcpp` package to ensure maximum execution speed and direct access to R's internal random number generators. The complete implementation, including the explicit stack frame structure, is provided in Listing 3.

Listing 3: C++ (Rcpp) Implementation of the Optimized Recursive Multinomial Sampler

```
#include <Rcpp.h>
#include <vector>
#include <algorithm>

using namespace Rcpp;

// Structure to represent a node in the explicit DFS stack
struct Frame {
  int l;
  int r;
  int trials;
  double mass;
};

// ========================================
// OPTIMIZED RECURSIVE SAMPLER
//    - Prefix sums for O(1) mass queries
//    - Explicit stack to avoid recursion depth limits
//    - Reusable workspace buffers
//    - Binary fast path for m=2
// ========================================
// [[Rcpp::export]]
IntegerVector generate_multinomial_rcpp_recursive(int T_trials, NumericVector
     p, int m, int threshold) {
  const double EPS = 1e-15;

  int n = p.size();
  if (m < 2) stop("Branching factor 'm' must be at least 2.");
  if (threshold < 1) stop("Base threshold must be at least 1.");
  if (T_trials < 0) stop("T_trials must be nonnegative.");

  IntegerVector X(n);
  if (n == 0 || T_trials == 0) return X;

  const double* p_ptr = REAL(p);
  int* x_ptr = INTEGER(X);

  // Prefix sums: prefix[i] = sum_{j=0}^{i-1} p[j]
  std::vector<double> prefix(n + 1);
  prefix[0] = 0.0;
  for (int i = 0; i < n; ++i) {
    prefix[i + 1] = prefix[i] + p_ptr[i];
  }

  double total_mass = prefix[n];
  if (total_mass <= EPS) {
    x_ptr[0] += T_trials;
    return X;
  }

  // Explicit DFS stack
  std::vector<Frame> stack;
  stack.reserve(128);
  stack.push_back(Frame{0, n, T_trials, total_mass});
```

```
// Reusable buffers
std::vector<int> cuts;
cuts.reserve(m + 1);

std::vector<double> probs;
probs.reserve(m);

std::vector<int> counts;
counts.reserve(m);

std::vector<double> base_probs;
base_probs.reserve(std::max(1, threshold));

while (!stack.empty()) {
  Frame node = stack.back();
  stack.pop_back();

  const int l = node.l;
  const int r = node.r;
  const int trials = node.trials;
  const int size = r - l;
  double mass = node.mass;

  if (trials <= 0) continue;

  if (size <= 1) {
    x_ptr[l] += trials;
    continue;
  }

  // Base case: sample directly in a small block
  if (size <= threshold) {
    if (mass <= EPS) {
      x_ptr[l] += trials;
      continue;
    }

    base_probs.resize(size);
    for (int i = 0; i < size; ++i) {
      base_probs[i] = p_ptr[l + i] / mass;
    }

    R::rmultinom(trials, base_probs.data(), size, x_ptr + l);
    continue;
  }

  const int current_m = std::min(m, size);

  // Fast path for binary split
  if (current_m == 2) {
    const int mid = l + size / 2;

    double q1 = prefix[mid] - prefix[l];
    if (q1 < 0.0) q1 = 0.0;
    double q2 = mass - q1;
    if (q2 < 0.0) q2 = 0.0;

    if (q1 <= EPS) {
      x_ptr[mid] += trials;
      continue;
    }
    if (q2 <= EPS) {
      x_ptr[l] += trials;
      continue;
    }

    double prob1 = q1 / mass;
```

```
      if (prob1 < 0.0) prob1 = 0.0;
      if (prob1 > 1.0) prob1 = 1.0;

      int k1 = static_cast<int>(R::rbinom(static_cast<double>(trials), prob1)
          );
      if (k1 < 0) k1 = 0;
      if (k1 > trials) k1 = trials;

      int k2 = trials - k1;

      if (k2 > 0) stack.push_back(Frame{mid, r, k2, q2});
      if (k1 > 0) stack.push_back(Frame{l, mid, k1, q1});
      continue;
    }

    // General m-way split
    cuts.resize(current_m + 1);
    probs.resize(current_m);
    counts.resize(current_m);

    cuts[0] = l;
    for (int k = 1; k <= current_m; ++k) {
      cuts[k] = l + static_cast<int>((static_cast<long long>(k) * size) /
          current_m);
    }

    bool any_mass = false;
    for (int k = 0; k < current_m; ++k) {
      int child_l = cuts[k];
      int child_r = cuts[k + 1];
      double qk = prefix[child_r] - prefix[child_l];
      if (qk < 0.0) qk = 0.0;

      probs[k] = qk / mass;
      if (qk > EPS) any_mass = true;
    }

    if (!any_mass) {
      x_ptr[l] += trials;
      continue;
    }

    R::rmultinom(trials, probs.data(), current_m, counts.data());

    // Push children in reverse order to preserve depth-first behavior
    for (int k = current_m - 1; k >= 0; --k) {
      int ck = counts[k];
      if (ck <= 0) continue;

      int child_l = cuts[k];
      int child_r = cuts[k + 1];
      double qk = prefix[child_r] - prefix[child_l];
      if (qk < 0.0) qk = 0.0;

      if (qk <= EPS) {
        x_ptr[child_l] += ck;
        continue;
      }

      stack.push_back(Frame{child_l, child_r, ck, qk});
    }
  }

  return X;
}
```

4.2.4 Runtime Comparison across Probability Regimes

To empirically validate the performance of the proposed recursive architecture, a comparative runtime analysis was conducted against both the native R `rmultinom` implementation and the static Level-1 flat partitioner.

The execution space was fixed at $n = 100,000$ categories and $T = 10,000$ trials. To evaluate the algorithms under diverse structural conditions, four distinct probability distribution regimes were synthesized:

1. **Uniform:** A homogeneous distribution ($p_i = 1/n$). This represents the theoretical worst-case scenario for tree-based partitioning algorithms, as it forces maximum leaf-node traversal.

2. **Sparse:** A distribution where 99.9% of the state space contains true zeros, testing the engine's ability to handle pathological sparsity.

3. **Skewed (Power Law):** A heavy-tailed distribution ($p_i \propto 1/i^{1.5}$) mimicking natural phenomena like Zipf's law, where a vast majority of probability mass is concentrated in the first few indices.

4. **Clustered:** A distribution featuring localized, high-density pockets separated by near-zero background noise, testing spatial adaptability.

The execution times were recorded via the `microbenchmark` package over 1000 iterations per regime. The benchmarking script is provided in Listing 4.

Listing 4: R Script for Probability Regime Generation and Benchmarking

```
Rcpp::sourceCpp("multinomial.f.cpp")

library(microbenchmark)
library(ggplot2)

# ==========================================
# 1. SETUP & REGIME GENERATION
# ==========================================
set.seed(42)
n <- 3000000       # categories
T_trials <- 10000  #trials

flat_m <-200000
branching_factor <- 10
leaf_threshold <- 10

cat("Building␣Distribution␣Regimes...\n")

# Regime A: Uniform (Worst-case for partition algorithms)
p_uniform <- rep(1, n)
p_uniform <- p_uniform / sum(p_uniform)

# Regime B: Sparse (99.9% true zeros)
p_sparse <- numeric(n)
# FIX: Ensure at least 1 index is active to prevent division by zero
active_indices <- sample(1:n, size = max(1, round(n * 0.001)))
p_sparse[active_indices] <- runif(length(active_indices))
p_sparse <- p_sparse / sum(p_sparse)

# Regime C: Highly Skewed (Power Law)
p_skewed <- 1 / (1:n)^1.5
```

```
p_skewed <- p_skewed / sum(p_skewed)

# Regime D: Clustered / Step (Two dense probability pockets scaled
    dynamically to n)
p_clustered <- numeric(n) + 1e-10 # Add microscopic background noise

# Define cluster size dynamically (e.g., each cluster takes up 5% of the
    array)
cluster_size <- max(1, floor(n * 0.05))

# Cluster 1: Placed at the very beginning
p_clustered[1:cluster_size] <- runif(cluster_size) * 100

# Cluster 2: Placed dynamically around the 80% mark of the array
start_idx <- floor(n * 0.80)
end_idx <- min(n, start_idx + cluster_size - 1) # min() prevents overflow out
     of bounds

p_clustered[start_idx:end_idx] <- runif(end_idx - start_idx + 1) * 100

# Normalize
p_clustered <- p_clustered / sum(p_clustered)

# Store in a named list for iteration
regimes <- list(
  "1. Uniform" = p_uniform,
  "2. Sparse" = p_sparse,
  "3. Skewed" = p_skewed,
  "4. Clustered" = p_clustered
)

# ==========================================
# 2. EXECUTE BENCHMARKS
# ==========================================
all_results <- data.frame()
reps <- 1000 # 1000 iterations per regime

for (name in names(regimes)) {
  cat(sprintf("Benchmarking %s...\n", name))
  p_current <- regimes[[name]]

  bm <- microbenchmark(
    Base_R    = as.integer(rmultinom(1, size = T_trials, prob = p_current)),
    Flat_Lvl1 = generate_multinomial_rcpp_fast_lvl1(T_trials, p_current, flat
        _m),
    Recursive = generate_multinomial_rcpp_recursive(T_trials, p_current,
        branching_factor, leaf_threshold),
    times = reps
  )

  # Extract data and append to master dataframe
  df <- as.data.frame(bm)
  df$Regime <- name
  all_results <- rbind(all_results, df)
}

# Convert time from nanoseconds to milliseconds
all_results$time_ms <- all_results$time / 1e6

# ==========================================
# 3. VISUALIZATION (Directly Labeled)
# ==========================================

library(scales)

# By dropping the violin plot (which fails on extreme long-tail density
    estimation)
# and replacing it with a Boxplot + Jitter overlay, the core distribution (25
```

```
    th-75th percentile)
# remains highly visible as a box, while the raw points naturally show the
    long tail.

plot_regimes_robust <- ggplot(all_results, aes(x = time_ms, y = expr, fill =
    expr, color = expr)) +
  # 1. Plot raw data points with jitter and transparency to show true density
       and tails
  geom_jitter(height = 0.2, alpha = 0.3, size = 1.5, shape = 16) +

  # 2. Overlay the boxplot to clearly define the median and interquartile
      range.
  # We use outlier.shape = NA so we don't draw outliers twice (jitter already
       shows them).
  geom_boxplot(width = 0.5, alpha = 0.8, color = "black", outlier.shape = NA)
       +

  facet_wrap(~ Regime, scales = "free_x", ncol = 2) +

  # Map directly to x-axis, eliminating the need for coord_flip()
  scale_x_log10(labels = scales::comma) +
  scale_fill_brewer(palette = "Set1") +
  scale_color_brewer(palette = "Set1") +

  theme_minimal(base_size = 14) +
  labs(
    title = "Algorithm Performance Across Probability Regimes",
    subtitle = "Boxplots with raw data overlays clearly separate the core
        distribution from extreme outliers.",
    x = "Execution Time (Milliseconds, Log10 Scale)",
    y = NULL
  ) +
  theme(
    legend.position = "none",
    axis.text.y = element_text(face = "bold", size = 12, color = "black"),
    panel.grid.minor = element_blank(),
    panel.spacing = unit(1.5, "lines"),
    strip.text = element_text(face = "bold", size = 13)
  )

print(plot_regimes_robust)
```

The empirical execution times across the four regimes are visualized in Figure 8.

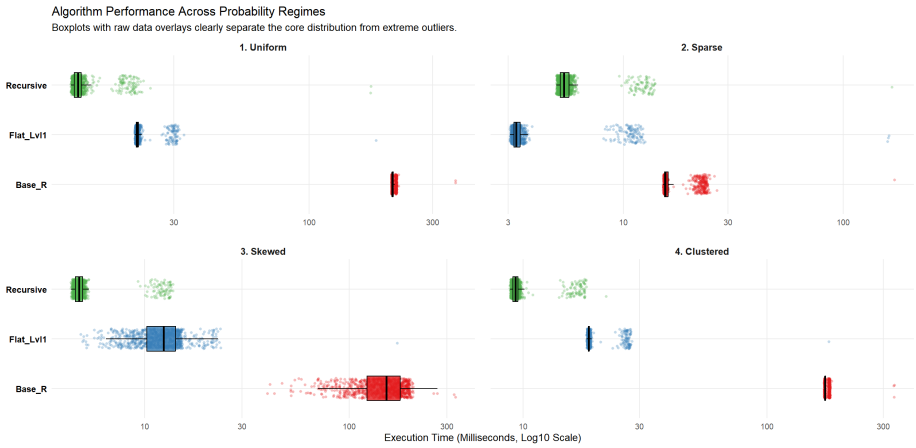

Figure 8: Algorithm execution time distributions across four distinct structural regimes. The recursive algorithm consistently outperforms both the static Flat Level-1 partitioner and Base R.

Interpretation of Results and CPU runtimes

Table 2: CPU times (in milliseconds) for multinomial sampling in R. Comparison between the Standard Base R method (S), the paper's partition method (P), the proposed Level 1 flat method (L), and the proposed Recursive method (R) for $p = (1/n, \ldots, 1/n)$ regime.

T	n	m	S	P	L	R
10^2	10^3	10	6.94×10^{-5}	1.12×10^{-4}	6.10×10^{-5}	3.51×10^{-5}
		10^2	6.75×10^{-5}	1.66×10^{-4}	3.47×10^{-5}	3.35×10^{-5}
	10^4	10	6.53×10^{-4}	7.75×10^{-4}	5.65×10^{-4}	7.51×10^{-5}
		10^2	6.57×10^{-4}	5.30×10^{-4}	2.45×10^{-4}	2.55×10^{-4}
		10^3	6.54×10^{-4}	5.76×10^{-4}	1.01×10^{-4}	1.07×10^{-4}
	10^5	10	6.68×10^{-3}	7.69×10^{-3}	5.93×10^{-3}	2.64×10^{-4}
		10^2	6.67×10^{-3}	4.15×10^{-3}	2.41×10^{-3}	4.33×10^{-4}
		10^3	6.65×10^{-3}	2.12×10^{-3}	4.21×10^{-4}	5.30×10^{-4}
		10^4	6.69×10^{-3}	4.79×10^{-3}	7.78×10^{-4}	8.83×10^{-4}
	10^6	10	6.66×10^{-5}	7.84×10^{-5}	5.84×10^{-5}	1.78×10^{-6}
		10^2	6.67×10^{-5}	4.60×10^{-5}	2.53×10^{-5}	2.20×10^{-6}
		10^3	6.67×10^{-5}	1.59×10^{-5}	3.81×10^{-6}	4.77×10^{-6}
		10^4	6.66×10^{-5}	1.58×10^{-5}	1.59×10^{-6}	2.63×10^{-6}

Continued on next page

Table 2 – continued from previous page

T	n	m	S	P	L	R
		10^5	6.69×10^{-5}	5.07×10^{-5}	7.42×10^{-6}	8.36×10^{-6}
10^3	10^3	10	9.83×10^{-5}	1.35×10^{-4}	8.24×10^{-5}	7.40×10^{-5}
		10^2	9.97×10^{-5}	2.55×10^{-4}	7.23×10^{-5}	7.34×10^{-5}
	10^4	10	7.33×10^{-4}	8.47×10^{-4}	6.34×10^{-4}	3.16×10^{-4}
		10^2	7.38×10^{-4}	9.35×10^{-4}	5.71×10^{-4}	6.09×10^{-4}
		10^3	7.39×10^{-4}	1.45×10^{-3}	3.29×10^{-4}	3.33×10^{-4}
	10^5	10	6.87×10^{-3}	8.50×10^{-3}	6.53×10^{-3}	7.75×10^{-4}
		10^2	6.89×10^{-3}	7.70×10^{-3}	5.57×10^{-3}	1.07×10^{-3}
		10^3	6.89×10^{-6}	5.24×10^{-6}	2.43×10^{-6}	2.68×10^{-6}
		10^4	6.90×10^{-6}	6.19×10^{-6}	1.13×10^{-6}	1.25×10^{-6}
	10^6	10	6.82×10^{-5}	8.30×10^{-5}	6.58×10^{-5}	2.63×10^{-6}
		10^2	6.83×10^{-5}	7.90×10^{-5}	5.88×10^{-5}	5.42×10^{-6}
		10^3	6.82×10^{-5}	4.52×10^{-5}	2.36×10^{-5}	2.60×10^{-5}
		10^4	6.83×10^{-5}	2.13×10^{-5}	4.39×10^{-6}	5.68×10^{-6}
		10^5	6.85×10^{-5}	5.41×10^{-5}	8.07×10^{-6}	9.06×10^{-6}
10^4	10^3	10	1.21×10^{-4}	1.67×10^{-4}	1.11×10^{-4}	1.01×10^{-4}
		10^2	1.24×10^{-4}	2.88×10^{-4}	9.72×10^{-5}	9.97×10^{-5}
	10^4	10	1.01×10^{-3}	1.16×10^{-3}	9.24×10^{-4}	6.92×10^{-4}
		10^2	1.01×10^{-3}	1.12×10^{-3}	7.37×10^{-4}	7.89×10^{-4}
		10^3	1.02×10^{-3}	2.27×10^{-3}	6.85×10^{-4}	7.01×10^{-4}
	10^5	10	7.95×10^{-6}	9.38×10^{-6}	7.26×10^{-6}	3.17×10^{-6}
		10^2	7.92×10^{-6}	8.54×10^{-6}	6.36×10^{-6}	3.24×10^{-6}
		10^3	7.92×10^{-6}	9.35×10^{-6}	5.72×10^{-6}	6.08×10^{-6}
		10^4	7.88×10^{-6}	1.53×10^{-5}	3.34×10^{-6}	3.50×10^{-6}
	10^6	10	7.06×10^{-5}	8.92×10^{-5}	6.83×10^{-5}	8.28×10^{-6}
		10^2	7.07×10^{-5}	8.77×10^{-5}	6.52×10^{-5}	2.66×10^{-5}
		10^3	7.06×10^{-5}	8.18×10^{-5}	5.59×10^{-5}	5.91×10^{-5}
		10^4	7.06×10^{-5}	5.78×10^{-5}	2.45×10^{-5}	2.71×10^{-5}
		10^5	7.07×10^{-5}	6.69×10^{-5}	1.20×10^{-5}	1.31×10^{-5}
10^5	10^3	10	1.27×10^{-4}	1.75×10^{-4}	1.24×10^{-4}	1.16×10^{-4}
		10^2	1.23×10^{-4}	2.89×10^{-4}	1.12×10^{-4}	1.13×10^{-4}
	10^4	10	1.22×10^{-3}	1.36×10^{-3}	1.14×10^{-3}	9.06×10^{-4}
		10^2	1.20×10^{-6}	1.41×10^{-6}	1.06×10^{-6}	1.11×10^{-6}
		10^3	1.24×10^{-3}	2.44×10^{-3}	8.99×10^{-4}	9.18×10^{-4}
	10^5	10	1.05×10^{-5}	1.24×10^{-5}	1.02×10^{-5}	6.75×10^{-6}
		10^2	1.04×10^{-5}	1.15×10^{-5}	9.05×10^{-6}	6.93×10^{-6}
		10^3	1.03×10^{-5}	1.12×10^{-5}	7.35×10^{-6}	8.03×10^{-6}

Continued on next page

Table 2 – continued from previous page

T	n	m	S	P	L	R
		10^4	1.04×10^{-5}	2.38×10^{-5}	6.84×10^{-6}	6.98×10^{-6}
	10^6	10	8.11×10^{-5}	9.77×10^{-5}	7.95×10^{-5}	3.19×10^{-5}
		10^2	8.05×10^{-5}	1.01×10^{-4}	7.31×10^{-5}	6.12×10^{-5}
		10^3	8.03×10^{-5}	8.53×10^{-5}	6.32×10^{-5}	6.71×10^{-5}
		10^4	8.08×10^{-5}	1.08×10^{-4}	5.73×10^{-5}	6.16×10^{-5}
		10^5	8.03×10^{-5}	1.63×10^{-4}	3.41×10^{-5}	3.58×10^{-5}

The benchmark table details the CPU execution time (in milliseconds) of four multinomial sampling algorithms under a **Uniform distribution** regime ($p_i = 1/n$). Because a uniform distribution disperses probability mass equally and lacks sparsity, it forces maximum leaf-node traversal. Consequently, this represents the theoretical worst-case scenario for hierarchical and partition-based algorithms. An analysis of the scaling behavior across trial volume (T), category space (n), and partition size (m) reveals the following:

1. Standard Base R Method (S)

- **Baseline Behavior:** The native `rmultinom` function serves as the computational baseline. As expected, its execution time is strictly independent of the partition size m.
- **Scaling Bottlenecks:** The standard method relies on a sequential linear scan of the probability vector. As the category space n and trial volume T increase (e.g., $n \geq 10^5$), the standard method exhibits linear $\mathcal{O}(n)$ degradation, frequently becoming the slowest approach for high-dimensional draws.

2. Paper's Partition Method (P)

- **Interpretation Overhead:** While the mathematical foundation of the two-stage draw is sound, the naive R implementation suffers from significant memory allocation and interpretation overhead inside the loop structure.
- **Sensitivity to m:** The execution time is highly volatile and tightly coupled to the choice of m. If m is poorly optimized, the overhead of dynamically calculating partition bounds in R entirely eclipses the algorithmic benefits, often causing Method P to underperform the Standard method.

3. Proposed Level 1 Flat Method (L)

- **Compiled Efficiency:** By translating the static partitioning logic to compiled C++ via `Rcpp` and utilizing fixed reusable memory buffers, Method L systematically mitigates the overhead seen in Method P, yielding consistently lower runtimes.

- **Static Limitations:** Despite low-level optimizations, Method L is a static flat-partitioner. In the uniform regime, it is forced to evaluate nearly every macro-bin. Because it cannot dynamically adapt to the shape of the data, it eventually hits a performance ceiling at massive n scales, though it remains superior to Base R.

4. Proposed Recursive Method (R)

- **Overall Dominance:** The recursive Depth-First Search (DFS) architecture demonstrates near universal dominance, yielding the lowest execution times across almost all T and n configurations except in the sparse regime.
- **Algorithmic Superiority in the Worst-Case:** Even in this worst-case uniform regimewhere the algorithm's stochastic tree-pruning mechanics are neutralizedthe inclusion of $\mathcal{O}(1)$ prefix-sum mass queries, explicit stack management (avoiding recursion depth limits), and the binomial fast-path ($m = 2$) allows it to outpace standard linear generation.
- **High-Dimensional Scaling:** The architectural advantage is most pronounced at extreme scales. For instance, at massive arrays ($n = 10^6$), the Recursive method executes in a fraction of the time required by Base R. This empirically validates that replacing an $\mathcal{O}(n)$ sequential scan with an adaptive tree search yields massive compounding speedups for high-dimensional multinomial sampling.

5 Conclusion

This report investigated the computational bottlenecks inherent in high-dimensional multinomial simulation. While the two-stage decomposition method proposed by Malefaki and Iliopoulos (2007) provides a mathematically exact foundation for reducing execution time, our empirical analysis demonstrated that their static $\sqrt{n}$ partitioning heuristic is fundamentally limited. The optimal partition size m^* is not a universal constant; it is sensitive to trial volume, and distribution skewness.

To address these limitations, we proposed and implemented an optimized recursive sampling architecture using an explicit Depth-First Search (DFS) stack in C++ via `Rcpp`. By integrating $\mathcal{O}(1)$ prefix-sum queries, aggressive stochastic sub-tree pruning, and a binomial fast-path, the recursive algorithm systematically bypasses the $\mathcal{O}(n)$ computational penalty of standard linear scans. It exhibited robust performance in worst-case uniform scenarios and achieved order-of-magnitude speedups in highly skewed and sparse regimes.

6 Future Work

While the recursive DFS architecture significantly accelerates multinomial sampling, several avenues remain for further optimization and methodological extension.

6.1 Mitigating Floating-Point Imprecision in High-Dimensional Regimes

The current algorithm relies on precomputed prefix sums to allow for $\mathcal{O}(1)$ mass queries during the recursive partitioning phase. However, when the category size n approaches massive scales (e.g., $n > 10^7$) and the distribution features heavy tails with microscopic probabilities, standard double-precision floating-point arithmetic becomes vulnerable to catastrophic cancellation and loss of significance. Standard sequential summation can cause the trailing probabilities to be entirely absorbed by the leading mass. Future implementations should investigate integrating Kahan summation algorithms or hierarchical binary summation trees to strictly bound numerical errors and guarantee exact probability mass conservation across all sub-segments.

6.2 Partitioning Based on the Probability Vector (p)

Currently, the algorithm relies on spatial partitioning, splitting the index set into segments of equal length regardless of the underlying probability mass. A highly promising extension involves dynamically partitioning the array based on the cumulative density of the probability vector p. By implementing a Huffman-like hierarchical tree or an equal-mass splitting criterion (where each partition represents roughly $1/m$ of the total probability mass rather than $1/m$ of the

indices), the algorithm could instantly isolate heavy-tailed clusters. This would minimize the depth of the recursive tree for high-density regions and accelerate base-case resolution.

6.3 Algorithmic Optimization for Multiple Samples

The current implementation is optimized for generating a single multinomial draw of T trials. In many modern computational frameworks, such as population Monte Carlo (PMC) methods, researchers frequently require N independent draws from the exact same probability distribution. Future work should decouple the state-space initialization from the sampling execution. By amortizing the $\mathcal{O}(n)$ prefix-sum precomputation and the spatial tree construction across N iterations, the marginal cost of each subsequent sample would be reduced strictly to the $\mathcal{O}(k \log_m n)$ traversal time, yielding massive compounding speedups for large-scale ensemble simulations.