Technical Manual

Pragmastat: Pragmatic Statistical Toolkit

22.02.2026

Andrey Akinshin · andrey.akinshin@gmail.com

Version 10.0.5 · DOI: 10.5281/zenodo.17236778

This manual presents a toolkit of statistical procedures that provide reliable results across diverse real-world distributions, with ready-to-use implementations and detailed explanations. The toolkit consists of renamed, recombined, and refined versions of existing methods.

Documentation	pragmastat-v10.0.5.pdf
	web-v10.0.5.zip
Implementations	py-v10.0.5.zip
	ts-v10.0.5.zip
	r-v10.0.5.zip
	cs-v10.0.5.zip
	kt-v10.0.5.zip
	rs-v10.0.5.zip
	go-v10.0.5.zip
Reference data	tests-v10.0.5.zip
	sim-v10.0.5.zip
Source code	pragmastat-10.0.5.zip

Andrey Akinshin
andrey.akinshin@gmail.com
DOI: 10.5281/zenodo.17236778

Source code and implementations are available at github.com/AndreyAkinshin/pragmastat.

Typeset with Typst. Text refined with LLM assistance.

Contents

Synopsis 4

1. One-Sample Estimators **6**
1.1. Center 6
1.2. CenterBounds 20
1.3. Spread 33
1.4. SpreadBounds 47

2. Two-Sample Estimators **56**
2.1. Shift 56
2.2. ShiftBounds 65
2.3. Ratio 71
2.4. RatioBounds 76
2.5. Disparity 81
2.6. DisparityBounds 85

3. Randomization **91**
3.1. Rng 91
3.2. UniformFloat 102
3.3. UniformInt 106
3.4. Sample 107
3.5. Resample 110
3.6. Shuffle 113

4. Distributions **116**
4.1. Additive ('Normal') 116
4.2. Multiplic 122
4.3. Exp 122
4.4. Power 123
4.5. Uniform 124

5. Implementations **126**
5.1. Python 126
5.2. TypeScript 128
5.3. R 129
5.4. C# 131
5.5. Kotlin 132
5.6. Rust 134
5.7. Go 136

6. Auxiliary **139**
6.1. AvgSpread 139
6.2. AvgSpreadBounds 143
6.3. Median 147

6.4. SignMargin 150
6.5. PairwiseMargin 155
6.6. SignedRankMargin 170

7. Appendix **178**
7.1. Assumptions 178
7.2. Foundations 185
7.3. Methodology 189
Bibliography 204

Synopsis

One-Sample Estimators

$\mathrm{Center}(\mathbf{x})$ — robust average
$\mathrm{CenterBounds}(\mathbf{x}, \mathrm{misrate})$ — confidence interval for center
$\mathrm{Spread}(\mathbf{x})$ — robust spread
$\mathrm{SpreadBounds}(\mathbf{x}, \mathrm{misrate})$ — confidence interval for spread

Two-Sample Estimators

$\mathrm{Shift}(\mathbf{x}, \mathbf{y})$ — robust difference
$\mathrm{ShiftBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate})$ — confidence interval for shift
$\mathrm{Ratio}(\mathbf{x}, \mathbf{y})$ — robust ratio
$\mathrm{RatioBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate})$ — confidence interval for ratio
$\mathrm{Disparity}(\mathbf{x}, \mathbf{y})$ — robust effect size $= \mathrm{Shift} \,/\, \mathrm{AvgSpread}$
$\mathrm{DisparityBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate})$ — confidence interval for disparity

Randomization

$r \leftarrow \mathrm{Rng}(s)$ — random number generator with seed s
$r.\,\mathrm{UniformFloat}()$ — uniform random value in $[0, 1)$
$r.\,\mathrm{UniformInt}(a, b)$ — uniform random integer in $[a, b)$
$r.\,\mathrm{Sample}(\mathbf{x}, k)$ — select k elements without replacement
$r.\,\mathrm{Resample}(\mathbf{x}, k)$ — select k elements with replacement
$r.\,\mathrm{Shuffle}(\mathbf{x})$ — uniformly random permutation

The table below maps each toolkit function to the underlying algorithm and its complexity.

One-Sample Estimators

Function	Algorithm	Complexity
Center	Monahan's implicit-matrix selection	$O(n \log n)$
CenterBounds	Binary search over pairwise averages + SignedRankMargin	$O(n \log n)$
Spread	Monahan's selection adapted for differences	$O(n \log n)$
SpreadBounds	Disjoint-pair sign-test inversion	$O(n \log n)$

Two-Sample Estimators

Function	Algorithm	Complexity
Shift	Value-space binary search over pairwise differences	$O((n+m) \log L)$
ShiftBounds	PairwiseMargin + Shift quantile selection	$O((n+m) \log L)$
Ratio	Log-exp transform + Shift	$O((n+m) \log L)$
RatioBounds	Log-exp transform + ShiftBounds	$O((n+m) \log L)$
Disparity	Composition: $\frac{\text{Shift}}{\text{AvgSpread}}$	$O((n+m) \log L + n \log n + m \log m)$
DisparityBounds	Bonferroni split: ShiftBounds + AvgSpreadBounds	$O((n+m) \log L + n \log n + m \log m)$

Randomization

Function	Algorithm	Complexity
UniformFloat	53-bit extraction from xoshiro256++ output	$O(1)$ per draw
UniformInt	Modulo reduction of raw 64-bit output	$O(1)$ per draw
Sample	Fan-Muller-Rezucha selection sampling	$O(n)$
Resample	Uniform integer sampling with replacement	$O(k)$
Shuffle	Fisher-Yates (Knuth shuffle)	$O(n)$

Auxiliary

Function	Algorithm	Complexity
AvgSpread	Weighted average of two Spread calls	$O(n \log n + m \log m)$
AvgSpreadBounds	Bonferroni combination of two SpreadBounds	$O(n \log n + m \log m)$
Median	Sort + pick middle	$O(n \log n)$
SignMargin	Binomial CDF inversion + randomized cutoff	$O(n)$
PairwiseMargin	Löffler recurrence (exact) / Edgeworth (approx)	$O(nm)$ / $O(\log(nm))$
SignedRankMargin	Dynamic programming (exact) / Edgeworth (approx)	$O(n^3)$ / $O(\log n)$

1. One-Sample Estimators

1.1. Center

$$\text{Center}(\mathbf{x}) = \text{Median}_{1 \le i \le j \le n} \frac{x_i + x_j}{2}$$

Robust measure of location (central tendency).

Also known as — Hodges-Lehmann estimator, pseudomedian
Asymptotic — median of the average of two random measurements from X
Complexity — $O(n \log n)$
Domain — any real numbers
Unit — same as measurements

Properties

Shift equivariance	$\text{Center}(\mathbf{x} + k) = \text{Center}(\mathbf{x}) + k$
Scale equivariance	$\text{Center}(k \cdot \mathbf{x}) = k \cdot \text{Center}(\mathbf{x})$

Example

```
Center([0, 2, 4, 6, 8]) = 4
Center(x + 10) = 14   Center(3x) = 12
```

Center is the recommended default for representing "where the data is." It works like the familiar mean but does not break when the data contains a few bad measurements or outliers. Up to 29% of data can be corrupted before Center becomes unreliable. When data is clean, Center is nearly as precise as the mean (95% efficiency), so the added protection comes at almost no cost. When uncertain whether to use mean, median, or something else, start with Center.

1.1.1. Algorithm

The Center estimator computes the median of all pairwise averages from a sample. Given a dataset $x = (x_1, x_2, ..., x_n)$, this estimator is defined as:

$$\text{Center}(\mathbf{x}) = \text{median}_{1 \le i \le j \le n} \frac{x_i + x_j}{2}$$

A direct implementation would generate all $\frac{n(n+1)}{2}$ pairwise averages and sort them. With $n = 10000$, this approach creates approximately 50 million values, requiring quadratic memory and $O(n^2 \log n)$ time.

The breakthrough came in 1984 when John Monahan developed an algorithm that reduces expected complexity to $O(n \log n)$ while using only linear memory (see Monahan (1984)). The algorithm exploits the inherent structure in pairwise sums rather than computing them explicitly. After sorting the values $x_1 \le x_2 \le ... \le x_n$, the algorithm considers the implicit upper triangular matrix M where $M_{i,j} = x_i + x_j$ for $i \le j$. This matrix has a crucial property: each row and column is sorted in non-decreasing order, enabling efficient median selection without storing the matrix.

Rather than sorting all pairwise sums, the algorithm uses a selection approach similar to quickselect. It maintains search bounds for each matrix row and iteratively narrows the search space. For each row i, the algorithm tracks active column indices from $i + 1$ to n, defining which pairwise sums remain candidates for the median. It selects a candidate sum as a pivot using randomized selection from active matrix elements, then counts how many pairwise sums fall below the pivot. Because both rows and columns are sorted, this counting takes only $O(n)$ time using a two-pointer sweep from the matrix's upper-right corner.

The median corresponds to rank $k = \lfloor \frac{N+1}{2} \rfloor$ where $N = \frac{n(n+1)}{2}$. If fewer than k sums lie below the pivot, the median must be larger; if more than k sums lie below the pivot, the median must be smaller. Based on this comparison, the algorithm eliminates portions of each row that cannot contain the median, shrinking the active search space while preserving the true median.

Real data often contain repeated values, which can cause the selection process to stall. When the algorithm detects no progress between iterations, it switches to a midrange strategy: find the smallest and largest pairwise sums still in the search space, then use their average as the next pivot. If the minimum equals the maximum, all remaining candidates are identical and the algorithm terminates. This tie-breaking mechanism ensures reliable convergence with discrete or duplicated data.

The algorithm achieves $O(n \log n)$ time complexity through linear partitioning (each pivot evaluation requires only $O(n)$ operations) and logarithmic iterations (randomized pivot selection leads to expected $O(\log n)$ iterations, similar to quickselect). The algorithm maintains only row bounds and counters, using $O(n)$ additional space. This matches the complexity of sorting a single array while avoiding the quadratic memory and time explosion of computing all pairwise combinations.

```
namespace Pragmastat.Algorithms;

internal static class FastCenter
{
  /// <summary>
```

```
  /// ACM Algorithm 616: fast computation of the Hodges-Lehmann location estimator
  /// </summary>
  /// <remarks>
  /// Computes the median of all pairwise averages (xi + xj)/2 efficiently.
  /// See: John F Monahan, "Algorithm 616: fast computation of the Hodges-Lehmann location
estimator"
  /// (1984) DOI: 10.1145/1271.319414
  /// </remarks>
  /// <param name="values">A sorted sample of values</param>
  /// <param name="random">Random number generator</param>
  /// <param name="isSorted">If values are sorted</param>
  /// <returns>Exact center value (Hodges-Lehmann estimator)</returns>
  public static double Estimate(IReadOnlyList<double> values, Random? random = null, bool
isSorted = false)
  {
    int n = values.Count;
    if (n == 1) return values[0];
    if (n == 2) return (values[0] + values[1]) / 2;
    random ??= new Random();
    if (!isSorted)
      values = values.OrderBy(x => x).ToList();

    // Calculate target median rank(s) among all pairwise sums
    long totalPairs = (long)n * (n + 1) / 2;
    long medianRankLow = (totalPairs + 1) / 2; // For odd totalPairs, this is the median
    long medianRankHigh =
      (totalPairs + 2) / 2; // For even totalPairs, average of ranks medianRankLow and
medianRankHigh

    // Initialize search bounds for each row in the implicit matrix
    long[] leftBounds = new long[n];
    long[] rightBounds = new long[n];
    long[] partitionCounts = new long[n];

    for (int i = 0; i < n; i++)
    {
      leftBounds[i] = i + 1; // Row i can pair with columns [i+1..n] (1-based indexing)
      rightBounds[i] = n; // Initially, all columns are available
    }

    // Start with a good pivot: sum of middle elements (handles both odd and even n)
    double pivot = values[(n - 1) / 2] + values[n / 2];
    long activeSetSize = totalPairs;
    long previousCount = 0;

    while (true)
    {
      // === PARTITION STEP ===
      // Count pairwise sums less than current pivot
      long countBelowPivot = 0;
      long currentColumn = n;

      for (int row = 1; row <= n; row++)
```

```
      {
        partitionCounts[row - 1] = 0;

        // Move left from current column until we find sums < pivot
        // This exploits the sorted nature of the matrix
        while (currentColumn >= row && values[row - 1] + values[(int)currentColumn - 1] >=
pivot)
          currentColumn--;

        // Count elements in this row that are < pivot
        if (currentColumn >= row)
        {
          long elementsBelow = currentColumn - row + 1;
          partitionCounts[row - 1] = elementsBelow;
          countBelowPivot += elementsBelow;
        }
      }

      // === CONVERGENCE CHECK ===
      // If no progress, we have ties - break them using midrange strategy
      if (countBelowPivot == previousCount)
      {
        double minActiveSum = double.MaxValue;
        double maxActiveSum = double.MinValue;

        // Find the range of sums still in the active search space
        for (int i = 0; i < n; i++)
        {
          if (leftBounds[i] > rightBounds[i]) continue; // Skip empty rows

          double rowValue = values[i];
          double smallestInRow = values[(int)leftBounds[i] - 1] + rowValue;
          double largestInRow = values[(int)rightBounds[i] - 1] + rowValue;

          minActiveSum = Min(minActiveSum, smallestInRow);
          maxActiveSum = Max(maxActiveSum, largestInRow);
        }

        pivot = (minActiveSum + maxActiveSum) / 2;
        if (pivot <= minActiveSum || pivot > maxActiveSum) pivot = maxActiveSum;

        // If all remaining values are identical, we're done
        if (minActiveSum == maxActiveSum || activeSetSize <= 2)
          return pivot / 2;

        continue;
      }

      // === TARGET CHECK ===
      // Check if we've found the median rank(s)
      bool atTargetRank = countBelowPivot == medianRankLow || countBelowPivot ==
medianRankHigh - 1;
      if (atTargetRank)
```

```
{
  // Find the boundary values: largest < pivot and smallest >= pivot
  double largestBelowPivot = double.MinValue;
  double smallestAtOrAbovePivot = double.MaxValue;

  for (int i = 1; i <= n; i++)
  {
    long countInRow = partitionCounts[i - 1];
    double rowValue = values[i - 1];
    long totalInRow = n - i + 1;

    // Find largest sum in this row that's < pivot
    if (countInRow > 0)
    {
      long lastBelowIndex = i + countInRow - 1;
      double lastBelowValue = rowValue + values[(int)lastBelowIndex - 1];
      largestBelowPivot = Max(largestBelowPivot, lastBelowValue);
    }

    // Find smallest sum in this row that's >= pivot
    if (countInRow < totalInRow)
    {
      long firstAtOrAboveIndex = i + countInRow;
      double firstAtOrAboveValue = rowValue + values[(int)firstAtOrAboveIndex - 1];
      smallestAtOrAbovePivot = Min(smallestAtOrAbovePivot, firstAtOrAboveValue);
    }
  }

  // Calculate final result based on whether we have odd or even number of pairs
  if (medianRankLow < medianRankHigh)
  {
    // Even total: average the two middle values
    return (smallestAtOrAbovePivot + largestBelowPivot) / 4;
  }
  else
  {
    // Odd total: return the single middle value
    bool needLargest = countBelowPivot == medianRankLow;
    return (needLargest ? largestBelowPivot : smallestAtOrAbovePivot) / 2;
  }
}

// === UPDATE BOUNDS ===
// Narrow the search space based on partition result
if (countBelowPivot < medianRankLow)
{
  // Too few values below pivot - eliminate smaller values, search higher
  for (int i = 0; i < n; i++)
    leftBounds[i] = i + partitionCounts[i] + 1;
}
else
{
  // Too many values below pivot - eliminate larger values, search lower
```

```
    for (int i = 0; i < n; i++)
      rightBounds[i] = i + partitionCounts[i];
  }

  // === PREPARE NEXT ITERATION ===
  previousCount = countBelowPivot;

  // Recalculate how many elements remain in the active search space
  activeSetSize = 0;
  for (int i = 0; i < n; i++)
  {
    long rowSize = rightBounds[i] - leftBounds[i] + 1;
    activeSetSize += Max(0, rowSize);
  }

  // Choose next pivot based on remaining active set size
  if (activeSetSize > 2)
  {
    // Use randomized row median strategy for efficiency
    // Handle large activeSetSize by using double precision for random selection
    double randomFraction = random.NextDouble();
    long targetIndex = (long)(randomFraction * activeSetSize);
    int selectedRow = 0;

    // Find which row contains the target index
    long cumulativeSize = 0;
    for (int i = 0; i < n; i++)
    {
      long rowSize = Max(0, rightBounds[i] - leftBounds[i] + 1);
      if (targetIndex < cumulativeSize + rowSize)
      {
        selectedRow = i;
        break;
      }

      cumulativeSize += rowSize;
    }

    // Use median element of the selected row as pivot
    long medianColumnInRow = (leftBounds[selectedRow] + rightBounds[selectedRow]) / 2;
    pivot = values[selectedRow] + values[(int)medianColumnInRow - 1];
  }
  else
  {
    // Few elements remain - use midrange strategy
    double minRemainingSum = double.MaxValue;
    double maxRemainingSum = double.MinValue;

    for (int i = 0; i < n; i++)
    {
      if (leftBounds[i] > rightBounds[i]) continue; // Skip empty rows

      double rowValue = values[i];
```

```
          double minInRow = values[(int)leftBounds[i] - 1] + rowValue;
          double maxInRow = values[(int)rightBounds[i] - 1] + rowValue;

          minRemainingSum = Min(minRemainingSum, minInRow);
          maxRemainingSum = Max(maxRemainingSum, maxInRow);
        }

        pivot = (minRemainingSum + maxRemainingSum) / 2;
        if (pivot <= minRemainingSum || pivot > maxRemainingSum)
          pivot = maxRemainingSum;

        if (minRemainingSum == maxRemainingSum)
          return pivot / 2;
      }
    }
  }
}
```

1.1.2. Notes

This section compares the toolkit's robust average estimator against traditional methods to demonstrate its advantages across diverse conditions.

Average Estimators

Mean (arithmetic average):

$$\text{Mean}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Median:

$$\text{Median}(\mathbf{x}) = \begin{cases} x_{((n+1)/2)} & \text{if } n \text{ is odd} \\ \frac{x_{(n/2)}+x_{(n/2+1)}}{2} & \text{if } n \text{ is even} \end{cases}$$

Center (Hodges-Lehmann estimator):

$$\text{Center}(\mathbf{x}) = \text{Median}_{1\le i\le j\le n}\left(\frac{x_i + x_j}{2}\right)$$

1.1.2.1. Breakdown

Heavy-tailed distributions naturally produce extreme outliers that completely distort traditional estimators. A single extreme measurement from the Power distribution can make the sample mean arbitrarily large. Real-world data can also contain corrupted measurements from instrument failures, recording errors, or transmission problems. Both natural extremes and data corruption create the same challenge: extracting reliable information when some measurements are too influential.

The breakdown point (Huber (2009)) is the fraction of a sample that can be replaced by arbitrarily large values without making an estimator arbitrarily large. The theoretical maximum is 50%; no estimator can guarantee reliable results when more than half the measurements are extreme or corrupted. In such cases, summary estimators are not applicable, and a more sophisticated approach is needed.

A 50% breakdown point is rarely needed in practice, as more conservative values also cover practical needs. Additionally, a high breakdown point corresponds to low precision (information is lost by neglecting part of the data). The optimal practical breakdown point should be between 0% (no robustness) and 50% (low precision).

The Center estimator achieves a 29% breakdown point, providing substantial protection against realistic contamination levels while maintaining good precision.

Asymptotic breakdown points for average estimators:

Mean	Median	Center
0%	50%	29%

1.1.2.2. Drift

Drift measures estimator precision by quantifying how much estimates scatter across repeated samples. It is based on the Spread of estimates and therefore has a breakdown point of approximately 29%.

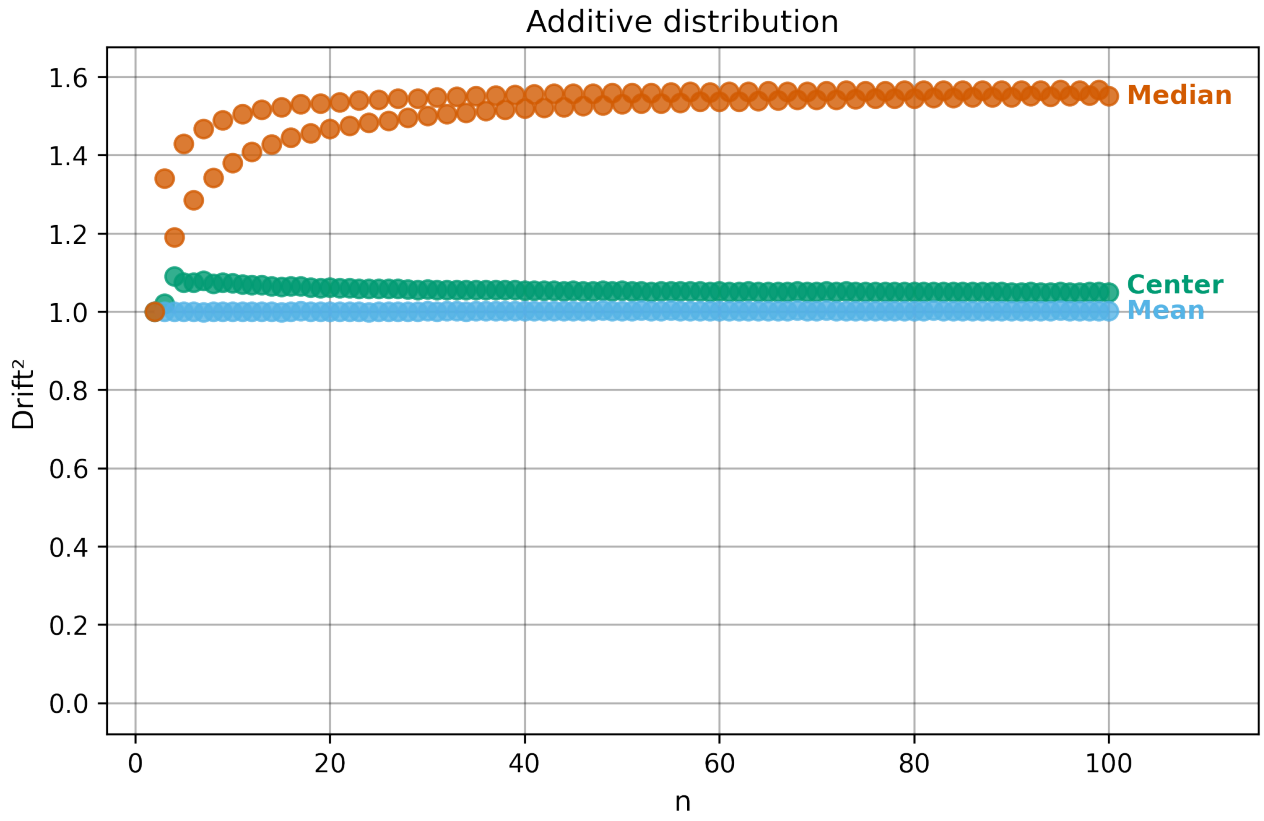
Drift is useful for comparing the precision of several estimators. To simplify the comparison, one of the estimators can be chosen as a baseline. A table of squared drift values, normalized by the baseline, shows the required sample size adjustment factor for switching from the baseline to another estimator. For example, if Center is the baseline and the rescaled drift square of Median is 1.5, this means that Median requires 1.5 times more data than Center to achieve the same precision. See From Statistical Efficiency to Drift for details.

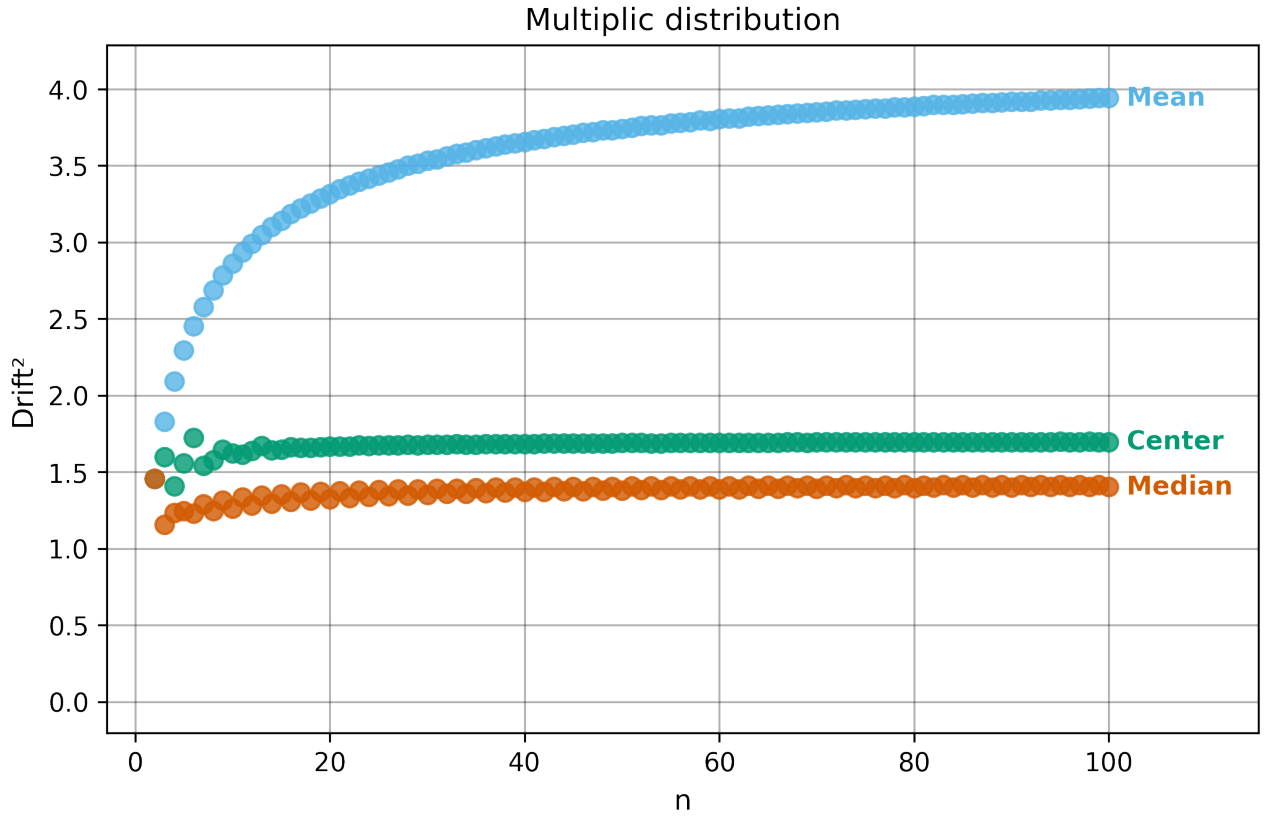
Squared Asymptotic Drift of Average Estimators (values are approximated):

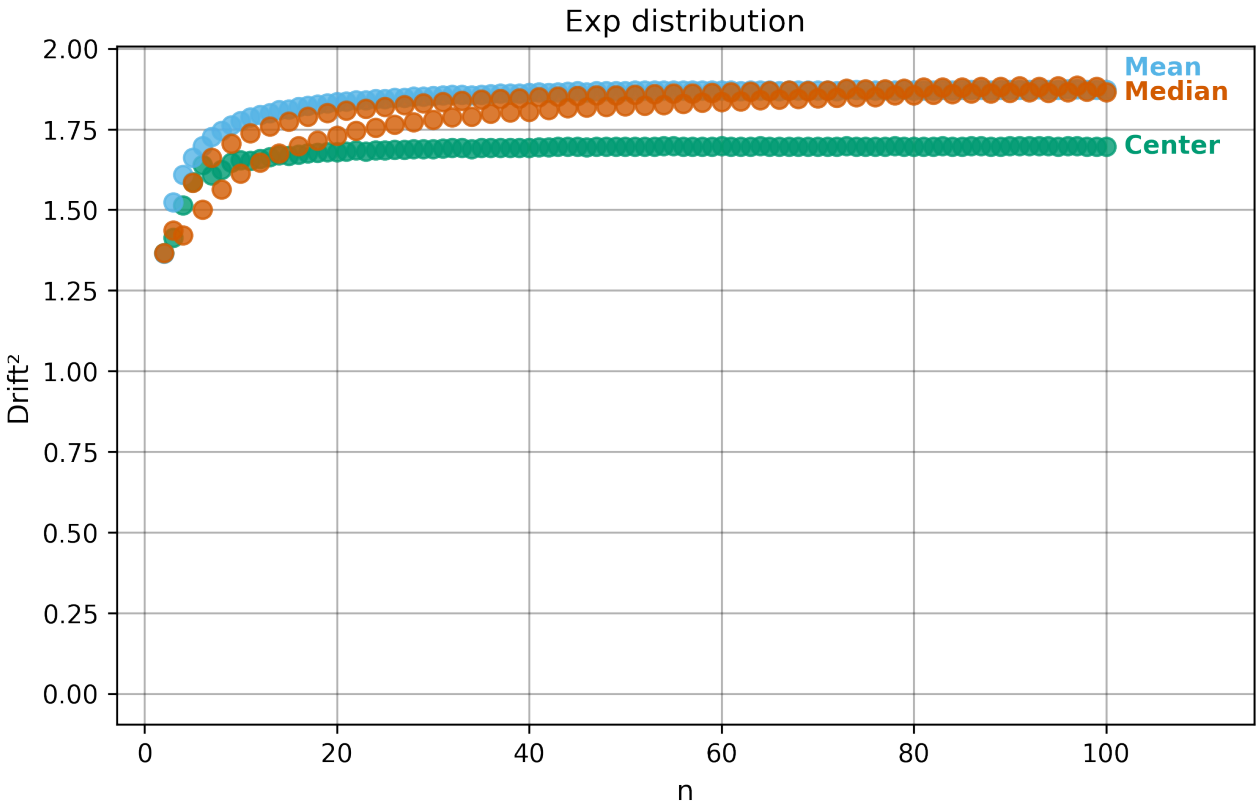
	Mean	Median	Center
Additive	1.0	1.571	1.047
Multiplic	3.95	1.40	1.7
Exp	1.88	1.88	1.69
Power	∞	0.9	2.1
Uniform	0.88	2.60	0.94

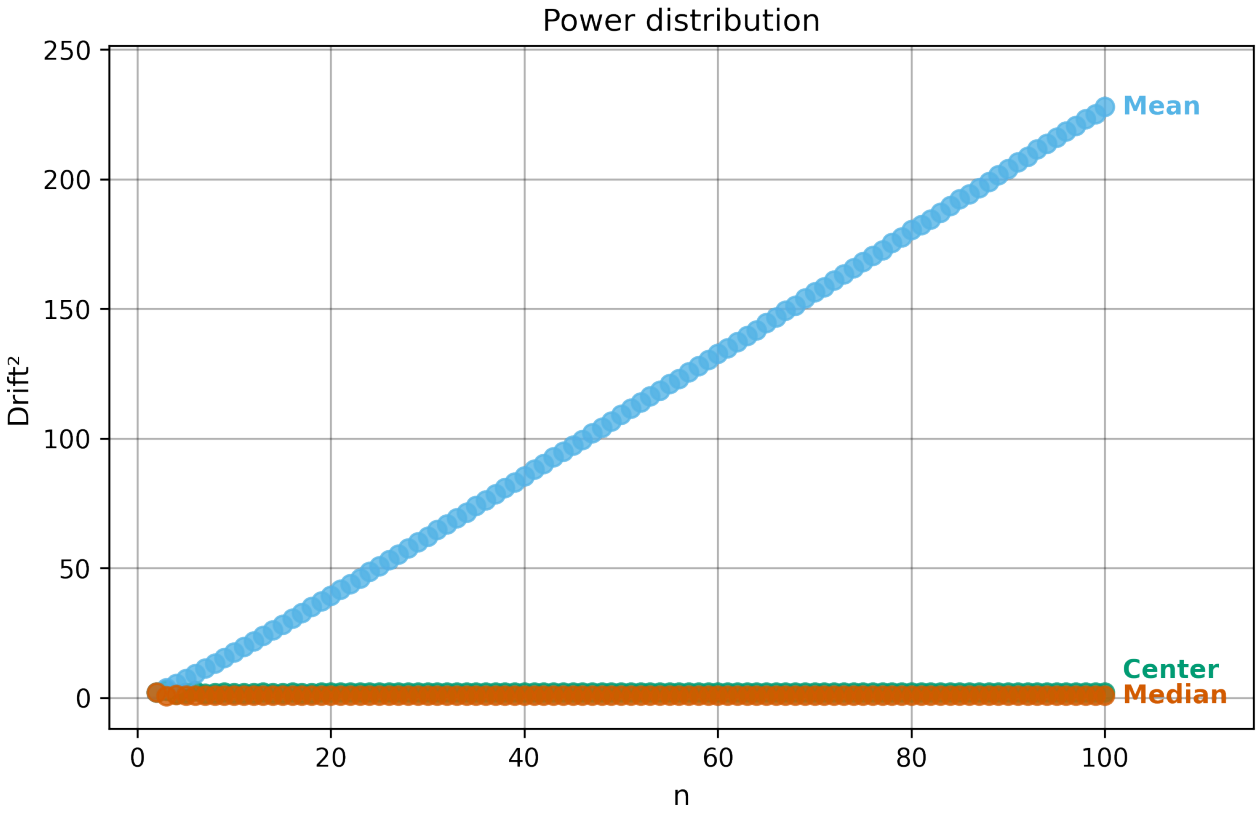
Rescaled to Center (sample size adjustment factors):

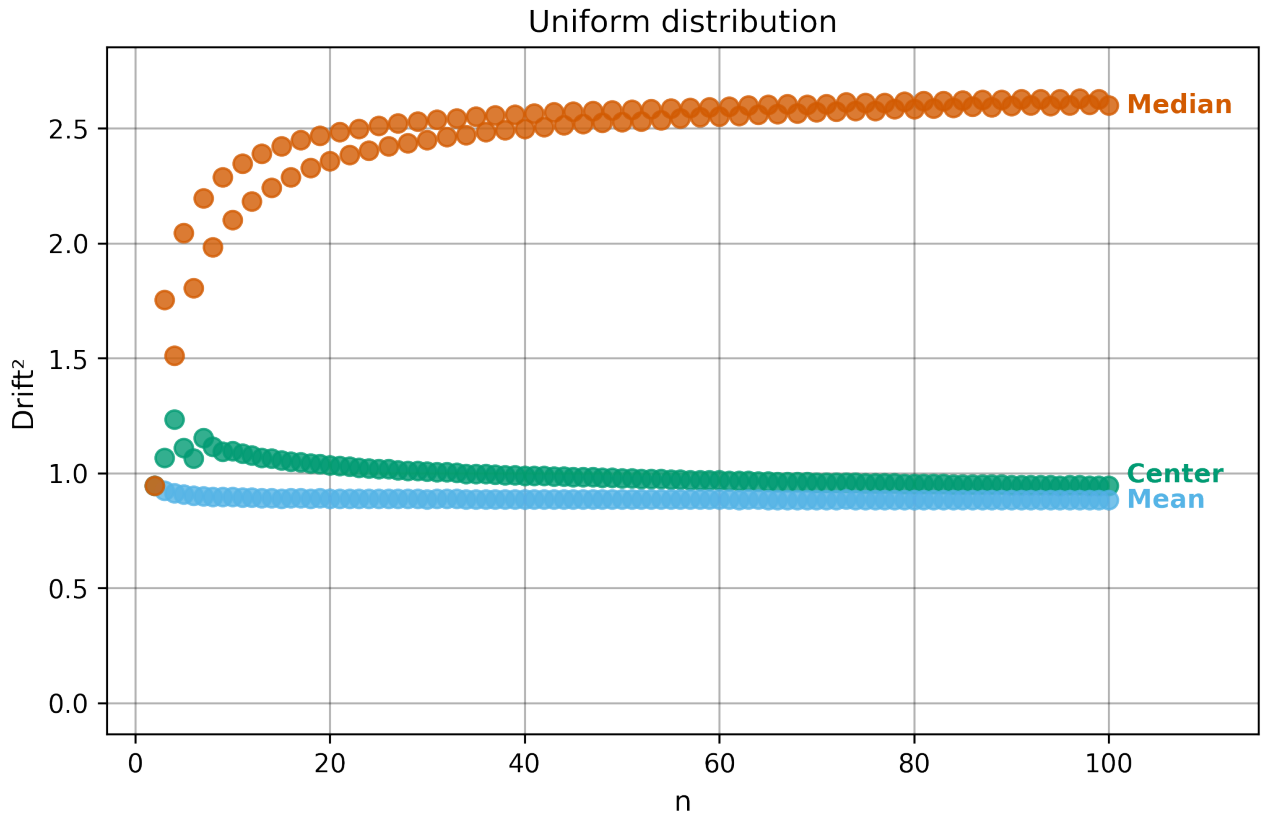
	Mean	Median	Center
Additive	0.96	1.50	1.0
Multiplic	2.32	0.82	1.0
Exp	1.11	1.11	1.0
Power	∞	0.43	1.0
Uniform	0.936	2.77	1.0


Additive distribution
Median
Center
Mean
Drift²
n
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
20
40
60
80
100


Multiplic distribution
Mean
Center
Median
Drift²
n
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20
40
60
80
100


Exp distribution
Mean
Median
Center
Drift²
n
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20
40
60
80
100


Power distribution
Mean
Center
Median
Drift²
n
250
200
150
100
50
0
0
20
40
60
80
100


Uniform distribution
Median
Center
Mean
Drift²
n
0.0
0.5
1.0
1.5
2.0
2.5
0
20
40
60
80
100

1.1.3. Tests

$$\mathrm{Center}(\mathbf{x}) = \mathrm{median}_{1 \le i \le j \le n} \frac{x_i + x_j}{2}$$

The Center test suite contains 38 correctness test cases stored in the repository (24 original + 14 unsorted), plus 1 performance test that should be implemented manually (see Test Framework).

Demo examples ($n = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (0, 2, 4, 6, 8)$, expected output: 4 (base case)
`demo-2`: $\mathbf{x} = (10, 12, 14, 16, 18)$ (= demo-1 + 10), expected output: 14 (location equivariance)
`demo-3`: $\mathbf{x} = (0, 6, 12, 18, 24)$ (= 3 × demo-1), expected output: 12 (scale equivariance)

Natural sequences ($n = 1, 2, 3, 4$) — canonical happy path examples:

`natural-1`: $\mathbf{x} = (1)$, expected output: 1
`natural-2`: $\mathbf{x} = (1, 2)$, expected output: 1.5
`natural-3`: $\mathbf{x} = (1, 2, 3)$, expected output: 2
`natural-4`: $\mathbf{x} = (1, 2, 3, 4)$, expected output: 2.5 (smallest even size with rich structure)

Negative values ($n = 3$) — sign handling validation:

`negative-3`: $\mathbf{x} = (-3, -2, -1)$, expected output: -2

Zero values ($n = 1, 2$) — edge case testing with zeros:

`zeros-1`: $\mathbf{x} = (0)$, expected output: 0
`zeros-2`: $\mathbf{x} = (0, 0)$, expected output: 0

Additive distribution ($n = 5, 10, 30$) — fuzzy testing with $\underline{\mathrm{Additive}}(10, 1)$:

`additive-5`, `additive-10`, `additive-30`: random samples generated with seed 0

Uniform distribution ($n = 5, 100$) — fuzzy testing with $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-5`, `uniform-100`: random samples generated with seed 1

The random samples validate that Center performs correctly on realistic distributions at various sample sizes. The progression from small ($n = 5$) to large ($n = 100$) samples helps identify issues that only manifest at specific scales.

Algorithm stress tests — edge cases for fast algorithm implementation:

`duplicates-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$ (all identical, stress stall handling)
`duplicates-10`: $\mathbf{x} = (1, 1, 1, 2, 2, 2, 3, 3, 3, 3)$ (many duplicates, stress tie-breaking)
`parity-odd-7`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$ (odd sample size for odd total pairs)
`parity-even-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6)$ (even sample size for even total pairs)
`parity-odd-49`: 49-element sequence $(1, 2, \ldots, 49)$ (large odd, 1225 pairs)
`parity-even-50`: 50-element sequence $(1, 2, \ldots, 50)$ (large even, 1275 pairs)

Extreme values — numerical stability and range tests:

`extreme-large-5`: $\mathbf{x} = (10^8, 2 \cdot 10^8, 3 \cdot 10^8, 4 \cdot 10^8, 5 \cdot 10^8)$ (very large values)
`extreme-small-5`: $\mathbf{x} = (10^{-8}, 2 \cdot 10^{-8}, 3 \cdot 10^{-8}, 4 \cdot 10^{-8}, 5 \cdot 10^{-8})$ (very small positive values)
`extreme-wide-5`: $\mathbf{x} = (0.001, 1, 100, 1000, 1000000)$ (wide range, tests precision)

Unsorted tests — verify sorting correctness (14 tests):

`unsorted-reverse-{n}` for $n \in \{2, 3, 4, 5, 7\}$: reverse sorted natural sequences (5 tests)
`unsorted-shuffle-3`: $\mathbf{x} = (2, 1, 3)$ (middle element first)
`unsorted-shuffle-4`: $\mathbf{x} = (3, 1, 4, 2)$ (interleaved)
`unsorted-shuffle-5`: $\mathbf{x} = (5, 2, 4, 1, 3)$ (complex shuffle)
`unsorted-last-first-5`: $\mathbf{x} = (5, 1, 2, 3, 4)$ (last moved to first)
`unsorted-first-last-5`: $\mathbf{x} = (2, 3, 4, 5, 1)$ (first moved to last)
`unsorted-duplicates-mixed-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$ (all identical, any order)
`unsorted-duplicates-unsorted-10`: $\mathbf{x} = (3, 1, 2, 3, 1, 3, 2, 1, 3, 2)$ (duplicates mixed)
`unsorted-extreme-large-unsorted-5`: $\mathbf{x} = (5 \cdot 10^8, 10^8, 4 \cdot 10^8, 2 \cdot 10^8, 3 \cdot 10^8)$ (large values unsorted)
`unsorted-parity-odd-reverse-7`: $\mathbf{x} = (7, 6, 5, 4, 3, 2, 1)$ (odd size reverse)

These tests ensure implementations correctly sort input data before computing pairwise averages. The variety of shuffle patterns (reverse, rotation, interleaving, single element displacement) catches common sorting bugs.

Performance test — validates the fast $O(n \log n)$ algorithm:

Input: $\mathbf{x} = (1, 2, 3, ..., 100000)$
Expected output: 50000.5
Time constraint: Must complete in under 5 seconds
Purpose: Ensures that the implementation uses the efficient algorithm rather than materializing all $\binom{n+1}{2} \approx 5$ billion pairwise averages

This test case is not stored in the repository because it generates a large JSON file (approximately 1.5 MB). Each language implementation should manually implement this test with the hardcoded expected result.

1.1.4. References

Hodges, J. L., & Lehmann, E. L. (1963). Estimates of Location Based on Rank Tests. *The Annals of Mathematical Statistics*, *34*(2), 598–611. http://projecteuclid.org/euclid.aoms/1177704172

Sen, P. K. (1963). On the Estimation of Relative Potency in Dilution (-Direct) Assays by Distribution-Free Methods. *Biometrics*, *19*(4), 532. https://www.jstor.org/stable/2527532

Monahan, J. F. (1984). Algorithm 616: fast computation of the Hodges-Lehmann location estimator. *ACM Transactions on Mathematical Software*, *10*(3), 265–270. https://dl.acm.org/doi/10.1145/1271.319414

Huber, P. J. (2009). Robust statistics. In *International encyclopedia of statistical science* (pp. 1248–1251). Springer.

1.2. CenterBounds

$$\text{CenterBounds}(\mathbf{x}, \text{misrate}) = \left[w_{(k_{\text{left}})}, w_{(k_{\text{right}})}\right]$$

where $\mathbf{w} = \{(x_i + x_j)/2\}$ (pairwise averages, sorted) for $i \le j$, $k_{\text{left}} = \lfloor \text{SignedRankMargin}/2 \rfloor + 1$, $k_{\text{right}} = N - \lfloor \text{SignedRankMargin}/2 \rfloor$, and $N = n(n+1)/2$

Robust bounds on $\text{Center}(\mathbf{x})$ with specified coverage.

Also known as — Wilcoxon signed-rank confidence interval for Hodges-Lehmann pseudomedian
Interpretation — misrate is probability that true center falls outside bounds
Domain — any real numbers, $n \ge 2$, $\text{misrate} \ge 2^{1-n}$
Unit — same as measurements
Note — assumes weak symmetry and weak continuity; exact for $n \le 63$, Edgeworth approximation for $n > 63$

Properties

Shift equivariance	$\text{CenterBounds}(\mathbf{x} + k, \text{misrate}) = \text{CenterBounds}(\mathbf{x}, \text{misrate}) + k$
Scale equivariance	$\text{CenterBounds}(k \cdot \mathbf{x}, \text{misrate}) = k \cdot \text{CenterBounds}(\mathbf{x}, \text{misrate})$

Example

`CenterBounds([1..10], 0.01) = [2.5, 8.5]` where `Center = 5.5`
Bounds fail to cover true center with probability $\approx$ misrate

CenterBounds provides not just the estimated center but also the uncertainty of that estimate. The function returns an interval of plausible center values given the data. Set misrate to control how often the bounds might fail to contain the true center: use 10^{-3} for everyday analysis or 10^{-6} for critical decisions where errors are costly. These bounds require weak symmetry but no specific distributional form. If the bounds exclude some reference value, that suggests the true center differs reliably from that value.

1.2.1. Algorithm

$\mathrm{CenterBounds}$ uses two components: SignedRankMargin to determine which order statistics to select, and a fast quantile algorithm to compute them.

The Center estimator computes the median of all pairwise averages $\frac{x_i+x_j}{2}$ for $i \le j$. For $\mathrm{CenterBounds}$, we need not just the median but specific order statistics: the k-th smallest pairwise average for bounds computation. Given a sample of size n, there are $N = \frac{n(n+1)}{2}$ such pairwise averages.

A naive approach would materialize all N pairwise averages, sort them, and extract the desired quantile. With $n = 10000$, this creates approximately 50 million values, requiring quadratic memory and $O(N \log N)$ time. The fast algorithm avoids materializing the pairs entirely.

The algorithm exploits the sorted structure of the implicit pairwise average matrix. After sorting the input to obtain $x_1 \le x_2 \le \ldots \le x_n$, the pairwise averages form a symmetric matrix where both rows and columns are sorted. Instead of searching through indices, the algorithm searches through values. It maintains a search interval $[\mathrm{lo}, \mathrm{hi}]$, initialized to $[x_1, x_n]$ (the range of all possible pairwise averages). At each iteration, it asks: "How many pairwise averages are at most this threshold?" Based on the count, it narrows the search interval. For finding the k-th smallest pairwise average: if $\mathrm{count} \ge k$, the target is at or below the threshold and the search continues in the lower half; if $\mathrm{count} < k$, the target is above the threshold and the search continues in the upper half.

The key operation is counting pairwise averages at or below a threshold t. For each i, we need to count how many $j \ge i$ satisfy $\frac{x_i+x_j}{2} \le t$, equivalently $x_j \le 2t - x_i$. Because the array is sorted, a single pass through the array suffices. For each i, a pointer j tracks the largest index where $x_j \le 2t - x_i$. As i increases, x_i increases, so the threshold $2t - x_i$ decreases, meaning j can only decrease (or stay the same). This two-pointer technique ensures each element is visited at most twice, making the counting operation $O(n)$ regardless of the threshold value.

Binary search converges to an approximate value, but bounds require exact pairwise averages. After the search converges, the algorithm identifies candidate exact values near the approximate threshold, then selects the one at the correct rank. The candidate generation examines positions near the boundary: for each row i, it finds the j values where the pairwise average crosses the threshold, collecting the actual pairwise average values. Sorting these candidates and verifying ranks ensures the exact quantile is returned.

$\mathrm{CenterBounds}$ needs two quantiles: $w_{(k_{\mathrm{left}})}$ and $w_{(k_{\mathrm{right}})}$. The algorithm computes each independently using the same technique. For symmetric bounds around the center, the lower and upper ranks are $k_{\mathrm{left}} = \left\lfloor \frac{\mathrm{SignedRankMargin}}{2} \right\rfloor + 1$ and $k_{\mathrm{right}} = N - \left\lfloor \frac{\mathrm{SignedRankMargin}}{2} \right\rfloor$.

The algorithm achieves $O(n \log n)$ time complexity for sorting, plus $O(n \log R)$ for binary search where R is the value range precision. Memory usage is $O(n)$ for the sorted array plus $O(n)$ for candidate generation. This is dramatically more efficient than the naive $O(n^2 \log n^2)$ approach. For $n = 10000$, the fast algorithm completes in milliseconds versus minutes for the naive approach.

The algorithm uses relative tolerance for convergence: $\mathrm{hi} - \mathrm{lo} \le 10^{-14} \cdot \max(1, |\mathrm{lo}|, |\mathrm{hi}|)$. The floor of 1 prevents degenerate tolerance when bounds are near zero. This ensures stable behavior across

different scales of input data. For candidate generation near the threshold, small tolerances prevent missing exact values due to floating-point imprecision.

```
using System.Diagnostics;

namespace Pragmastat.Algorithms;

/// <summary>
/// Efficiently computes quantiles from all pairwise averages (x[i] + x[j]) / 2 for i ≤ j.
/// Uses binary search with counting function to avoid materializing all N(N+1)/2 pairs.
/// </summary>
internal static class FastCenterQuantiles
{
  private static bool IsSorted(IReadOnlyList<double> list)
  {
    for (int i = 1; i < list.Count; i++)
      if (list[i] < list[i - 1])
        return false;
    return true;
  }
  /// <summary>
  /// Relative epsilon for floating-point comparisons in binary search convergence.
  /// </summary>
  private const double RelativeEpsilon = 1e-14;
  /// <summary>
  /// Compute specified quantile from pairwise averages.
  /// </summary>
  /// <param name="sorted">Sorted input array.</param>
  /// <param name="k">1-based rank of the desired quantile.</param>
  /// <returns>The k-th smallest pairwise average.</returns>
  public static double Quantile(IReadOnlyList<double> sorted, long k)
  {
    Debug.Assert(
      IsSorted(sorted),
      "FastCenterQuantiles.Quantile: input must be sorted");
    int n = sorted.Count;
    if (n == 0)
      throw new ArgumentException("Input cannot be empty", nameof(sorted));

    long totalPairs = (long)n * (n + 1) / 2;
    if (k < 1 || k > totalPairs)
      throw new ArgumentOutOfRangeException(nameof(k), $"k must be in range [1,
{totalPairs}]");

    if (n == 1)
      return sorted[0];

    return FindExactQuantile(sorted, k);
  }

  /// <summary>
  /// Compute both lower and upper bounds from pairwise averages.
  /// </summary>
```

```
  /// <param name="sorted">Sorted input array.</param>
  /// <param name="marginLo">Rank of lower bound (1-based).</param>
  /// <param name="marginHi">Rank of upper bound (1-based).</param>
  /// <returns>Lower and upper quantiles.</returns>
  public static (double Lo, double Hi) Bounds(IReadOnlyList<double> sorted, long marginLo,
long marginHi)
  {
    Debug.Assert(
      IsSorted(sorted),
      "FastCenterQuantiles.Bounds: input must be sorted");
    int n = sorted.Count;
    if (n == 0)
      throw new ArgumentException("Input cannot be empty", nameof(sorted));

    long totalPairs = (long)n * (n + 1) / 2;

    marginLo = Max(1, Min(marginLo, totalPairs));
    marginHi = Max(1, Min(marginHi, totalPairs));

    double lo = FindExactQuantile(sorted, marginLo);
    double hi = FindExactQuantile(sorted, marginHi);

    return (Min(lo, hi), Max(lo, hi));
  }

  /// <summary>
  /// Count pairwise averages ≤ target value.
  /// </summary>
  private static long CountPairsLessOrEqual(IReadOnlyList<double> sorted, double target)
  {
    int n = sorted.Count;
    long count = 0;
    // j is not reset: as i increases, threshold decreases monotonically
    int j = n - 1;

    for (int i = 0; i < n; i++)
    {
      double threshold = 2 * target - sorted[i];

      while (j >= 0 && sorted[j] > threshold)
        j--;

      if (j >= i)
        count += j - i + 1;
    }

    return count;
  }

  /// <summary>
  /// Find exact k-th pairwise average using selection algorithm.
  /// </summary>
  private static double FindExactQuantile(IReadOnlyList<double> sorted, long k)
```

```
  {
    int n = sorted.Count;
    long totalPairs = (long)n * (n + 1) / 2;

    if (n == 1)
      return sorted[0];

    if (k == 1)
      return sorted[0];

    if (k == totalPairs)
      return sorted[n - 1];

    double lo = sorted[0];
    double hi = sorted[n - 1];
    const double eps = RelativeEpsilon;

    var candidates = new List<double>(n);

    while (hi - lo > eps * Max(1.0, Max(Abs(lo), Abs(hi))))
    {
      double mid = (lo + hi) / 2;
      long countLessOrEqual = CountPairsLessOrEqual(sorted, mid);

      if (countLessOrEqual >= k)
        hi = mid;
      else
        lo = mid;
    }

    double target = (lo + hi) / 2;

    for (int i = 0; i < n; i++)
    {
      double threshold = 2 * target - sorted[i];

      int left = i;
      int right = n;

      while (left < right)
      {
        int m = (left + right) / 2;
        if (sorted[m] < threshold - eps)
          left = m + 1;
        else
          right = m;
      }

      if (left < n && left >= i && Abs(sorted[left] - threshold) < eps * Max(1.0,
Abs(threshold)))
        candidates.Add((sorted[i] + sorted[left]) / 2);

      if (left > i)
```

```
      {
        double avgBefore = (sorted[i] + sorted[left - 1]) / 2;
        if (avgBefore <= target + eps)
          candidates.Add(avgBefore);
      }
    }

    if (candidates.Count == 0)
      return target;

    candidates.Sort();

    foreach (double candidate in candidates)
    {
      long countAtCandidate = CountPairsLessOrEqual(sorted, candidate);
      if (countAtCandidate >= k)
        return candidate;
    }

    return target;
  }
}
```

1.2.2. Notes

1.2.2.1. On Bootstrap for Center Bounds

A natural question arises: can bootstrap resampling improve $\mathrm{CenterBounds}$ coverage for asymmetric distributions where the weak symmetry assumption fails?

The idea is appealing. The signed-rank approach computes bounds from order statistics of Walsh averages using a margin derived from the Wilcoxon distribution, which assumes symmetric deviations from the center. Bootstrap makes no symmetry assumption: resample the data with replacement, compute Center on each resample, and take quantiles of the bootstrap distribution as bounds. This should yield valid bounds regardless of distributional shape.

This manual deliberately does not provide a bootstrap-based alternative to $\mathrm{CenterBounds}$. The reasons are both computational and statistical.

Computational cost

$\mathrm{CenterBounds}$ computes bounds in $O(n \log n)$ time: a single pass through the Walsh averages guided by the signed-rank margin. No resampling, no iteration.

A bootstrap version requires B resamples (typically $B = 10000$ for stable tail quantiles), each computing Center on the resample. Center itself costs $O(n \log n)$ via the fast selection algorithm on the implicit pairwise matrix. The total cost becomes $O(B \cdot n \log n)$ per call — roughly $10000\times$ slower than the signed-rank approach.

For $n = 5$, each call to Center operates on 15 Walsh averages. The bootstrap recomputes this 10000 times. The computation is not deep — it is merely wasteful. For $n = 100$, there are 5050 Walsh averages per resample, and 10000 resamples produce 5×10^7 selection operations per bounds call. In a simulation study that evaluates bounds across many samples, this cost becomes prohibitive.

Statistical quality

Bootstrap bounds are *nominal*, not exact. The percentile method has well-documented undercoverage for small samples: requesting 95% confidence ($\mathrm{misrate} = 0.05$) typically yields 85–92% actual coverage for $n < 30$. This is inherent to the bootstrap percentile method — the quantile estimates from B resamples are biased toward the sample and underrepresent tail behavior. Refined methods (BCa, bootstrap-t) partially address this but add complexity and still provide only asymptotic guarantees.

Meanwhile, $\mathrm{CenterBounds}$ provides exact distribution-free coverage under symmetry. For $n = 5$ requesting $\mathrm{misrate} = 0.1$, the signed-rank method delivers exactly 10% misrate. A bootstrap method, requesting the same 10%, typically delivers 12–15% misrate. The exact method is simultaneously faster and more accurate.

Behavior under asymmetry

Under asymmetric distributions, the signed-rank margin is no longer calibrated: the Wilcoxon distribution assumes symmetric deviations, and asymmetry shifts the actual distribution of comparison counts.

However, the coverage degradation is gradual, not catastrophic. Mild asymmetry produces mild coverage drift. The bounds remain meaningful — they still bracket the pseudomedian using order statistics of the Walsh averages — but the actual misrate differs from the requested value.

This is the same situation as bootstrap, which also provides only approximate coverage. The practical difference is that the signed-rank approach achieves this approximate coverage in $O(n \log n)$ time, while bootstrap achieves comparable approximate coverage in $O(B \cdot n \log n)$ time.

Why not both?

One might argue for providing both methods: the signed-rank approach as default, and a bootstrap variant for cases where symmetry is severely violated.

This creates a misleading choice. If the bootstrap method offered substantially better coverage under asymmetry, the complexity would be justified. But for the distributions practitioners encounter (Multiplic, Exp, and other moderate asymmetries), the coverage difference between the two approaches is small relative to the $10000\times$ cost difference. For extreme asymmetries where the signed-rank coverage genuinely breaks down, the sign test provides an alternative foundation for median bounds (see On Misrate Efficiency of MedianBounds), but its $O\left(n^{-1/2}\right)$ efficiency convergence makes it impractical for moderate sample sizes.

The toolkit therefore provides CenterBounds as the single bounds estimator. The weak symmetry assumption means the method performs well under approximate symmetry and degrades gracefully under moderate asymmetry. There is no useful middle ground that justifies a $10000\times$ computational penalty for marginally different approximate coverage.

1.2.2.2. On Misrate Efficiency of MedianBounds

This note analyzes MedianBounds, a bounds estimator for the population median based on the sign test, and explains why pragmastat omits it in favor of CenterBounds.

Definition

$$\text{MedianBounds}(\mathbf{x}, \text{misrate}) = \left[x_{(k)}, x_{(n-k+1)}\right]$$

where k is the largest integer satisfying $2 \cdot \Pr(B \le k-1) \le \text{misrate}$ and $B \sim \text{Binomial}(n, 0.5)$. The interval brackets the population median using order statistics, with misrate controlling the probability that the true median falls outside the bounds.

MedianBounds requires no symmetry assumption — only weak continuity — making it applicable to arbitrarily skewed distributions. This is its principal advantage over CenterBounds, which assumes weak symmetry.

Sign test foundation

The method is equivalent to inverting the sign test. Under weak continuity, each observation independently falls above or below the true median with probability $1/2$. The number of observations below the median follows $\text{Binomial}(n, 0.5)$, and the order statistics $x_{(k)}$ and $x_{(n-k+1)}$ form a confidence interval whose coverage is determined exactly by the binomial CDF.

Because the binomial CDF is a step function, the achievable misrate values form a discrete set. The algorithm rounds down to the nearest achievable level, inevitably wasting part of the requested misrate budget. This note derives the resulting efficiency loss and its convergence rate.

Achievable misrate levels

The achievable misrate values for sample size n are:

$$m_k = 2 \cdot \Pr(B \leq k), \quad k = 0, 1, 2, ...$$

The algorithm selects the largest k satisfying $m_k \leq \text{misrate}$. The *efficiency* $\eta = m_k / \text{misrate}$ measures how much of the requested budget is used. Efficiency $\eta = 1$ means the bounds are as tight as the misrate allows; $\eta = 0.5$ means half the budget is wasted, producing unnecessarily wide bounds.

Spacing between consecutive levels

The gap between consecutive achievable misrates is:

$$\Delta m_k = m_{k+1} - m_k = 2 \cdot \Pr(B = k+1) = 2 \cdot \binom{n}{k+1} / 2^n$$

For a target misrate α, the relevant index k satisfies $m_k \approx \alpha$. By the normal approximation to the binomial, $B \approx \mathcal{N}(n/2, n/4)$, the binomial CDF near this index changes by approximately:

$$\Delta m \approx \frac{4\varphi(z_\alpha)}{\sqrt{n}}$$

where $z_\alpha = \Phi^{-1}(\alpha/2)$ is the corresponding standard normal quantile and φ is the standard normal density. This spacing governs how coarsely the achievable misrates are distributed near the target.

Expected efficiency

The requested misrate α falls at a uniformly random position within a gap of width Δm. On average, the algorithm wastes $\Delta m/2$, giving expected efficiency:

$$\mathbb{E}[\eta] \approx 1 - \frac{\Delta m}{2\alpha} = 1 - \frac{2\varphi(z_\alpha)}{\alpha\sqrt{n}}$$

Define the misrate-dependent constant:

$$C(\alpha) = \frac{2\varphi(\Phi^{-1}(\alpha/2))}{\alpha}$$

Then the expected efficiency has the form:

$$\mathbb{E}[\eta] \approx 1 - \frac{C(\alpha)}{\sqrt{n}}$$

The convergence rate is $O\big(n^{-1/2}\big)$: efficiency improves as the square root of sample size.

Values of $C(\alpha)$

The constant $C(\alpha)$ increases for smaller misrates, meaning tighter error tolerances require proportionally larger samples for efficient bounds:

α	0.1	0.05	0.01	0.005	0.001
z_α	-1.64	-1.96	-2.58	-2.81	-3.29
$\varphi(z_\alpha)$	0.103	0.058	0.015	0.008	0.002
$C(\alpha)$	2.06	2.33	2.94	3.17	3.45

For $\alpha = 0.1$ and $n = 50$: $\mathbb{E}[\eta] \approx 1 - 2.06/\sqrt{50} \approx 0.71$. Achieving 90% efficiency on average requires $n > (C(\alpha)/0.1)^2$. For $\alpha = 0.1$ this gives $n > 424$; for $\alpha = 0.001$ this gives $n > 1190$.

Comparison with CenterBounds

$\mathrm{CenterBounds}$ uses the signed-rank statistic W with range $[0, n(n+1)/2]$. Under the null hypothesis, W has variance $\sigma^2 = n(n+1)(2n+1)/24 \approx n^3/12$. The CDF spacing at the relevant quantile is:

$$\Delta m_W \approx \frac{2\sqrt{12} \cdot \varphi(z_\alpha)}{n^{3/2}}$$

The expected efficiency for $\mathrm{CenterBounds}$ is therefore:

$$\mathbb{E}[\eta_W] \approx 1 - \frac{\sqrt{12} \cdot \varphi(z_\alpha)}{\alpha \cdot n^{3/2}}$$

This converges at rate $O\big(n^{-3/2}\big)$ — three polynomial orders faster than $\mathrm{MedianBounds}$. The difference arises because the signed-rank distribution has $n(n+1)/2$ discrete levels compared to the binomial's n levels, providing fundamentally finer resolution.

Why pragmastat omits MedianBounds

The efficiency loss of $\mathrm{MedianBounds}$ is not an implementation artifact. It reflects a structural limitation of the sign test: using only the signs of $(X_i - \theta)$ discards magnitude information, leaving only n binary observations to determine coverage. The signed-rank test used by $\mathrm{CenterBounds}$ exploits both signs and ranks, producing $n(n+1)/2$ comparison outcomes and correspondingly finer misrate resolution.

For applications requiring tight misrate control on the median, large samples ($n > 500$) are needed to ensure efficient use of the misrate budget. For smaller samples, the bounds remain valid but conservative: the actual misrate is guaranteed to not exceed the requested value, even though it may be substantially below it.

$\mathrm{CenterBounds}$ with its $O\big(n^{-3/2}\big)$ convergence achieves near-continuous misrate control even for moderate n, at the cost of requiring weak symmetry. For the distributions practitioners typically encounter, this tradeoff favors $\mathrm{CenterBounds}$ as the single bounds estimator in the toolkit. When symmetry is severely violated, the coverage drift of $\mathrm{CenterBounds}$ is gradual — mild asymmetry produces mild drift — making it a robust default without the efficiency penalty inherent to the sign test approach.

1.2.3. Tests

$$\text{CenterBounds}(\mathbf{x}, \text{misrate}) = \left[w_{(k_{\text{left}})}, w_{(k_{\text{right}})}\right]$$

where $\mathbf{w} = \{(x_i + x_j)/2\}$ (pairwise averages, sorted) for $i \le j$, $k_{\text{left}} = \lfloor \text{SignedRankMargin}/2 \rfloor + 1$, $k_{\text{right}} = N - \lfloor \text{SignedRankMargin}/2 \rfloor$, and $N = n(n+1)/2$.

The CenterBounds test suite contains 43 test cases (3 demo + 4 natural + 5 property + 7 edge + 4 symmetric + 4 asymmetric + 2 additive + 2 uniform + 4 misrate + 6 unsorted + 2 error cases). Since CenterBounds returns bounds rather than a point estimate, tests validate that bounds contain $\text{Center}(\mathbf{x})$ and satisfy equivariance properties. Each test case output is a JSON object with `lower` and `upper` fields representing the interval bounds.

Demo examples — from manual introduction, validating basic bounds:

`demo-1`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\text{misrate} = 0.1$, expected output: $[1.5, 4.5]$
`demo-2`: $\mathbf{x} = (1, \ldots, 10)$, $\text{misrate} = 0.01$, expected output: $[2.5, 8.5]$
`demo-3`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\text{misrate} = 0.1$

These cases illustrate how tighter misrates produce wider bounds.

Natural sequences ($n \in \{5, 7, 10, 20\}$, $\text{misrate} = 0.01$) — 4 tests:

`natural-5`: $\mathbf{x} = (1, 2, 3, 4, 5)$, bounds containing $\text{Center} = 3$
`natural-7`: $\mathbf{x} = (1, \ldots, 7)$, bounds containing $\text{Center} = 4$
`natural-10`: $\mathbf{x} = (1, \ldots, 10)$, expected output: $[2.5, 8.5]$
`natural-20`: $\mathbf{x} = (1, \ldots, 20)$, bounds containing $\text{Center} = 10.5$

Property validation ($n = 5$, $\text{misrate} = 0.05$) — 5 tests:

`property-identity`: $\mathbf{x} = (1, 2, 3, 4, 5)$, bounds must contain $\text{Center} = 3$
`property-centered`: $\mathbf{x} = (-2, -1, 0, 1, 2)$, bounds must contain $\text{Center} = 0$
`property-location-shift`: $\mathbf{x} = (11, 12, 13, 14, 15)$ (= demo-1 + 10), bounds must be demo-1 bounds + 10
`property-scale-2x`: $\mathbf{x} = (2, 4, 6, 8, 10)$ (= 2 × demo-1), bounds must be 2× demo-1 bounds
`property-mixed-signs`: $\mathbf{x} = (-2, -1, 0, 1, 2)$, validates bounds crossing zero

Edge cases — boundary conditions and extreme scenarios (7 tests):

`edge-two-elements`: $\mathbf{x} = (1, 2)$, $\text{misrate} = 0.5$ (minimum meaningful sample)
`edge-three-elements`: $\mathbf{x} = (1, 2, 3)$, $\text{misrate} = 0.25$ (small sample)
`edge-loose-misrate`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\text{misrate} = 0.5$ (permissive bounds)
`edge-strict-misrate`: $\mathbf{x} = (1, \ldots, 10)$, $\text{misrate} = 0.002$ (near-minimum misrate for $n = 10$)
`edge-duplicates-10`: $\mathbf{x} = (5, 5, 5, 5, 5, 5, 5, 5, 5, 5)$, $\text{misrate} = 0.01$ (all identical, bounds $= [5, 5]$)
`edge-negative`: $\mathbf{x} = (-5, -4, -3, -2, -1)$, $\text{misrate} = 0.05$ (negative values)
`edge-wide-range`: $\mathbf{x} = (1, 10, 100, 1000, 10000)$, $\text{misrate} = 0.1$ (extreme value range)

Symmetric distributions ($\text{misrate} = 0.05$) — 4 tests with symmetric data:

`symmetric-5`: $\mathbf{x} = (-2, -1, 0, 1, 2)$, bounds centered around 0
`symmetric-7`: $\mathbf{x} = (-3, -2, -1, 0, 1, 2, 3)$, bounds centered around 0
`symmetric-10`: $n = 10$ symmetric around 0
`symmetric-15`: $n = 15$ symmetric around 0

These tests validate that symmetric data produces symmetric bounds around the center.

Asymmetric distributions ($n = 5, \text{misrate} = 0.1$) — 4 tests validating bounds with asymmetric data:

`asymmetric-left-skew`: $\mathbf{x} = (1, 7, 8, 9, 10)$, expected output: $[4, 9.5]$
`asymmetric-right-skew`: $\mathbf{x} = (1, 2, 3, 4, 10)$, expected output: $[1.5, 7]$
`asymmetric-bimodal`: $\mathbf{x} = (1, 1, 5, 9, 9)$, expected output: $[1, 9]$
`asymmetric-outlier`: $\mathbf{x} = (1, 2, 3, 4, 100)$, expected output: $[1.5, 52]$

These tests validate that $\mathrm{CenterBounds}$ handles asymmetric data correctly, complementing the symmetric test cases.

Additive distribution ($\text{misrate} = 0.01$) — 2 tests with $\underline{\mathrm{Additive}}(10, 1)$:

`additive-10`: $n = 10$, seed 0
`additive-20`: $n = 20$, seed 0

Uniform distribution ($\text{misrate} = 0.01$) — 2 tests with $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-10`: $n = 10$, seed 1
`uniform-20`: $n = 20$, seed 1

Misrate variation ($\mathbf{x} = (1, ..., 10)$) — 4 tests with varying misrates:

`misrate-1e-1`: $\text{misrate} = 0.1$
`misrate-5e-2`: $\text{misrate} = 0.05$
`misrate-1e-2`: $\text{misrate} = 0.01$
`misrate-5e-3`: $\text{misrate} = 0.005$

These tests validate monotonicity: smaller misrates produce wider bounds.

Unsorted tests — verify sorting independence (6 tests):

`unsorted-reverse-5`: $\mathbf{x} = (5, 4, 3, 2, 1)$, must equal `natural-5` output
`unsorted-reverse-7`: $\mathbf{x} = (7, 6, 5, 4, 3, 2, 1)$, must equal `natural-7` output
`unsorted-shuffle-5`: $\mathbf{x}$ shuffled, must equal sorted counterpart
`unsorted-shuffle-7`: $\mathbf{x}$ shuffled, must equal sorted counterpart
`unsorted-negative-5`: negative values unsorted
`unsorted-mixed-signs-5`: mixed signs unsorted

These tests validate that $\mathrm{CenterBounds}$ produces identical results regardless of input order.

Error cases — 2 tests validating input validation:

`error-single-element`: $\mathbf{x} = (1), \text{misrate} = 0.5$ (minimum sample size violation)

`error-invalid-misrate`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\text{misrate} = 0.001$ (misrate below minimum achievable)

1.3. Spread

$$\text{Spread}(\mathbf{x}) = \text{Median}_{1 \le i < j \le n} |x_i - x_j|$$

Robust measure of dispersion (variability, scatter).

Also known as — Shamos scale estimator
Asymptotic — median of the absolute difference between two random measurements from X
Complexity — $O(n \log n)$
Domain — any real numbers
Assumptions — `sparity(x)`
Unit — same as measurements

Properties

Shift invariance	$\text{Spread}(\mathbf{x} + k) = \text{Spread}(\mathbf{x})$
Scale equivariance	$\text{Spread}(k \cdot \mathbf{x}) = \lvert k \rvert \cdot \text{Spread}(\mathbf{x})$
Non-negativity	$\text{Spread}(\mathbf{x}) \ge 0$

Example

```
Spread([0, 2, 4, 6, 8]) = 4
Spread(x + 10) = 4   Spread(2x) = 8
```

Spread measures how much measurements vary from each other. It serves the same purpose as standard deviation but does not explode with outliers or heavy-tailed data. The result comes in the same units as the measurements, so if Spread is 5 milliseconds, that indicates how much values typically differ. Like Center, it tolerates up to 29% corrupted data. When comparing variability across datasets, Spread gives a reliable answer even when standard deviation would be misleading or infinite. When all values are positive, Spread can be conveniently expressed in relative units by dividing by |Center|: the ratio $\frac{\text{Spread}(\mathbf{x})}{|\text{Center}(\mathbf{x})|}$ is a robust alternative to the coefficient of variation that is scale-invariant.

1.3.1. Algorithm

The Spread estimator computes the median of all pairwise absolute differences. Given a sample $x = (x_1, x_2, ..., x_n)$, this estimator is defined as:

$$\mathrm{Spread}(\mathbf{x}) = \mathrm{median}_{1 \le i < j \le n} \left| x_i - x_j \right|$$

Like Center, computing Spread naively requires generating all $\frac{n(n-1)}{2}$ pairwise differences, sorting them, and finding the median — a quadratic approach that becomes computationally prohibitive for large datasets.

The same structural principles that accelerate Center computation also apply to pairwise differences, yielding an exact $O(n \log n)$ algorithm. After sorting the input to obtain $y_1 \le y_2 \le ... \le y_n$, all pairwise absolute differences $\left| x_i - x_j \right|$ with $i < j$ become positive differences $y_j - y_i$. This allows considering the implicit upper triangular matrix D where $D_{i,j} = y_j - y_i$ for $i < j$. This matrix has a crucial structural property: for a fixed row i, differences increase monotonically, while for a fixed column j, differences decrease as i increases. This sorted structure enables linear-time counting of elements below any threshold.

The algorithm applies Monahan's selection strategy (Monahan (1984)), adapted for differences rather than sums. For each row i, it tracks active column indices representing differences still under consideration, initially spanning columns $i + 1$ through n. It chooses candidate differences from the active set using weighted random row selection, maintaining expected logarithmic convergence while avoiding expensive pivot computations. For any pivot value p, the number of differences falling below p is counted using a single sweep; the monotonic structure ensures this counting requires only $O(n)$ operations. While counting, the largest difference below p and the smallest difference at or above p are maintained — these boundary values become the exact answer when the target rank is reached.

The algorithm naturally handles both odd and even cases. For an odd number of differences, it returns the single middle element when the count exactly hits the median rank. For an even number of differences, it returns the average of the two middle elements; boundary tracking during counting provides both values simultaneously. Unlike approximation methods, this algorithm returns the precise median of all pairwise differences; randomness affects only performance, not correctness.

The algorithm includes the same stall-handling mechanisms as the center algorithm. It tracks whether the count below the pivot changes between iterations; when progress stalls due to tied values, it computes the range of remaining active differences and pivots to their midrange. This midrange strategy ensures convergence even with highly discrete data or datasets with many identical values.

Several optimizations make the implementation practical for production use. A global column pointer that never moves backward during counting exploits the matrix structure to avoid redundant comparisons. Exact boundary values are captured during each counting pass, eliminating the need for additional searches when the target rank is reached. Using only $O(n)$ additional space for row bounds and counters, the algorithm achieves $O(n \log n)$ time complexity with minimal memory overhead, making robust scale estimation practical for large datasets.

```
namespace Pragmastat.Algorithms;
```

```
internal static class FastSpread
{
  /// <summary>
  /// Shamos "Spread".  Expected O(n log n) time, O(n) extra space. Exact.
  /// </summary>
  public static double Estimate(IReadOnlyList<double> values, Random? random = null, bool
isSorted = false)
  {
    int n = values.Count;
    if (n <= 1) return 0;
    if (n == 2) return Abs(values[1] - values[0]);
    random ??= new Random();

    // Prepare a sorted working copy.
    double[] a = isSorted ? CopySorted(values) : EnsureSorted(values);

    // Total number of pairwise differences with i < j
    long N = (long)n * (n - 1) / 2;
    long kLow = (N + 1) / 2; // 1-based rank of lower middle
    long kHigh = (N + 2) / 2; // 1-based rank of upper middle

    // Per-row active bounds over columns j (0-based indices).
    // Row i allows j in [i+1, n-1] initially.
    int[] L = new int[n];
    int[] R = new int[n];
    long[] rowCounts = new long[n]; // # of elements in row i that are < pivot (for
current partition)

    for (int i = 0; i < n; i++)
    {
      L[i] = Min(i + 1, n); // n means empty
      R[i] = n - 1; // inclusive
      if (L[i] > R[i])
      {
        L[i] = 1;
        R[i] = 0;
      } // mark empty
    }

    // A reasonable initial pivot: a central gap
    double pivot = a[n / 2] - a[(n - 1) / 2];

    long prevCountBelow = -1;

    while (true)
    {
      // === PARTITION: count how many differences are < pivot; also track boundary
neighbors ===
      long countBelow = 0;
      double largestBelow = double.NegativeInfinity; // max difference < pivot
      double smallestAtOrAbove = double.PositiveInfinity; // min difference >= pivot

      int j = 1; // global two-pointer (non-decreasing across rows)
```

```
      for (int i = 0; i < n - 1; i++)
      {
        if (j < i + 1) j = i + 1;
        while (j < n && a[j] - a[i] < pivot) j++;

        long cntRow = j - (i + 1);
        if (cntRow < 0) cntRow = 0;
        rowCounts[i] = cntRow;
        countBelow += cntRow;

        // boundary elements for this row
        if (cntRow > 0)
        {
          // last < pivot in this row is (j-1)
          double candBelow = a[j - 1] - a[i];
          if (candBelow > largestBelow) largestBelow = candBelow;
        }

        if (j < n)
        {
          double candAtOrAbove = a[j] - a[i];
          if (candAtOrAbove < smallestAtOrAbove) smallestAtOrAbove = candAtOrAbove;
        }
      }

      // === TARGET CHECK ===
      // If we've split exactly at the middle, we can return using the boundaries we just
found.
      bool atTarget =
        (countBelow == kLow) || // lower middle is the largest < pivot
        (countBelow == (kHigh - 1)); // upper middle is the smallest >= pivot

      if (atTarget)
      {
        if (kLow < kHigh)
        {
          // Even N: average the two central order stats.
          return 0.5 * (largestBelow + smallestAtOrAbove);
        }
        else
        {
          // Odd N: pick the single middle depending on which side we hit.
          bool needLargest = (countBelow == kLow);
          return needLargest ? largestBelow : smallestAtOrAbove;
        }
      }

      // === STALL HANDLING (ties / no progress) ===
      if (countBelow == prevCountBelow)
      {
        // Compute min/max remaining difference in the ACTIVE set and pivot to their
midrange.
        double minActive = double.PositiveInfinity;
```

```
        double maxActive = double.NegativeInfinity;
        long active = 0;

        for (int i = 0; i < n - 1; i++)
        {
          int Li = L[i], Ri = R[i];
          if (Li > Ri) continue;

          double rowMin = a[Li] - a[i];
          double rowMax = a[Ri] - a[i];
          if (rowMin < minActive) minActive = rowMin;
          if (rowMax > maxActive) maxActive = rowMax;
          active += (Ri - Li + 1);
        }

        if (active <= 0)
        {
          // No active candidates left: the only consistent answer is the boundary implied
by counts.
          // Fall back to neighbors from this partition.
          if (kLow < kHigh) return 0.5 * (largestBelow + smallestAtOrAbove);
          return (countBelow >= kLow) ? largestBelow : smallestAtOrAbove;
        }

        if (maxActive <= minActive) return minActive; // all remaining equal

        double mid = 0.5 * (minActive + maxActive);
        pivot = (mid > minActive && mid <= maxActive) ? mid : maxActive;
        prevCountBelow = countBelow;
        continue;
      }

      // === SHRINK ACTIVE WINDOW ===
      // --- SHRINK ACTIVE WINDOW (fixed) ---
      if (countBelow < kLow)
      {
        // Need larger differences: discard all strictly below pivot.
        for (int i = 0; i < n - 1; i++)
        {
          // First j with a[j] - a[i] >= pivot is j = i + 1 + cntRow (may be n => empty
row)
          int newL = i + 1 + (int)rowCounts[i];
          if (newL > L[i]) L[i] = newL; // do NOT clamp; allow L[i] == n to mean empty
          if (L[i] > R[i])
          {
            L[i] = 1;
            R[i] = 0;
          } // mark empty
        }
      }
      else
      {
        // Too many below: keep only those strictly below pivot.
```

```
    for (int i = 0; i < n - 1; i++)
    {
      // Last j with a[j] - a[i] < pivot is j = i + cntRow  (not cntRow-1!)
      int newR = i + (int)rowCounts[i];
      if (newR < R[i]) R[i] = newR; // shrink downward to the true last-below
      if (R[i] < i + 1)
      {
        L[i] = 1;
        R[i] = 0;
      } // empty row if none remain
    }
  }

  prevCountBelow = countBelow;

  // === CHOOSE NEXT PIVOT FROM ACTIVE SET (weighted random row, then row median) ===
  long activeSize = 0;
  for (int i = 0; i < n - 1; i++)
  {
    if (L[i] <= R[i]) activeSize += (R[i] - L[i] + 1);
  }

  if (activeSize <= 2)
  {
    // Few candidates left: return midrange of remaining exactly.
    double minRem = double.PositiveInfinity, maxRem = double.NegativeInfinity;
    for (int i = 0; i < n - 1; i++)
    {
      if (L[i] > R[i]) continue;
      double lo = a[L[i]] - a[i];
      double hi = a[R[i]] - a[i];
      if (lo < minRem) minRem = lo;
      if (hi > maxRem) maxRem = hi;
    }

    if (activeSize <= 0) // safety net; fall back to boundary from last partition
    {
      if (kLow < kHigh) return 0.5 * (largestBelow + smallestAtOrAbove);
      return (countBelow >= kLow) ? largestBelow : smallestAtOrAbove;
    }

    if (kLow < kHigh) return 0.5 * (minRem + maxRem);
    return (Abs((kLow - 1) - countBelow) <= Abs(countBelow - kLow)) ? minRem : maxRem;
  }
  else
  {
    long t = NextIndex(random, activeSize); // 0..activeSize-1
    long acc = 0;
    int row = 0;
    for (; row < n - 1; row++)
    {
      if (L[row] > R[row]) continue;
      long size = R[row] - L[row] + 1;
```

```
          if (t < acc + size) break;
          acc += size;
        }

        // Median column of the selected row
        int col = (L[row] + R[row]) >> 1;
        pivot = a[col] - a[row];
      }
    }
  }
  // --- Helpers ---

  private static double[] CopySorted(IReadOnlyList<double> values)
  {
    var a = new double[values.Count];
    for (int i = 0; i < a.Length; i++)
    {
      double v = values[i];
      if (double.IsNaN(v)) throw new ArgumentException("NaN not allowed.",
nameof(values));
      a[i] = v;
    }

    Array.Sort(a);
    return a;
  }

  private static double[] EnsureSorted(IReadOnlyList<double> values)
  {
    // Trust caller; still copy to array for fast indexed access.
    var a = new double[values.Count];
    for (int i = 0; i < a.Length; i++)
    {
      double v = values[i];
      if (double.IsNaN(v)) throw new ArgumentException("NaN not allowed.",
nameof(values));
      a[i] = v;
    }

    return a;
  }

  private static long NextIndex(Random rng, long limitExclusive)
  {
    // Uniform 0..limitExclusive-1 even for large ranges.
    // Use rejection sampling for correctness.
    ulong uLimit = (ulong)limitExclusive;
    if (uLimit <= int.MaxValue)
    {
      return rng.Next((int)uLimit);
    }

    while (true)
```

```
    {
      ulong u = ((ulong)(uint)rng.Next() << 32) | (uint)rng.Next();
      ulong r = u % uLimit;
      if (u - r <= ulong.MaxValue - (ulong.MaxValue % uLimit)) return (long)r;
    }
  }
}
```

1.3.2. Notes

This section compares the toolkit's robust dispersion estimator against traditional methods to demonstrate its advantages across diverse conditions.

Dispersion Estimators

Standard Deviation:

$$\mathrm{StdDev}(\mathbf{x}) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mathrm{Mean}(\mathbf{x})\right)^2}$$

Median Absolute Deviation (around the median):

$$\mathrm{MAD}(\mathbf{x}) = \mathrm{Median}(|x_i - \mathrm{Median}(\mathbf{x})|)$$

Spread (Shamos scale estimator):

$$\mathrm{Spread}(\mathbf{x}) = \mathrm{Median}_{1\le i<j\le n}|x_i - x_j|$$

1.3.2.1. Breakdown

Heavy-tailed distributions naturally produce extreme outliers that completely distort traditional estimators. A single extreme measurement from the Power distribution can make the standard deviation arbitrarily large. Real-world data can also contain corrupted measurements from instrument failures, recording errors, or transmission problems. Both natural extremes and data corruption create the same challenge: extracting reliable information when some measurements are too influential.

The breakdown point (Huber (2009)) is the fraction of a sample that can be replaced by arbitrarily large values without making an estimator arbitrarily large. The theoretical maximum is 50%; no estimator can guarantee reliable results when more than half the measurements are extreme or corrupted.

The Spread estimator achieves a 29% breakdown point, providing substantial protection against realistic contamination levels while maintaining good precision.

Asymptotic breakdown points for dispersion estimators:

StdDev	MAD	Spread
0%	50%	29%

1.3.2.2. Drift

Drift measures estimator precision by quantifying how much estimates scatter across repeated samples. It is based on the Spread of estimates and therefore has a breakdown point of approximately 29%.

Drift is useful for comparing the precision of several estimators. To simplify the comparison, one of the estimators can be chosen as a baseline. A table of squared drift values, normalized by the baseline, shows the required sample size adjustment factor for switching from the baseline to another estimator. For example, if Spread is the baseline and the rescaled drift square of MAD is 1.25, this means that

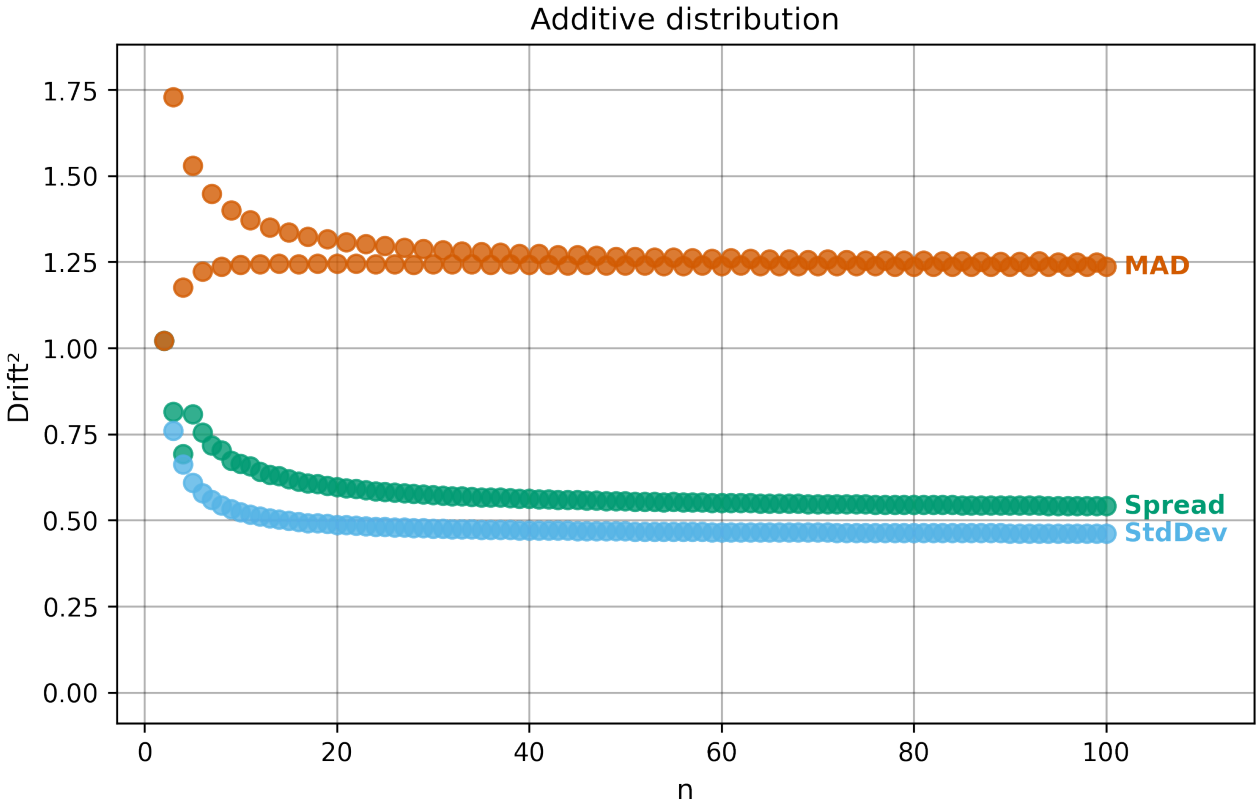
MAD requires 1.25 times more data than Spread to achieve the same precision. See From Statistical Efficiency to Drift for details.

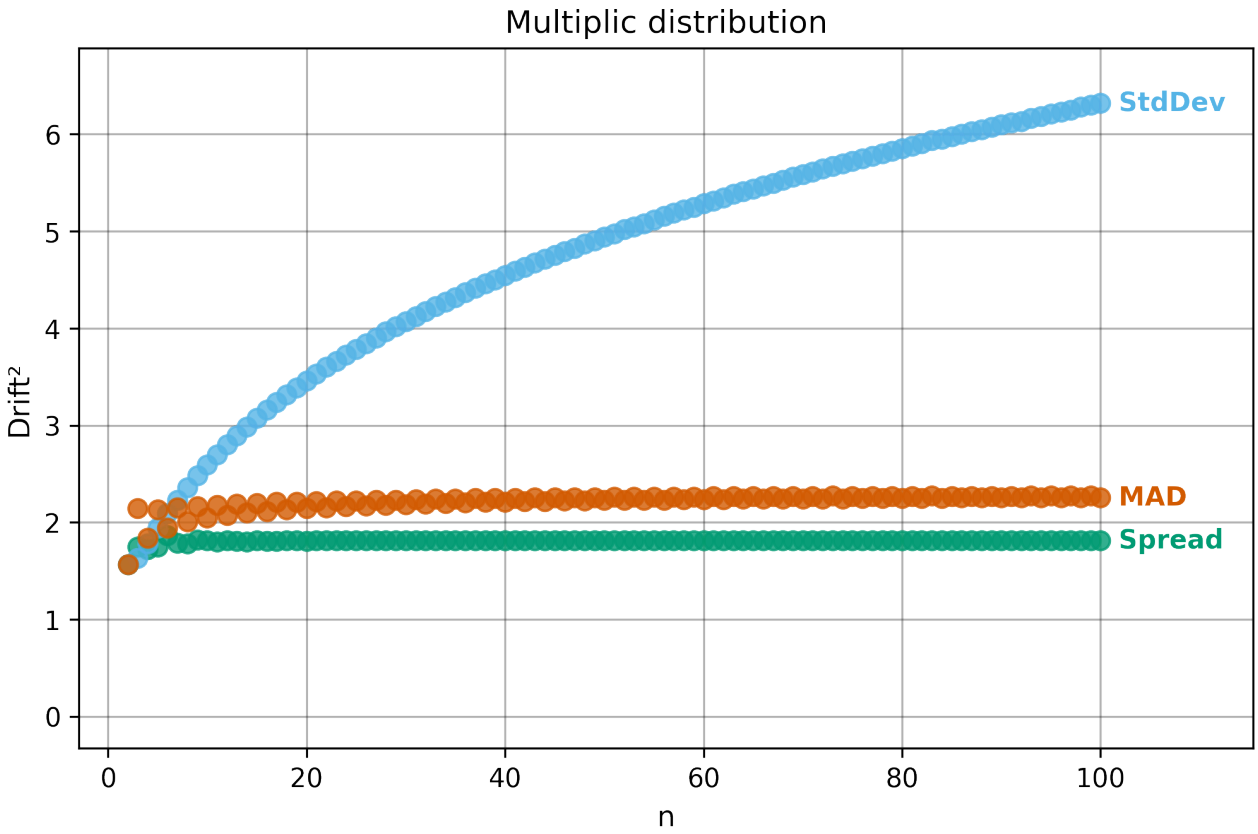
Squared Asymptotic Drift of Dispersion Estimators (values are approximated):

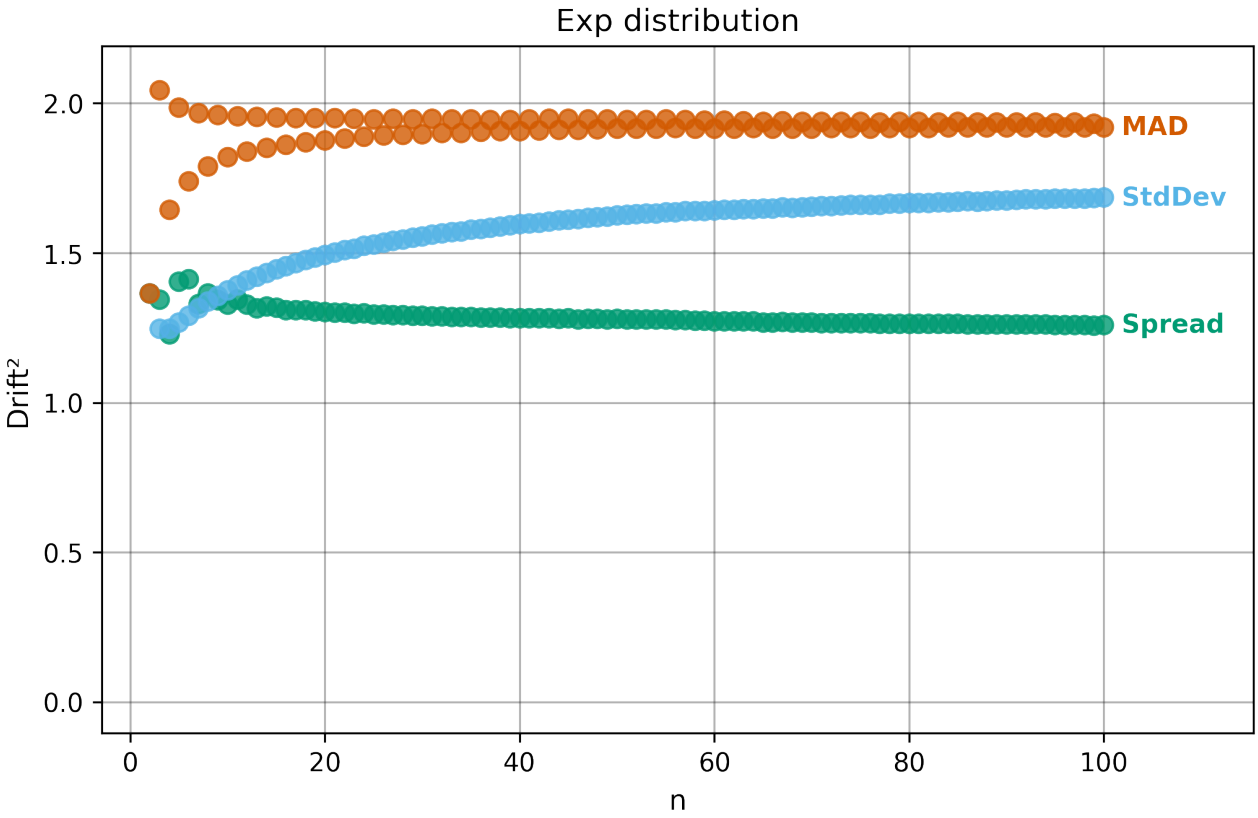
	StdDev	MAD	Spread
Additive	0.45	1.22	0.52
Multiplic	∞	2.26	1.81
Exp	1.69	1.92	1.26
Power	∞	3.5	4.4
Uniform	0.18	0.90	0.43

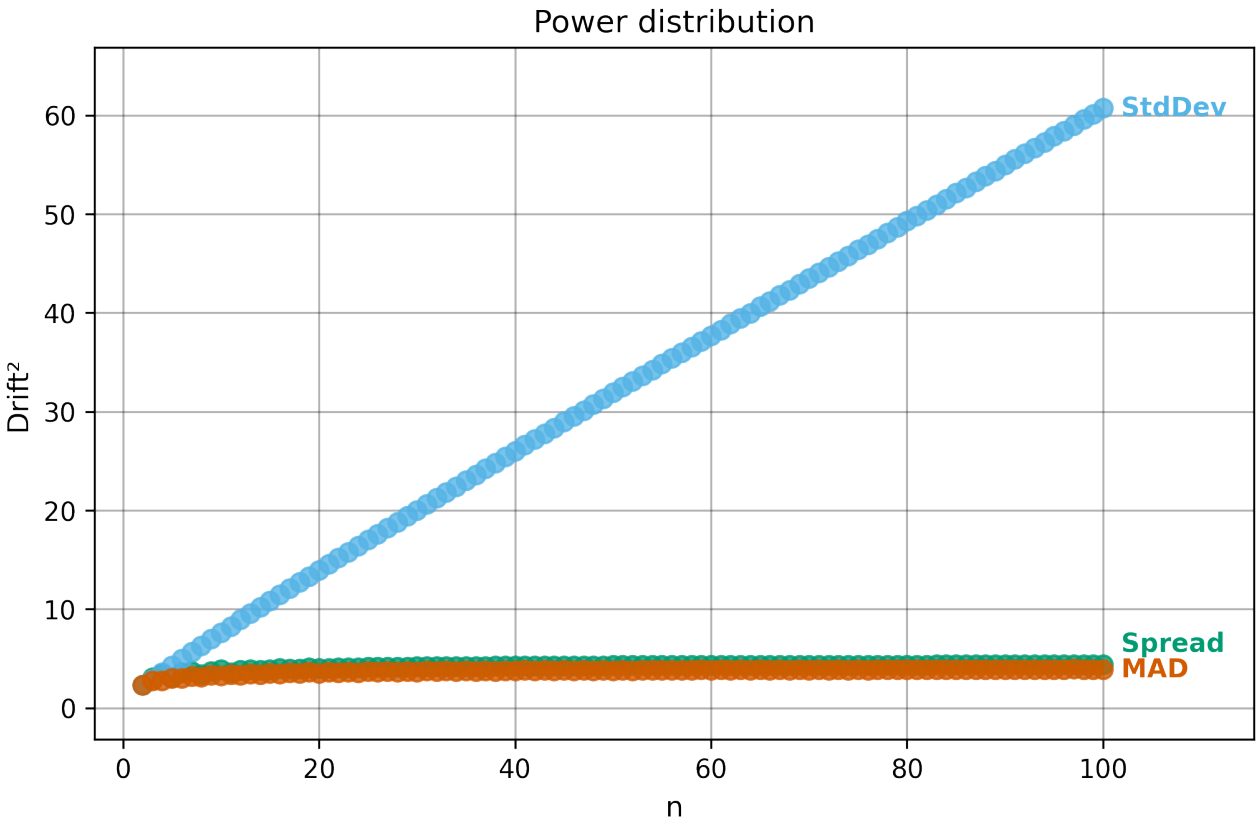
Rescaled to Spread (sample size adjustment factors):

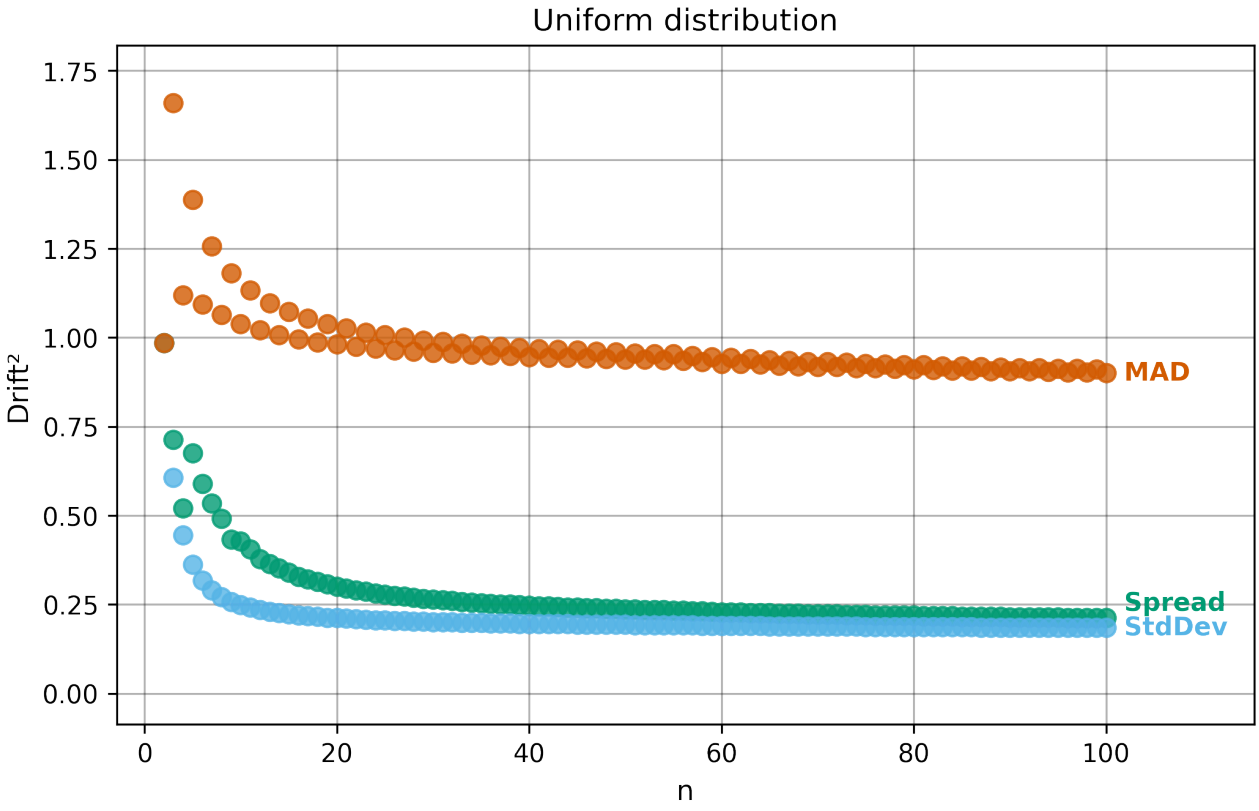
	StdDev	MAD	Spread
Additive	0.87	2.35	1.0
Multiplic	∞	1.25	1.0
Exp	1.34	1.52	1.0
Power	∞	0.80	1.0
Uniform	0.42	2.09	1.0




Multiplic distribution
StdDev
MAD
Spread
Drift²
n
0
1
2
3
4
5
6
0
20
40
60
80
100


Exp distribution
MAD
StdDev
Spread
Drift²
n
0.0
0.5
1.0
1.5
2.0
0
20
40
60
80
100


Power distribution
StdDev
Spread
MAD
Drift²
n
0
10
20
30
40
50
60
0
20
40
60
80
100


Uniform distribution
MAD
Spread
StdDev
Drift²
n
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
20
40
60
80
100

1.3.3. Tests

$$\mathrm{Spread}(\mathbf{x}) = \mathrm{median}_{1 \le i < j \le n} |x_i - x_j|$$

The Spread test suite contains 30 correctness test cases stored in the repository (20 original + 10 unsorted), plus 1 performance test that should be implemented manually (see Test Framework).

Demo examples ($n = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (0, 2, 4, 6, 8)$, expected output: 4 (base case)
`demo-2`: $\mathbf{x} = (10, 12, 14, 16, 18)$ (= demo-1 + 10), expected output: 4 (location invariance)
`demo-3`: $\mathbf{x} = (0, 4, 8, 12, 16)$ (= 2 × demo-1), expected output: 8 (scale equivariance)

Natural sequences ($n = 2, 3, 4$):

`natural-2`: $\mathbf{x} = (1, 2)$, expected output: 1
`natural-3`: $\mathbf{x} = (1, 2, 3)$, expected output: 1
`natural-4`: $\mathbf{x} = (1, 2, 3, 4)$, expected output: 1.5 (smallest even size with rich structure)

Negative values ($n = 3$) — sign handling validation:

`negative-3`: $\mathbf{x} = (-3, -2, -1)$, expected output: 1

Additive distribution ($n = 5, 10, 30$) — $\underline{\mathrm{Additive}}(10, 1)$:

`additive-5`, `additive-10`, `additive-30`: random samples generated with seed 0

Uniform distribution ($n = 5, 100$) — $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-5`, `uniform-100`: random samples generated with seed 1

The natural sequence cases validate the basic pairwise difference calculation. Constant samples and $n = 1$ are excluded because Spread requires $\mathrm{Spread}(\mathbf{x}) > 0$.

Algorithm stress tests — edge cases for fast algorithm implementation:

`duplicates-10`: $\mathbf{x} = (1, 1, 1, 2, 2, 2, 3, 3, 3, 3)$ (many duplicates, stress tie-breaking)
`parity-odd-7`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$ (odd sample size, 21 differences)
`parity-even-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6)$ (even sample size, 15 differences)
`parity-odd-49`: 49-element sequence $(1, 2, \ldots, 49)$ (large odd, 1176 differences)
`parity-even-50`: 50-element sequence $(1, 2, \ldots, 50)$ (large even, 1225 differences)

Extreme values — numerical stability and range tests:

`extreme-large-5`: $\mathbf{x} = (10^8, 2 \cdot 10^8, 3 \cdot 10^8, 4 \cdot 10^8, 5 \cdot 10^8)$ (very large values)
`extreme-small-5`: $\mathbf{x} = (10^{-8}, 2 \cdot 10^{-8}, 3 \cdot 10^{-8}, 4 \cdot 10^{-8}, 5 \cdot 10^{-8})$ (very small positive values)
`extreme-wide-5`: $\mathbf{x} = (0.001, 1, 100, 1000, 1000000)$ (wide range, tests precision)

Unsorted tests — verify sorting correctness (10 tests):

`unsorted-reverse-{n}` for $n \in \{2, 3, 4, 5, 7\}$: reverse sorted natural sequences (5 tests)

`unsorted-shuffle-3`: $\mathbf{x} = (3, 1, 2)$ (rotated)
`unsorted-shuffle-4`: $\mathbf{x} = (4, 2, 1, 3)$ (mixed order)
`unsorted-shuffle-5`: $\mathbf{x} = (5, 1, 3, 2, 4)$ (partial shuffle)
`unsorted-duplicates-unsorted-10`: $\mathbf{x} = (2, 3, 1, 3, 2, 1, 2, 3, 1, 3)$ (duplicates mixed)
`unsorted-extreme-wide-unsorted-5`: $\mathbf{x} = (1000, 0.001, 1000000, 100, 1)$ (wide range unsorted)

These tests verify that implementations correctly sort input before computing pairwise differences. Since Spread uses absolute differences, order-dependent bugs would manifest differently than in Center.

Performance test — validates the fast $O(n \log n)$ algorithm:

Input: $\mathbf{x} = (1, 2, 3, ..., 100000)$
Expected output: 29290
Time constraint: Must complete in under 5 seconds
Purpose: Ensures that the implementation uses the efficient algorithm rather than materializing all $\binom{n}{2} \approx 5$ billion pairwise differences

This test case is not stored in the repository because it generates a large JSON file (approximately 1.5 MB). Each language implementation should manually implement this test with the hardcoded expected result.

1.3.4. References

Shamos, M. I. (1976). *Geometry and Statistics: Problems at the Interface*.
Huber, P. J. (2009). Robust statistics. In *International encyclopedia of statistical science* (pp. 1248–1251). Springer.
Monahan, J. F. (1984). Algorithm 616: fast computation of the Hodges-Lehmann location estimator. *ACM Transactions on Mathematical Software*, *10*(3), 265–270. https://dl.acm.org/doi/10.1145/1271.319414

1.4. SpreadBounds

$$\text{SpreadBounds}(\mathbf{x}, \text{misrate}) = \left[d_{(k_L)}, d_{(k_U)}\right]$$

where $m = \lfloor \frac{n}{2} \rfloor$, $\mathbf{d}$ is the sorted absolute differences from a random disjoint pairing, $k_L = r + 1$, $k_U = m - r$, and r is the randomized sign-test cutoff for $\text{Binomial}\left(m, \frac{1}{2}\right)$.

Robust bounds on $\text{Spread}(\mathbf{x})$ with specified coverage.

Interpretation — misrate is probability that true spread falls outside bounds
Domain — any real numbers, $n \geq 2$, misrate $\geq 2^{1-m}$
Assumptions — `sparity(x)`
Unit — same as measurements
Note — disjoint-pair sign-test inversion; randomized cutoff matches requested misrate exactly under weak continuity; conservative with ties

Properties

Shift invariance $\text{SpreadBounds}(\mathbf{x} + c, \text{misrate}) = \text{SpreadBounds}(\mathbf{x}, \text{misrate})$
Scale equivariance $\text{SpreadBounds}(c \cdot \mathbf{x}, \text{misrate}) = |c| \cdot \text{SpreadBounds}(\mathbf{x}, \text{misrate})$
Non-negativity $\text{SpreadBounds}(\mathbf{x}, \text{misrate}) = [a, b]$ where $a \geq 0, b \geq 0$
Monotonicity in misrate smaller misrate produces wider bounds

Example

`SpreadBounds([1..30], 0.01)` where `Spread = 9`
Bounds fail to cover true spread with probability $\approx$ misrate

SpreadBounds provides distribution-free bounds on the Spread estimate. It uses disjoint pairs and an exact sign-test inversion, which guarantees coverage regardless of the underlying distribution. Set misrate to control how often the bounds might fail to contain the true spread: use 10^{-3} for everyday analysis or 10^{-6} for critical decisions. The cutoff r is clamped so that $\lfloor \frac{r}{2} \rfloor \leq \frac{m-1}{2}$, ensuring the lower and upper order statistics remain within the sorted differences.

Minimum sample size — the sign test on $m = \lfloor \frac{n}{2} \rfloor$ pairs has minimum achievable misrate 2^{1-m}:

misrate	10^{-1}	10^{-2}	10^{-3}	10^{-6}
$n_{\min}$	10	16	22	42

1.4.1. Algorithm

The $\mathrm{SpreadBounds}$ estimator constructs distribution-free bounds on $\mathrm{Spread}(\mathbf{x})$ by inverting a sign test on disjoint pairs.

Given a sample $\mathbf{x} = (x_1, \ldots, x_n)$, the algorithm proceeds as follows:

1. **Random disjoint pairing** — Randomly pair the n observations into $m = \lfloor \frac{n}{2} \rfloor$ disjoint pairs. If n is odd, one observation is discarded. The randomization ensures that the pairing does not depend on the data ordering.

2. **Absolute differences** — For each pair $\left(x_{a_i}, x_{b_i}\right)$, compute the absolute difference $d_i = \left|x_{a_i} - x_{b_i}\right|$. Under the sparity assumption, these m absolute differences are exchangeable.

3. **Sort** — Sort the differences to obtain $d_{(1)} \leq d_{(2)} \leq \ldots \leq d_{(m)}$.

4. **SignMargin cutoff** — Compute $r = \mathrm{SignMargin}(m, \mathrm{misrate})$ (see SignMargin). This determines how many extreme order statistics to exclude from each tail.

5. **Order statistic selection** — Return $\left[d_{(k_L)}, d_{(k_U)}\right]$ where $k_L = \lfloor \frac{r}{2} \rfloor + 1$ and $k_U = m - \lfloor \frac{r}{2} \rfloor$. Clamping ensures the indices remain within $[1, m]$.

The key insight is that disjoint pairs provide independence under the symmetry assumption. Under weak symmetry around the true spread, each absolute difference is equally likely to exceed or fall below the population spread. This makes the count of differences exceeding the spread a $\mathrm{Binomial}(m, 1/2)$ variable, enabling exact coverage control via the sign test.

The randomized cutoff from $\mathrm{SignMargin}$ matches the requested misrate exactly under weak continuity. With tied values, the bounds become conservative (actual coverage exceeds $1 - \mathrm{misrate}$).

```
using Pragmastat.Algorithms;
using Pragmastat.Functions;
using Pragmastat.Exceptions;
using Pragmastat.Internal;
using Pragmastat.Randomization;

namespace Pragmastat.Estimators;

/// <summary>
/// Distribution-free bounds for Spread using disjoint pairs with sign-test inversion.
/// Randomizes the cutoff between adjacent ranks to match the requested misrate.
/// Requires misrate >= 2^(1 - floor(n/2)).
/// </summary>
public class SpreadBoundsEstimator : IOneSampleBoundsEstimator
{
  public static readonly SpreadBoundsEstimator Instance = new();

  public Bounds Estimate(Sample x, Probability misrate)
  {
    return Estimate(x, misrate, null);
  }
```

```
  public Bounds Estimate(Sample x, Probability misrate, string? seed)
  {
    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
      throw AssumptionException.Domain(Subject.Misrate);

    int n = x.Size;
    int m = n / 2;
    double minMisrate = MinAchievableMisrate.OneSample(m);
    if (misrate < minMisrate)
      throw AssumptionException.Domain(Subject.Misrate);

    if (x.Size < 2)
      throw AssumptionException.Sparity(Subject.X);
    if (FastSpread.Estimate(x.SortedValues, isSorted: true) <= 0)
      throw AssumptionException.Sparity(Subject.X);

    var rng = seed == null ? new Rng() : new Rng(seed);

    int margin = SignMargin.Instance.CalcRandomized(m, misrate, rng);
    int halfMargin = margin / 2;
    int maxHalfMargin = (m - 1) / 2;
    if (halfMargin > maxHalfMargin)
      halfMargin = maxHalfMargin;

    int kLeft = halfMargin + 1;
    int kRight = m - halfMargin;

    int[] indices = Enumerable.Range(0, n).ToArray();
    var shuffled = rng.Shuffle(indices);
    var diffs = new double[m];
    for (int i = 0; i < m; i++)
    {
      int a = shuffled[2 * i];
      int b = shuffled[2 * i + 1];
      diffs[i] = Math.Abs(x.Values[a] - x.Values[b]);
    }
    Array.Sort(diffs);

    double lower = diffs[kLeft - 1];
    double upper = diffs[kRight - 1];

    return new Bounds(lower, upper, x.Unit);
  }
}
```

1.4.2. Notes

SpreadBounds targets the population spread $\text{Spread} = \text{median}|X_1 - X_2|$ with distribution-free coverage. This note records the approaches we discussed and tried, why most of them fail, and which theoretical limits force the final design. Here "distribution-free" means the stated misrate bound holds for **all** distributions of X, not just for a parametric family or under smoothness assumptions.

Setup and notation

Let $X_1, .., X_n$ be i.i.d. real-valued observations. Let $m = \lfloor \frac{n}{2} \rfloor$ and let pi be a random disjoint pairing of indices independent of the values. Define

$$d_i = \left| X_{\{\pi(2i-1)\}} - X_{\{\pi(2i)\}} \right|$$

for $i = 1..m$, and let $d_{(1)} \leq .. \leq d_{(m)}$ be their order statistics.

Let G be the distribution of $D = |X_1 - X_2|$ and let θ be any median of G. The target parameter is $\text{Spread} = \theta$.

Valid pivot and exact coverage

Assume weak continuity of G (no ties), so $P(D \leq \theta) = \frac{1}{2}$. Then the sign count S (the number of indices i with $d_i \leq \theta$) has distribution $B \sim \text{Binomial}\left(m, \frac{1}{2}\right)$. For any integer r,

$$P\left(\theta < d_{(r+1)}\right) = P(S \leq r), \wedge P\left(\theta > d_{(m-r)}\right) = P(S \leq r).$$

Therefore the interval $\left[d_{(r+1)}, d_{(m-r)}\right]$ has coverage $1 - 2P(S \leq r)$ for **all** distributions of X. This is the core distribution-free pivot used by SpreadBounds.

If G has atoms (ties), then $P(D \leq \theta) \geq \frac{1}{2}$ for any median θ. The binomial pivot becomes conservative: the same interval still covers with probability at least $1 - 2P(S \leq r)$, but exact matching of a requested misrate is impossible.

Approaches that look reasonable but fail

All pairwise differences as if independent. Construct $N = \frac{n(n-1)}{2}$ absolute differences and apply a sign-test or binomial confidence interval as if those N values were i.i.d. This ignores strong dependence between pairs, so coverage can be arbitrarily wrong. Correcting for dependence would require the joint law of all pairwise differences, which depends on the unknown distribution and is not distribution-free.

U-statistic inequalities for the median. Hoeffding-type bounds give distribution-free confidence bands for U-statistics, but for a median of pairwise differences they are extremely conservative. Intervals frequently collapse to almost $[\min, \max]$, which is unusable in practice. Asymptotic U-quantile results require smoothness and density assumptions, so they are not distribution-free.

Deterministic pairing based on order or sorting. Pairing consecutive elements in the given order is value-independent, but it is not permutation-invariant: changing the input order changes the result.

If the order is adversarial (or structured), coverage can deviate arbitrarily. Pairing after sorting (or pairing extremes) is value-dependent, so the distribution-free pivot no longer applies.

Choose the "best" pairing based on the data. Searching many pairings and picking the tightest interval is data-dependent. The selection event depends on the observed values, so coverage is not unconditional. Coverage can become arbitrarily low.

Bootstrap, asymptotic normality, or variance estimation. These are not distribution-free and can be anti-conservative for heavy tails or small samples. Studentization helps asymptotically but still depends on distributional regularity (density, smoothness, finite moments).

Mid-p or continuity-corrected binomial intervals. These reduce discreteness but do not guarantee the nominal misrate even for the binomial model itself, so the distribution-free guarantee is lost.

Deterministic disjoint pairs: why it is conservative

With disjoint pairs, the pivot is valid, but the binomial CDF is discrete. The achievable misrates form a grid

$$2 * P(B \leq r)$$

. If the requested misrate falls between grid points, a deterministic method must round down, which is conservative by design. The minimum achievable misrate is 2^{1-m}.

Deterministic "interpolation" between neighboring order statistics would use value distances instead of ranks, making coverage distribution-dependent and invalidating the distribution-free guarantee. As m grows, the grid becomes finer (step size is $O\left(\frac{1}{\sqrt{m}}\right)$), so deterministic rounding converges asymptotically, but exact matching remains impossible for any finite n.

Working approach: randomized cutoff

The final design keeps disjoint pairs but randomizes the cutoff r between adjacent grid points. Let $t = \frac{\text{misrate}}{2}$. Define $F(r) = P(B \leq r)$ and $f(r) = P(B = r)$. Let r_l be the largest integer such that $F(r_l) \leq t$ and let $r_h = r_l + 1$. Choose $r = r_h$ with probability

$$p = \frac{t - F(r_l)}{f(r_h)}$$

and $r = r_l$ otherwise. This makes the tail probability exactly t.

Randomization affects only the cutoff and is independent of data values, so the distribution-free property is preserved.

Theoretical limits

> **Distribution-free + deterministic + exact misrate is impossible** for finite n. The pivot statistic is integer-valued, so the coverage function takes only finitely many values. Exact matching requires randomization.

Exact matching for all distributions is impossible because ties break the binomial pivot. Under weak continuity (no ties), randomized cutoffs achieve exact misrate; with atoms, coverage is only guaranteed to be conservative.

Deterministic exact misrate is possible only with extra assumptions (parametric family, smooth density at the target, known variance), which violates distribution-free guarantees.

Any method that treats all pairwise differences as independent is invalid because dependence destroys the binomial pivot.

Any data-dependent choice of pairing or cutoff breaks unconditional coverage and can make the method arbitrarily anti-conservative.

1.4.3. Tests

$$\text{SpreadBounds}(\mathbf{x}, \text{misrate}) = [L, U]$$

Let $m = \lfloor \frac{n}{2} \rfloor$. Draw a value-independent random disjoint pairing of the sample, compute $\mathbf{d} = \left\{ \left| x_{\{\pi(2i-1)\}} - x_{\{\pi(2i)\}} \right| \right\}$ for $i = 1..m$, and sort ascending.

Let r be the largest integer such that $\sum_{i=0}^{r} \frac{\binom{m}{i}}{2^m} \leq \frac{\text{misrate}}{2}$. If the target lies between two adjacent CDF steps, r is randomized between r and $r+1$ to match the requested misrate. Define $k_L = r + 1$ and $k_U = m - r$.

Return $[L, U] = \left[d_{(k_L)}, d_{(k_U)} \right]$.

The SpreadBounds test suite contains 46 test cases (3 demo + 4 natural + 4 property + 7 edge + 3 additive + 2 uniform + 5 misrate + 5 conservatism + 8 unsorted + 5 error cases). Since SpreadBounds returns bounds rather than a point estimate, tests validate that bounds are well-formed and satisfy equivariance properties under a fixed seed. Each test case output is a JSON object with `lower` and `upper` fields representing the interval bounds. Because pairing and cutoff selection are randomized, tests fix `seed` to keep outputs deterministic.

Demo examples — from manual introduction, validating basic bounds:

`demo-1`: $\mathbf{x} = (1, 2, ..., 30)$, $\text{misrate} = 0.01$, bounds containing $\text{Spread} = 9$
`demo-2`: $\mathbf{x} = (1, 2, ..., 30)$, $\text{misrate} = 0.002$, wider bounds (tighter misrate)
`demo-3`: $\mathbf{x} = (1, 2, ..., 15)$, $\text{misrate} = 0.07$

These cases illustrate how tighter misrates produce wider bounds and how sample size affects bound width.

Natural sequences (misrate varies by size) — 4 tests:

`natural-10`: $\mathbf{x} = (1, 2, ..., 10)$, $\text{misrate} = 0.15$
`natural-15`: $\mathbf{x} = (1, 2, ..., 15)$, $\text{misrate} = 0.05$
`natural-20`: $\mathbf{x} = (1, 2, ..., 20)$, $\text{misrate} = 0.05$
`natural-30`: $\mathbf{x} = (1, 2, ..., 30)$, $\text{misrate} = 0.05$

Property validation ($n = 10$, $\text{misrate} = 0.2$) — 4 tests:

`property-identity`: $\mathbf{x} = (1, 2, ..., 10)$, bounds must contain Spread
`property-location-shift`: $\mathbf{x} = (11, 12, ..., 20)$ (= identity + 10), bounds must equal identity bounds (shift invariance)
`property-scale-2x`: $\mathbf{x} = (2, 4, ..., 20)$ (= 2 $\times$ identity), bounds must be $2\times$ identity bounds (scale equivariance)
`property-scale-neg`: $\mathbf{x} = (-10, -9, ..., -1)$ (= $-1 \times$ identity), bounds must equal identity bounds ($|k|$ scaling)

Edge cases — boundary conditions and extreme scenarios (7 tests):

`edge-small-non-trivial`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\text{misrate} = 0.8$ (small but non-trivial bounds)

`edge-large-misrate`: $\mathbf{x} = (1, 2, ..., 10)$, $\text{misrate} = 0.5$ (permissive bounds)
`edge-duplicates-mixed`: $\mathbf{x} = (1, 1, 1, 2, 3, 4, 5)$, $\text{misrate} = 0.5$ (partial ties)
`edge-wide-range`: $\mathbf{x} = (1, 10, 100, 1000, 10000)$, $\text{misrate} = 0.8$ (extreme value range)
`edge-negative`: $\mathbf{x} = (-5, -4, -3, -2, -1)$, $\text{misrate} = 0.8$ (negative values)
`edge-large-n`: $\mathbf{x} = (1, 2, ..., 100)$, $\text{misrate} = 0.01$ (large sample, tighter sign-test bounds)
`edge-n2`: $\mathbf{x} = (1, 3)$, $\text{misrate} = 1.0$ (minimum sample size, only valid misrate is 1.0)

Additive distribution (misrate varies by size) — 3 tests with $\underline{\text{Additive}}(10, 1)$:

`additive-20`: $n = 20$, $\text{misrate} = 0.02$
`additive-30`: $n = 30$, $\text{misrate} = 0.01$
`additive-50`: $n = 50$, $\text{misrate} = 0.01$

Uniform distribution (misrate varies by size) — 2 tests with $\underline{\text{Uniform}}(0, 1)$:

`uniform-20`: $n = 20$, $\text{misrate} = 0.02$
`uniform-50`: $n = 50$, $\text{misrate} = 0.01$

Misrate variation ($\mathbf{x} = (1, 2, ..., 25)$) — 5 tests with varying misrates:

`misrate-5e-1`: $\text{misrate} = 0.5$
`misrate-1e-1`: $\text{misrate} = 0.1$
`misrate-5e-2`: $\text{misrate} = 0.05$
`misrate-1e-2`: $\text{misrate} = 0.01$
`misrate-2e-3`: $\text{misrate} = 0.002$

These tests validate monotonicity: smaller misrates produce wider bounds.

Conservatism tests ($\text{misrate} = 0.1$) — 5 tests unique to SpreadBounds:

`conservatism-12`: $\mathbf{x} = (1, 2, ..., 12)$, sign-test bounds are wide relative to Spread
`conservatism-15`: $\mathbf{x} = (1, 2, ..., 15)$
`conservatism-20`: $\mathbf{x} = (1, 2, ..., 20)$
`conservatism-30`: $\mathbf{x} = (1, 2, ..., 30)$
`conservatism-50`: $\mathbf{x} = (1, 2, ..., 50)$

These tests document how discreteness-driven conservatism decreases with increasing sample size. For small n, bounds may span a large part of the pairwise-difference range. For large n, bounds tighten to a practical interval around Spread.

Unsorted tests — verify stable behavior on non-sorted inputs (8 tests):

`unsorted-reverse-10`: $\mathbf{x} = (10, 9, ..., 1)$, $\text{misrate} = 0.2$
`unsorted-reverse-15`: $\mathbf{x} = (15, 14, ..., 1)$, $\text{misrate} = 0.07$
`unsorted-shuffle-10`: $\mathbf{x}$ shuffled, $\text{misrate} = 0.2$
`unsorted-shuffle-15`: $\mathbf{x}$ shuffled, $\text{misrate} = 0.07$
`unsorted-negative-5`: negative values unsorted ($\text{misrate} = 0.8$)
`unsorted-mixed-signs-5`: mixed signs unsorted ($\text{misrate} = 0.8$)
`unsorted-duplicates`: $\mathbf{x} = (1, 3, 1, 3, 2)$, unsorted with duplicates ($\text{misrate} = 0.8$)

`unsorted-wide-range`: $\mathbf{x} = (1000, 1, 100, 10, 10000)$, unsorted wide range ($\text{misrate} = 0.8$)

These tests validate that SpreadBounds produces sensible bounds for arbitrary input order under a fixed seed.

Error cases — inputs that violate assumptions (5 tests):

`error-empty-array`: $\mathbf{x} = ()$, $\text{misrate} = 0.5$ — empty array violates validity
`error-single-element`: $\mathbf{x} = (1)$, $\text{misrate} = 0.5$ — $m = 0$ violates domain (n too small to form pairs)
`error-misrate-zero`: $\mathbf{x} = (1, 2, ..., 10)$, $\text{misrate} = 0$ — below minimum achievable misrate
`error-invalid-misrate`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\text{misrate} = 0.001$ — below minimum achievable misrate ($2^{1-2} = 0.5$)
`error-constant-sample`: $\mathbf{x} = (1, 1, 1, 1, 1)$, $\text{misrate} = 0.5$ — constant sample violates sparity ($\text{Spread} = 0$)

Note: SpreadBounds has a minimum misrate constraint. The sign-test inversion requires $\text{misrate} \geq 2^{1-m}$.

2. Two-Sample Estimators

2.1. Shift

$$\operatorname{Shift}(\mathbf{x}, \mathbf{y}) = \operatorname{Median}_{1 \le i \le n, 1 \le j \le m}(x_i - y_j)$$

Robust measure of location difference between two samples.

Also known as — Hodges-Lehmann estimator for two samples
Asymptotic — median of the difference between random measurements from X and Y
Complexity — $O((m+n)\log L)$
Domain — any real numbers
Unit — same as measurements

Properties

Self-difference $\operatorname{Shift}(\mathbf{x}, \mathbf{x}) = 0$
Shift equivariance $\operatorname{Shift}(\mathbf{x} + k_x, \mathbf{y} + k_y) = \operatorname{Shift}(\mathbf{x}, \mathbf{y}) + k_x - k_y$
Scale equivariance $\operatorname{Shift}(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = k \cdot \operatorname{Shift}(\mathbf{x}, \mathbf{y})$
Antisymmetry $\operatorname{Shift}(\mathbf{x}, \mathbf{y}) = -\operatorname{Shift}(\mathbf{y}, \mathbf{x})$

Example

```
Shift([0, 2, 4, 6, 8], [10, 12, 14, 16, 18]) = -10
Shift(y, x) = -Shift(x, y)
```

Shift measures how much one group differs from another. When comparing response times between version A and version B, Shift tells by how many milliseconds A is faster or slower than B. A negative result means the first group tends to be lower; positive means it tends to be higher. Unlike comparing means, Shift handles outliers gracefully and works well with skewed data. The result comes in the same units as your measurements, making it easy to interpret.

2.1.1. Algorithm

The Shift estimator measures the median of all pairwise differences between elements of two samples. Given samples $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_m)$, this estimator is defined as:

$$\mathrm{Shift}(\mathbf{x}, \mathbf{y}) = \mathrm{median}_{1 \le i \le n, 1 \le j \le m}(x_i - y_j)$$

This definition represents a special case of a more general problem: computing arbitrary quantiles of all pairwise differences. For samples of size n and m, the total number of pairwise differences is $n \times m$. A naive approach would materialize all differences, sort them, and extract the desired quantile. With $n = m = 10000$, this approach creates 100 million values, requiring quadratic memory and $O(nm \log(nm))$ time.

The presented algorithm avoids materializing pairwise differences by exploiting the sorted structure of the samples. After sorting both samples to obtain $x_1 \le x_2 \le ... \le x_n$ and $y_1 \le y_2 \le ... \le y_m$, the key insight is that it's possible to count how many pairwise differences fall below any threshold without computing them explicitly. This counting operation enables a binary search over the continuous space of possible difference values, iteratively narrowing the search range until it converges to the exact quantile.

The algorithm operates through a value-space search rather than index-space selection. It maintains a search interval $[\mathrm{searchMin}, \mathrm{searchMax}]$, initialized to the range of all possible differences: $[x_1 - y_m, x_n - y_1]$. At each iteration, it selects a candidate value within this interval and counts how many pairwise differences are less than or equal to this threshold. For the median (quantile $p = 0.5$), if fewer than half the differences lie below the threshold, the median must be larger; if more than half lie below, the median must be smaller. Based on this comparison, the search space is reduced by eliminating portions that cannot contain the target quantile.

The counting operation achieves linear complexity via a two-pointer sweep. For a given threshold t, the number of pairs (i, j) satisfying $x_i - y_j \le t$ is counted. This is equivalent to counting pairs where $y_j \ge x_i - t$. For each row i in the implicit matrix of differences, a column pointer advances through the sorted y array while $x_i - y_j > t$, stopping at the first position where $x_i - y_j \le t$. All subsequent positions in that row satisfy the condition, contributing $(m - j)$ pairs to the count for row i. Because both samples are sorted, the column pointer advances monotonically and never backtracks across rows, making each counting pass $O(n + m)$ regardless of the total number of differences.

During each counting pass, the algorithm tracks boundary values: the largest difference at or below the threshold and the smallest difference above it. When the count exactly matches the target rank (or the two middle ranks for even-length samples), these boundary values provide the exact answer without additional searches. For Type-7 quantile computation (Hyndman & Fan (1996)), which interpolates between order statistics, the algorithm collects the necessary boundary values in a single pass and performs linear interpolation: $(1 - w) \cdot \mathrm{lower} + w \cdot \mathrm{upper}$.

Real datasets often contain discrete or repeated values that can cause search stagnation. The algorithm detects when the search interval stops shrinking between iterations, indicating that multiple pairwise differences share the same value. When the closest difference below the threshold equals the closest above, all remaining candidates are identical and the algorithm terminates immediately. Otherwise, it

uses the boundary values from the counting pass to snap the search interval to actual difference values, ensuring reliable convergence even with highly discrete data.

The binary search employs numerically stable midpoint calculations and terminates when the search interval collapses to a single value or when boundary tracking confirms convergence. Iteration limits are included as a safety mechanism, though convergence typically occurs much earlier due to the exponential narrowing of the search space.

The algorithm naturally generalizes to multiple quantiles by computing each one independently. For k quantiles with samples of size n and m, the total complexity is $O(k(n+m)\log L)$, where L represents the convergence precision. This is dramatically more efficient than the naive $O(nm\log(nm))$ approach, especially for large n and m with small k. The algorithm requires only $O(1)$ additional space beyond the input arrays, making it practical for large-scale statistical analysis where memory constraints prohibit materializing quadratic data structures.

```
namespace Pragmastat.Algorithms;

using System;
using System.Collections.Generic;
using System.Linq;

public static class FastShift
{
  /// <summary>
  /// Computes quantiles of all pairwise differences { x_i - y_j }.
  /// Time: O((m + n) * log(precision)) per quantile. Space: O(1).
  /// </summary>
  /// <param name="p">Probabilities in [0, 1].</param>
  /// <param name="assumeSorted">If false, collections will be sorted.</param>
  public static double[] Estimate(IReadOnlyList<double> x, IReadOnlyList<double> y,
double[] p, bool assumeSorted = false)
  {
    if (x == null || y == null || p == null)
      throw new ArgumentNullException();
    if (x.Count == 0 || y.Count == 0)
      throw new ArgumentException("x and y must be non-empty.");

    foreach (double pk in p)
      if (double.IsNaN(pk) || pk < 0.0 || pk > 1.0)
        throw new ArgumentOutOfRangeException(nameof(p), "Probabilities must be within [0,
1].");

    double[] xs, ys;
    if (assumeSorted)
    {
      xs = x as double[] ?? x.ToArray();
      ys = y as double[] ?? y.ToArray();
    }
    else
    {
      xs = x.OrderBy(v => v).ToArray();
```

```
      ys = y.OrderBy(v => v).ToArray();
    }

    int m = xs.Length;
    int n = ys.Length;
    long total = (long)m * n;

    // Type-7 quantile: h = 1 + (n-1)*p, then interpolate between floor(h) and ceil(h)
    var requiredRanks = new SortedSet<long>();
    var interpolationParams = new (long lowerRank, long upperRank, double weight)
[p.Length];

    for (int i = 0; i < p.Length; i++)
    {
      double h = 1.0 + (total - 1) * p[i];
      long lowerRank = (long)Math.Floor(h);
      long upperRank = (long)Math.Ceiling(h);
      double weight = h - lowerRank;
      if (lowerRank < 1) lowerRank = 1;
      if (upperRank > total) upperRank = total;
      interpolationParams[i] = (lowerRank, upperRank, weight);
      requiredRanks.Add(lowerRank);
      requiredRanks.Add(upperRank);
    }

    var rankValues = new Dictionary<long, double>();
    foreach (long rank in requiredRanks)
      rankValues[rank] = SelectKthPairwiseDiff(xs, ys, rank);

    var result = new double[p.Length];
    for (int i = 0; i < p.Length; i++)
    {
      var (lowerRank, upperRank, weight) = interpolationParams[i];
      double lower = rankValues[lowerRank];
      double upper = rankValues[upperRank];
      result[i] = weight == 0.0 ? lower : (1.0 - weight) * lower + weight * upper;
    }

    return result;
  }

  // Binary search in [min_diff, max_diff] that snaps to actual discrete values.
  // Avoids materializing all m*n differences.
  private static double SelectKthPairwiseDiff(double[] x, double[] y, long k)
  {
    int m = x.Length;
    int n = y.Length;
    long total = (long)m * n;

    if (k < 1 || k > total)
      throw new ArgumentOutOfRangeException(nameof(k));

    double searchMin = x[0] - y[n - 1];
```

```
    double searchMax = x[m - 1] - y[0];

    if (double.IsNaN(searchMin) || double.IsNaN(searchMax))
      throw new InvalidOperationException("NaN in input values.");

    const int maxIterations = 128; // Sufficient for double precision convergence
    double prevMin = double.NegativeInfinity;
    double prevMax = double.PositiveInfinity;

    for (int iter = 0; iter < maxIterations && searchMin != searchMax; iter++)
    {
      double mid = Midpoint(searchMin, searchMax);
      CountAndNeighbors(x, y, mid, out long countLessOrEqual, out double closestBelow, out
double closestAbove);

      if (closestBelow == closestAbove)
        return closestBelow;

      // No progress means we're stuck between two discrete values
      if (searchMin == prevMin && searchMax == prevMax)
        return countLessOrEqual >= k ? closestBelow : closestAbove;

      prevMin = searchMin;
      prevMax = searchMax;

      if (countLessOrEqual >= k)
        searchMax = closestBelow;
      else
        searchMin = closestAbove;
    }

    if (searchMin != searchMax)
      throw new InvalidOperationException("Convergence failure (pathological input).");

    return searchMin;
  }

  // Two-pointer algorithm: counts pairs where x[i] - y[j] <= threshold, and tracks
  // the closest actual differences on either side of threshold.
  private static void CountAndNeighbors(
    double[] x, double[] y, double threshold,
    out long countLessOrEqual, out double closestBelow, out double closestAbove)
  {
    int m = x.Length, n = y.Length;
    long count = 0;
    double maxBelow = double.NegativeInfinity;
    double minAbove = double.PositiveInfinity;

    int j = 0;
    for (int i = 0; i < m; i++)
    {
      while (j < n && x[i] - y[j] > threshold)
        j++;
```

```
      count += (n - j);

      if (j < n)
      {
        double diff = x[i] - y[j];
        if (diff > maxBelow) maxBelow = diff;
      }

      if (j > 0)
      {
        double diff = x[i] - y[j - 1];
        if (diff < minAbove) minAbove = diff;
      }
    }

    // Fallback to actual min/max if no boundaries found (shouldn't happen in normal
operation)
    if (double.IsNegativeInfinity(maxBelow))
      maxBelow = x[0] - y[n - 1];
    if (double.IsPositiveInfinity(minAbove))
      minAbove = x[m - 1] - y[0];

    countLessOrEqual = count;
    closestBelow = maxBelow;
    closestAbove = minAbove;
  }

  private static double Midpoint(double a, double b) => a + (b - a) * 0.5;
}
```

2.1.2. Tests

$$\mathrm{Shift}(\mathbf{x}, \mathbf{y}) = \mathrm{median}_{1 \le i \le n, 1 \le j \le m}(x_i - y_j)$$

The Shift test suite contains 60 correctness test cases stored in the repository (42 original + 18 unsorted), plus 1 performance test that should be implemented manually (see Test Framework).

Demo examples ($n = m = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\mathbf{y} = (10, 12, 14, 16, 18)$, expected output: -10 (base case)
`demo-2`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\mathbf{y} = (0, 2, 4, 6, 8)$, expected output: 0 (identity property)
`demo-3`: $\mathbf{x} = (7, 9, 11, 13, 15)$, $\mathbf{y} = (13, 15, 17, 19, 21)$ (= demo-1 + [7,3]), expected output: -6 (location equivariance)
`demo-4`: $\mathbf{x} = (0, 4, 8, 12, 16)$, $\mathbf{y} = (20, 24, 28, 32, 36)$ (= 2 × demo-1), expected output: -20 (scale equivariance)
`demo-5`: $\mathbf{x} = (10, 12, 14, 16, 18)$, $\mathbf{y} = (0, 2, 4, 6, 8)$ (= reversed demo-1), expected output: 10 (anti-symmetry)

Natural sequences ($[n, m] \in \{1, 2, 3\} \times \{1, 2, 3\}$) — 9 combinations:

`natural-1-1`: $\mathbf{x} = (1)$, $\mathbf{y} = (1)$, expected output: 0
`natural-1-2`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2)$, expected output: -0.5
`natural-1-3`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2, 3)$, expected output: -1
`natural-2-1`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1)$, expected output: 0.5
`natural-2-2`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2)$, expected output: 0
`natural-2-3`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2, 3)$, expected output: -0.5
`natural-3-1`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1)$, expected output: 1
`natural-3-2`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2)$, expected output: 0.5
`natural-3-3`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2, 3)$, expected output: 0

Negative values ($[n, m] = [2, 2]$) — sign handling validation:

`negative-2-2`: $\mathbf{x} = (-2, -1)$, $\mathbf{y} = (-2, -1)$, expected output: 0

Mixed-sign values ($[n, m] = [2, 2]$) — validates anti-symmetry across zero:

`mixed-2-2`: $\mathbf{x} = (-1, 1)$, $\mathbf{y} = (-1, 1)$, expected output: 0

Zero values ($[n, m] \in \{1, 2\} \times \{1, 2\}$) — 4 combinations:

`zeros-1-1`, `zeros-1-2`, `zeros-2-1`, `zeros-2-2`: all produce output 0

Additive distribution ($[n, m] \in \{5, 10, 30\} \times \{5, 10, 30\}$) — 9 combinations with $\underline{\mathrm{Additive}}(10, 1)$:

`additive-5-5`, `additive-5-10`, `additive-5-30`
`additive-10-5`, `additive-10-10`, `additive-10-30`
`additive-30-5`, `additive-30-10`, `additive-30-30`
Random generation: $\mathbf{x}$ uses seed 0, $\mathbf{y}$ uses seed 1

Uniform distribution ($[n, m] \in \{5, 100\} \times \{5, 100\}$) — 4 combinations with $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-5-5`, `uniform-5-100`, `uniform-100-5`, `uniform-100-100`
Random generation: $\mathbf{x}$ uses seed 2, $\mathbf{y}$ uses seed 3

The natural sequences validate anti-symmetry ($\text{Shift}(\mathbf{x}, \mathbf{y}) = -\text{Shift}(\mathbf{y}, \mathbf{x})$) and the identity property ($\text{Shift}(\mathbf{x}, \mathbf{x}) = 0$). The asymmetric size combinations test the two-sample algorithm with unbalanced inputs.

Algorithm stress tests — edge cases for fast binary search algorithm:

`duplicates-5-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (3, 3, 3, 3, 3)$ (all identical, expected output: 0)
`duplicates-10-10`: $\mathbf{x} = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5)$, $\mathbf{y} = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5)$ (many duplicates)
`parity-odd-7-7`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6, 7)$ (odd sizes, 49 differences, expected output: 0)
`parity-even-6-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6)$ (even sizes, 36 differences, expected output: 0)
`parity-asymmetric-7-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6)$ (mixed parity, 42 differences)
`parity-large-49-50`: $\mathbf{x} = (1, 2, ..., 49)$, $\mathbf{y} = (1, 2, ..., 50)$ (large asymmetric, 2450 differences)

Extreme asymmetry — tests with very unbalanced sample sizes:

`asymmetry-1-100`: $\mathbf{x} = (50)$, $\mathbf{y} = (1, 2, ..., 100)$ (single vs many, 100 differences)
`asymmetry-2-50`: $\mathbf{x} = (10, 20)$, $\mathbf{y} = (1, 2, ..., 50)$ (tiny vs medium, 100 differences)
`asymmetry-constant-varied`: $\mathbf{x} = (5, 5, 5, 5, 5)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ (constant vs varied)

Unsorted tests — verify independent sorting of each sample (18 tests):

`unsorted-x-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4), (5, 5)\}$: X unsorted (reversed), Y sorted (3 tests)
`unsorted-y-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4), (5, 5)\}$: X sorted, Y unsorted (reversed) (3 tests)
`unsorted-both-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4), (5, 5)\}$: both unsorted (reversed) (3 tests)
`unsorted-reverse-3-3`: $\mathbf{x} = (3, 2, 1)$, $\mathbf{y} = (3, 2, 1)$ (both reversed)
`unsorted-x-shuffle-3-3`: $\mathbf{x} = (2, 1, 3)$, $\mathbf{y} = (1, 2, 3)$ (X shuffled, Y sorted)
`unsorted-y-shuffle-3-3`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (3, 1, 2)$ (X sorted, Y shuffled)
`unsorted-both-shuffle-4-4`: $\mathbf{x} = (3, 1, 4, 2)$, $\mathbf{y} = (4, 2, 1, 3)$ (both shuffled)
`unsorted-duplicates-mixed-5-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (3, 3, 3, 3, 3)$ (all identical)
`unsorted-x-unsorted-duplicates`: $\mathbf{x} = (2, 1, 3, 2, 1)$, $\mathbf{y} = (1, 1, 2, 2, 3)$ (X has unsorted duplicates)
`unsorted-y-unsorted-duplicates`: $\mathbf{x} = (1, 1, 2, 2, 3)$, $\mathbf{y} = (3, 2, 1, 3, 2)$ (Y has unsorted duplicates)
`unsorted-asymmetric-unsorted-2-5`: $\mathbf{x} = (2, 1)$, $\mathbf{y} = (5, 2, 4, 1, 3)$ (asymmetric sizes, both unsorted)
`unsorted-negative-unsorted-3-3`: $\mathbf{x} = (-1, -3, -2)$, $\mathbf{y} = (-2, -3, -1)$ (negative unsorted)

These tests are critical for two-sample estimators because they verify that $\mathbf{x}$ and $\mathbf{y}$ are sorted **independently**. The variety includes cases where only one sample is unsorted, ensuring implementations don't incorrectly assume pre-sorted input or sort samples together.

Performance test — validates the fast $O((m + n) \log L)$ binary search algorithm:

Input: $\mathbf{x} = (1, 2, 3, ..., 100000)$, $\mathbf{y} = (1, 2, 3, ..., 100000)$
Expected output: 0
Time constraint: Must complete in under 5 seconds
Purpose: Ensures that the implementation uses the efficient algorithm rather than materializing all $mn = 10$ billion pairwise differences

This test case is not stored in the repository because it generates a large JSON file (approximately 1.5 MB). Each language implementation should manually implement this test with the hardcoded expected result.

2.1.3. References

Hodges, J. L., & Lehmann, E. L. (1963). Estimates of Location Based on Rank Tests. *The Annals of Mathematical Statistics*, *34*(2), 598–611. http://projecteuclid.org/euclid.aoms/1177704172
Sidak, Z., Sen, P. K., & Hajek, J. (1999). *Theory of rank tests*. Elsevier.
Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. *The American Statistician*, *50*(4), 361. https://www.jstor.org/stable/2684934

2.2. ShiftBounds

$$\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \left[z_{(k_{\text{left}})}, z_{(k_{\text{right}})}\right]$$

where $\mathbf{z} = \{x_i - y_j\}$ (sorted), $k_{\text{left}} = \left\lfloor \frac{\text{PairwiseMargin}}{2} \right\rfloor + 1$, $k_{\text{right}} = nm - \left\lfloor \frac{\text{PairwiseMargin}}{2} \right\rfloor$

Robust bounds on $\text{Shift}(\mathbf{x}, \mathbf{y})$ with specified coverage.

Also known as — distribution-free confidence interval for Hodges-Lehmann
Interpretation — misrate is probability that true shift falls outside bounds
Domain — any real numbers, $\text{misrate} \geq \frac{2}{\binom{n+m}{n}}$
Unit — same as measurements
Note — assumes weak continuity (ties from measurement resolution are tolerated but may yield conservative bounds)

Properties

Shift invariance $\text{ShiftBounds}(\mathbf{x} + k, \mathbf{y} + k, \text{misrate}) = \text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Scale equivariance $\text{ShiftBounds}(k \cdot \mathbf{x}, k \cdot \mathbf{y}, \text{misrate}) = k \cdot \text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$

Example

`ShiftBounds([1..30], [21..50], 1e-4) = [-30, -10]` where `Shift = -20`
Bounds fail to cover true shift with probability $\approx$ misrate

ShiftBounds provides not just the estimated shift but also the uncertainty of that estimate. The function returns an interval of plausible shift values given the data. Set misrate to control how often the bounds might fail to contain the true shift: use 10^{-3} for everyday analysis or 10^{-6} for critical decisions where errors are costly. These bounds require no assumptions about your data distribution, so they remain valid for any continuous measurements. If the bounds exclude zero, that suggests a reliable difference between the two groups.

2.2.1. Algorithm

The $\mathrm{ShiftBounds}$ estimator constructs distribution-free bounds on $\mathrm{Shift}(\mathbf{x}, \mathbf{y})$ by selecting specific order statistics from the pairwise differences.

Given samples $\mathbf{x} = (x_1, ..., x_n)$ and $\mathbf{y} = (y_1, ..., y_m)$, the algorithm proceeds as follows:

1. **Compute the margin** — Call $\mathrm{PairwiseMargin}(n, m, \mathrm{misrate})$ (see PairwiseMargin) to determine how many extreme pairwise differences to exclude from each tail.

2. **Determine quantile ranks** — From the margin M, compute $k_{\text{left}} = \left\lfloor \frac{M}{2} \right\rfloor + 1$ and $k_{\text{right}} = nm - \left\lfloor \frac{M}{2} \right\rfloor$. These are the ranks of the order statistics that form the bounds.

3. **Compute quantiles via Shift** — Use the Shift algorithm to compute the k_{left}-th and k_{right}-th order statistics of all nm pairwise differences $x_i - y_j$. The Shift algorithm's value-space binary search finds these quantiles in $O((n + m) \log L)$ time without materializing all differences.

4. **Return bounds** — Return $\left[z_{(k_{\text{left}})}, z_{(k_{\text{right}})}\right]$.

The $\mathrm{PairwiseMargin}$ function encodes the statistical theory: it determines which order statistics provide bounds with coverage $1 - \mathrm{misrate}$. The Shift algorithm provides the computational machinery: it extracts those specific order statistics efficiently from the implicit matrix of pairwise differences.

2.2.2. Tests

$$\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \left[z_{(k_{\text{left}})}, z_{(k_{\text{right}})}\right]$$

where

$$\mathbf{z} = \{x_i - y_j\}_{1 \le i \le n, 1 \le j \le m} \quad \text{(sorted)}$$

$$k_{\text{left}} = \left\lfloor \frac{\text{PairwiseMargin}(n, m, \text{misrate})}{2} \right\rfloor + 1$$

$$k_{\text{right}} = nm - \left\lfloor \frac{\text{PairwiseMargin}(n, m, \text{misrate})}{2} \right\rfloor$$

The ShiftBounds test suite contains 61 correctness test cases (3 demo + 9 natural + 6 property + 10 edge + 9 additive + 4 uniform + 5 misrate + 15 unsorted). Since ShiftBounds returns bounds rather than a point estimate, tests validate that the bounds contain $\text{Shift}(\mathbf{x}, \mathbf{y})$ and satisfy equivariance properties. Each test case output is a JSON object with `lower` and `upper` fields representing the interval bounds. The domain constraint $\text{misrate} \ge \frac{2}{\binom{n+m}{n}}$ is enforced; inputs violating this return a domain error.

Demo examples ($n = m = 5$) — from manual introduction, validating basic bounds:

`demo-1`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, $\text{misrate} = 0.05$, expected output: $[-4, 0]$
`demo-2`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, $\text{misrate} = 0.01$, expected output: $[-5, 1]$
`demo-3`: $\mathbf{x} = (3, 4, 5, 6, 7)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, $\text{misrate} = 0.05$, expected output: bounds containing 0 (identity case)

These cases illustrate how tighter misrates produce wider bounds and validate the identity property where identical samples yield bounds containing zero.

Natural sequences ($[n, m] \in \{5, 8, 10\} \times \{5, 8, 10\}$, $\text{misrate} = 10^{-2}$) — 9 combinations:

`natural-5-5`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 5)$, expected bounds containing 0
`natural-5-8`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 8)$
`natural-5-10`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 10)$
`natural-8-5`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 5)$
`natural-8-8`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 8)$, expected bounds containing 0
`natural-8-10`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 10)$
`natural-10-5`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 5)$
`natural-10-8`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 8)$
`natural-10-10`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 10)$, expected bounds containing 0

These sizes are chosen to satisfy $\text{misrate} \ge \frac{2}{\binom{n+m}{n}}$ for all combinations.

Property validation ($n = m = 10$, $\text{misrate} = 10^{-3}$) — 6 tests:

`property-identity`: $\mathbf{x} = (0, 2, 4, ..., 18)$, $\mathbf{y} = (0, 2, 4, ..., 18)$, bounds must contain 0
`property-location-shift`: $\mathbf{x} = (7, 9, 11, ..., 25)$, $\mathbf{y} = (13, 15, 17, ..., 31)$

Must produce same bounds as base case (location invariance)

`property-scale-2x`: $\mathbf{x} = (2, 4, 6, ..., 20)$, $\mathbf{y} = (6, 8, 10, ..., 24)$

Bounds must be 2× the base case bounds (scale equivariance)

`property-antisymmetry`: $\mathbf{x} = (3, 4, ..., 12)$, $\mathbf{y} = (1, 2, ..., 10)$

Bounds must be negated: if original is $[a, b]$, this yields $[-b, -a]$

`property-negative`: $\mathbf{x} = (-10, -9, ..., -1)$, $\mathbf{y} = (-12, -11, ..., -3)$

Validates sign handling with all negative values

`property-mixed-signs`: $\mathbf{x} = (-4, -3, ..., 5)$, $\mathbf{y} = (-3, -2, ..., 6)$

Validates bounds crossing zero with mixed-sign samples

Edge cases — boundary conditions and extreme scenarios (10 tests):

`edge-min-samples`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (6, 7, 8, 9, 10)$, $\text{misrate} = 0.05$

`edge-permissive-misrate`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, $\text{misrate} = 0.5$ (very wide bounds)

`edge-strict-misrate`: $n = m = 20$, $\text{misrate} = 10^{-6}$ (very narrow bounds)

`edge-zero-shift`: $n = m = 10$, all values $= 5$, $\text{misrate} = 10^{-3}$ (bounds around 0)

`edge-asymmetric-3-100`: $n = 3$, $m = 100$, $\text{misrate} = 10^{-2}$ (extreme size difference)

`edge-asymmetric-5-50`: $n = 5$, $m = 50$, $\text{misrate} = 10^{-3}$ (highly unbalanced)

`edge-duplicates`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (5, 5, 5, 5, 5)$, $\text{misrate} = 10^{-2}$ (all duplicates, bounds around -2)

`edge-wide-range`: $n = m = 10$, values spanning 10^{-3} to 10^{8}, $\text{misrate} = 10^{-3}$ (extreme value range)

`edge-tiny-values`: $n = m = 10$, values $\approx 10^{-8}$, $\text{misrate} = 10^{-3}$ (numerical precision)

`edge-large-values`: $n = m = 10$, values $\approx 10^{8}$, $\text{misrate} = 10^{-3}$ (large magnitude)

These edge cases stress-test boundary conditions, numerical stability, and the margin calculation with extreme parameters.

Additive distribution ($[n, m] \in \{10, 30, 50\} \times \{10, 30, 50\}$, $\text{misrate} = 10^{-3}$) — 9 combinations with $\underline{\text{Additive}}(10, 1)$:

`additive-10-10`, `additive-10-30`, `additive-10-50`
`additive-30-10`, `additive-30-30`, `additive-30-50`
`additive-50-10`, `additive-50-30`, `additive-50-50`

Random generation: $\mathbf{x}$ uses seed 0, $\mathbf{y}$ uses seed 1

These fuzzy tests validate that bounds properly encompass the shift estimate for realistic normally-distributed data at various sample sizes.

Uniform distribution ($[n, m] \in \{10, 100\} \times \{10, 100\}$, $\text{misrate} = 10^{-4}$) — 4 combinations with $\underline{\text{Uniform}}(0, 1)$:

`uniform-10-10`, `uniform-10-100`, `uniform-100-10`, `uniform-100-100`

Random generation: $\mathbf{x}$ uses seed 2, $\mathbf{y}$ uses seed 3

The asymmetric size combinations are particularly important for testing margin calculation with unbalanced samples.

Misrate variation ($n = m = 20$, $\mathbf{x} = (0, 2, 4, ..., 38)$, $\mathbf{y} = (10, 12, 14, ..., 48)$) — 5 tests with varying misrates:

- `misrate-1e-2`: misrate $= 10^{-2}$
- `misrate-1e-3`: misrate $= 10^{-3}$
- `misrate-1e-4`: misrate $= 10^{-4}$
- `misrate-1e-5`: misrate $= 10^{-5}$
- `misrate-1e-6`: misrate $= 10^{-6}$

These tests use identical samples with varying misrates to validate the monotonicity property: smaller misrates (higher confidence) produce wider bounds. The sequence demonstrates how bound width increases as misrate decreases, helping implementations verify correct margin calculation.

Unsorted tests — verify independent sorting of $\mathbf{x}$ and $\mathbf{y}$ (15 tests):

- `unsorted-x-natural-5-5`: $\mathbf{x} = (5, 3, 1, 4, 2)$, $\mathbf{y} = (1, 2, 3, 4, 5)$, misrate $= 10^{-2}$ (X reversed, Y sorted)
- `unsorted-y-natural-5-5`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (5, 3, 1, 4, 2)$, misrate $= 10^{-2}$ (X sorted, Y reversed)
- `unsorted-both-natural-5-5`: $\mathbf{x} = (5, 3, 1, 4, 2)$, $\mathbf{y} = (5, 3, 1, 4, 2)$, misrate $= 10^{-2}$ (both reversed)
- `unsorted-x-shuffle-5-5`: $\mathbf{x} = (3, 1, 5, 4, 2)$, $\mathbf{y} = (1, 2, 3, 4, 5)$, misrate $= 10^{-2}$ (X shuffled)
- `unsorted-y-shuffle-5-5`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (4, 2, 5, 1, 3)$, misrate $= 10^{-2}$ (Y shuffled)
- `unsorted-both-shuffle-5-5`: $\mathbf{x} = (3, 1, 5, 4, 2)$, $\mathbf{y} = (2, 4, 1, 5, 3)$, misrate $= 10^{-2}$ (both shuffled)
- `unsorted-demo-unsorted-x`: $\mathbf{x} = (5, 1, 4, 2, 3)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, misrate $= 0.05$ (demo-1 X unsorted)
- `unsorted-demo-unsorted-y`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (7, 3, 6, 4, 5)$, misrate $= 0.05$ (demo-1 Y unsorted)
- `unsorted-demo-both-unsorted`: $\mathbf{x} = (4, 1, 5, 2, 3)$, $\mathbf{y} = (6, 3, 7, 4, 5)$, misrate $= 0.05$ (demo-1 both unsorted)
- `unsorted-identity-unsorted`: $\mathbf{x} = (4, 1, 5, 2, 3)$, $\mathbf{y} = (5, 1, 4, 3, 2)$, misrate $= 10^{-2}$ (identity property, both unsorted)
- `unsorted-negative-unsorted`: $\mathbf{x} = (-1, -5, -3, -2, -4)$, $\mathbf{y} = (-2, -4, -3, -5, -1)$, misrate $= 10^{-2}$ (negative values unsorted)
- `unsorted-asymmetric-5-10`: $\mathbf{x} = (2, 5, 1, 3, 4)$, $\mathbf{y} = (10, 5, 2, 8, 4, 1, 9, 3, 7, 6)$, misrate $= 10^{-2}$ (asymmetric sizes, both unsorted)
- `unsorted-duplicates`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (5, 5, 5, 5, 5)$, misrate $= 10^{-2}$ (all duplicates, any order)
- `unsorted-mixed-duplicates-x`: $\mathbf{x} = (2, 1, 3, 2, 1)$, $\mathbf{y} = (1, 1, 2, 2, 3)$, misrate $= 10^{-2}$ (X has unsorted duplicates)
- `unsorted-mixed-duplicates-y`: $\mathbf{x} = (1, 1, 2, 2, 3)$, $\mathbf{y} = (3, 2, 1, 3, 2)$, misrate $= 10^{-2}$ (Y has unsorted duplicates)

These unsorted tests are critical because ShiftBounds computes bounds from pairwise differences, requiring both samples to be sorted independently. The variety ensures implementations don't incorrectly assume pre-sorted input or sort samples together. Each test must produce identical output to its sorted counterpart, validating that the implementation correctly handles the sorting step.

No performance test — ShiftBounds uses the FastShift algorithm internally, which is already validated by the Shift performance test. Since bounds computation involves only two quantile calculations from the pairwise differences (at positions determined by PairwiseMargin), the performance

characteristics are equivalent to computing two Shift estimates, which completes efficiently for large samples.

2.3. Ratio

$$\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) = \exp(\mathrm{Shift}(\log \mathbf{x}, \log \mathbf{y}))$$

Robust measure of scale ratio between two samples — the multiplicative dual of Shift.

Asymptotic — geometric median of pairwise ratios $\frac{x_i}{y_j}$ (via log-space aggregation)
Domain — $x_i > 0, y_j > 0$
Assumptions — `positivity(x)`, `positivity(y)`
Unit — dimensionless
Complexity — $O((m+n)\log L)$

Properties

Self-ratio $\mathrm{Ratio}(\mathbf{x}, \mathbf{x}) = 1$
Scale equivariance $\mathrm{Ratio}(k_x \cdot \mathbf{x}, k_y \cdot \mathbf{y}) = \left(\frac{k_x}{k_y}\right) \cdot \mathrm{Ratio}(\mathbf{x}, \mathbf{y})$
Multiplicative antisymmetry $\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) = \frac{1}{\mathrm{Ratio}(\mathbf{y}, \mathbf{x})}$

Example

```
Ratio([1, 2, 4, 8, 16], [2, 4, 8, 16, 32]) = 0.5
Ratio(x, x) = 1  Ratio(2x, 5y) = 0.4 · Ratio(x, y)
```

Relationship to Shift

Ratio is the multiplicative analog of Shift. While Shift computes the median of pairwise differences $x_i - y_j$, Ratio computes the median of pairwise ratios $\frac{x_i}{y_j}$ via log-transformation. This relationship is expressed formally as:

$$\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) = \exp(\mathrm{Shift}(\log \mathbf{x}, \log \mathbf{y}))$$

The log-transformation converts multiplicative relationships to additive ones, allowing the fast Shift algorithm to compute the result efficiently. The exp-transformation converts back to the ratio scale.

Ratio is appropriate for multiplicative relationships rather than additive differences. If one system is "twice as fast" or prices are "30% lower," the underlying thinking is in ratios. A result of 0.5 means the first group is typically half the size of the second; 2.0 means twice as large. This estimator is appropriate for quantities like prices, response times, and concentrations where relative comparisons make more sense than absolute ones. Both samples must contain strictly positive values.

2.3.1. Algorithm

The Ratio estimator measures the typical multiplicative relationship between elements of two samples. Given samples $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_m)$ with all positive values, this estimator is defined as:

$$\text{Ratio}(\mathbf{x}, \mathbf{y}) = \exp\big(\text{median}_{i,j}\big(\log x_i - \log y_j\big)\big) = \exp(\text{Shift}(\log \mathbf{x}, \log \mathbf{y}))$$

A naive approach would compute all $n \times m$ ratios, sort them, and extract the median. With $n = m = 10000$, this creates 100 million values, requiring quadratic memory and $O(nm \log(nm))$ time.

The presented algorithm avoids this cost by exploiting the multiplicative-additive duality. Taking the logarithm of a ratio yields a difference:

$$\log\left(\frac{x_i}{y_j}\right) = \log(x_i) - \log(y_j)$$

This transforms the problem of finding the median of pairwise ratios into finding the median of pairwise differences in log-space.

The algorithm operates in three steps:

1. **Log-transform** — Apply $\log$ to each element of both samples. If $\mathbf{x}$ was sorted, $\log \mathbf{x}$ remains sorted (log is monotonically increasing for positive values).

2. **Delegate to Shift** — Use the Shift algorithm to compute the desired quantile of pairwise differences in log-space. This leverages the $O((m + n) \log L)$ complexity of the Shift algorithm.

3. **Exp-transform** — Apply $\exp$ to convert the result back to ratio-space. If the log-space median is d, then $\exp(d) = \exp\big(\log(x_i) - \log\big(y_j\big)\big) = \frac{x_i}{y_j}$ is the original ratio.

The total complexity is $O((m + n) \log L)$ per quantile, where L represents the convergence precision in the log-space binary search. This is dramatically more efficient than the naive $O(nm \log(nm))$ approach.

Memory usage is $O(m + n)$ for storing the log-transformed samples, compared to $O(nm)$ for materializing all pairwise ratios.

```
namespace Pragmastat.Algorithms;

using System;
using System.Collections.Generic;
using System.Linq;
using Pragmastat.Exceptions;
using Pragmastat.Internal;

/// <summary>
/// Computes quantiles of pairwise ratios via log-transformation and FastShift delegation.
/// Ratio(x, y) = exp(Shift(log(x), log(y)))
/// </summary>
```

```
public static class FastRatio
{
  /// <summary>
  /// Computes quantiles of all pairwise ratios { x_i / y_j }.
  /// Time: O((m + n) * log(precision)) per quantile. Space: O(m + n).
  /// </summary>
  /// <remarks>
  /// Log-transformation preserves sort order for positive values.
  /// </remarks>
  /// <param name="x">First sample (must be strictly positive).</param>
  /// <param name="y">Second sample (must be strictly positive).</param>
  /// <param name="p">Probabilities in [0, 1].</param>
  /// <param name="assumeSorted">If true, assumes x and y are already sorted in ascending
order.</param>
  /// <returns>Quantile values for each probability in p.</returns>
  public static double[] Estimate(IReadOnlyList<double> x, IReadOnlyList<double> y,
double[] p, bool assumeSorted = false)
  {
    // Log-transform both samples (includes positivity check)
    var logX = MathExtensions.Log(x, Subject.X);
    var logY = MathExtensions.Log(y, Subject.Y);

    // Delegate to FastShift in log-space
    var logResult = FastShift.Estimate(logX, logY, p, assumeSorted);

    // Exp-transform back to ratio-space
    return logResult.Select(v => Math.Exp(v)).ToArray();
  }
}
```

2.3.2. Tests

$$\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) = \exp(\mathrm{Shift}(\log \mathbf{x}, \log \mathbf{y}))$$

The Ratio test suite contains 37 test cases (25 original + 12 unsorted), excluding zero values due to division constraints. The new definition uses geometric interpolation (via log-space), which affects expected values for even $m \times n$ cases.

Demo examples ($n = m = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (1, 2, 4, 8, 16)$, $\mathbf{y} = (2, 4, 8, 16, 32)$, expected output: 0.5 (base case, odd $m \times n$)
`demo-2`: $\mathbf{x} = (1, 2, 4, 8, 16)$, $\mathbf{y} = (1, 2, 4, 8, 16)$, expected output: 1 (identity property)
`demo-3`: $\mathbf{x} = (2, 4, 8, 16, 32)$, $\mathbf{y} = (10, 20, 40, 80, 160)$ (=[2×demo-1.x, 5×demo-1.y]), expected output: 0.2 (scale property)

Natural sequences ($[n, m] \in \{1, 2, 3\} \times \{1, 2, 3\}$) — 9 combinations:

`natural-1-1`: $\mathbf{x} = (1)$, $\mathbf{y} = (1)$, expected output: 1
`natural-1-2`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2)$, expected output: ≈ 0.707 ($= \sqrt{0.5}$, geometric interpolation)
`natural-1-3`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2, 3)$, expected output: 0.5
`natural-2-1`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1)$, expected output: ≈ 1.414 ($= \sqrt{2}$, geometric interpolation)
`natural-2-2`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2)$, expected output: 1
`natural-2-3`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2, 3)$, expected output: ≈ 0.816 (geometric interpolation)
`natural-3-1`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1)$, expected output: 2
`natural-3-2`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2)$, expected output: ≈ 1.225 (geometric interpolation)
`natural-3-3`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2, 3)$, expected output: 1

Additive distribution ($[n, m] \in \{5, 10, 30\} \times \{5, 10, 30\}$) — 9 combinations with $\underline{\mathrm{Additive}}(10, 1)$:

`additive-5-5`, `additive-5-10`, `additive-5-30`
`additive-10-5`, `additive-10-10`, `additive-10-30`
`additive-30-5`, `additive-30-10`, `additive-30-30`
Random generation: x uses seed 0, y uses seed 1

Uniform distribution ($[n, m] \in \{5, 100\} \times \{5, 100\}$) — 4 combinations with $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-5-5`, `uniform-5-100`, `uniform-100-5`, `uniform-100-100`
Random generation: x uses seed 2, y uses seed 3
Note: all generated values are strictly positive (no zeros); values near zero test numerical stability of log-transformation

The natural sequences verify the identity property ($\mathrm{Ratio}(\mathbf{x}, \mathbf{x}) = 1$) and validate ratio calculations with simple integer inputs. Note that implementations should handle the practical constraint of avoiding division by values near zero.

Unsorted tests — verify independent sorting for ratio calculation (12 tests):

`unsorted-x-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X unsorted (reversed), Y sorted (2 tests)
`unsorted-y-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X sorted, Y unsorted (reversed) (2 tests)

`unsorted-both-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: both unsorted (reversed) (2 tests)
`unsorted-demo-unsorted-x`: $\mathbf{x} = (16, 1, 8, 2, 4)$, $\mathbf{y} = (2, 4, 8, 16, 32)$ (demo-1 with X unsorted)
`unsorted-demo-unsorted-y`: $\mathbf{x} = (1, 2, 4, 8, 16)$, $\mathbf{y} = (32, 2, 16, 4, 8)$ (demo-1 with Y unsorted)
`unsorted-demo-both-unsorted`: $\mathbf{x} = (8, 1, 16, 4, 2)$, $\mathbf{y} = (16, 32, 2, 8, 4)$ (demo-1 both unsorted)
`unsorted-identity-unsorted`: $\mathbf{x} = (4, 1, 8, 2, 16)$, $\mathbf{y} = (16, 1, 8, 4, 2)$ (identity property, both unsorted)
`unsorted-asymmetric-unsorted-2-3`: $\mathbf{x} = (2, 1)$, $\mathbf{y} = (3, 1, 2)$ (asymmetric, both unsorted)
`unsorted-power-unsorted-5`: $\mathbf{x} = (16, 2, 8, 1, 4)$, $\mathbf{y} = (32, 4, 16, 2, 8)$ (powers of 2 unsorted)

2.4. RatioBounds

$$\text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \exp(\text{ShiftBounds}(\log \mathbf{x}, \log \mathbf{y}, \text{misrate}))$$

Robust bounds on $\text{Ratio}(\mathbf{x}, \mathbf{y})$ with specified coverage — the multiplicative dual of ShiftBounds.

Also known as — distribution-free confidence interval for Hodges-Lehmann ratio
Interpretation — misrate is probability that true ratio falls outside bounds
Domain — $x_i > 0, y_j > 0, \text{misrate} \geq \frac{2}{\binom{n+m}{n}}$
Assumptions — `positivity(x)`, `positivity(y)`
Unit — dimensionless
Note — assumes weak continuity (ties from measurement resolution are tolerated but may yield conservative bounds)

Properties

Scale invariance $\text{RatioBounds}(k \cdot \mathbf{x}, k \cdot \mathbf{y}, \text{misrate}) = \text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Scale equivariance $\text{RatioBounds}\left(k_x \cdot \mathbf{x}, k_y \cdot \mathbf{y}, \text{misrate}\right) = \left(\frac{k_x}{k_y}\right) \cdot \text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Multiplicative antisymmetry $\text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \frac{1}{\text{RatioBounds}(\mathbf{y}, \mathbf{x}, \text{misrate})}$ (bounds reversed)

Example

`RatioBounds([1..30], [10..40], 1e-4)` where `Ratio ≈ 0.5` yields bounds containing 0.5
Bounds fail to cover true ratio with probability $\approx$ misrate

Relationship to ShiftBounds

RatioBounds is computed via log-transformation:

$$\text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \exp(\text{ShiftBounds}(\log \mathbf{x}, \log \mathbf{y}, \text{misrate}))$$

This means if ShiftBounds returns $[a, b]$ for the log-transformed samples, RatioBounds returns $[e^a, e^b]$.

RatioBounds provides not just the estimated ratio but also the uncertainty of that estimate. The function returns an interval of plausible ratio values given the data. Set misrate to control how often the bounds might fail to contain the true ratio: use 10^{-3} for everyday analysis or 10^{-6} for critical decisions where errors are costly. These bounds require no assumptions about your data distribution, so they remain valid for any continuous positive measurements. If the bounds exclude 1, that suggests a reliable multiplicative difference between the two groups.

2.4.1. Algorithm

The $\mathrm{RatioBounds}$ estimator uses the same log-exp transformation as Ratio, delegating to ShiftBounds in log-space:

$$\mathrm{RatioBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate}) = \exp(\mathrm{ShiftBounds}(\log \mathbf{x}, \log \mathbf{y}, \mathrm{misrate}))$$

The algorithm operates in three steps:

1. **Log-transform** — Apply $\log$ to each element of both samples. Positivity is required so that the logarithm is defined.

2. **Delegate to ShiftBounds** — Compute $[a, b] = \mathrm{ShiftBounds}(\log \mathbf{x}, \log \mathbf{y}, \mathrm{misrate})$. This provides distribution-free bounds on the shift in log-space.

3. **Exp-transform** — Return $[e^a, e^b]$, converting the additive bounds back to multiplicative bounds.

Because $\log$ and $\exp$ are monotone, the coverage guarantee of $\mathrm{ShiftBounds}$ transfers directly: the probability that the true ratio falls outside $[e^a, e^b]$ equals the probability that the true log-shift falls outside $[a, b]$, which is at most $\mathrm{misrate}$.

2.4.2. Tests

$$\text{RatioBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \exp(\text{ShiftBounds}(\log \mathbf{x}, \log \mathbf{y}, \text{misrate}))$$

The RatioBounds test suite contains 61 correctness test cases (3 demo + 9 natural + 6 property + 10 edge + 9 multiplic + 4 uniform + 5 misrate + 15 unsorted). Since RatioBounds returns bounds rather than a point estimate, tests validate that the bounds contain $\text{Ratio}(\mathbf{x}, \mathbf{y})$ and satisfy equivariance properties. Each test case output is a JSON object with `lower` and `upper` fields representing the interval bounds. All samples must contain strictly positive values. The domain constraint $\text{misrate} \geq \frac{2}{\binom{n+m}{n}}$ is enforced; inputs violating this return a domain error.

Demo examples ($n = m = 5$, positive samples) — 3 tests:

`demo-1`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (2, 3, 4, 5, 6)$, $\text{misrate} = 0.05$
`demo-2`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (2, 3, 4, 5, 6)$, $\text{misrate} = 0.01$, expected: wider bounds than demo-1
`demo-3`: $\mathbf{x} = (2, 3, 4, 5, 6)$, $\mathbf{y} = (2, 3, 4, 5, 6)$, $\text{misrate} = 0.05$, expected: bounds containing 1 (identity case)

These cases illustrate how tighter misrates produce wider bounds and validate the identity property where identical samples yield bounds containing one.

Natural sequences ($[n, m] \in \{5, 8, 10\} \times \{5, 8, 10\}$, $\text{misrate} = 10^{-2}$) — 9 combinations:

`natural-5-5`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 5)$, expected bounds containing 1
`natural-5-8`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 8)$
`natural-5-10`: $\mathbf{x} = (1, ..., 5)$, $\mathbf{y} = (1, ..., 10)$
`natural-8-5`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 5)$
`natural-8-8`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 8)$, expected bounds containing 1
`natural-8-10`: $\mathbf{x} = (1, ..., 8)$, $\mathbf{y} = (1, ..., 10)$
`natural-10-5`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 5)$
`natural-10-8`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 8)$
`natural-10-10`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 10)$, expected bounds containing 1

These sizes are chosen to satisfy $\text{misrate} \geq \frac{2}{\binom{n+m}{n}}$ for all combinations.

Property validation ($n = m = 10$, $\text{misrate} = 10^{-3}$) — 6 tests:

`property-identity`: $\mathbf{x} = (1, 2, ..., 10)$, $\mathbf{y} = (1, 2, ..., 10)$, bounds must contain 1
`property-scale-2x`: $\mathbf{x} = (2, 4, ..., 20)$, $\mathbf{y} = (1, 2, ..., 10)$, bounds must contain 2
`property-reciprocal`: $\mathbf{x} = (1, 2, ..., 10)$, $\mathbf{y} = (2, 4, ..., 20)$, bounds must contain 0.5 (reciprocal of scale-2x)
`property-common-scale`: $\mathbf{x} = (10, 20, ..., 100)$, $\mathbf{y} = (20, 40, ..., 200)$
Same ratio as property-reciprocal (common scale invariance)
`property-small-values`: $\mathbf{x} = (0.1, 0.2, ..., 1.0)$, $\mathbf{y} = (0.2, 0.4, ..., 2.0)$
Same ratio as property-reciprocal (small value handling)
`property-mixed-scales`: $\mathbf{x} = (0.01, 0.1, 1, 10, 100, 1000, 0.5, 5, 50, 500)$, $\mathbf{y} = (0.1, 1, 10, 100, 1000, 10000, 5, 50, 500, 5000)$
Wide range validation

Edge cases — boundary conditions and extreme scenarios (10 tests):

`edge-min-samples`: $\mathbf{x} = (2, 3, 4, 5, 6)$, $\mathbf{y} = (3, 4, 5, 6, 7)$, $\text{misrate} = 0.05$
`edge-permissive-misrate`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (2, 3, 4, 5, 6)$, $\text{misrate} = 0.5$ (very wide bounds)
`edge-strict-misrate`: $n = m = 20$, $\text{misrate} = 10^{-6}$ (very narrow bounds)
`edge-unity-ratio`: $n = m = 10$, all values $= 5$, $\text{misrate} = 10^{-3}$ (bounds around 1)
`edge-asymmetric-3-100`: $n = 3$, $m = 100$, $\text{misrate} = 10^{-2}$ (extreme size difference)
`edge-asymmetric-5-50`: $n = 5$, $m = 50$, $\text{misrate} = 10^{-3}$ (highly unbalanced)
`edge-duplicates`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (5, 5, 5, 5, 5)$, $\text{misrate} = 10^{-2}$ (all duplicates, bounds around 0.6)
`edge-wide-range`: $n = m = 10$, values spanning 10^{-3} to 10^{8}, $\text{misrate} = 10^{-3}$ (extreme value range)
`edge-tiny-values`: $n = m = 10$, values $\approx 10^{-6}$, $\text{misrate} = 10^{-3}$ (numerical precision)
`edge-large-values`: $n = m = 10$, values $\approx 10^{8}$, $\text{misrate} = 10^{-3}$ (large magnitude)

These edge cases stress-test boundary conditions, numerical stability, and the margin calculation with extreme parameters.

Multiplic distribution ($[n, m] \in \{10, 30, 50\} \times \{10, 30, 50\}$, $\text{misrate} = 10^{-3}$) — 9 combinations with $\underline{\text{Multiplic}}(1, 0.5)$:

`multiplic-10-10`, `multiplic-10-30`, `multiplic-10-50`
`multiplic-30-10`, `multiplic-30-30`, `multiplic-30-50`
`multiplic-50-10`, `multiplic-50-30`, `multiplic-50-50`
Random generation: $\mathbf{x}$ uses seed 0, $\mathbf{y}$ uses seed 1

These fuzzy tests validate that bounds properly encompass the ratio estimate for realistic log-normally-distributed data at various sample sizes.

Uniform distribution ($[n, m] \in \{10, 100\} \times \{10, 100\}$, $\text{misrate} = 10^{-4}$) — 4 combinations with $\underline{\text{Uniform}}(1, 10)$:

`uniform-10-10`, `uniform-10-100`, `uniform-100-10`, `uniform-100-100`
Random generation: $\mathbf{x}$ uses seed 2, $\mathbf{y}$ uses seed 3
Note: positive range $[1, 10)$ used for ratio compatibility

The asymmetric size combinations are particularly important for testing margin calculation with unbalanced samples.

Misrate variation ($n = m = 20$, $\mathbf{x} = (1, 2, ..., 20)$, $\mathbf{y} = (2, 4, ..., 40)$) — 5 tests with varying misrates:

`misrate-1e-2`: $\text{misrate} = 10^{-2}$
`misrate-1e-3`: $\text{misrate} = 10^{-3}$
`misrate-1e-4`: $\text{misrate} = 10^{-4}$
`misrate-1e-5`: $\text{misrate} = 10^{-5}$
`misrate-1e-6`: $\text{misrate} = 10^{-6}$

These tests use identical samples with varying misrates to validate the monotonicity property: smaller misrates (higher confidence) produce wider bounds. The sequence demonstrates how bound width increases as misrate decreases, helping implementations verify correct margin calculation.

Unsorted tests — verify independent sorting of $\mathbf{x}$ and $\mathbf{y}$ (15 tests):

`unsorted-x-natural-5-5`: $\mathbf{x} = (5, 3, 1, 4, 2)$, $\mathbf{y} = (1, 2, 3, 4, 5)$, $\text{misrate} = 10^{-2}$ (X reversed, Y sorted)
`unsorted-y-natural-5-5`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (5, 3, 1, 4, 2)$, $\text{misrate} = 10^{-2}$ (X sorted, Y reversed)
`unsorted-both-natural-5-5`: $\mathbf{x} = (5, 3, 1, 4, 2)$, $\mathbf{y} = (5, 3, 1, 4, 2)$, $\text{misrate} = 10^{-2}$ (both reversed)
`unsorted-x-shuffle-5-5`: $\mathbf{x} = (3, 1, 5, 4, 2)$, $\mathbf{y} = (1, 2, 3, 4, 5)$, $\text{misrate} = 10^{-2}$ (X shuffled)
`unsorted-y-shuffle-5-5`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (4, 2, 5, 1, 3)$, $\text{misrate} = 10^{-2}$ (Y shuffled)
`unsorted-both-shuffle-5-5`: $\mathbf{x} = (3, 1, 5, 4, 2)$, $\mathbf{y} = (2, 4, 1, 5, 3)$, $\text{misrate} = 10^{-2}$ (both shuffled)
`unsorted-demo-unsorted-x`: $\mathbf{x} = (5, 1, 4, 2, 3)$, $\mathbf{y} = (2, 3, 4, 5, 6)$, $\text{misrate} = 0.05$ (demo-1 X unsorted)
`unsorted-demo-unsorted-y`: $\mathbf{x} = (1, 2, 3, 4, 5)$, $\mathbf{y} = (6, 2, 5, 3, 4)$, $\text{misrate} = 0.05$ (demo-1 Y unsorted)
`unsorted-demo-both-unsorted`: $\mathbf{x} = (4, 1, 5, 2, 3)$, $\mathbf{y} = (5, 2, 6, 3, 4)$, $\text{misrate} = 0.05$ (demo-1 both unsorted)
`unsorted-identity-unsorted`: $\mathbf{x} = (4, 1, 5, 2, 3)$, $\mathbf{y} = (5, 1, 4, 3, 2)$, $\text{misrate} = 10^{-2}$ (identity property, both unsorted)
`unsorted-scale-unsorted`: $\mathbf{x} = (10, 30, 20)$, $\mathbf{y} = (15, 5, 10)$, $\text{misrate} = 0.5$ (scale relationship, both unsorted)
`unsorted-asymmetric-5-10`: $\mathbf{x} = (2, 5, 1, 3, 4)$, $\mathbf{y} = (10, 5, 2, 8, 4, 1, 9, 3, 7, 6)$, $\text{misrate} = 10^{-2}$ (asymmetric sizes, both unsorted)
`unsorted-duplicates`: $\mathbf{x} = (3, 3, 3, 3, 3)$, $\mathbf{y} = (5, 5, 5, 5, 5)$, $\text{misrate} = 10^{-2}$ (all duplicates, any order)
`unsorted-mixed-duplicates-x`: $\mathbf{x} = (2, 1, 3, 2, 1)$, $\mathbf{y} = (1, 1, 2, 2, 3)$, $\text{misrate} = 10^{-2}$ (X has unsorted duplicates)
`unsorted-mixed-duplicates-y`: $\mathbf{x} = (1, 1, 2, 2, 3)$, $\mathbf{y} = (3, 2, 1, 3, 2)$, $\text{misrate} = 10^{-2}$ (Y has unsorted duplicates)

These unsorted tests are critical because RatioBounds computes bounds from pairwise ratios, requiring both samples to be sorted independently. The variety ensures implementations don't incorrectly assume pre-sorted input or sort samples together. Each test must produce identical output to its sorted counterpart, validating that the implementation correctly handles the sorting step.

No performance test — RatioBounds uses the FastRatio algorithm internally, which delegates to FastShift in log-space. Since bounds computation involves only two quantile calculations from the pairwise differences (at positions determined by PairwiseMargin), the performance characteristics are equivalent to computing two Ratio estimates, which completes efficiently for large samples.

2.5. Disparity

$$\text{Disparity}(\mathbf{x}, \mathbf{y}) = \frac{\text{Shift}(\mathbf{x}, \mathbf{y})}{\text{AvgSpread}(\mathbf{x}, \mathbf{y})}$$

where $\text{AvgSpread}(\mathbf{x}, \mathbf{y}) = \frac{n \cdot \text{Spread}(\mathbf{x}) + m \cdot \text{Spread}(\mathbf{y})}{n+m}$ is the weighted average of dispersions (pooled scale).

Robust effect size (shift normalized by pooled dispersion).

Also known as — robust Cohen's d (Cohen (1988); estimates differ due to robust construction)
Domain — $\text{AvgSpread}(\mathbf{x}, \mathbf{y}) > 0$
Assumptions — `sparity(x)`, `sparity(y)`
Unit — spread units

Properties

Location invariance $\text{Disparity}(\mathbf{x} + k, \mathbf{y} + k) = \text{Disparity}(\mathbf{x}, \mathbf{y})$
Scale invariance $\text{Disparity}(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = \text{sign}(k) \cdot \text{Disparity}(\mathbf{x}, \mathbf{y})$
Antisymmetry $\text{Disparity}(\mathbf{x}, \mathbf{y}) = -\text{Disparity}(\mathbf{y}, \mathbf{x})$

Example

`Disparity(x, y) = 0.4` where `Shift = 2`, `AvgSpread = 5`
`Disparity(x + c, y + c) = Disparity(x, y)` `Disparity(kx, ky) = Disparity(x, y)`

Disparity expresses a difference between groups in a way that does not depend on the original measurement units. A disparity of 0.5 means the groups differ by half a spread unit; 1.0 means one full spread unit. Being dimensionless allows comparison of effect sizes across different studies, metrics, or measurement scales. What counts as a "large" or "small" disparity depends entirely on the domain and what matters practically in a given application. Do not rely on universal thresholds; interpret the number in context.

2.5.1. Algorithm

The Disparity estimator is a composition of Shift and Spread:

$$\text{Disparity}(\mathbf{x}, \mathbf{y}) = \frac{\text{Shift}(\mathbf{x}, \mathbf{y})}{\text{AvgSpread}(\mathbf{x}, \mathbf{y})}$$

where $\text{AvgSpread}(\mathbf{x}, \mathbf{y}) = \frac{n \cdot \text{Spread}(\mathbf{x}) + m \cdot \text{Spread}(\mathbf{y})}{n+m}$ is the pooled scale.

The algorithm proceeds as follows:

1. **Compute Spread for each sample** — Delegate to the Spread algorithm for $\mathbf{x}$ and $\mathbf{y}$ independently.
2. **Compute AvgSpread** — Form the weighted average $\text{AvgSpread} = \frac{n \cdot \text{Spread}(\mathbf{x}) + m \cdot \text{Spread}(\mathbf{y})}{n+m}$.
3. **Domain check** — Verify that $\text{AvgSpread} > 0$. If the pooled spread is zero, the division is undefined.
4. **Compute Shift** — Delegate to the Shift algorithm for the pair $(\mathbf{x}, \mathbf{y})$.
5. **Divide** — Return $\frac{\text{Shift}(\mathbf{x},\mathbf{y})}{\text{AvgSpread}(\mathbf{x},\mathbf{y})}$.

```
using Pragmastat.Algorithms;
using Pragmastat.Exceptions;
using Pragmastat.Internal;
using Pragmastat.Metrology;

namespace Pragmastat.Estimators;

public class DisparityEstimator : ITwoSampleEstimator
{
  public static readonly DisparityEstimator Instance = new();

  public Measurement Estimate(Sample x, Sample y)
  {
    Assertion.MatchedUnit(x, y);

    var spreadX = FastSpread.Estimate(x.SortedValues, isSorted: true);
    if (spreadX <= 0)
      throw AssumptionException.Sparity(Subject.X);
    var spreadY = FastSpread.Estimate(y.SortedValues, isSorted: true);
    if (spreadY <= 0)
      throw AssumptionException.Sparity(Subject.Y);

    // Calculate shift (we know inputs are valid)
    var shiftVal = FastShift.Estimate(x.SortedValues, y.SortedValues, [0.5], true)[0];
    var avgSpreadVal = (x.Size * spreadX + y.Size * spreadY) / (x.Size + y.Size);

    return (shiftVal / avgSpreadVal).WithUnit(DisparityUnit.Instance);
  }
}
```

2.5.2. Tests

$$\text{Disparity}(\mathbf{x}, \mathbf{y}) = \frac{\text{Shift}(\mathbf{x}, \mathbf{y})}{\text{AvgSpread}(\mathbf{x}, \mathbf{y})}$$

The Disparity test suite contains 28 test cases (16 original + 12 unsorted). Since Disparity combines Shift and AvgSpread, unsorted tests verify both components handle sorting correctly.

Demo examples ($n = m = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (0, 2, 4, 6, 8)$, expected output: 0.4 (base case: $2/5$)
`demo-2`: $\mathbf{x} = (5, 8, 11, 14, 17)$, $\mathbf{y} = (5, 7, 9, 11, 13)$ (= demo-1 + 5), expected output: 0.4 (location invariance)
`demo-3`: $\mathbf{x} = (0, 6, 12, 18, 24)$, $\mathbf{y} = (0, 4, 8, 12, 16)$ (= 2 × demo-1), expected output: 0.4 (scale invariance)
`demo-4`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\mathbf{y} = (0, 3, 6, 9, 12)$ (= reversed demo-1), expected output: -0.4 (anti-symmetry)

Natural sequences ($[n, m] \in \{2, 3\} \times \{2, 3\}$) — 4 combinations:

`natural-2-2`, `natural-2-3`, `natural-3-2`, `natural-3-3`
Minimum size $n, m \geq 2$ required for meaningful dispersion calculations

Negative values ($[n, m] = [2, 2]$) — end-to-end validation with negative values:

`negative-2-2`: $\mathbf{x} = (-2, -1)$, $\mathbf{y} = (-2, -1)$, expected output: 0

Uniform distribution ($[n, m] \in \{5, 100\} \times \{5, 100\}$) — 4 combinations with $\underline{\text{Uniform}}(0, 1)$:

`uniform-5-5`, `uniform-5-100`, `uniform-100-5`, `uniform-100-100`
Random generation: $\mathbf{x}$ uses seed 0, $\mathbf{y}$ uses seed 1

The smaller test set for Disparity reflects implementation confidence. Since Disparity combines Shift and AvgSpread, correct implementation of those components ensures Disparity correctness. The test cases validate the division operation and confirm scale-free properties.

Composite estimator stress tests — edge cases for effect size calculation:

`composite-small-avgspread`: $\mathbf{x} = (10.001, 10.002, 10.003)$, $\mathbf{y} = (10.004, 10.005, 10.006)$ (tiny spread, large shift)
`composite-large-avgspread`: $\mathbf{x} = (1, 100, 200)$, $\mathbf{y} = (50, 150, 250)$ (large spread, small shift)
`composite-extreme-disparity`: $\mathbf{x} = (1, 1.001)$, $\mathbf{y} = (100, 100.001)$ (extreme ratio, tests precision)

Unsorted tests — verify both Shift and AvgSpread handle sorting (12 tests):

`unsorted-x-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X unsorted (reversed), Y sorted (2 tests)
`unsorted-y-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X sorted, Y unsorted (reversed) (2 tests)
`unsorted-both-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: both unsorted (reversed) (2 tests)
`unsorted-demo-unsorted-x`: $\mathbf{x} = (12, 0, 6, 3, 9)$, $\mathbf{y} = (0, 2, 4, 6, 8)$ (demo-1 with X unsorted)
`unsorted-demo-unsorted-y`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (8, 0, 4, 2, 6)$ (demo-1 with Y unsorted)

`unsorted-demo-both-unsorted`: $\mathbf{x} = (9, 0, 12, 3, 6)$, $\mathbf{y} = (6, 0, 8, 2, 4)$ (demo-1 both unsorted)
`unsorted-location-invariance-unsorted`: $\mathbf{x} = (17, 5, 11, 8, 14)$, $\mathbf{y} = (13, 5, 9, 7, 11)$ (demo-2 unsorted)
`unsorted-scale-invariance-unsorted`: $\mathbf{x} = (24, 0, 12, 6, 18)$, $\mathbf{y} = (16, 0, 8, 4, 12)$ (demo-3 unsorted)
`unsorted-anti-symmetry-unsorted`: $\mathbf{x} = (8, 0, 4, 2, 6)$, $\mathbf{y} = (12, 0, 6, 3, 9)$ (demo-4 reversed and unsorted)

As a composite estimator, Disparity tests both the numerator (Shift) and denominator (AvgSpread). Unsorted variants verify end-to-end correctness including invariance properties.

2.5.3. References

Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Lawrence Erlbaum Associates.

2.6. DisparityBounds

$$\text{DisparityBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = [L_D, U_D]$$

Let $\min_S = \frac{2}{\binom{n+m}{n}}$ and $\min_A = 2 \cdot \max\left(2^{1-\lfloor \frac{n}{2} \rfloor}, 2^{1-\lfloor \frac{m}{2} \rfloor}\right)$. Require $\text{misrate} \geq \min_S + \min_A$. Let $\text{extra} = \text{misrate} - (\min_S + \min_A)$, $\alpha_S = \min_S + \frac{\text{extra}}{2}$, $\alpha_A = \min_A + \frac{\text{extra}}{2}$. Compute $[L_S, U_S] = \text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \alpha_S)$ and $[L_A, U_A] = \text{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \alpha_A)$.

If $L_A > 0$, return $[L_D, U_D] = \left[\min\left(\frac{L_S}{L_A}, \frac{L_S}{U_A}, \frac{U_S}{L_A}, \frac{U_S}{U_A}\right), \max\left(\frac{L_S}{L_A}, \frac{L_S}{U_A}, \frac{U_S}{L_A}, \frac{U_S}{U_A}\right)\right]$.

If $L_A = 0$, return the tightest single interval that is always valid:

$L_S > 0$: $\left[\frac{L_S}{U_A}, +\infty\right)$
$U_S < 0$: $\left(-\infty, \frac{U_S}{U_A}\right]$
$L_S = 0$ and $U_S = 0$: $[0, 0]$
$L_S = 0$ and $U_S > 0$: $[0, +\infty)$
$L_S < 0$ and $U_S = 0$: $(-\infty, 0]$
otherwise: $(-\infty, +\infty)$

If $U_A = 0$, use the sign-only rule: $[0, +\infty)$ if $L_S \geq 0$, $(-\infty, 0]$ if $U_S \leq 0$, $(-\infty, +\infty)$ otherwise (with $[0, 0]$ when $L_S = U_S = 0$).

Robust bounds on $\text{Disparity}(\mathbf{x}, \mathbf{y})$ with specified coverage.

Interpretation — misrate is probability that true disparity falls outside bounds
Domain — any real numbers, $n \geq 2$, $m \geq 2$, $\text{misrate} \geq \min_S + \min_A$
Assumptions — `sparity(x)`, `sparity(y)`
Unit — dimensionless (spread units)
Note — Bonferroni split between shift and avg-spread bounds; no independence assumption needed; bounds may be unbounded when pooled spread cannot be certified positive

Properties

Location invariance $\text{DisparityBounds}(\mathbf{x} + k, \mathbf{y} + k, \text{misrate}) = \text{DisparityBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Scale invariance $\text{DisparityBounds}(k \cdot \mathbf{x}, k \cdot \mathbf{y}, \text{misrate}) = \text{sign}(k) \cdot \text{DisparityBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Antisymmetry $\text{DisparityBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = -\text{DisparityBounds}(\mathbf{y}, \mathbf{x}, \text{misrate})$ (bounds reversed)
Monotonicity in misrate smaller misrate produces wider bounds

Example

`DisparityBounds([1..30], [21..50], 0.02)` returns bounds containing `Disparity`

2.6.1. Algorithm

The DisparityBounds estimator constructs bounds on $\mathrm{Disparity}(\mathbf{x}, \mathbf{y})$ by combining ShiftBounds and AvgSpreadBounds through a Bonferroni split.

Misrate allocation

The total misrate budget is split between the shift and avg-spread components. Let $\min_S = \frac{2}{\binom{n+m}{n}}$ (minimum for ShiftBounds) and $\min_A = 2 \cdot \max\left(2^{1-\lfloor n/2 \rfloor}, 2^{1-\lfloor m/2 \rfloor}\right)$ (minimum for AvgSpreadBounds). The extra budget beyond the minimums is split equally:

$$\alpha_S = \min_S + \frac{\text{misrate} - \min_S - \min_A}{2}, \quad \alpha_A = \min_A + \frac{\text{misrate} - \min_S - \min_A}{2}$$

Component bounds

Compute $[L_S, U_S] = \mathrm{ShiftBounds}(\mathbf{x}, \mathbf{y}, \alpha_S)$ and $[L_A, U_A] = \mathrm{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \alpha_A)$. By Bonferroni's inequality, the probability that both intervals simultaneously contain their respective true values is at least $1 - \alpha_S - \alpha_A = 1 - \text{misrate}$.

Interval division

When $L_A > 0$, the disparity bounds are obtained by dividing the shift interval by the avg-spread interval. Since dividing by a positive interval can flip the ordering depending on the sign of the numerator endpoints, the algorithm computes all four combinations and takes the extremes:

$$[L_D, U_D] = \left[\min\left(\frac{L_S}{L_A}, \frac{L_S}{U_A}, \frac{U_S}{L_A}, \frac{U_S}{U_A}\right), \max\left(\frac{L_S}{L_A}, \frac{L_S}{U_A}, \frac{U_S}{L_A}, \frac{U_S}{U_A}\right)\right]$$

Edge cases

When $L_A = 0$ (the avg-spread interval includes zero), the bounds become partially or fully unbounded depending on the sign of $[L_S, U_S]$:

$L_S > 0$: $\left[\frac{L_S}{U_A}, +\infty\right)$
$U_S < 0$: $\left(-\infty, \frac{U_S}{U_A}\right]$
$L_S = U_S = 0$: $[0, 0]$
otherwise: $(-\infty, +\infty)$

When $U_A = 0$ (the avg-spread interval collapses to zero), only the sign of the shift determines the result.

```
using Pragmastat.Algorithms;
using Pragmastat.Exceptions;
using Pragmastat.Internal;
using Pragmastat.Metrology;

using static Pragmastat.Functions.MinAchievableMisrate;

namespace Pragmastat.Estimators;
```

```
/// <summary>
/// Distribution-free bounds for disparity using Bonferroni combination.
/// </summary>
public class DisparityBoundsEstimator : ITwoSampleBoundsEstimator
{
  public static readonly DisparityBoundsEstimator Instance = new();

  public Bounds Estimate(Sample x, Sample y, Probability misrate)
  {
    return Estimate(x, y, misrate, null);
  }

  public Bounds Estimate(Sample x, Sample y, Probability misrate, string? seed)
  {
    Assertion.MatchedUnit(x, y);

    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
      throw AssumptionException.Domain(Subject.Misrate);

    int n = x.Size;
    int m = y.Size;
    if (n < 2)
      throw AssumptionException.Domain(Subject.X);
    if (m < 2)
      throw AssumptionException.Domain(Subject.Y);

    double minShift = TwoSample(n, m);
    double minX = OneSample(n / 2);
    double minY = OneSample(m / 2);
    double minAvg = 2.0 * Math.Max(minX, minY);

    if (misrate < minShift + minAvg)
      throw AssumptionException.Domain(Subject.Misrate);

    double extra = misrate - (minShift + minAvg);
    double alphaShift = minShift + extra / 2.0;
    double alphaAvg = minAvg + extra / 2.0;

    if (FastSpread.Estimate(x.SortedValues, isSorted: true) <= 0)
      throw AssumptionException.Sparity(Subject.X);
    if (FastSpread.Estimate(y.SortedValues, isSorted: true) <= 0)
      throw AssumptionException.Sparity(Subject.Y);

    var shiftBounds = ShiftBoundsEstimator.Instance.Estimate(x, y, alphaShift);
    var avgBounds = seed == null
      ? AvgSpreadBoundsEstimator.Instance.Estimate(x, y, alphaAvg)
      : AvgSpreadBoundsEstimator.Instance.Estimate(x, y, alphaAvg, seed);

    double la = avgBounds.Lower;
    double ua = avgBounds.Upper;
    double ls = shiftBounds.Lower;
    double us = shiftBounds.Upper;
```

```
    if (la > 0.0)
    {
      double r1 = ls / la;
      double r2 = ls / ua;
      double r3 = us / la;
      double r4 = us / ua;
      double lower = Math.Min(Math.Min(r1, r2), Math.Min(r3, r4));
      double upper = Math.Max(Math.Max(r1, r2), Math.Max(r3, r4));
      return new Bounds(lower, upper, DisparityUnit.Instance);
    }

    if (ua <= 0.0)
    {
      if (ls == 0.0 && us == 0.0)
        return new Bounds(0.0, 0.0, DisparityUnit.Instance);
      if (ls >= 0.0)
        return new Bounds(0.0, double.PositiveInfinity, DisparityUnit.Instance);
      if (us <= 0.0)
        return new Bounds(double.NegativeInfinity, 0.0, DisparityUnit.Instance);
      return new Bounds(double.NegativeInfinity, double.PositiveInfinity,
DisparityUnit.Instance);
    }

    if (ls > 0.0)
      return new Bounds(ls / ua, double.PositiveInfinity, DisparityUnit.Instance);
    if (us < 0.0)
      return new Bounds(double.NegativeInfinity, us / ua, DisparityUnit.Instance);
    if (ls == 0.0 && us == 0.0)
      return new Bounds(0.0, 0.0, DisparityUnit.Instance);
    if (ls == 0.0 && us > 0.0)
      return new Bounds(0.0, double.PositiveInfinity, DisparityUnit.Instance);
    if (ls < 0.0 && us == 0.0)
      return new Bounds(double.NegativeInfinity, 0.0, DisparityUnit.Instance);

    return new Bounds(double.NegativeInfinity, double.PositiveInfinity,
DisparityUnit.Instance);
  }
}
```

2.6.2. Tests

$$\text{DisparityBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \frac{\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})}{\text{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})}$$

The DisparityBounds test suite contains 39 test cases (3 demo + 5 natural + 6 property + 5 edge + 5 misrate + 2 distro + 6 unsorted + 7 error). Since DisparityBounds returns bounds rather than a point estimate, tests validate that the bounds contain $\text{Disparity}(\mathbf{x}, \mathbf{y})$ and satisfy equivariance properties. Each test case output is a JSON object with `lower` and `upper` fields representing the interval bounds. Because the denominator (AvgSpreadBounds) uses randomized SpreadBounds, tests fix a `seed` to keep outputs deterministic.

Demo examples ($n = m = 30$, $n = m = 20$) — from manual introduction:

`demo-1`: $\mathbf{x} = (1, ..., 30)$, $\mathbf{y} = (21, ..., 50)$, $\text{misrate} = 0.02$
`demo-2`: $\mathbf{x} = (1, ..., 30)$, $\mathbf{y} = (21, ..., 50)$, $\text{misrate} = 0.005$, wider bounds (tighter misrate)
`demo-3`: $\mathbf{x} = (1, ..., 20)$, $\mathbf{y} = (5, ..., 24)$, $\text{misrate} = 0.05$

These cases illustrate how tighter misrates produce wider bounds.

Natural sequences ($\text{misrate} = 0.2$) — 5 tests:

`natural-10-10`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 10)$, bounds containing 0
`natural-10-15`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (1, ..., 15)$
`natural-15-10`: $\mathbf{x} = (1, ..., 15)$, $\mathbf{y} = (1, ..., 10)$
`natural-15-15`: $\mathbf{x} = (1, ..., 15)$, $\mathbf{y} = (1, ..., 15)$, bounds containing 0
`natural-20-20`: $\mathbf{x} = (1, ..., 20)$, $\mathbf{y} = (1, ..., 20)$, bounds containing 0

Property validation ($n = m = 10$, $\text{misrate} = 0.2$) — 6 tests:

`property-identity`: $\mathbf{x} = (0, 2, ..., 18)$, $\mathbf{y} = (0, 2, ..., 18)$, bounds must contain 0
`property-location-shift`: $\mathbf{x}$ and $\mathbf{y}$ shifted by constant, same bounds as identity (location invariance)
`property-scale-2x`: $\mathbf{x}$ and $\mathbf{y}$ scaled by 2, same bounds as identity (scale invariance)
`property-scale-neg`: $\mathbf{x}$ and $\mathbf{y}$ negated, bounds preserved (abs scaling)
`property-symmetry`: $\mathbf{x} = (1, ..., 10)$, $\mathbf{y} = (6, ..., 15)$, observed bounds
`property-symmetry-swapped`: $\mathbf{x}$ and $\mathbf{y}$ swapped, bounds negated (anti-symmetry)

Edge cases — boundary conditions (5 tests):

`edge-small`: $n = m = 6$, $\text{misrate} = 0.6$ (small samples)
`edge-negative`: negative values for both samples
`edge-mixed-signs`: mixed positive/negative values
`edge-wide-range`: extreme value range
`edge-asymmetric-10-20`: $n = 10$, $m = 20$ (unbalanced sizes)

Misrate variation ($\mathbf{x} = (1, ..., 20)$, $\mathbf{y} = (5, ..., 24)$) — 5 tests:

`misrate-2e-1`: $\text{misrate} = 0.2$

`misrate-1e-1`: $\text{misrate} = 0.1$
`misrate-5e-2`: $\text{misrate} = 0.05$
`misrate-2e-2`: $\text{misrate} = 0.02$
`misrate-1e-2`: $\text{misrate} = 0.01$

These tests validate monotonicity: smaller misrates produce wider bounds.

Distribution tests (misrate varies) — 2 tests:

`additive-20-20`: $n = m = 20$, $\underline{\text{Additive}}(10, 1)$
`uniform-20-20`: $n = m = 20$, $\underline{\text{Uniform}}(0, 1)$

Unsorted tests — verify independent sorting of $\mathbf{x}$ and $\mathbf{y}$ (6 tests):

`unsorted-reverse-x`: X reversed, Y sorted
`unsorted-reverse-y`: X sorted, Y reversed
`unsorted-reverse-both`: both reversed
`unsorted-shuffle-x`: X shuffled, Y sorted
`unsorted-shuffle-y`: X sorted, Y shuffled
`unsorted-wide-range`: wide value range, both unsorted

These tests validate that DisparityBounds produces identical results regardless of input order.

Error cases — inputs that violate assumptions (7 tests):

`error-empty-x`: $\mathbf{x} = ()$ (empty X array)
`error-empty-y`: $\mathbf{y} = ()$ (empty Y array)
`error-single-element-x`: $|\mathbf{x}| = 1$ (too few elements for pairing)
`error-single-element-y`: $|\mathbf{y}| = 1$ (too few elements for pairing)
`error-constant-x`: constant $\mathbf{x}$ violates sparity ($\text{Spread} = 0$)
`error-constant-y`: constant $\mathbf{y}$ violates sparity ($\text{Spread} = 0$)
`error-misrate-below-min`: misrate below minimum achievable

3. Randomization

3.1. Rng

All randomization in the toolkit is driven by Rng — a deterministic pseudorandom number generator. Creating a generator from a seed produces an object whose methods (UniformFloat, UniformInt, Sample, Resample, Shuffle) yield identical sequences across all seven supported languages.

This chapter starts with the generator object itself, then introduces the primitive draw (UniformFloat) and its integer counterpart (UniformInt), followed by sampling utilities.

$$r = \mathrm{Rng}(s)$$

Seed types — integer seed or string seed (hashed via FNV-1a)
Determinism — identical sequences across all supported languages
Period — $2^{256} - 1$

Example

`r = Rng("experiment-1")` — create generator from string seed
`r = Rng(42)` — create generator from integer seed

The underlying algorithm is xoshiro256++ seeded via SplitMix64. See UniformFloat → Algorithm for implementation details.

3.1.1. Implementation

```
using System;
using System.Collections.Generic;

namespace Pragmastat.Randomization;

/// <summary>
/// A deterministic random number generator.
/// </summary>
/// <remarks>
/// <para>
/// Rng uses xoshiro256++ internally and guarantees identical output sequences
/// across all Pragmastat language implementations when initialized with the same seed.
/// </para>
/// <para>
/// <b>Thread safety:</b> Rng instances are <b>not</b> thread-safe. Each thread
/// must use its own instance. Sharing an instance across threads without
/// external synchronization produces undefined (non-reproducible) output.
/// </para>
/// </remarks>
public sealed class Rng
{
  private readonly Xoshiro256PlusPlus _inner;

  /// <summary>
  /// Create a new Rng with system entropy (non-deterministic).
  /// </summary>
  public Rng()
      : this(DateTime.UtcNow.Ticks)
  {
  }

  /// <summary>
  /// Create a new Rng from an integer seed.
  /// The same seed always produces the same sequence of random numbers.
  /// </summary>
  /// <param name="seed">The seed value.</param>
  public Rng(long seed)
  {
    _inner = new Xoshiro256PlusPlus((ulong)seed);
  }

  /// <summary>
  /// Create a new Rng from a string seed.
  /// The string is hashed using FNV-1a to produce a numeric seed.
  /// </summary>
  /// <param name="seed">The string seed.</param>
  /// <exception cref="ArgumentNullException">Thrown if seed is null.</exception>
  public Rng(string seed)
  {
    if (seed == null)
      throw new ArgumentNullException(nameof(seed));
```

```
    _inner = new Xoshiro256PlusPlus(Fnv1a.Hash(seed));
  }

  // ========================================================================
  // Floating Point Methods
  // ========================================================================

  /// <summary>
  /// Generate a uniform random double in [0, 1).
  /// Uses 53 bits of precision for the mantissa.
  /// </summary>
  /// <returns>A random value in [0, 1).</returns>
  public double UniformDouble()
  {
    return _inner.UniformDouble();
  }

  /// <summary>
  /// Generate a uniform random double in [min, max).
  /// </summary>
  /// <param name="min">Minimum value (inclusive).</param>
  /// <param name="max">Maximum value (exclusive).</param>
  /// <returns>A random value in [min, max). Returns min if min >= max.</returns>
  public double UniformDouble(double min, double max)
  {
    return _inner.UniformDouble(min, max);
  }

  /// <summary>
  /// Generate a uniform random float in [0, 1).
  /// Uses 24 bits for float mantissa precision.
  /// </summary>
  /// <returns>A random value in [0, 1).</returns>
  public float UniformSingle()
  {
    return _inner.UniformSingle();
  }

  /// <summary>
  /// Generate a uniform random float in [min, max).
  /// </summary>
  /// <param name="min">Minimum value (inclusive).</param>
  /// <param name="max">Maximum value (exclusive).</param>
  /// <returns>A random value in [min, max). Returns min if min >= max.</returns>
  public float UniformSingle(float min, float max)
  {
    return _inner.UniformSingle(min, max);
  }

  // ========================================================================
  // Signed Integer Methods
  // ========================================================================
```

```
/// <summary>
/// Generate a uniform random long in [min, max).
/// </summary>
/// <remarks>
/// Uses modulo reduction which introduces slight bias for ranges that don't
/// evenly divide 2^64. This bias is negligible for statistical simulations
/// but not suitable for cryptographic applications.
/// </remarks>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random long in [min, max). Returns min if min >= max.</returns>
public long UniformInt64(long min, long max)
{
  return _inner.UniformInt64(min, max);
}

/// <summary>
/// Generate a uniform random int in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random int in [min, max). Returns min if min >= max.</returns>
public int UniformInt32(int min, int max)
{
  return _inner.UniformInt32(min, max);
}

/// <summary>
/// Generate a uniform random short in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random short in [min, max). Returns min if min >= max.</returns>
public short UniformInt16(short min, short max)
{
  return _inner.UniformInt16(min, max);
}

/// <summary>
/// Generate a uniform random sbyte in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random sbyte in [min, max). Returns min if min >= max.</returns>
[CLSCompliant(false)]
public sbyte UniformInt8(sbyte min, sbyte max)
{
  return _inner.UniformInt8(min, max);
}

// ================================================================
// Unsigned Integer Methods
// ================================================================
```

```
/// <summary>
/// Generate a uniform random ulong in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random ulong in [min, max). Returns min if min >= max.</returns>
[CLSCompliant(false)]
public ulong UniformUInt64(ulong min, ulong max)
{
  return _inner.UniformUInt64(min, max);
}

/// <summary>
/// Generate a uniform random uint in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random uint in [min, max). Returns min if min >= max.</returns>
[CLSCompliant(false)]
public uint UniformUInt32(uint min, uint max)
{
  return _inner.UniformUInt32(min, max);
}

/// <summary>
/// Generate a uniform random ushort in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random ushort in [min, max). Returns min if min >= max.</returns>
[CLSCompliant(false)]
public ushort UniformUInt16(ushort min, ushort max)
{
  return _inner.UniformUInt16(min, max);
}

/// <summary>
/// Generate a uniform random byte in [min, max).
/// </summary>
/// <param name="min">Minimum value (inclusive).</param>
/// <param name="max">Maximum value (exclusive).</param>
/// <returns>A random byte in [min, max). Returns min if min >= max.</returns>
public byte UniformByte(byte min, byte max)
{
  return _inner.UniformByte(min, max);
}

// ================================================================================
// Boolean Methods
// ================================================================================

/// <summary>
```

```
  /// Generate a uniform random boolean with P(true) = 0.5.
  /// </summary>
  /// <returns>A random boolean.</returns>
  public bool UniformBool()
  {
    return _inner.UniformBool();
  }

  // ============================================================================
  // Collection Methods
  // ============================================================================

  /// <summary>
  /// Sample k elements from the input list without replacement.
  /// Uses selection sampling to maintain order of first appearance.
  /// Returns all elements if k >= x.Count.
  /// </summary>
  /// <typeparam name="T">Element type.</typeparam>
  /// <param name="x">Input list to sample from.</param>
  /// <param name="k">Number of elements to sample. Must be positive.</param>
  /// <returns>List of k sampled elements.</returns>
  /// <exception cref="ArgumentOutOfRangeException">Thrown if k is not positive.</
exception>
  public List<T> Sample<T>(IReadOnlyList<T> x, int k)
  {
    if (k <= 0)
      throw new ArgumentOutOfRangeException(nameof(k), k, "k must be positive");
    if (x.Count == 0)
      throw new ArgumentException("Cannot sample from empty list", nameof(x));

    int n = x.Count;
    if (k >= n)
    {
      return new List<T>(x);
    }

    var result = new List<T>(k);
    int remaining = k;

    for (int i = 0; i < n && remaining > 0; i++)
    {
      int available = n - i;
      // Probability of selecting this item: remaining / available
      if (UniformDouble() * available < remaining)
      {
        result.Add(x[i]);
        remaining--;
      }
    }

    return result;
  }
```

```
  /// <summary>
  /// Sample k elements from the sample values without replacement.
  /// </summary>
  /// <param name="sample">Input sample to sample from.</param>
  /// <param name="k">Number of elements to sample. Must be positive.</param>
  /// <returns>New sample with k sampled values.</returns>
  /// <exception cref="ArgumentOutOfRangeException">Thrown if k is not positive.</
exception>
  public Sample Sample(Sample sample, int k)
  {
    if (k <= 0)
      throw new ArgumentOutOfRangeException(nameof(k), k, "k must be positive");
    if (sample.IsWeighted)
      throw new NotSupportedException("Weighted samples are not supported by Rng.Sample");
    int n = sample.Size;
    if (k >= n)
    {
      return sample;
    }

    var values = new List<double>(k);
    int remaining = k;

    for (int i = 0; i < n && remaining > 0; i++)
    {
      int available = n - i;
      if (UniformDouble() * available < remaining)
      {
        values.Add(sample.Values[i]);
        remaining--;
      }
    }

    return new Sample(values, sample.Unit);
  }

  // ========================================================================
  // Bootstrap (With-Replacement) Methods
  // ========================================================================

  /// <summary>
  /// Resample k elements from the input list with replacement (bootstrap sampling).
  /// </summary>
  /// <typeparam name="T">Element type.</typeparam>
  /// <param name="x">Input list to sample from.</param>
  /// <param name="k">Number of elements to sample. Must be positive.</param>
  /// <returns>List of k sampled elements (may contain duplicates).</returns>
  /// <exception cref="ArgumentOutOfRangeException">Thrown if k is not positive.</
exception>
  /// <exception cref="ArgumentException">Thrown if input list is empty.</exception>
  public List<T> Resample<T>(IReadOnlyList<T> x, int k)
  {
    if (k <= 0)
```

```
      throw new ArgumentOutOfRangeException(nameof(k), k, "k must be positive");
    if (x.Count == 0)
      throw new ArgumentException("Cannot resample from empty list", nameof(x));

    var result = new List<T>(k);
    for (int i = 0; i < k; i++)
      result.Add(x[(int)UniformInt64(0, x.Count)]);
    return result;
  }

  /// <summary>
  /// Resample k elements from the sample values with replacement (bootstrap sampling).
  /// </summary>
  /// <param name="sample">Input sample to resample from.</param>
  /// <param name="k">Number of elements to sample. Must be positive.</param>
  /// <returns>New sample with k resampled values (may contain duplicates).</returns>
  /// <exception cref="ArgumentOutOfRangeException">Thrown if k is not positive.</
exception>
  public Sample Resample(Sample sample, int k)
  {
    if (k <= 0)
      throw new ArgumentOutOfRangeException(nameof(k), k, "k must be positive");
    if (sample.IsWeighted)
      throw new NotSupportedException("Weighted samples are not supported by
Rng.Resample");
    if (sample.Size == 0)
      throw new ArgumentException("Cannot resample from empty sample", nameof(sample));

    var values = new List<double>(k);
    for (int i = 0; i < k; i++)
      values.Add(sample.Values[(int)UniformInt64(0, sample.Size)]);
    return new Sample(values, sample.Unit);
  }

  /// <summary>
  /// Return a shuffled copy of the input list.
  /// Uses the Fisher-Yates shuffle algorithm for uniform distribution.
  /// The original list is not modified.
  /// </summary>
  /// <typeparam name="T">Element type.</typeparam>
  /// <param name="x">Input list to shuffle.</param>
  /// <returns>Shuffled copy of the input.</returns>
  public List<T> Shuffle<T>(IReadOnlyList<T> x)
  {
    if (x.Count == 0)
      throw new ArgumentException("Cannot shuffle empty list", nameof(x));
    var result = new List<T>(x);
    int n = result.Count;

    // Fisher-Yates shuffle (backwards)
    for (int i = n - 1; i > 0; i--)
    {
      int j = (int)UniformInt64(0, i + 1);
```

```
      (result[i], result[j]) = (result[j], result[i]);
    }

    return result;
  }

  /// <summary>
  /// Return a shuffled copy of the sample values.
  /// </summary>
  /// <param name="sample">Input sample to shuffle.</param>
  /// <returns>New sample with shuffled values.</returns>
  public Sample Shuffle(Sample sample)
  {
    if (sample.IsWeighted)
      throw new NotSupportedException("Weighted samples are not supported by
Rng.Shuffle");
    var shuffled = Shuffle(sample.Values);
    return new Sample(shuffled, sample.Unit);
  }
}
```

3.1.2. Tests

The Rng test suite contains 55 test cases validating the deterministic pseudo-random number generator across seven output categories. All tests verify reproducibility: given the same seed, every language implementation must produce identical sequences. Seeds can be integers or strings (string seeds are hashed to produce an integer seed).

uniform-seed (10 tests) — base $\underline{\text{Uniform}}(0, 1)$ generation from integer seeds:

Seeds: -2147483648, -42, -1, 0, 1, 123, 999, 1729, 12345, 2147483647
Each generates 20 values in $[0, 1)$
Covers int32 boundary values (min, max), negative seeds, and common seeds

uniform-f32 (7 tests) — single-precision $\underline{\text{Uniform}}(0, 1)$ generation:

Seeds: -42, -1, 0, 1, 123, 999, 1729
Each generates 20 values in $[0, 1)$ at f32 precision
Validates that f32 output matches the truncated f64 sequence

uniform-bool (7 tests) — boolean generation ($\underline{\text{Uniform}} < 0.5$):

Seeds: -42, -1, 0, 1, 123, 999, 1729
Each generates 100 boolean values
Validates the threshold-based boolean conversion

uniform-range (7 tests) — $\underline{\text{Uniform}}(\text{min}, \text{max})$ generation with real-valued bounds:

Seeds and ranges: seed 0 with $[-1, 1]$; seed 123 with $[0, 1]$; seed 999 with $[0, 100]$; seed 1729 with $[-1, 1]$, $[-50, 50]$, $[0, 1]$, $[0, 100]$
Each generates 20 values in $[\text{min}, \text{max})$
Validates affine transformation of base uniform

uniform-int (8 tests) — uniform integer generation in $[\text{min}, \text{max})$:

Seeds and ranges: seed -42 with $[0, 100]$; seed 0 with $[0, 100]$; seed 123 with $[0, 100]$; seed 999 with $[-100, 100]$; seed 1729 with $[-50, 50]$, $[0, 10]$, $[0, 100]$, $[1000, 2000]$
Each generates 20 integer values
Validates modulo reduction of raw 64-bit output

uniform-i32 (5 tests) — 32-bit signed integer generation:

Seeds and ranges: seed 0 with $[-500, 500]$; seed 123 with $[0, 1000]$; seed 999 with $[0, 100]$; seed 1729 with $[-500, 500]$ and $[0, 1000]$
Each generates 20 values
Validates i32-specific truncation behavior

uniform-string (11 tests) — string-seeded $\underline{\text{Uniform}}(0, 1)$ generation:

Seeds: `""` (empty), `"a"`, `"abc"`, `"test"`, `"hello_world"`, `"pragmastat"`, `"Rng"`, `"experiment-1"`, plus 3 UTF-8 seeds (π, `"hello"` in Chinese, `"hello"` in German)

Each generates 20 values in $[0, 1)$
Validates string-to-seed hashing, including empty strings, case sensitivity, and multi-byte UTF-8

3.1.3. References

Blackman, D., & Vigna, S. (2021). Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software*, *47*(4), 1–32. https://dl.acm.org/doi/10.1145/3460772

Steele, G. L., Lea, D., & Flood, C. H. (2014). *Fast Splittable Pseudorandom Number Generators* (pp. 453–472). ACM. https://dl.acm.org/doi/10.1145/2660193.2660195

Fowler, G., Noll, L. C., & Vo, K.-P. (1991,). *FNV Hash*. http://www.isthe.com/chongo/tech/comp/fnv/

3.2. UniformFloat

$$r.\,\mathrm{UniformFloat}()$$

Draw a uniform random value in $[0, 1)$ using generator r. UniformFloat is the primitive draw; all other randomization functions and distribution samplers build on top of it.

Distribution — $\underline{\mathrm{Uniform}}(0, 1)$
Range — $[0, 1)$ (includes 0, excludes 1)
Precision — 53-bit mantissa (2^{53} distinct values)
Complexity — $O(1)$ per draw

Properties

Determinism same generator state produces same value
Independence successive draws are uncorrelated
Uniformity all representable values in $[0, 1)$ equally likely

Example (conceptual)

Call — `Rng("demo").UniformFloat()` (conceptual name; see mapping below)
Repeat — 10 successive calls produce 10 independent values

Implementation names

Language	Method
C#	`UniformDouble()`
Go	`UniformFloat64()`
Kotlin	`uniformDouble()`
Rust	`uniform_f64()`
Python	`uniform_float()`
R	`uniform_float()`
TypeScript	`uniformFloat()`

UniformFloat is the fundamental operation of the random number generator. All other randomization functions — Sample, Shuffle, Resample, and the distribution samplers — are built on top of uniform draws. See Naming for why the toolkit uses the name UniformFloat instead of the traditional `.Next()`.

3.2.1. Algorithm

Generator Core

The core random number generator uses the xoshiro256++ algorithm (see Blackman & Vigna (2021)), a member of the xoshiro/xoroshiro family developed by David Blackman and Sebastiano Vigna. This algorithm was selected for several reasons:

Quality: passes all tests in the BigCrush test suite from TestU01
Speed: extremely fast due to simple bitwise operations (shifts, rotations, XORs)
Period: period of $2^{256} - 1$, sufficient for parallel simulations
Adoption: used by .NET 6+, Julia, and Rust's rand crate

The generator maintains a 256-bit state (s_0, s_1, s_2, s_3) and produces 64-bit outputs. Each step updates the state through a combination of XOR, shift, and rotate operations.

Seed Initialization

Converting a single seed value into the full 256-bit state requires a seeding algorithm. The toolkit uses SplitMix64 (see Steele et al. (2014)) for this purpose:

$$\begin{aligned} x &\leftarrow x + \text{0x9e3779b97f4a7c15} \\ z &\leftarrow (x \oplus (x \gg 30)) \times \text{0xbf58476d1ce4e5b9} \\ z &\leftarrow (z \oplus (z \gg 27)) \times \text{0x94d049bb133111eb} \\ \text{output} &\leftarrow z \oplus (z \gg 31) \end{aligned}$$

Four consecutive outputs from SplitMix64 initialize the xoshiro256++ state (s_0, s_1, s_2, s_3). This approach provides high-quality initial states from simple integer seeds.

String Seeds

For named experiments (e.g., $\text{Rng}(\text{experiment-1})$), string seeds are converted to integers using FNV-1a hash (see Fowler et al. (1991)):

$$\begin{aligned} &\text{hash} \leftarrow \text{0xcbf29ce484222325} \quad (\text{offset basis}) \\ \text{for each byte } b: \quad &\text{hash} \leftarrow (\text{hash} \oplus b) \times \text{0x00000100000001b3} \quad (\text{FNV prime}) \end{aligned}$$

This enables meaningful experiment identifiers while maintaining determinism.

UniformFloat Generation

To generate uniform values in $[0, 1)$, the upper 53 bits of a 64-bit output are used:

$$r.\,\text{UniformFloat}() = (\text{next}() \gg 11) \times 2^{-53}$$

The 53-bit mantissa of IEEE 754 double precision ensures all representable values in $[0, 1)$ are reachable.

UniformInt Mapping

$\mathrm{UniformInt}$ maps a uniform 64-bit value into $[a, b)$ using modulo reduction. Ranges that do not divide 2^{64} introduce a slight bias (acceptable for simulation, not for cryptographic use).

Numerical Constants

Distribution sampling requires handling edge cases where floating-point operations would be undefined. Two constants are used across all language implementations:

Machine Epsilon ($\varepsilon_{\text{mach}}$): The smallest ε such that $1 + \varepsilon \neq 1$ in float64 arithmetic.

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16}$$

Used when $U = 1$ to avoid $\ln(0)$ or division by zero in inverse transform sampling.

Smallest Positive Subnormal (ε_{sub}): The smallest positive value representable in IEEE 754 binary64.

$$\varepsilon_{\text{sub}} = 2^{-1074} \approx 4.94 \times 10^{-324}$$

Used when $U = 0$ to avoid $\ln(0)$ in transforms that take a logarithm.

All language implementations use the same literal values for these constants (not language-specific builtins like `Number.EPSILON` or `f64::EPSILON`) to ensure bit-identical outputs across languages.

Distribution Sampling

All distribution samplers consume one or more $\mathrm{UniformFloat}$ draws. The specific transforms are documented on the corresponding distribution pages.

3.2.2. Notes

Most programming languages expose the primary PRNG operation as `.Next()`, `.random()`, or `rand()` — a method name that describes the **mechanism** (advance the internal state and return a value) rather than the **result** (a uniformly distributed number on $[0, 1)$).

This toolkit names the operations UniformFloat and UniformInt — combining the distribution name with the return type. The reasons are both pedagogical and practical. In actual code the method names are language-specific and type-suffixed; see the UniformFloat and UniformInt pages for the mapping.

The name communicates the contract. Calling $r.\,\mathrm{UniformFloat}()$ immediately tells the reader what distribution the returned value follows and what type it produces. Calling `r.Next()` says only that something comes next; the distribution, range, and precision are left to documentation. In a library that manipulates multiple distributions (Additive, Multiplic, Exp, Power, Uniform), naming the uniform draw explicitly makes it a peer of the other distributions rather than a special primitive hidden behind a generic verb.

The name prevents a category error. When `random()` returns a value in $[0, 1)$, users sometimes treat it as "a random number" without recognizing that it samples from a specific distribution. Making the distribution explicit in the name reinforces that Uniform is one choice among many and that other distributions require different transformations.

The name preserves the URL namespace. Using UniformFloat for the function frees `/uniform` for the Uniform distribution page, avoiding ambiguity between the function (which draws a single value) and the distribution family (which defines the parametric model).

Composition becomes self-documenting. When a distribution is built from uniform draws, the code reads naturally as "take UniformFloat draws and apply a transformation," keeping the distribution explicit. Replacing UniformFloat with `.Next()` in these descriptions obscures the mathematical structure.

Precedent. Scientific computing libraries (NumPy's `random.uniform`, R's `runif`, Julia's `rand(Uniform())`) already use "uniform" when the distribution matters. The toolkit follows this convention consistently: every random draw is named after its distribution, starting with the simplest one.

3.2.3. References

Blackman, D., & Vigna, S. (2021). Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software*, *47*(4), 1–32. https://dl.acm.org/doi/10.1145/3460772

Steele, G. L., Lea, D., & Flood, C. H. (2014). *Fast Splittable Pseudorandom Number Generators* (pp. 453–472). ACM. https://dl.acm.org/doi/10.1145/2660193.2660195

Fowler, G., Noll, L. C., & Vo, K.-P. (1991,). *FNV Hash*. http://www.isthe.com/chongo/tech/comp/fnv/

3.3. UniformInt

$$r.\,\mathrm{UniformInt}(a, b)$$

Generate a uniform random integer in $[a, b)$ using generator r. This draw is derived from the underlying uniform float stream.

Range — $[a, b)$ (includes a, excludes b)
Complexity — $O(1)$ per draw

Example (conceptual)

Call — `Rng(42).UniformInt(a, b)` (conceptual name; see mapping below)
Range — integer in $[0, 100)$ or $[-50, 50)$

Implementation names

Language	Method
C#	`UniformInt32()` / `UniformInt64()` / `UniformInt16()` / `UniformInt8()`
Go	`UniformIntN()` / `UniformInt64()` / `UniformInt32()` / `UniformInt16()` / `UniformInt8()`
Kotlin	`uniformInt()` / `uniformLong()` / `uniformShort()` / `uniformByte()`
Rust	`uniform_i32()` / `uniform_i64()` / `uniform_i16()` / `uniform_i8()`
Python	`uniform_int()`
R	`uniform_int()`
TypeScript	`uniformInt()`

UniformInt draws are derived from the same generator core as UniformFloat and map a uniform 64-bit value into $[a, b)$. The implementation uses modulo reduction; ranges that do not divide 2^{64} introduce a slight bias (acceptable for simulation, not for cryptographic use). See UniformFloat → Algorithm for the core generator details.

Unsigned variants are available in languages that support them.

3.4. Sample

$$r.\,\mathrm{Sample}(\mathbf{x}, k)$$

Select k elements from sample $\mathbf{x}$ without replacement using generator r.

Algorithm — selection sampling (Fan, Muller, Rezucha 1962), see Sample
Complexity — $O(n)$ time, single pass
Output — preserves original order of selected elements
Domain — $k \geq 0$ (clamped to n if $k > n$)

Notation

$\mathbf{x} = (x_1, \ldots, x_n)$ sample ($n \geq 0$)
x_i individual measurements

Properties

Simple random sample each k-subset has equal probability
Order preservation selected elements appear in order of first occurrence
Determinism same generator state produces same selection

Example

`Rng("demo-sample").Sample([1, 2, 3, 4, 5], 3)` — select 3 elements
`r.Sample(x, n) = x` — selecting all elements returns original order

Implementation names

Language	Method
C#	`Rng.Sample()`
Go	`Sample()`
Kotlin	`Rng.sample()`
Rust	`Rng::sample()`
Python	`Rng.sample()`
R	`rng$sample()`
TypeScript	`Rng.sample()`

Sample picks a random subset of data without replacement. Common uses include random subsetting, creating cross-validation splits, or reducing a large dataset to a manageable size. Every possible subset of size k has equal probability of being selected, and the selected elements keep their original order. To make your subsampling reproducible, combine it with a seeded generator: `Sample(data, 100, Rng("training-set"))` will always select the same 100 elements.

3.4.1. Algorithm

The Sample function uses selection sampling (see Fan et al. (1962)) to select k elements from n without replacement.

The algorithm makes a single pass through the data, deciding independently for each element whether to include it, using the Rng generator for random decisions:

```
seen = 0, selected = 0
for each element x at position i:
    if uniform() < (k - selected) / (n - seen):
        output x
        selected += 1
    seen += 1
```

This algorithm preserves the original order of elements (order of first appearance) and requires only a single pass through the data. Each element is selected independently with the correct marginal probability, producing a simple random sample.

3.4.2. Tests

$$\text{Sample}(\text{seed}, \mathbf{x}, k)$$

The Sample test suite contains 15 test cases validating sampling without replacement. Given a seed, input array x of size n, and draw count k, Sample returns k distinct elements from x, preserving their original order. All tests verify reproducibility: the same seed, input, and k must produce the same output across all language implementations.

Seed variation ($n = 10, k = 3$) — 3 tests with different seeds:

`seed-0-n10-k3`: seed $= 0$
`seed-123-n10-k3`: seed $= 123$
`seed-999-n10-k3`: seed $= 999$

These tests validate that different seeds produce different samples from the same input.

Parameter variation (seed $= 1729$) — 12 tests exploring n and k:

`seed-1729-n1-k1`: $n = 1, k = 1$ (trivial case, single element)
`seed-1729-n2-k1`: $n = 2, k = 1$ (draw one from two)
`seed-1729-n5-k3`: $n = 5, k = 3$ (standard draw)
`seed-1729-n10-k1`: $n = 10, k = 1$ (single draw from many)
`seed-1729-n10-k3`: $n = 10, k = 3$ (standard draw)
`seed-1729-n10-k5`: $n = 10, k = 5$ (half draw)
`seed-1729-n10-k10`: $n = 10, k = 10$ (full permutation)
`seed-1729-n10-k15`: $n = 10, k = 15$ ($k > n$, clamped to n)
`seed-1729-n20-k5`: $n = 20, k = 5$
`seed-1729-n20-k10`: $n = 20, k = 10$
`seed-1729-n100-k10`: $n = 100, k = 10$ (large pool, small draw)
`seed-1729-n100-k25`: $n = 100, k = 25$ (large pool, moderate draw)

The progression from $k = 1$ to $k = n$ to $k > n$ validates boundary handling. When $k \geq n$, the result is a copy of x in its original order.

Seed-based validation — All seed $= 1729$ tests share the same underlying RNG state. Cross-seed tests ($0, 123, 999$) confirm that different seeds yield different permutation sequences.

3.4.3. References

Fan, C. T., Muller, M. E., & Rezucha, I. (1962). Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers. *Journal of the American Statistical Association*, *57*(298), 387–402. https://www.tandfonline.com/doi/abs/10.1080/01621459.1962.10480667

3.5. Resample

$$r.\,\mathrm{Resample}(\mathbf{x}, k)$$

Select k elements from sample $\mathbf{x}$ with replacement using generator r.

Algorithm — uniform sampling with replacement, see Resample
Complexity — $O(k)$ time
Output — new array with k elements (may contain duplicates)
Domain — $k \geq 0$, sample size $n \geq 1$

Notation

$\mathbf{x} = (x_1, ..., x_n)$ sample $(n \geq 1)$
x_i individual measurements

Properties

Independence each selection is independent with equal probability $1/n$
Duplicates same element may appear multiple times in output
Determinism same generator state produces same selection

Example

`Rng("demo-resample").Resample([1, 2, 3, 4, 5], 3)` — select 3 with replacement
`r.Resample(x, n)` — bootstrap sample of same size as original

Implementation names

Language	Method
C#	`Rng.Resample()`
Go	`Resample()`
Kotlin	`Rng.resample()`
Rust	`Rng::resample()`
Python	`Rng.resample()`
R	`rng$resample()`
TypeScript	`Rng.resample()`

Resample picks elements with replacement, allowing the same element to be selected multiple times. This is essential for bootstrap methods where we simulate new samples from the observed data. Unlike Sample (without replacement), Resample can produce outputs larger than the input and will typically contain duplicate values. For reproducible bootstrap, combine with a seeded generator: `Resample(data, n, Rng("bootstrap-1"))`.

3.5.1. Algorithm

The $\mathrm{Resample}$ function selects k elements from a sample of size n with replacement.

The algorithm generates k independent uniform random integers in $[0, n)$ using the Rng generator, and collects the corresponding elements:

```
result = new array of size k
for i from 0 to k-1:
    j = uniform_int(0, n)
    result[i] = x[j]
```

Each selection is independent with equal probability $1/n$ for every element. The same element may appear multiple times in the output. Time complexity is $O(k)$ with $O(k)$ additional space for the result array.

3.5.2. Tests

$$\text{Resample}(\text{seed}, \mathbf{x}, k)$$

The Resample test suite contains 19 test cases validating sampling with replacement (bootstrap resampling). Given a seed, input array x of size n, and draw count k, Resample returns k elements drawn independently and uniformly from x (with replacement, so duplicates are possible). All tests verify reproducibility: the same seed, input, and k must produce the same output across all language implementations.

Seed variation — 6 tests with different seeds:

`seed-0-n10-k3`: seed $= 0$, $n = 10$, $k = 3$
`seed-42-n10-k5`: seed $= 42$, $n = 10$, $k = 5$
`seed-123-n10-k3`: seed $= 123$, $n = 10$, $k = 3$
`seed-314-n10-k10`: seed $= 314$, $n = 10$, $k = 10$
`seed-999-n10-k3`: seed $= 999$, $n = 10$, $k = 3$
`seed-2718-n100-k25`: seed $= 2718$, $n = 100$, $k = 25$

These tests validate that different seeds produce different bootstrap samples from the same input.

Parameter variation (seed $= 1729$) — 13 tests exploring n and k:

`seed-1729-n1-k1`: $n = 1$, $k = 1$ (trivial case)
`seed-1729-n2-k1`: $n = 2$, $k = 1$ (single draw from two)
`seed-1729-n5-k3`: $n = 5$, $k = 3$ (standard draw)
`seed-1729-n5-k7`: $n = 5$, $k = 7$ ($k > n$, valid for resampling)
`seed-1729-n10-k1`: $n = 10$, $k = 1$ (single draw from many)
`seed-1729-n10-k3`: $n = 10$, $k = 3$ (standard draw)
`seed-1729-n10-k5`: $n = 10$, $k = 5$ (half draw)
`seed-1729-n10-k10`: $n = 10$, $k = 10$ (draw equal to pool size)
`seed-1729-n10-k15`: $n = 10$, $k = 15$ ($k > n$, exercises repeated sampling)
`seed-1729-n20-k5`: $n = 20$, $k = 5$
`seed-1729-n20-k10`: $n = 20$, $k = 10$
`seed-1729-n100-k10`: $n = 100$, $k = 10$ (large pool, small draw)
`seed-1729-n100-k25`: $n = 100$, $k = 25$ (large pool, moderate draw)

Unlike Sample (without replacement), Resample allows $k > n$ since each draw is independent. The $k > n$ cases (`n5-k7`, `n10-k15`) are unique to resampling and validate that the output can contain repeated values from the input.

3.5.3. References

Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, *7*(1), 1–26. https://projecteuclid.org/euclid.aos/1176344552

3.6. Shuffle

$$r.\,\mathrm{Shuffle}(\mathbf{x})$$

Uniformly random permutation of sample $\mathbf{x}$ using generator r.

Algorithm — Fisher-Yates (Knuth shuffle), see Shuffle
Complexity — $O(n)$ time, $O(n)$ space (returns new array)
Output — new array (does not modify input)

Properties

Uniformity each of $n!$ permutations has equal probability
Determinism same generator state produces same permutation

Example

`Rng("demo-shuffle").Shuffle([1, 2, 3, 4, 5])` — shuffled copy
`r.Shuffle(x)` preserves multiset (same elements, different order)

Implementation names

Language	Method
C#	`Rng.Shuffle()`
Go	`Shuffle()`
Kotlin	`Rng.shuffle()`
Rust	`Rng::shuffle()`
Python	`Rng.shuffle()`
R	`rng$shuffle()`
TypeScript	`Rng.shuffle()`

Shuffle produces a random reordering of data. This is essential for permutation tests and useful for eliminating any bias from the original ordering. Every possible arrangement has exactly equal probability, which is required for valid statistical inference. The function returns a new shuffled array and leaves the original data unchanged. For reproducible results, pass a seeded generator: `Shuffle(data, Rng("experiment-1"))` will always produce the same permutation.

3.6.1. Algorithm

The Shuffle function uses the Fisher-Yates algorithm (see Fisher & Yates (1938), Knuth (1997)), also known as the Knuth shuffle, with the Rng generator for random decisions:

```
for i from n-1 down to 1:
    j = uniform_int(0, i+1)
    swap(array[i], array[j])
```

This produces a uniformly random permutation in $O(n)$ time with $O(n)$ additional space (the input is copied). The algorithm is unbiased: each of the $n!$ permutations has equal probability.

3.6.2. Tests

$$\text{Shuffle}(\text{seed}, \mathbf{x})$$

The Shuffle test suite contains 12 test cases validating random permutation. Given a seed and input array $\mathbf{x}$, Shuffle returns a permutation of $\mathbf{x}$ using the Fisher–Yates algorithm. All tests verify reproducibility: the same seed and input must produce the same permutation across all language implementations.

Seed variation ($n = 5$, $\mathbf{x} = (1, 2, 3, 4, 5)$) — 3 tests with different seeds:

`seed-0-n5-basic`: seed $= 0$
`seed-123-n5-basic`: seed $= 123$
`seed-999-n5-basic`: seed $= 999$

These tests validate that different seeds produce different permutations of the same input.

Fixed seed (seed $= 1729$) — 9 tests exploring different input sizes and content:

`seed-1729-n1-single`: $\mathbf{x} = (1)$ (trivial case, single element)
`seed-1729-n2-basic`: $\mathbf{x} = (1, 2)$ (minimum non-trivial case)
`seed-1729-n5-basic`: $\mathbf{x} = (1, 2, 3, 4, 5)$ (standard case)
`seed-1729-n5-zeros`: $\mathbf{x} = (0, 0, 0, 0, 0)$ (all identical, permutation preserves content)
`seed-1729-n6-neg`: $\mathbf{x} = (-5, -3, -1, 1, 3, 5)$ (negative and positive values)
`seed-1729-n10-seq`: $\mathbf{x} = (0, 1, \ldots, 9)$ (10-element sequential)
`seed-1729-n20-seq`: $\mathbf{x} = (0, 1, \ldots, 19)$ (20-element sequential)
`seed-1729-n100-seq`: $\mathbf{x} = (0, 1, \ldots, 99)$ (large array)
`seed-123-n10-seq`: seed $= 123$, $\mathbf{x} = (0, 1, \ldots, 9)$ (different seed, same size as n10-seq)

The progression from $n = 1$ to $n = 100$ validates that the Fisher–Yates implementation scales correctly. The zero-valued and negative-valued tests verify that shuffling operates on positions, not values.

3.6.3. References

Fisher, R. A., & Yates, F. (1938). *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (3rd ed.). Addison-Wesley.

4. Distributions

Distributions are parametrized random generators with well-defined statistical properties. Each distribution describes a family of random variables characterized by specific parameters.

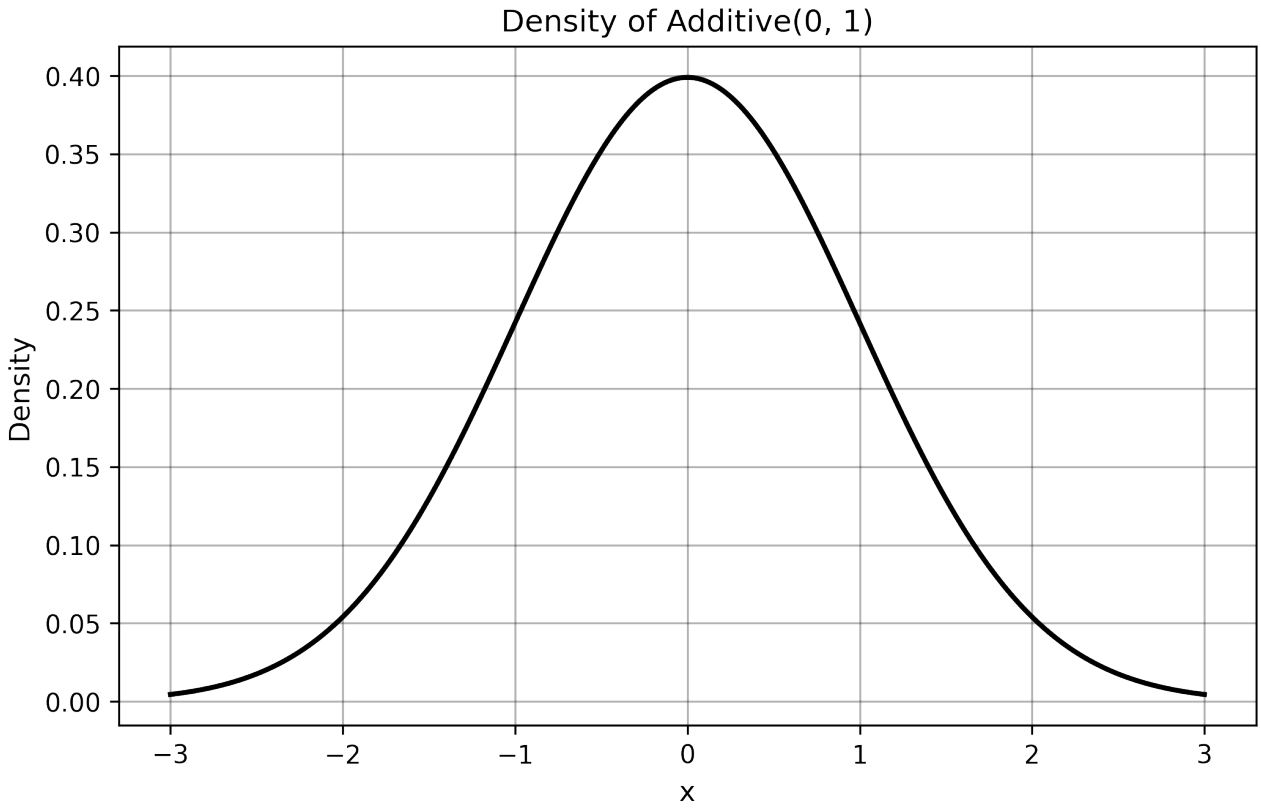
Notation

$X \sim \underline{\text{Additive}}(0, 1)$ — X is distributed as standard normal
$\text{Estimator}(\mathbf{x})$ — estimate computed from sample
$\text{Estimator}[X]$ — true value (asymptotic limit)
$n \to \infty$ — asymptotic case (large sample approximation)

4.1. Additive ('Normal')

$$\underline{\text{Additive}}(\text{mean}, \text{stdDev})$$

mean: location parameter (center of the distribution), consistent with Center
stdDev: scale parameter (standard deviation), can be rescaled to Spread



Formation: the sum of many variables $X_1 + X_2 + ... + X_n$ under mild CLT (Central Limit Theorem) conditions (e.g., Lindeberg-Feller).
Origin: historically called 'Normal' or 'Gaussian' distribution after Carl Friedrich Gauss and others.
Rename Motivation: renamed to Additive to reflect its formation mechanism through addition.
Properties: symmetric, bell-shaped, characterized by central limit theorem convergence.

Applications: measurement errors, heights and weights in populations, test scores, temperature variations.
Characteristics: symmetric around the mean, light tails, finite variance.
Caution: no perfectly additive distributions exist in real data; all real-world measurements contain deviations. Traditional estimators like Mean and StdDev lack robustness to outliers; use them only when strong evidence supports approximate additivity with no extreme measurements.

4.1.1. Notes

The Additive ('Normal') distribution has two parameters: the mean and the standard deviation, written as $\underline{\mathrm{Additive}}(\mathrm{mean}, \mathrm{stdDev})$.

4.1.1.1. Sampling (Box-Muller Transform)

The toolkit samples Additive values using the Box-Muller transform (see Box & Muller (1958)), which converts two independent $\mathrm{UniformFloat}$ draws into standard normal values. Given $U_1, U_2 \in [0, 1)$:

$$Z_0 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$$
$$Z_1 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2)$$

Both Z_0 and Z_1 are independent standard normal values. The implementation uses only Z_0 to maintain cross-language determinism.

4.1.1.2. Asymptotic Spread Value

Consider two independent draws X and Y from the $\underline{\mathrm{Additive}}(\mathrm{mean}, \mathrm{stdDev})$ distribution. The goal is to find the median of their absolute difference $|X - Y|$. Define the difference $D = X - Y$. By linearity of expectation, $\mathbb{E}[D] = 0$. By independence, $\mathrm{Var}[D] = 2 \cdot \mathrm{stdDev}^2$. Thus D has distribution $\underline{\mathrm{Additive}}\left(0, \sqrt{2} \cdot \mathrm{stdDev}\right)$, and the problem reduces to finding the median of $|D|$. The location parameter mean disappears, as expected, because absolute differences are invariant to shifts.

Let $\tau = \sqrt{2} \cdot \mathrm{stdDev}$, so that $D \sim \underline{\mathrm{Additive}}(0, \tau)$. The random variable $|D|$ then follows the Half-Additive ('Folded Normal') distribution with scale τ. Its cumulative distribution function for $z \geq 0$ becomes

$$F_{|D|}(z) = \Pr(|D| \leq z) = 2\Phi(z/\tau) - 1$$

where Φ denotes the standard Additive ('Normal') CDF.

The median m is the point at which this cdf equals $1/2$. Setting $F_{|D|}(m) = 1/2$ gives

$$2\Phi(m/\tau) - 1 = 1/2 \Rightarrow \Phi(m/\tau) = 3/4$$

Applying the inverse cdf yields $m/\tau = z_{0.75}$. Substituting back $\tau = \sqrt{2} \cdot \mathrm{stdDev}$ produces

$$\mathrm{Median}(|X - Y|) = \sqrt{2} \cdot z_{0.75} \cdot \mathrm{stdDev}$$

Define $z_{0.75} := \Phi^{-1}(0.75) \approx 0.6744897502$. Numerically, the median absolute difference is approximately $\sqrt{2} \cdot z_{0.75} \cdot \mathrm{stdDev} \approx 0.9538725524 \cdot \mathrm{stdDev}$. This expression depends only on the scale parameter stdDev, not on the mean, reflecting the translation invariance of the problem.

4.1.1.3. Lemma: Average Estimator Drift Formula

For average estimators T_n with asymptotic standard deviation $a \cdot \mathrm{stdDev}\ /\sqrt{n}$ around the mean μ, define $\mathrm{RelSpread}[T_n] := \mathrm{Spread}[T_n]/\mathrm{Spread}[X]$. In the Additive ('Normal') case, $\mathrm{Spread}[X] = \sqrt{2} \cdot z_{0.75} \cdot \mathrm{stdDev}$.

For any average estimator T_n with asymptotic standard deviation $a \cdot \mathrm{stdDev}\ /\sqrt{n}$ around the mean μ, the drift calculation follows:

The spread of two independent estimates: $\mathrm{Spread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a \cdot \mathrm{stdDev}\ /\sqrt{n}$
The relative spread: $\mathrm{RelSpread}[T_n] = a/\sqrt{n}$
The asymptotic drift: $\mathrm{Drift}(T, X) = a$

4.1.1.4. Asymptotic Mean Drift

For the sample mean $\mathrm{Mean}(\mathbf{x}) = 1/n \sum_{i=1}^{n} x_i$ applied to samples from Additive$(\mathrm{mean}, \mathrm{stdDev})$, the sampling distribution of Mean is also additive with mean mean and standard deviation $\mathrm{stdDev}/\sqrt{n}$.

Using the lemma with $a = 1$ (since the standard deviation is $\mathrm{stdDev}/\sqrt{n}$):

$$\mathrm{Drift}(\mathrm{Mean}, X) = 1$$

Mean achieves unit drift under the Additive ('Normal') distribution, serving as the natural baseline for comparison. Mean is the optimal estimator under the Additive ('Normal') distribution: no other estimator achieves lower Drift.

4.1.1.5. Asymptotic Median Drift

For the sample median $\mathrm{Median}(\mathbf{x})$ applied to samples from Additive$(\mathrm{mean}, \mathrm{stdDev})$, the asymptotic sampling distribution of Median is approximately Additive ('Normal') with mean mean and standard deviation $\sqrt{\pi/2} \cdot \mathrm{stdDev}/\sqrt{n}$.

This result follows from the asymptotic theory of order statistics. For the median of a sample from a continuous distribution with density f and cumulative distribution F, the asymptotic variance is $1/\left(4n[f(F^{-1}(0.5))]^2\right)$. For the Additive ('Normal') distribution with standard deviation stdDev, the density at the median (which equals the mean) is $1/\left(\mathrm{stdDev}\ \sqrt{2\pi}\right)$. Thus the asymptotic variance becomes $\pi \cdot \mathrm{stdDev}^2/(2n)$.

Using the lemma with $a = \sqrt{\pi/2}$:

$$\mathrm{Drift}(\mathrm{Median}, X) = \sqrt{\pi/2}$$

Numerically, $\sqrt{\pi/2} \approx 1.2533$, so the median has approximately 25% higher drift than the mean under the Additive ('Normal') distribution.

4.1.1.6. Asymptotic Center Drift

For the sample center $\mathrm{Center}(\mathbf{x}) = \mathrm{Median}_{1 \le i \le j \le n}\left(x_i + x_j\right)/2$ applied to samples from Additive$(\mathrm{mean}, \mathrm{stdDev})$, its asymptotic sampling distribution must be determined.

The center estimator computes all pairwise averages (including $i = j$) and takes their median. For the Additive ('Normal') distribution, asymptotic theory shows that the center estimator is asymptotically Additive ('Normal') with mean mean.

The exact asymptotic variance of the center estimator for the Additive ('Normal') distribution is:

$$\mathrm{Var}[\mathrm{Center}(X_{1:n})] = (\pi \cdot \mathrm{stdDev}^2)/(3n)$$

This gives an asymptotic standard deviation of:

$$\mathrm{StdDev}[\mathrm{Center}(X_{1:n})] = \sqrt{\pi/3} \cdot \mathrm{stdDev}/\sqrt{n}$$

Using the lemma with $a = \sqrt{\pi/3}$:

$$\mathrm{Drift}(\mathrm{Center}, X) = \sqrt{\pi/3}$$

Numerically, $\sqrt{\pi/3} \approx 1.0233$, so the center estimator achieves a drift very close to 1 under the Additive ('Normal') distribution, performing nearly as well as the mean while offering greater robustness to outliers.

4.1.1.7. Lemma: Dispersion Estimator Drift Formula

For dispersion estimators T_n with asymptotic center $b \cdot \mathrm{stdDev}$ and standard deviation $a \cdot \mathrm{stdDev}\ /\sqrt{n}$, define $\mathrm{RelSpread}[T_n] := \mathrm{Spread}[T_n]/(b \cdot \mathrm{stdDev})$.

For any dispersion estimator T_n with asymptotic distribution $T_n \sim \mathrm{approx}\, \mathrm{Additive}(b \cdot \mathrm{stdDev}, (a \cdot \mathrm{stdDev})^2/n)$, the drift calculation follows:

The spread of two independent estimates: $\mathrm{Spread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a \cdot \mathrm{stdDev}\ /\sqrt{n}$
The relative spread: $\mathrm{RelSpread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a/(b\sqrt{n})$
The asymptotic drift: $\mathrm{Drift}(T, X) = \sqrt{2} \cdot z_{0.75} \cdot a/b$

Note: The $\sqrt{2}$ factor comes from the standard deviation of the difference $D = T_1 - T_2$ of two independent estimates, and the $z_{0.75}$ factor converts this standard deviation to the median absolute difference.

4.1.1.8. Asymptotic StdDev Drift

For the sample standard deviation $\mathrm{StdDev}(\mathbf{x}) = \sqrt{1/(n-1)\sum_{i=1}^{n}(x_i - \mathrm{Mean}(\mathbf{x}))^2}$ applied to samples from $\mathrm{Additive}(\mathrm{mean}, \mathrm{stdDev})$, the sampling distribution of StdDev is approximately Additive ('Normal') for large n with mean stdDev and standard deviation $\mathrm{stdDev}/\sqrt{2n}$.

Applying the lemma with $a = 1/\sqrt{2}$ and $b = 1$:

$$\mathrm{Spread}[\mathrm{StdDev}(X_{1:n})] = \sqrt{2} \cdot z_{0.75} \cdot 1/\sqrt{2} \cdot \mathrm{stdDev}/\sqrt{n} = z_{0.75} \cdot \mathrm{stdDev}/\sqrt{n}$$

For the dispersion drift, we use the relative spread formula:

$$\mathrm{RelSpread}[\mathrm{StdDev}(X_{1:n})] = \mathrm{Spread}[\mathrm{StdDev}(X_{1:n})]/\,\mathrm{Center}[\mathrm{StdDev}(X_{1:n})]$$

Since $\mathrm{Center}[\mathrm{StdDev}(X_{1:n})] \approx \mathrm{stdDev}$ asymptotically:

$$\mathrm{RelSpread}[\mathrm{StdDev}(X_{1:n})] = \left(z_{0.75} \cdot \mathrm{stdDev}/\sqrt{n}\right)/\mathrm{stdDev} = z_{0.75}/\sqrt{n}$$

Therefore:

$$\text{Drift}(\text{StdDev}, X) = \lim_{n\to\infty} \sqrt{n} \cdot \text{RelSpread}[\text{StdDev}(X_{1:n})] = z_{0.75}$$

Numerically, $z_{0.75} \approx 0.67449$.

4.1.1.9. Asymptotic MAD Drift

For the median absolute deviation $\text{MAD}(\mathbf{x}) = \text{Median}(|x_i - \text{Median}(\mathbf{x})|)$ applied to samples from $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$, the asymptotic distribution is approximately $\underline{\text{Additive}}$ ('Normal').

For the $\underline{\text{Additive}}$ ('Normal') distribution, the population MAD equals $z_{0.75} \cdot \text{stdDev}$. The asymptotic standard deviation of the sample MAD is:

$$\text{StdDev}[\text{MAD}(X_{1:n})] = c_{\text{mad}} \cdot \text{stdDev}/\sqrt{n}$$

where $c_{\text{mad}} \approx 0.78$.

Applying the lemma with $a = c_{\text{mad}}$ and $b = z_{0.75}$:

$$\text{Spread}[\text{MAD}(X_{1:n})] = \sqrt{2} \cdot z_{0.75} \cdot c_{\text{mad}} \cdot \text{stdDev}/\sqrt{n}$$

Since $\text{Center}[\text{MAD}(X_{1:n})] \approx z_{0.75} \cdot \text{stdDev}$ asymptotically:

$$\text{RelSpread}[\text{MAD}(X_{1:n})] = \left(\sqrt{2} \cdot z_{0.75} \cdot c_{\text{mad}} \cdot \text{stdDev}/\sqrt{n}\right)/(z_{0.75} \cdot \text{stdDev}) = \left(\sqrt{2} \cdot c_{\text{mad}}\right)/\sqrt{n}$$

Therefore:

$$\text{Drift}(\text{MAD}, X) = \lim_{n\to\infty} \sqrt{n} \cdot \text{RelSpread}[\text{MAD}(X_{1:n})] = \sqrt{2} \cdot c_{\text{mad}}$$

Numerically, $\sqrt{2} \cdot c_{\text{mad}} \approx \sqrt{2} \cdot 0.78 \approx 1.10$.

4.1.1.10. Asymptotic Spread Drift

For the sample spread $\text{Spread}(\mathbf{x}) = \text{Median}_{1\le i<j\le n}|x_i - x_j|$ applied to samples from $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$, the asymptotic distribution is approximately $\underline{\text{Additive}}$ ('Normal').

The spread estimator computes all pairwise absolute differences and takes their median. For the $\underline{\text{Additive}}$ ('Normal') distribution, the population spread equals $\sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}$ as derived in the Asymptotic Spread Value section.

The asymptotic standard deviation of the sample spread for the $\underline{\text{Additive}}$ ('Normal') distribution is:

$$\text{StdDev}[\text{Spread}(X_{1:n})] = c_{\text{spr}} \cdot \text{stdDev}/\sqrt{n}$$

where $c_{\text{spr}} \approx 0.72$.

Applying the lemma with $a = c_{\text{spr}}$ and $b = \sqrt{2} \cdot z_{0.75}$:

$$\text{Spread}[\text{Spread}(X_{1:n})] = \sqrt{2} \cdot z_{0.75} \cdot c_{\text{spr}} \cdot \text{stdDev}/\sqrt{n}$$

Since $\text{Center}[\text{Spread}(X_{1:n})] \approx \sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}$ asymptotically:

$$\mathrm{RelSpread}[\mathrm{Spread}(X_{1:n})] = \left(\sqrt{2}\cdot z_{0.75}\cdot c_{\mathrm{spr}}\cdot \mathrm{stdDev}/\sqrt{n}\right)/\left(\sqrt{2}\cdot z_{0.75}\cdot \mathrm{stdDev}\right) = c_{\mathrm{spr}}/\sqrt{n}$$

Therefore:

$$\mathrm{Drift}(\mathrm{Spread}, X) = \lim_{n\to\infty}\sqrt{n}\cdot \mathrm{RelSpread}[\mathrm{Spread}(X_{1:n})] = c_{\mathrm{spr}}$$

Numerically, $c_{\mathrm{spr}} \approx 0.72$.

4.1.1.11. Summary

Summary for average estimators:

Estimator	$\mathrm{Drift}(E, X)$	$\mathrm{Drift}^2(E, X)$	$1/\mathrm{Drift}^2(E, X)$
Mean	1	1	1
Median	≈ 1.253	$\pi/2 \approx 1.571$	$2/\pi \approx 0.637$
Center	≈ 1.023	$\pi/3 \approx 1.047$	$3/\pi \approx 0.955$

The squared drift values indicate the sample size adjustment needed when switching estimators. For instance, switching from Mean to Median while maintaining the same precision requires increasing the sample size by a factor of $\pi/2 \approx 1.571$ (about 57% more observations). Similarly, switching from Mean to Center requires only about 5% more observations.

The inverse squared drift (rightmost column) equals the classical statistical efficiency relative to the Mean. The Mean achieves optimal performance (unit efficiency) for the Additive ('Normal') distribution, as expected from classical theory. The Center maintains 95.5% efficiency while offering greater robustness to outliers, making it an attractive alternative when some contamination is possible. The Median, while most robust, operates at only 63.7% efficiency under purely Additive ('Normal') conditions.

Summary for dispersion estimators:

For the Additive ('Normal') distribution, the asymptotic drift values reveal the relative precision of different dispersion estimators:

Estimator	$\mathrm{Drift}(E, X)$	$\mathrm{Drift}^2(E, X)$	$1/\mathrm{Drift}^2(E, X)$
StdDev	≈ 0.67	≈ 0.45	≈ 2.22
MAD	≈ 1.10	≈ 1.22	≈ 0.82
Spread	≈ 0.72	≈ 0.52	≈ 1.92

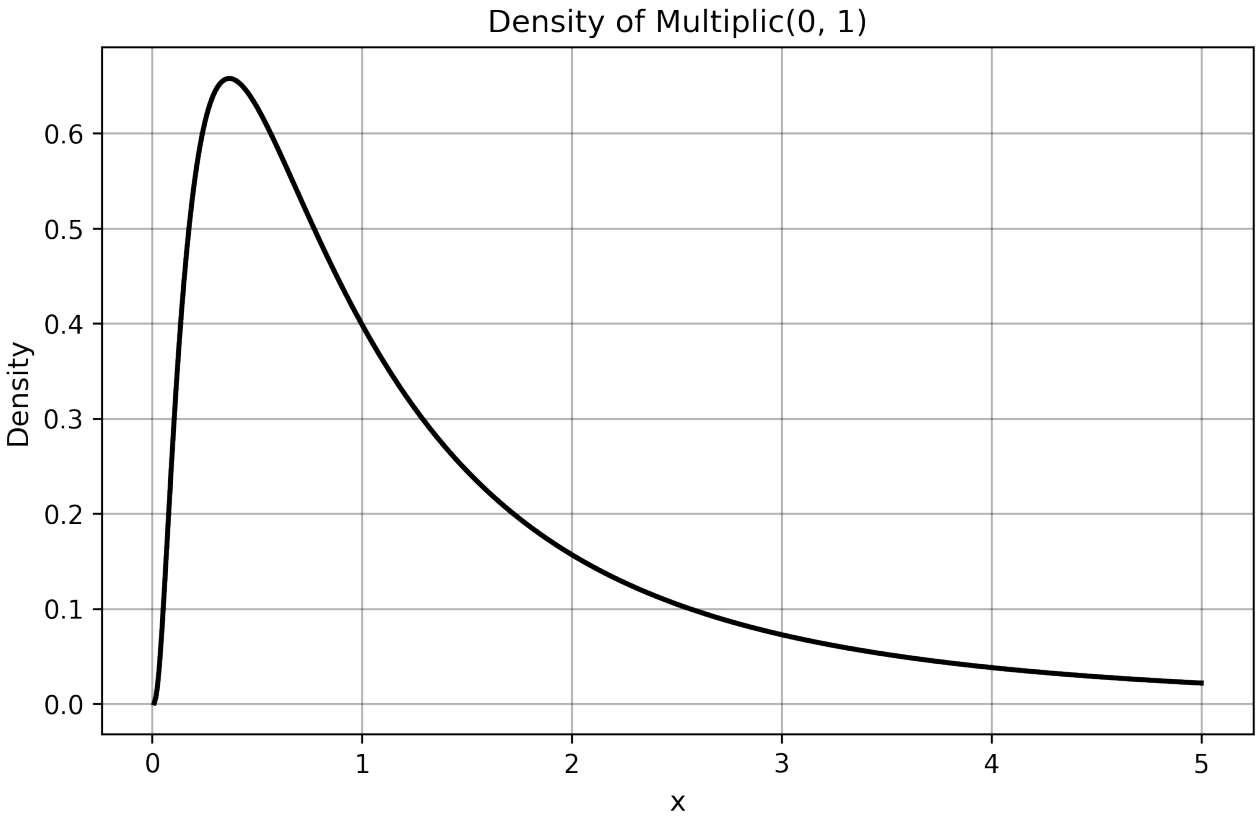
The squared drift values indicate the sample size adjustment needed when switching estimators. For instance, switching from StdDev to MAD while maintaining the same precision requires increasing the sample size by a factor of $1.22/0.45 \approx 2.71$ (more than doubling the observations). Similarly, switching from StdDev to Spread requires a factor of $0.52/0.45 \approx 1.16$.

The StdDev achieves optimal performance for the Additive ('Normal') distribution. The MAD requires about 2.7 times more data to match the StdDev precision while offering greater robustness to outliers. The Spread requires about 1.16 times more data to match the StdDev precision under purely Additive ('Normal') conditions while maintaining robustness.

4.2. Multiplic

$$\underline{\text{Multiplic}}(\text{logMean}, \text{logStdDev})$$

logMean: mean of log values (location parameter; e^{logMean} equals the geometric mean)
logStdDev: standard deviation of log values (scale parameter; controls multiplicative spread)

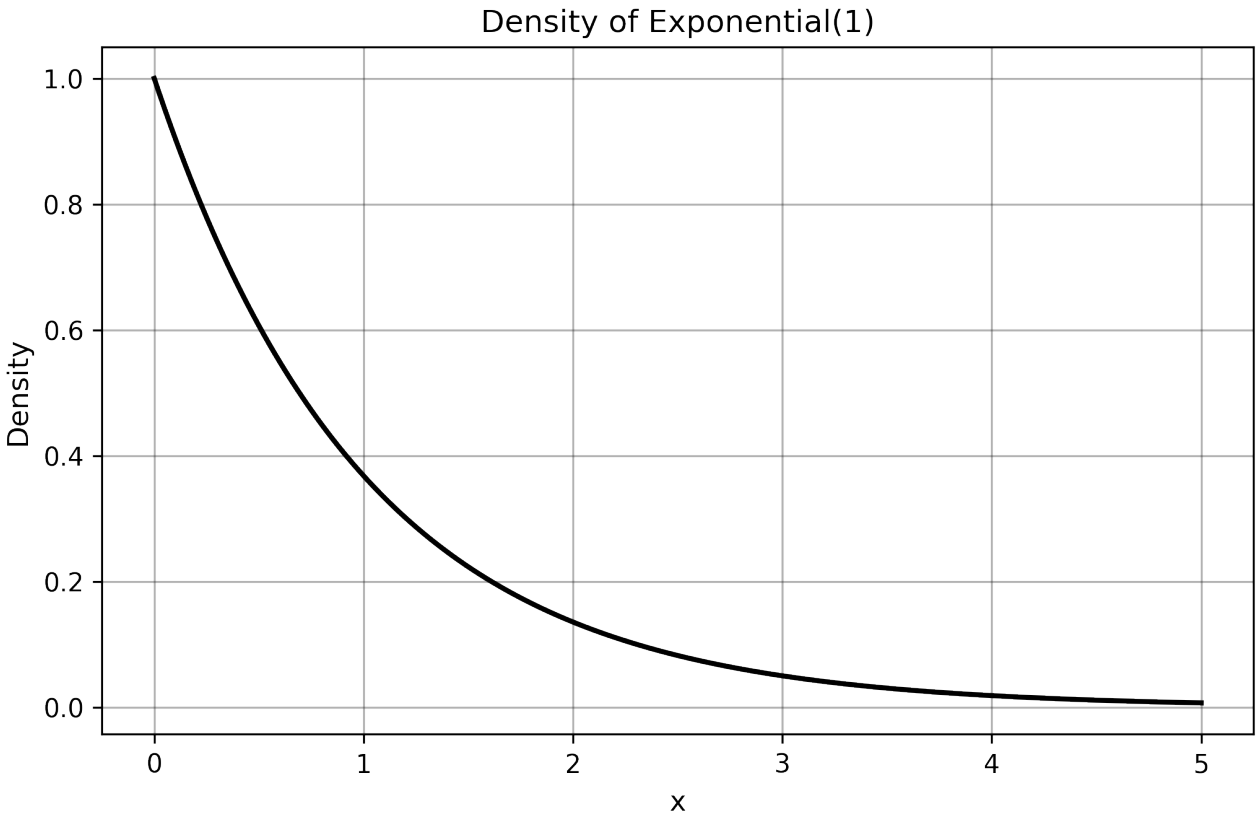


Formation: the product of many positive variables $X_1 \cdot X_2 \cdot ... \cdot X_n$ with mild conditions (e.g., finite variance of $\log X$).
Origin: historically called 'Log-Normal' or 'Galton' distribution after Francis Galton.
Rename Motivation: renamed to Multiplic to reflect its formation mechanism through multiplication.
Properties: logarithm of a Multiplic ('LogNormal') variable follows an Additive ('Normal') distribution.
Applications: stock prices, file sizes, reaction times, income distributions, biological growth rates.
Caution: no perfectly multiplic distributions exist in real data; all real-world measurements contain deviations. Traditional estimators struggle with the inherent skewness and heavy right tail.

4.3. Exp

$$\underline{\text{Exp}}(\text{rate})$$

rate: rate parameter ($\lambda > 0$, controls decay speed; mean = $1/\text{rate}$)

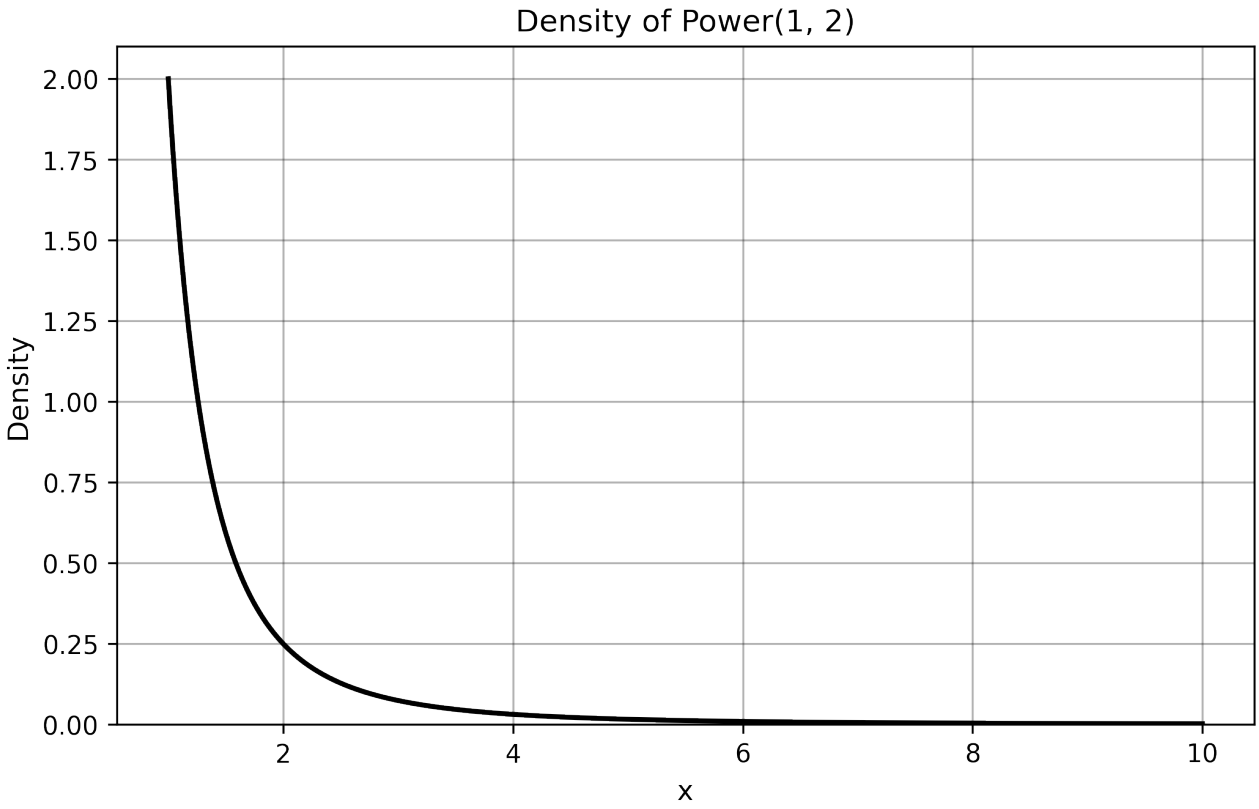


Formation: the waiting time between events in a Poisson process.
Origin: naturally arises from memoryless processes where the probability of an event occurring is constant over time.
Properties: memoryless (past events do not affect future probabilities).
Applications: time between failures, waiting times in queues, radioactive decay, customer service times.
Characteristics: always positive, right-skewed with a light (exponential) tail.
Caution: extreme skewness makes traditional location estimators like Mean unreliable; robust estimators provide more stable results.

4.4. Power

$$\underline{\mathrm{Power}}(\mathrm{min}, \mathrm{shape})$$

min: **minimum value (lower bound,** $\mathrm{min} > 0$**)**
shape: **shape parameter (**$\alpha > 0$**, controls tail heaviness; smaller values = heavier tails)**

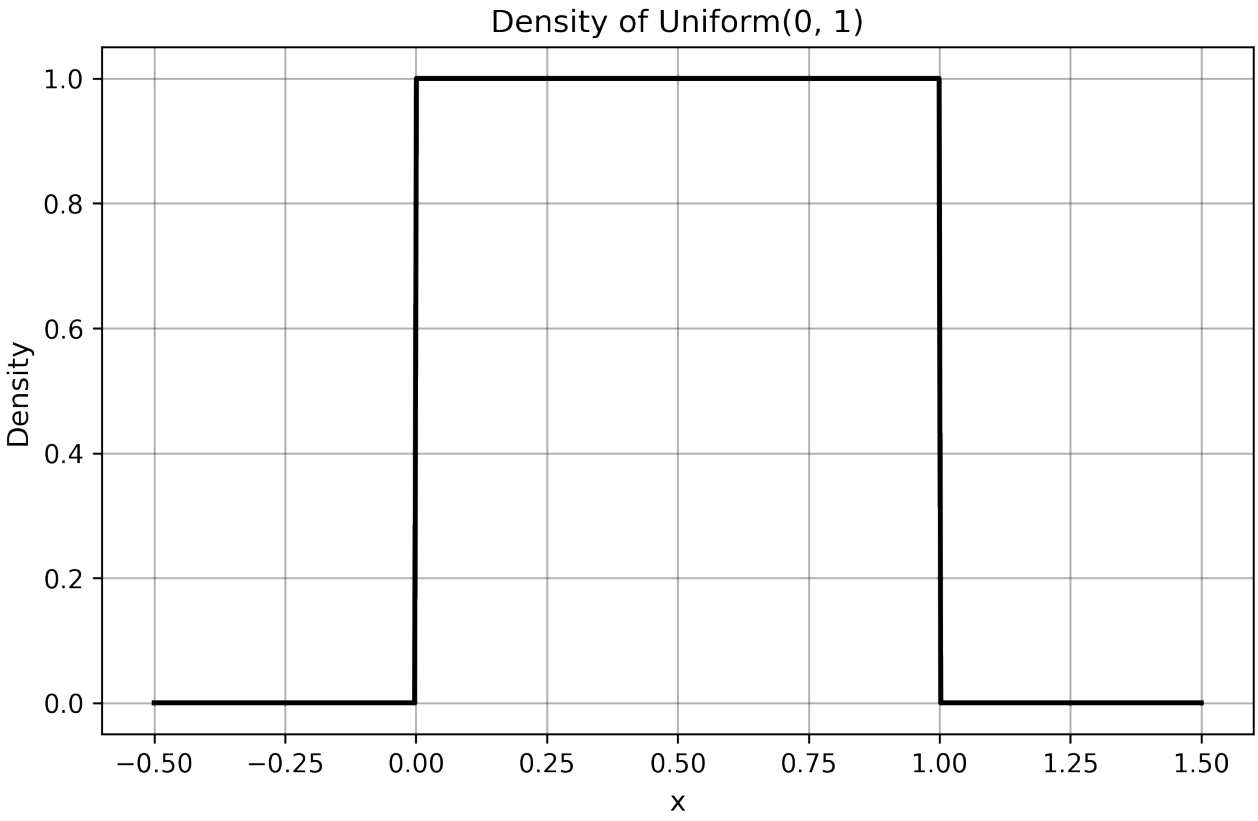


Formation: follows a power-law relationship where large values are rare but possible.
Origin: historically called 'Pareto' distribution after Vilfredo Pareto's work on wealth distribution.
Rename Motivation: renamed to Power to reflect its connection with power-law.
Properties: exhibits scale invariance and extremely heavy tails.
Applications: wealth distribution, city population sizes, word frequencies, earthquake magnitudes, website traffic.
Characteristics: infinite variance for many parameter values; extreme outliers are common.
Caution: traditional variance-based estimators completely fail; robust estimators are essential for reliable analysis.

4.5. Uniform

$$\text{Uniform}(\min, \max)$$

$\min$: lower bound of the support interval
$\max$: upper bound of the support interval $(\max > \min)$



Formation: all values within a bounded interval have equal probability.
Origin: represents complete uncertainty within known bounds.
Properties: rectangular probability density, finite support with hard boundaries.
Applications: random number generation, round-off errors, arrival times within known intervals.
Characteristics: symmetric, bounded, no tail behavior.
Note: traditional estimators work reasonably well due to symmetry and bounded nature.

5. Implementations

5.1. Python

Install from PyPI:

```
pip install pragmastat==10.0.5
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/py

Pragmastat on PyPI: https://pypi.org/project/pragmastat/

Demo:

```
from pragmastat import (
    Rng,
    center,
    spread,
    shift,
    ratio,
    disparity,
    center_bounds,
    shift_bounds,
    ratio_bounds,
    spread_bounds,
    disparity_bounds,
)
from pragmastat.distributions import Additive, Exp, Multiplic, Power, Uniform


def main():
    # --- One-Sample ---

    x = list(range(1, 21))

    print(center(x))  # 10.5
    bounds = center_bounds(x, 0.05)
    print(
        f"Bounds(lower={bounds.lower}, upper={bounds.upper})"
    )  # Bounds(lower=7.5, upper=13.5)
    print(spread(x))  # 6.0
    bounds = spread_bounds(x, 0.05, seed="demo")
    print(
        f"Bounds(lower={bounds.lower}, upper={bounds.upper})"
    )  # Bounds(lower=2.0, upper=10.0)

    # --- Two-Sample ---

    x = list(range(1, 31))
    y = list(range(21, 51))

    print(shift(x, y))  # -20.0
    bounds = shift_bounds(x, y, 0.05)
```

```
    print(
        f"Bounds(lower={bounds.lower}, upper={bounds.upper})"
    )  # Bounds(lower=-25.0, upper=-15.0)
    print(ratio(x, y))  # 0.43669798282695127
    bounds = ratio_bounds(x, y, 0.05)
    print(
        f"Bounds(lower={bounds.lower}, upper={bounds.upper})"
    )  # Bounds(lower=0.31250000000000006, upper=0.5599999999999999)
    print(disparity(x, y))  # -2.2222222222222223
    bounds = disparity_bounds(x, y, 0.05, seed="demo")
    print(
        f"Bounds(lower={bounds.lower}, upper={bounds.upper})"
    )  # Bounds(lower=-13.0, upper=-0.8235294117647058)

    # --- Randomization ---

    rng = Rng("demo-uniform")
    print(rng.uniform_float())  # 0.2640554428629759
    print(rng.uniform_float())  # 0.9348534835582796

    rng = Rng("demo-uniform-int")
    print(rng.uniform_int(0, 100))  # 41

    rng = Rng("demo-sample")
    print(rng.sample([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 3))  # [3, 8, 9]

    rng = Rng("demo-resample")
    print(rng.resample([1, 2, 3, 4, 5], 7))  # [3, 1, 3, 2, 4, 1, 2]

    rng = Rng("demo-shuffle")
    print(rng.shuffle([1, 2, 3, 4, 5]))  # [4, 2, 3, 5, 1]

    # --- Distributions ---

    rng = Rng("demo-dist-additive")
    print(Additive(0, 1).sample(rng))  # 0.17410448679568188

    rng = Rng("demo-dist-multiplic")
    print(Multiplic(0, 1).sample(rng))  # 1.1273244602673853

    rng = Rng("demo-dist-exp")
    print(Exp(1).sample(rng))  # 0.6589065267276553

    rng = Rng("demo-dist-power")
    print(Power(1, 2).sample(rng))  # 1.023677535537084

    rng = Rng("demo-dist-uniform")
    print(Uniform(0, 10).sample(rng))  # 6.54043657816832


if __name__ == "__main__":
    main()
```

5.2. TypeScript

Install from npm:

```
npm i pragmastat@10.0.5
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/ts

Pragmastat on npm: https://www.npmjs.com/package/pragmastat

Demo:

```
import {
    center, spread, shift, ratio, disparity,
    centerBounds, shiftBounds, ratioBounds,
    spreadBounds, disparityBounds,
    Rng, Additive, Multiplic, Exp, Power, Uniform
} from '../src';

function main() {
    // --- One-Sample ---

    let x = Array.from({ length: 20 }, (_, i) => i + 1);

    console.log(center(x));             // 10.5
    console.log(centerBounds(x, 0.05)); // { lower: 7.5, upper: 13.5 }
    console.log(spread(x));             // 6
    console.log(spreadBounds(x, 0.05, "demo")); // { lower: 2, upper: 10 }

    // --- Two-Sample ---

    x = Array.from({ length: 30 }, (_, i) => i + 1);
    let y = Array.from({ length: 30 }, (_, i) => i + 21);

    console.log(shift(x, y));             // -20
    console.log(shiftBounds(x, y, 0.05)); // { lower: -25, upper: -15 }
    console.log(ratio(x, y));             // 0.4366979828269513
    console.log(ratioBounds(x, y, 0.05)); // { lower: 0.31250000000000006, upper:
0.5600000000000003 }
    console.log(disparity(x, y));         // -2.2222222222222223
    console.log(disparityBounds(x, y, 0.05, "demo")); // { lower: -13, upper:
-0.8235294117647058 }

    // --- Randomization ---

    let rng = new Rng("demo-uniform");
    console.log(rng.uniformFloat()); // 0.2640554428629759
    console.log(rng.uniformFloat()); // 0.9348534835582796

    rng = new Rng("demo-uniform-int");
    console.log(rng.uniformInt(0, 100)); // 41

    rng = new Rng("demo-sample");
    console.log(rng.sample([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 3)); // [3, 8, 9]
```

```
    rng = new Rng("demo-resample");
    console.log(rng.resample([1, 2, 3, 4, 5], 7)); // [3, 1, 3, 2, 4, 1, 2]

    rng = new Rng("demo-shuffle");
    console.log(rng.shuffle([1, 2, 3, 4, 5])); // [4, 2, 3, 5, 1]

    // --- Distributions ---

    rng = new Rng("demo-dist-additive");
    console.log(new Additive(0, 1).sample(rng)); // 0.17410448679568188

    rng = new Rng("demo-dist-multiplic");
    console.log(new Multiplic(0, 1).sample(rng)); // 1.1273244602673853

    rng = new Rng("demo-dist-exp");
    console.log(new Exp(1).sample(rng)); // 0.6589065267276553

    rng = new Rng("demo-dist-power");
    console.log(new Power(1, 2).sample(rng)); // 1.023677535537084

    rng = new Rng("demo-dist-uniform");
    console.log(new Uniform(0, 10).sample(rng)); // 6.54043657816832
}

main();
```

5.3. R

Install from GitHub:

```
install.packages("remotes") # If 'remotes' is not installed
remotes::install_github("AndreyAkinshin/pragmastat",
                        subdir = "r/pragmastat", ref = "v10.0.5")
library(pragmastat)
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/r

Demo:

```
library(pragmastat)

# --- One-Sample ---

x <- 1:20

print(center(x)) # 10.5
bounds <- center_bounds(x, 0.05)
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep="")) # [7.5, 13.5]
print(spread(x)) # 6
bounds <- spread_bounds(x, 0.05, seed = "demo")
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep="")) # [2, 10]
```

```
# --- Two-Sample ---

x <- 1:30
y <- 21:50

print(shift(x, y)) # -20
bounds <- shift_bounds(x, y, 0.05)
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep="")) # [-25, -15]
print(ratio(x, y)) # 0.43669798282695127
bounds <- ratio_bounds(x, y, 0.05)
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep="")) # [0.31250000000000006,
0.5599999999999999]
print(disparity(x, y)) # -2.222222222222223
bounds <- disparity_bounds(x, y, 0.05, seed = "demo")
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep="")) # [-13,
-0.8235294117647058]

# --- Randomization ---

r <- rng("demo-uniform")
print(r$uniform_float()) # 0.2640554428629759
print(r$uniform_float()) # 0.9348534835582796

r <- rng("demo-uniform-int")
print(r$uniform_int(0, 100)) # 41

r <- rng("demo-sample")
print(r$sample(0:9, 3)) # [3, 8, 9]

r <- rng("demo-resample")
print(r$resample(c(1, 2, 3, 4, 5), 7)) # [3, 1, 3, 2, 4, 1, 2]

r <- rng("demo-shuffle")
print(r$shuffle(c(1, 2, 3, 4, 5))) # [4, 2, 3, 5, 1]

# --- Distributions ---

r <- rng("demo-dist-additive")
dist <- dist_additive(0, 1)
print(dist$sample(r)) # 0.17410448679568188

r <- rng("demo-dist-multiplic")
dist <- dist_multiplic(0, 1)
print(dist$sample(r)) # 1.1273244602673853

r <- rng("demo-dist-exp")
dist <- dist_exp(1)
print(dist$sample(r)) # 0.6589065267276553

r <- rng("demo-dist-power")
dist <- dist_power(1, 2)
print(dist$sample(r)) # 1.023677535537084
```

```
r <- rng("demo-dist-uniform")
dist <- dist_uniform(0, 10)
print(dist$sample(r)) # 6.54043657816832
```

5.4. C#

Install from NuGet via .NET CLI:

```
dotnet add package Pragmastat --version 10.0.5
```

Install from NuGet via Package Manager Console:

```
NuGet\Install-Package Pragmastat -Version 10.0.5
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/cs

Pragmastat on NuGet: https://www.nuget.org/packages/Pragmastat/

Demo:

```
using static System.Console;
using Pragmastat.Distributions;
using Pragmastat.Randomization;

namespace Pragmastat.Demo;

class Program
{
  static void Main()
  {
    // --- One-Sample ---

    var x = new Sample(
      1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15, 16, 17, 18, 19, 20);

    WriteLine(Toolkit.Center(x));             // 10.5
    WriteLine(Toolkit.CenterBounds(x, 0.05)); // [7.5;13.5]
    WriteLine(Toolkit.Spread(x));             // 6
    WriteLine(Toolkit.SpreadBounds(x, 0.05, "demo")); // [2;10]

    // --- Two-Sample ---

    x = new Sample(
      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
      16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30);
    var y = new Sample(
      21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
      36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50);

    WriteLine(Toolkit.Shift(x, y));             // -20
    WriteLine(Toolkit.ShiftBounds(x, y, 0.05)); // [-25;-15]
    WriteLine(Toolkit.Ratio(x, y));             // 0.43669798282695127
```

```
    WriteLine(Toolkit.RatioBounds(x, y, 0.05)); //
[0.31250000000000006;0.5599999999999999]
    WriteLine(Toolkit.Disparity(x, y));         // -2.2222222222222223
    WriteLine(Toolkit.DisparityBounds(x, y, 0.05, "demo")); // [-13;-0.8235294117647058]

    // --- Randomization ---

    var rng = new Rng("demo-uniform");
    WriteLine(rng.UniformDouble()); // 0.2640554428629759
    WriteLine(rng.UniformDouble()); // 0.9348534835582796

    rng = new Rng("demo-uniform-int");
    WriteLine(rng.UniformInt32(0, 100)); // 41

    rng = new Rng("demo-sample");
    WriteLine(string.Join(", ", rng.Sample([0.0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 3))); // 3,
8, 9

    rng = new Rng("demo-resample");
    WriteLine(string.Join(", ", rng.Resample([1.0, 2, 3, 4, 5], 7))); // 3, 1, 3, 2, 4, 1,
2

    rng = new Rng("demo-shuffle");
    WriteLine(string.Join(", ", rng.Shuffle([1.0, 2, 3, 4, 5]))); // 4, 2, 3, 5, 1

    // --- Distributions ---

    rng = new Rng("demo-dist-additive");
    WriteLine(new Additive(0, 1).Sample(rng)); // 0.17410448679568188

    rng = new Rng("demo-dist-multiplic");
    WriteLine(new Multiplic(0, 1).Sample(rng)); // 1.1273244602673853

    rng = new Rng("demo-dist-exp");
    WriteLine(new Exp(1).Sample(rng)); // 0.6589065267276553

    rng = new Rng("demo-dist-power");
    WriteLine(new Power(1, 2).Sample(rng)); // 1.023677535537084

    rng = new Rng("demo-dist-uniform");
    WriteLine(new Uniform(0, 10).Sample(rng)); // 6.54043657816832
  }
}
```

5.5. Kotlin

Install from Maven Central Repository via Apache Maven:

```
<dependency>
    <groupId>dev.pragmastat</groupId>
    <artifactId>pragmastat</artifactId>
    <version>10.0.5</version>
</dependency>
```

Install from Maven Central Repository via Gradle:

```
implementation 'dev.pragmastat:pragmastat:10.0.5'
```

Install from Maven Central Repository via Gradle (Kotlin):

```
implementation("dev.pragmastat:pragmastat:10.0.5")
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/kt

Pragmastat on Maven Central Repository: https://central.sonatype.com/artifact/dev.pragmastat/pragmastat/overview

Demo:

```
package dev.pragmastat.demo

import dev.pragmastat.*
import dev.pragmastat.distributions.*

fun main() {
    // --- One-Sample ---

    var x = (1..20).map { it.toDouble() }

    println(center(x))                    // 10.5
    println(centerBounds(x, 0.05))        // Bounds(lower=7.5, upper=13.5)
    println(spread(x))                    // 6.0
    println(spreadBounds(x, 0.05, "demo")) // Bounds(lower=2.0, upper=10.0)

    // --- Two-Sample ---

    x = (1..30).map { it.toDouble() }
    var y = (21..50).map { it.toDouble() }

    println(shift(x, y))                    // -20
    println(shiftBounds(x, y, 0.05))        // Bounds(lower=-25.0, upper=-15.0)
    println(ratio(x, y))                    // 0.43669798282695127
    println(ratioBounds(x, y, 0.05))        // Bounds(lower=0.31250000000000006,
upper=0.5599999999999999)
    println(disparity(x, y))                // -2.2222222222222223
    println(disparityBounds(x, y, 0.05, "demo")) // Bounds(lower=-13.0,
upper=-0.8235294117647058)

    // --- Randomization ---

    var rng = Rng("demo-uniform")
    println(rng.uniformDouble()) // 0.2640554428629759
    println(rng.uniformDouble()) // 0.9348534835582796

    rng = Rng("demo-uniform-int")
    println(rng.uniformInt(0, 100)) // 41
```

```
    rng = Rng("demo-sample")
    println(rng.sample(listOf(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0), 3)) //
[3.0, 8.0, 9.0]

    rng = Rng("demo-resample")
    println(rng.resample(listOf(1.0, 2.0, 3.0, 4.0, 5.0), 7)) // [3.0, 1.0, 3.0, 2.0, 4.0,
1.0, 2.0]

    rng = Rng("demo-shuffle")
    println(rng.shuffle(listOf(1.0, 2.0, 3.0, 4.0, 5.0))) // [4.0, 2.0, 3.0, 5.0, 1.0]

    // --- Distributions ---

    rng = Rng("demo-dist-additive")
    println(Additive(0.0, 1.0).sample(rng)) // 0.17410448679568188

    rng = Rng("demo-dist-multiplic")
    println(Multiplic(0.0, 1.0).sample(rng)) // 1.1273244602673853

    rng = Rng("demo-dist-exp")
    println(Exp(1.0).sample(rng)) // 0.6589065267276553

    rng = Rng("demo-dist-power")
    println(Power(1.0, 2.0).sample(rng)) // 1.023677535537084

    rng = Rng("demo-dist-uniform")
    println(Uniform(0.0, 10.0).sample(rng)) // 6.54043657816832
}
```

5.6. Rust

Install from crates.io via cargo:

```
cargo add pragmastat@10.0.5
```

Install from crates.io via `Cargo.toml`:

```
[dependencies]
pragmastat = "10.0.5"
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/rs

Pragmastat on crates.io: https://crates.io/crates/pragmastat

Demo:

```
use pragmastat::distributions::{Additive, Distribution, Exp, Multiplic, Power, Uniform};
use pragmastat::*;

fn main() {
    // --- One-Sample ---

    let x: Vec<f64> = (1..=20).map(|i| i as f64).collect();
```

```
    println!("{}", center(&x).unwrap()); // 10.5
    let bounds = center_bounds(&x, 0.05).unwrap();
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper); // {lower: 7.5,
upper: 13.5}
    println!("{}", spread(&x).unwrap()); // 6
    let bounds = spread_bounds_with_seed(&x, 0.05, "demo").unwrap();
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper); // {lower: 2, upper:
10}

    // --- Two-Sample ---

    let x: Vec<f64> = (1..=30).map(|i| i as f64).collect();
    let y: Vec<f64> = (21..=50).map(|i| i as f64).collect();

    println!("{}", shift(&x, &y).unwrap()); // -20
    let bounds = shift_bounds(&x, &y, 0.05).unwrap();
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper); // {lower: -25,
upper: -15}
    println!("{}", ratio(&x, &y).unwrap()); // 0.43669798282695127
    let bounds = ratio_bounds(&x, &y, 0.05).unwrap();
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper); // {lower:
0.31250000000000006, upper: 0.5599999999999999}
    println!("{}", disparity(&x, &y).unwrap()); // -2.2222222222222223
    let bounds = disparity_bounds_with_seed(&x, &y, 0.05, "demo").unwrap();
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper); // {lower: -13,
upper: -0.8235294117647058}

    // --- Randomization ---

    let mut rng = Rng::from_string("demo-uniform");
    println!("{}", rng.uniform_f64()); // 0.2640554428629759
    println!("{}", rng.uniform_f64()); // 0.9348534835582796

    let mut rng = Rng::from_string("demo-uniform-int");
    println!("{}", rng.uniform_i64(0, 100)); // 41

    let mut rng = Rng::from_string("demo-sample");
    println!(
        "{:?}",
        rng.sample(&[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0], 3)
    ); // [3, 8, 9]

    let mut rng = Rng::from_string("demo-resample");
    println!("{:?}", rng.resample(&[1.0, 2.0, 3.0, 4.0, 5.0], 7)); // [3, 1, 3, 2, 4, 1,
2]

    let mut rng = Rng::from_string("demo-shuffle");
    println!("{:?}", rng.shuffle(&[1.0, 2.0, 3.0, 4.0, 5.0])); // [4, 2, 3, 5, 1]

    // --- Distributions ---

    let mut rng = Rng::from_string("demo-dist-additive");
```

```
    println!("{}", Additive::new(0.0, 1.0).sample(&mut rng)); // 0.17410448679568188

    let mut rng = Rng::from_string("demo-dist-multiplic");
    println!("{}", Multiplic::new(0.0, 1.0).sample(&mut rng)); // 1.1273244602673853

    let mut rng = Rng::from_string("demo-dist-exp");
    println!("{}", Exp::new(1.0).sample(&mut rng)); // 0.6589065267276553

    let mut rng = Rng::from_string("demo-dist-power");
    println!("{}", Power::new(1.0, 2.0).sample(&mut rng)); // 1.023677535537084

    let mut rng = Rng::from_string("demo-dist-uniform");
    println!("{}", Uniform::new(0.0, 10.0).sample(&mut rng)); // 6.54043657816832
}
```

5.7. Go

Install from GitHub:

```
go get github.com/AndreyAkinshin/pragmastat/go/v10@v10.0.5
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v10.0.5/go

Demo:

```
package main

import (
  "fmt"
  "log"

  pragmastat "github.com/AndreyAkinshin/pragmastat/go/v10"
)

func must[T any](val T, err error) T {
  if err != nil {
    log.Fatal(err)
  }
  return val
}

func main() {
  // --- One-Sample ---

  x := []float64{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20}


fmt.Println(must(pragmastat.Center(x)))                                    //
10.5
  fmt.Println(must(pragmastat.CenterBounds(x, pragmastat.BoundsConfig{Misrate:
0.05})))               // {7.5 13.5}
```

```
fmt.Println(must(pragmastat.Spread(x)))                                                    //
6
  fmt.Println(must(pragmastat.SpreadBounds(x, pragmastat.BoundsConfig{Misrate: 0.05, Seed:
"demo"}))) // {2 10}

  // --- Two-Sample ---

  x = []float64{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
  y := []float64{
    21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50}

  fmt.Println(must(pragmastat.Shift(x,
y)))                                                                  // -20
  fmt.Println(must(pragmastat.ShiftBounds(x, y, pragmastat.BoundsConfig{Misrate:
0.05})))                  // {-25 -15}
  fmt.Println(must(pragmastat.Ratio(x,
y)))                                                                  // 0.4366979828269513
  fmt.Println(must(pragmastat.RatioBounds(x, y, pragmastat.BoundsConfig{Misrate:
0.05})))                  // {0.31250000000000006 0.5600000000000003}
  fmt.Println(must(pragmastat.Disparity(x,
y)))                                                              // -2.2222222222222223
  fmt.Println(must(pragmastat.DisparityBounds(x, y, pragmastat.BoundsConfig{Misrate: 0.05,
Seed: "demo"}))) // {-13 -0.8235294117647058}

  // --- Randomization ---

  rng := pragmastat.NewRngFromString("demo-uniform")
  fmt.Println(rng.UniformFloat64()) // 0.2640554428629759
  fmt.Println(rng.UniformFloat64()) // 0.9348534835582796

  rng = pragmastat.NewRngFromString("demo-uniform-int")
  fmt.Println(rng.UniformInt64(0, 100)) // 41

  rng = pragmastat.NewRngFromString("demo-sample")
  fmt.Println(pragmastat.Sample(rng, []float64{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 3)) // [3 8
9]

  rng = pragmastat.NewRngFromString("demo-resample")
  fmt.Println(pragmastat.Resample(rng, []float64{1, 2, 3, 4, 5}, 7)) // [3 1 3 2 4 1 2]

  rng = pragmastat.NewRngFromString("demo-shuffle")
  fmt.Println(pragmastat.Shuffle(rng, []float64{1, 2, 3, 4, 5})) // [4 2 3 5 1]

  // --- Distributions ---

  rng = pragmastat.NewRngFromString("demo-dist-additive")
  fmt.Println(pragmastat.NewAdditive(0, 1).Sample(rng)) // 0.1741044867956819

  rng = pragmastat.NewRngFromString("demo-dist-multiplic")
  fmt.Println(pragmastat.NewMultiplic(0, 1).Sample(rng)) // 1.1273244602673853
```

```
  rng = pragmastat.NewRngFromString("demo-dist-exp")
  fmt.Println(pragmastat.NewExp(1).Sample(rng)) // 0.6589065267276553

  rng = pragmastat.NewRngFromString("demo-dist-power")
  fmt.Println(pragmastat.NewPower(1, 2).Sample(rng)) // 1.023677535537084

  rng = pragmastat.NewRngFromString("demo-dist-uniform")
  fmt.Println(pragmastat.NewUniform(0, 10).Sample(rng)) // 6.54043657816832
}
```

6. Auxiliary

6.1. AvgSpread

$$\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) = \frac{n \cdot \mathrm{Spread}(\mathbf{x}) + m \cdot \mathrm{Spread}(\mathbf{y})}{n + m}$$

Weighted average of dispersions (pooled scale).

Also known as — robust pooled standard deviation
Domain — any real numbers
Assumptions — `sparity(x)`, `sparity(y)`
Unit — same as measurements
Caveat — $\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) \neq \mathrm{Spread}(\mathbf{x} \cup \mathbf{y})$ (pooled scale, not concatenated spread)

Properties

Self-average $\mathrm{AvgSpread}(\mathbf{x}, \mathbf{x}) = \mathrm{Spread}(\mathbf{x})$
Symmetry $\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) = \mathrm{AvgSpread}(\mathbf{y}, \mathbf{x})$
Scale equivariance $\mathrm{AvgSpread}(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = |k| \cdot \mathrm{AvgSpread}(\mathbf{x}, \mathbf{y})$
Mixed scaling $\mathrm{AvgSpread}(k_1 \cdot \mathbf{x}, k_2 \cdot \mathbf{x}) = \frac{|k_1|+|k_2|}{2} \cdot \mathrm{Spread}(\mathbf{x})$

Example

`AvgSpread(x, y) = 5` where `Spread(x) = 6`, `Spread(y) = 4`, `n = m`
`AvgSpread(x, y) = AvgSpread(y, x)`

AvgSpread provides a single number representing the typical variability across two groups. It combines the spread of both samples, giving more weight to larger samples since they provide more reliable estimates. This pooled spread serves as a common reference scale, essential for expressing a difference in relative terms. Disparity uses AvgSpread internally to normalize the shift into a scale-free effect size.

6.1.1. Algorithm

The $\mathrm{AvgSpread}$ function computes the weighted average of per-sample spreads:

$$\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) = \frac{n \cdot \mathrm{Spread}(\mathbf{x}) + m \cdot \mathrm{Spread}(\mathbf{y})}{n + m}$$

The algorithm delegates to the Spread algorithm independently for each sample, then forms the weighted linear combination with weights $\frac{n}{n+m}$ and $\frac{m}{n+m}$.

```
using Pragmastat.Algorithms;
using Pragmastat.Exceptions;
using Pragmastat.Internal;
using Pragmastat.Metrology;

namespace Pragmastat.Estimators;

internal class AvgSpreadEstimator : ITwoSampleEstimator
{
  public static readonly AvgSpreadEstimator Instance = new();

  public Measurement Estimate(Sample x, Sample y)
  {
    Assertion.MatchedUnit(x, y);

    var spreadX = FastSpread.Estimate(x.SortedValues, isSorted: true);
    if (spreadX <= 0)
      throw AssumptionException.Sparity(Subject.X);
    var spreadY = FastSpread.Estimate(y.SortedValues, isSorted: true);
    if (spreadY <= 0)
      throw AssumptionException.Sparity(Subject.Y);

    return ((x.Size * spreadX + y.Size * spreadY) / (x.Size + y.Size)).WithUnitOf(x);
  }
}
```

6.1.2. Tests

$$\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) = \frac{n \cdot \mathrm{Spread}(\mathbf{x}) + m \cdot \mathrm{Spread}(\mathbf{y})}{n + m}$$

The AvgSpread test suite contains 36 test cases (5 demo + 4 natural + 1 negative + 9 additive + 4 uniform + 1 composite + 12 unsorted). Since AvgSpread is a weighted average of two Spread estimates, tests validate both the individual spread calculations and the weighting formula.

Demo examples ($n = m = 5$) — from manual introduction, validating properties:

`demo-1`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (0, 2, 4, 6, 8)$, expected output: 5 (base case)
`demo-2`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (0, 3, 6, 9, 12)$, expected output: 6 (equal samples)
`demo-3`: $\mathbf{x} = (0, 6, 12, 18, 24)$, $\mathbf{y} = (0, 9, 18, 27, 36)$, expected output: 15 (scale equivariance, $3 \times$ demo-1)
`demo-4`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\mathbf{y} = (0, 3, 6, 9, 12)$, expected output: 5 (swap symmetry with demo-1)
`demo-5`: $\mathbf{x} = (0, 6, 12, 18, 24)$, $\mathbf{y} = (0, 4, 8, 12, 16)$, expected output: 10 (scale, $2 \times$ demo-1)

Natural sequences ($[n, m] \in \{2, 3\} \times \{2, 3\}$) — 4 combinations:

`natural-2-2`, `natural-2-3`, `natural-3-2`, `natural-3-3`
Minimum size $n, m \geq 2$ required for meaningful dispersion

Negative values ($[n, m] = [2, 2]$) — sign handling validation:

`negative-2-2`: $\mathbf{x} = (-2, -1)$, $\mathbf{y} = (-2, -1)$, expected output: 1

Additive distribution ($[n, m] \in \{5, 10, 30\} \times \{5, 10, 30\}$) — 9 combinations with $\underline{\mathrm{Additive}}(10, 1)$:

`additive-5-5`, `additive-5-10`, `additive-5-30`
`additive-10-5`, `additive-10-10`, `additive-10-30`
`additive-30-5`, `additive-30-10`, `additive-30-30`
Random generation: x uses seed 0, y uses seed 1

Uniform distribution ($[n, m] \in \{5, 100\} \times \{5, 100\}$) — 4 combinations with $\underline{\mathrm{Uniform}}(0, 1)$:

`uniform-5-5`, `uniform-5-100`, `uniform-100-5`, `uniform-100-100`
Random generation: x uses seed 0, y uses seed 1

Composite estimator stress test — 1 test:

`composite-asymmetric-weights`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (3, 4, 5, 6, 7, 8, 9, 10)$ ($n = 2$, $m = 8$, highly asymmetric weights $w_x = 0.2$, $w_y = 0.8$)

Unsorted tests — verify sorting independence (12 tests):

`unsorted-x-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X unsorted (reversed), Y sorted (2 tests)
`unsorted-y-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X sorted, Y unsorted (reversed) (2 tests)
`unsorted-both-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: both unsorted (reversed) (2 tests)
`unsorted-demo-unsorted-x`: demo-1 with X unsorted

`unsorted-demo-unsorted-y`: demo-1 with Y unsorted
`unsorted-demo-both-unsorted`: demo-1 with both unsorted
`unsorted-identity-unsorted`: equal samples, both unsorted
`unsorted-negative-unsorted`: negative values, both unsorted
`unsorted-asymmetric-weights-unsorted`: asymmetric weights, both unsorted

As a composite estimator, AvgSpread tests both individual Spread computations and the weighted combination. Unsorted variants verify end-to-end correctness including the weighting formula.

6.2. AvgSpreadBounds

$$\text{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = [L_A, U_A]$$

where $\alpha = \frac{\text{misrate}}{2}$, $[L_x, U_x] = \text{SpreadBounds}(\mathbf{x}, \alpha)$, $\left[L_y, U_y\right] = \text{SpreadBounds}(\mathbf{y}, \alpha)$, $w_x = \frac{n}{n+m}$, $w_y = \frac{m}{n+m}$, and $[L_A, U_A] = \left[w_x L_x + w_y L_y, w_x U_x + w_y U_y\right]$.

Robust bounds on $\text{AvgSpread}(\mathbf{x}, \mathbf{y})$ with specified coverage.

Interpretation — misrate is probability that true avg spread falls outside bounds
Domain — any real numbers, $n \geq 2$, $m \geq 2$, $\alpha \geq 2^{1-\lfloor \frac{n}{2} \rfloor}$ and $\alpha \geq 2^{1-\lfloor \frac{m}{2} \rfloor}$
Assumptions — `sparity(x)`, `sparity(y)`
Unit — same as measurements
Note — Bonferroni combination of two SpreadBounds calls with equal split $\alpha = \frac{\text{misrate}}{2}$; no independence assumption needed; randomized pairing and cutoff, conservative with ties

Properties

Symmetry $\text{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \text{misrate}) = \text{AvgSpreadBounds}(\mathbf{y}, \mathbf{x}, \text{misrate})$ (equal split)
Shift invariance adding constants to $\mathbf{x}$ and/or $\mathbf{y}$ does not change bounds
Scale equivariance $\text{AvgSpreadBounds}(k \cdot \mathbf{x}, k \cdot \mathbf{y}, \text{misrate}) = |k| \cdot \text{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$
Non-negativity bounds are non-negative
Monotonicity in misrate smaller misrate produces wider bounds

Example

`AvgSpreadBounds([1..30], [21..50], 0.02)` returns bounds containing `AvgSpread`

AvgSpreadBounds provides distribution-free uncertainty bounds for the pooled spread: the weighted average of the two sample spreads. The algorithm computes separate SpreadBounds for each sample using an equal Bonferroni split and then combines them linearly with weights $\frac{n}{n+m}$ and $\frac{m}{n+m}$. This guarantees that the probability of missing the true AvgSpread is at most misrate without requiring independence between samples.

Minimum misrate — because $\alpha = \frac{\text{misrate}}{2}$ must satisfy the per-sample minimum, the overall misrate must be large enough for both samples:

$$\text{misrate} \geq 2 \cdot \max\left(2^{1-\lfloor \frac{n}{2} \rfloor}, 2^{1-\lfloor \frac{m}{2} \rfloor}\right)$$

6.2.1. Algorithm

The AvgSpreadBounds estimator constructs bounds on the pooled spread by combining two independent SpreadBounds calls through a Bonferroni split.

The algorithm proceeds as follows:

1. **Equal Bonferroni split** — Set $\alpha = \frac{\text{misrate}}{2}$. Each per-sample bounds call uses half the total error budget.
2. **Per-sample bounds** — Compute $[L_x, U_x] = \text{SpreadBounds}(\mathbf{x}, \alpha)$ and $[L_y, U_y] = \text{SpreadBounds}(\mathbf{y}, \alpha)$ (see SpreadBounds).
3. **Weighted linear combination** — With weights $w_x = \frac{n}{n+m}$ and $w_y = \frac{m}{n+m}$, return:

$$[L_A, U_A] = \left[w_x L_x + w_y L_y, w_x U_x + w_y U_y\right]$$

By Bonferroni's inequality, the probability that both per-sample bounds simultaneously cover their respective true spreads is at least $1 - 2\alpha = 1 - \text{misrate}$. Since AvgSpread is a weighted average of the individual spreads, the linear combination of the bounds covers the true AvgSpread whenever both individual bounds hold.

```
using Pragmastat.Algorithms;
using Pragmastat.Exceptions;
using Pragmastat.Functions;
using Pragmastat.Internal;

namespace Pragmastat.Estimators;

/// <summary>
/// Distribution-free bounds for AvgSpread via Bonferroni combination of SpreadBounds.
/// Uses equal split alpha = misrate / 2.
/// </summary>
internal class AvgSpreadBoundsEstimator : ITwoSampleBoundsEstimator
{
  internal static readonly AvgSpreadBoundsEstimator Instance = new();

  public Bounds Estimate(Sample x, Sample y, Probability misrate)
  {
    return Estimate(x, y, misrate, null);
  }

  public Bounds Estimate(Sample x, Sample y, Probability misrate, string? seed)
  {
    Assertion.MatchedUnit(x, y);

    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
      throw AssumptionException.Domain(Subject.Misrate);

    int n = x.Size;
    int m = y.Size;
    if (n < 2)
```

```
      throw AssumptionException.Domain(Subject.X);
    if (m < 2)
      throw AssumptionException.Domain(Subject.Y);

    double alpha = misrate / 2.0;
    double minX = MinAchievableMisrate.OneSample(n / 2);
    double minY = MinAchievableMisrate.OneSample(m / 2);
    if (alpha < minX || alpha < minY)
      throw AssumptionException.Domain(Subject.Misrate);

    if (FastSpread.Estimate(x.SortedValues, isSorted: true) <= 0)
      throw AssumptionException.Sparity(Subject.X);
    if (FastSpread.Estimate(y.SortedValues, isSorted: true) <= 0)
      throw AssumptionException.Sparity(Subject.Y);

    Bounds boundsX = seed == null
      ? SpreadBoundsEstimator.Instance.Estimate(x, alpha)
      : SpreadBoundsEstimator.Instance.Estimate(x, alpha, seed);
    Bounds boundsY = seed == null
      ? SpreadBoundsEstimator.Instance.Estimate(y, alpha)
      : SpreadBoundsEstimator.Instance.Estimate(y, alpha, seed);

    double weightX = (double)n / (n + m);
    double weightY = (double)m / (n + m);

    double lower = weightX * boundsX.Lower + weightY * boundsY.Lower;
    double upper = weightX * boundsX.Upper + weightY * boundsY.Upper;
    return new Bounds(lower, upper, x.Unit);
  }
}
```

6.2.2. Tests

$$\mathrm{AvgSpreadBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate}) = [L_A, U_A]$$

Let $\alpha = \frac{\mathrm{misrate}}{2}$ (equal Bonferroni split). Compute $[L_x, U_x] = \mathrm{SpreadBounds}(\mathbf{x}, \alpha)$ and $[L_y, U_y] = \mathrm{SpreadBounds}(\mathbf{y}, \alpha)$ using disjoint-pair sign-test inversion (see SpreadBounds). Let $w_x = \frac{n}{n+m}$ and $w_y = \frac{m}{n+m}$. Return $[L_A, U_A] = \left[w_x L_x + w_y L_y, w_x U_x + w_y U_y\right]$.

The AvgSpreadBounds test suite validates:

- bounds are well-formed ($L_A \leq U_A$ and non-negative)
- shift invariance and scale equivariance
- monotonicity in misrate
- symmetry under swapping x and y (with equal split)
- error cases for invalid inputs and misrate domain violations

Because SpreadBounds is randomized, tests fix a seed to make outputs deterministic. Both SpreadBounds calls use the same seed (two identical RNG streams).

Minimum misrate constraint — the equal split requires

$$\alpha \geq 2^{1-\lfloor \frac{n}{2} \rfloor}$$

and

$$\alpha \geq 2^{1-\lfloor \frac{m}{2} \rfloor}$$

,

so

$$\mathrm{misrate} \geq 2 \cdot \max\left(2^{1-\lfloor \frac{n}{2} \rfloor}, 2^{1-\lfloor \frac{m}{2} \rfloor}\right)$$

.

6.3. Median

$$\text{Median}(\mathbf{x}) = \begin{cases} x_{((n+1)/2)} & \text{if } n \text{ is odd} \\ \frac{x_{(n/2)}+x_{(n/2+1)}}{2} & \text{if } n \text{ is even} \end{cases}$$

The value splitting a sorted sample into two equal parts.

Also known as — 50th percentile, second quartile (Q2)
Asymptotic — value where $P(X \leq \text{Median}) = 0.5$
Complexity — $O(n)$
Domain — any real numbers
Unit — same as measurements

Notation

$x_{(1)}, \ldots, x_{(n)}$	order statistics (sorted sample)

Properties

Shift equivariance	$\text{Median}(\mathbf{x} + k) = \text{Median}(\mathbf{x}) + k$
Scale equivariance	$\text{Median}(k \cdot \mathbf{x}) = k \cdot \text{Median}(\mathbf{x})$

Example

```
Median([1, 2, 3, 4, 5]) = 3
Median([1, 2, 3, 4]) = 2.5
```

Median provides maximum protection against outliers and corrupted data. It achieves a 50% breakdown point, meaning that up to half of the data can be arbitrarily bad before the estimate becomes meaningless. However, this extreme robustness comes at a cost: the median is less precise than Center when data is clean. For most practical applications, Center offers a better tradeoff (29% breakdown with 95% efficiency). Reserve Median for situations with suspected contamination levels above 29% or when the strongest possible robustness guarantee is needed.

6.3.1. Algorithm

The Median algorithm sorts the sample and selects the middle element:

1. **Sort** — Arrange the sample to obtain $x_{(1)} \leq x_{(2)} \leq \ldots \leq x_{(n)}$.
2. **Select** — If n is odd, return $x_{((n+1)/2)}$. If n is even, return $\frac{x_{(n/2)}+x_{(n/2+1)}}{2}$.

The time complexity is $O(n \log n)$ dominated by the sort. This standard algorithm is used as a building block by other estimators; no specialized implementation is needed beyond the language's built-in sort.

6.3.2. Tests

$$\mathrm{Median}(\mathbf{x})$$

No reference test data exists for Median yet. The `tests/median/` directory has not been created. Test cases will be added when the Median test generator is implemented.

6.4. SignMargin

$$\mathrm{SignMargin}(n, \mathrm{misrate})$$

Randomized exclusion count for disjoint-pair sign-test bounds.

Purpose — determines extreme pairwise absolute differences to exclude when constructing bounds
Based on — $\mathrm{Binomial}\left(n, \frac{1}{2}\right)$ CDF inversion with randomized cutoff between adjacent grid points
Returns — total margin split evenly between lower and upper tails
Used by — SpreadBounds to select appropriate order statistics of disjoint-pair differences
Complexity — exact for all n
Domain — $n \geq 1$, $\mathrm{misrate} \geq 2^{1-n}$
Unit — count
Note — randomized to match the requested misrate exactly (deterministic rounding is conservative)

Properties

Bounds $0 \leq \mathrm{SignMargin}(n, \mathrm{misrate}) \leq 2n$
Monotonicity lower misrate $\to$ smaller margin $\to$ wider bounds

Example

Each call returns one of two adjacent grid points (randomized):

`SignMargin(10, 0.05)` returns 2 or 4
`SignMargin(15, 0.01)` returns 4 or 6
`SignMargin(30, 1e-4)` returns 8 or 10

This is a supporting function that SpreadBounds uses internally, so most users do not need to call it directly. It calculates how many extreme disjoint-pair absolute differences should be excluded when constructing bounds, based on the number of pairs and the desired error rate. Because the $\mathrm{Binomial}\left(n, \frac{1}{2}\right)$ CDF is discrete, exact matching of an arbitrary misrate requires randomizing the cutoff between two adjacent integer values. A lower misrate (higher confidence) results in a smaller margin, which produces wider bounds.

6.4.1. Algorithm

The SignMargin function determines the exclusion count for disjoint-pair sign-test bounds by inverting the $\mathrm{Binomial}(n, 1/2)$ CDF.

Given n pairs and a desired misrate, the algorithm finds the number of extreme order statistics to exclude so that the resulting bounds contain the true parameter with probability $1 - \mathrm{misrate}$.

Binomial CDF computation

Under the symmetry assumption, the number of positive signs among n disjoint-pair differences follows $\mathrm{Binomial}(n, 1/2)$. The CDF is computed exactly:

$$\Pr(W \le k) = \sum_{i=0}^{k} \binom{n}{i} 2^{-n}$$

The algorithm evaluates this sum incrementally, accumulating probabilities until the cumulative probability reaches $\frac{\mathrm{misrate}}{2}$.

Grid point identification

Because the Binomial CDF is a step function, the exact misrate typically falls between two adjacent grid points k and $k+1$. The algorithm identifies these adjacent values: k_{lo} is the largest integer where $\Pr(W \le k_{\mathrm{lo}}) < \frac{\mathrm{misrate}}{2}$ and $k_{\mathrm{hi}} = k_{\mathrm{lo}} + 1$.

Randomized cutoff

To match the requested misrate exactly rather than conservatively, the algorithm interpolates between the two grid points. It computes a probability p such that using margin $2k_{\mathrm{lo}}$ with probability p and margin $2k_{\mathrm{hi}}$ with probability $1 - p$ yields an expected coverage of exactly $1 - \mathrm{misrate}$. A uniform random draw determines which margin to return.

This randomization ensures that the bounds are calibrated exactly at the requested error rate under weak continuity, rather than being conservative due to the discreteness of the Binomial distribution.

```
using Pragmastat.Exceptions;

namespace Pragmastat.Functions;

/// <summary>
/// SignMargin function for one-sample bounds based on Binomial(n, 0.5).
/// </summary>
internal class SignMargin
{
  public static readonly SignMargin Instance = new();

  public int CalcRandomized(int n, double misrate, Pragmastat.Randomization.Rng rng)
  {
    if (n <= 0)
      throw AssumptionException.Domain(Subject.X);
    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
```

```
      throw AssumptionException.Domain(Subject.Misrate);

    double minMisrate = MinAchievableMisrate.OneSample(n);
    if (misrate < minMisrate)
      throw AssumptionException.Domain(Subject.Misrate);

    double target = misrate / 2;
    if (target <= 0)
      return 0;
    if (target >= 1)
      return checked(n * 2);

    var split = CalcSplit(n, target);
    int rLow = split.RLow;
    double logCdf = split.LogCdf;
    double logPmfHigh = split.LogPmfHigh;

    double logTarget = Math.Log(target);
    double logNum = logTarget > logCdf ? LogSubExp(logTarget, logCdf) :
double.NegativeInfinity;
    double p = (IsFinite(logPmfHigh) && IsFinite(logNum))
      ? Math.Exp(logNum - logPmfHigh)
      : 0.0;
    if (p < 0)
      p = 0;
    else if (p > 1)
      p = 1;

    double u = rng.UniformDouble();
    int r = u < p ? rLow + 1 : rLow;
    return checked(r * 2);
  }

  private readonly struct SplitResult
  {
    public readonly int RLow;
    public readonly double LogCdf;
    public readonly double LogPmfHigh;

    public SplitResult(int rLow, double logCdfLow, double logPmfHigh)
    {
      RLow = rLow;
      LogCdf = logCdfLow;
      LogPmfHigh = logPmfHigh;
    }
  }

  private static SplitResult CalcSplit(int n, double target)
  {
    double logTarget = Math.Log(target);

    double logPmf = -n * Math.Log(2);
    double logCdf = logPmf;
```

```
    int rLow = 0;

    if (logCdf > logTarget)
      return new SplitResult(0, logCdf, logPmf);

    for (int k = 1; k <= n; k++)
    {
      double logPmfNext = logPmf + Math.Log(n - k + 1) - Math.Log(k);
      double logCdfNext = LogAddExp(logCdf, logPmfNext);

      if (logCdfNext > logTarget)
        return new SplitResult(rLow, logCdf, logPmfNext);

      rLow = k;
      logPmf = logPmfNext;
      logCdf = logCdfNext;
    }

    return new SplitResult(rLow, logCdf, double.NegativeInfinity);
  }

  private static double LogAddExp(double a, double b)
  {
    if (double.IsNegativeInfinity(a))
      return b;
    if (double.IsNegativeInfinity(b))
      return a;
    double m = Math.Max(a, b);
    return m + Math.Log(Math.Exp(a - m) + Math.Exp(b - m));
  }

  private static double LogSubExp(double a, double b)
  {
    if (double.IsNegativeInfinity(b))
      return a;
    double diff = Math.Exp(b - a);
    if (diff >= 1.0)
      return double.NegativeInfinity;
    return a + Math.Log(1.0 - diff);
  }

  private static bool IsFinite(double value)
  {
    return !double.IsNaN(value) && !double.IsInfinity(value);
  }
}
```

6.4.2. Tests

$$\mathrm{SignMargin}(n, \mathrm{misrate})$$

No reference test data exists for SignMargin yet. The `tests/sign-margin/` directory has not been created. Test cases will be added when the SignMargin test generator is implemented.

6.5. PairwiseMargin

$$\text{PairwiseMargin}(n, m, \text{misrate})$$

Exclusion count for dominance-based bounds.

Purpose — determines extreme pairwise differences to exclude when constructing bounds
Based on — distribution of $\text{Dominance}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}(x_i > y_j)$ under random sampling
Returns — total margin split evenly between lower and upper tails
Used by — ShiftBounds to select appropriate order statistics
Complexity — exact for small samples, approximated for large
Domain — $n, m \geq 1$, $\text{misrate} > \frac{2}{\binom{n+m}{n}}$ (minimum achievable)
Unit — count
Note — assumes weak continuity (ties from measurement resolution are tolerated)

Properties

Symmetry $\text{PairwiseMargin}(n, m, \text{misrate}) = \text{PairwiseMargin}(m, n, \text{misrate})$
Bounds $0 \leq \text{PairwiseMargin}(n, m, \text{misrate}) \leq nm$

Example

```
PairwiseMargin(30, 30, 1e-6) = 276
PairwiseMargin(30, 30, 1e-4) = 390
PairwiseMargin(30, 30, 1e-3) = 464
```

This is a supporting function that ShiftBounds uses internally, so most users do not need to call it directly. It calculates how many extreme pairwise differences should be excluded when constructing bounds, based on sample sizes and the desired error rate. A lower misrate (higher confidence) results in a smaller margin, which produces wider bounds. The function automatically chooses between exact computation for small samples and a fast approximation for large samples.

6.5.1. Algorithm

The PairwiseMargin function determines how many extreme pairwise differences to exclude when constructing bounds around $\text{Shift}(\mathbf{x}, \mathbf{y})$. Given samples $\mathbf{x} = (x_1, ..., x_n)$ and $\mathbf{y} = (y_1, ..., y_m)$, the ShiftBounds estimator computes all nm pairwise differences $z_{ij} = x_i - y_j$ and sorts them. The bounds select specific order statistics from this sorted sequence: $\left[z_{(k_{\text{left}})}, z_{(k_{\text{right}})}\right]$. The challenge lies in determining which order statistics produce bounds that contain the true shift $\text{Shift}[X, Y]$ with probability $1 - \text{misrate}$.

Random sampling creates natural variation in pairwise differences. Even when populations have identical distributions, sampling variation produces both positive and negative differences. The margin function quantifies this sampling variability: it specifies how many extreme pairwise differences could occur by chance with probability misrate. For symmetric bounds, this margin splits evenly between the tails, giving $k_{\text{left}} = \left\lfloor \frac{\text{PairwiseMargin}(n,m,\text{misrate})}{2} \right\rfloor + 1$ and $k_{\text{right}} = nm - \left\lfloor \frac{\text{PairwiseMargin}(n,m,\text{misrate})}{2} \right\rfloor$.

Computing the margin requires understanding the distribution of pairwise comparisons. Each pairwise difference corresponds to a comparison: $x_i - y_j > 0$ exactly when $x_i > y_j$. This connection motivates the dominance function:

$$\text{Dominance}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}(x_i > y_j)$$

The dominance function counts how many pairwise comparisons favor $\mathbf{x}$ over $\mathbf{y}$. Both Shift and Dominance operate on the same collection of nm pairwise differences. The Shift estimator examines difference values, returning the median as a location estimate. The Dominance function examines difference signs, counting how many comparisons produce positive differences. While Shift provides the estimate itself, Dominance determines which order statistics form reliable bounds around that estimate.

When populations have equivalent distributions, Dominance concentrates near $nm/2$ by symmetry. The distribution of Dominance across all possible sample orderings determines reliable bounds. If Dominance deviates from $nm/2$ by at least $k/2$ with probability misrate, then the interval excluding the k most extreme pairwise differences contains zero with probability $1 - \text{misrate}$. Translation invariance extends this relationship to arbitrary shifts: the margin computed from the comparison distribution applies regardless of the true shift value.

Two computational approaches provide the distribution of Dominance: exact calculation for small samples and approximation for large samples.

Exact method

Small sample sizes allow exact computation without approximation. The exact approach exploits a fundamental symmetry: under equivalent populations, all $\binom{n+m}{n}$ orderings of the combined measurements occur with equal probability. This symmetry enables direct calculation of how many orderings produce each comparison count.

Direct computation faces a combinatorial challenge. Enumerating all orderings to count comparison outcomes requires substantial resources. For samples beyond a few dozen measurements, naive implementation becomes impractical.

Löffler's recurrence relation (Löffler (1982)) resolves this through algebraic structure. The recurrence exploits cycle properties in the comparison distribution, reducing memory requirements while maintaining exact calculation. The algorithm builds cumulative probabilities sequentially until reaching the threshold corresponding to the desired error rate. This approach extends practical exact computation to combined sample sizes of several hundred.

Define $p_{n,m}(c)$ as the number of orderings producing exactly c comparisons favoring x. The probability mass function becomes:

$$\Pr(\text{Dominance} = c) = p_{n,m} \frac{c}{\binom{n+m}{n}}$$

A direct recurrence follows from considering the largest measurement. The rightmost element comes from either x (contributing m comparisons) or y (contributing zero):

$$p_{n,m}(c) = p_{n-1,m}(c - m) + p_{n,m-1}(c)$$

with base cases $p_{n,0}(0) = 1$ and $p_{0,m}(0) = 1$.

Direct implementation requires $O(n \cdot m \cdot nm)$ time and $O(nm)$ memory. An alternative recurrence (Löffler (1982)) exploits cycle structure:

$$p_{n,m}(c) = \frac{1}{c} \sum_{i=0}^{c-1} p_{n,m}(i) \cdot \sigma_{n,m}(c - i)$$

where $\sigma_{n,m}(d)$ captures structural properties through divisors:

$$\sigma_{n,m}(d) = \sum_{k|d} \varepsilon_k \cdot k, \quad \varepsilon_k = \begin{cases} 1 & \text{if } 1 \le k \le n \\ -1 & \text{if } m+1 \le k \le m+n \\ 0 & \text{otherwise} \end{cases}$$

This reduces memory to $O(nm)$ and enables efficient computation through $c = nm$.

The algorithm computes cumulative probabilities $\Pr(\text{Dominance} \le c)$ sequentially until the threshold $\text{misrate}/2$ is exceeded. By symmetry, the lower and upper thresholds determine the total margin $\text{PairwiseMargin} = 2c$.

The sequential computation proceeds incrementally. Starting from $u = 0$ with base probability $p_{n,m}(0) = 1$, the algorithm computes $p_{n,m}(1)$, then $p_{n,m}(2)$, and so on, accumulating the cumulative distribution function with each step. The loop terminates as soon as $\Pr(\text{Dominance} \le u)$ reaches $\text{misrate}/2$, returning the threshold value u without computing further probabilities.

This sequential approach performs particularly well for small misrates. For $\text{misrate} = 10^{-6}$, the threshold u typically remains small even with large sample sizes, requiring only a few iterations regardless of whether n and m equal 50 or 200. The algorithm computes only the extreme tail probabil-

ities needed to reach the threshold, never touching the vast majority of probability mass concentrated near $nm/2$. This efficiency advantage grows as misrates decrease: stricter bounds require fewer computed values, making exact calculation particularly attractive for high-confidence applications.

Approximate method

Large samples make exact computation impractical. The dominance count $\mathrm{Dominance}$ concentrates near $nm/2$ with variance $nm(n+m+1)/12$. A basic $\underline{\text{Additive}}$ (Normal) approximation suffices asymptotically:

$$\mathrm{Dominance} \approx \underline{\text{Additive}}\Big(nm/2, \sqrt{nm(n+m+1)/12}\Big)$$

This approximation underestimates tail probabilities for moderate sample sizes. The $\underline{\text{Additive}}$ (Normal) approximation provides a baseline but fails to capture the true distribution shape in the tails, producing mis-calibrated probabilities that become problematic for small error rates.

The Edgeworth expansion refines this approximation through moment-based corrections (Fix & Hodges (1955)). The expansion starts with the $\underline{\text{Additive}}$ (Normal) cumulative distribution as a baseline, then adds correction terms that account for the distribution's asymmetry (skewness) and tail weight (kurtosis). These corrections use Hermite polynomials to adjust the baseline curve where the $\underline{\text{Additive}}$ (Normal) approximation deviates most from the true distribution. The first few correction terms typically achieve the practical balance between accuracy and computational cost, substantially improving tail probability estimates compared to the basic approximation.

The standardized comparison count (with continuity correction):

$$z = \frac{c - nm/2 - 0.5}{\sqrt{nm(n+m+1)/12}}$$

produces the approximated cumulative distribution:

$$\Pr(\mathrm{Dominance} \leq c) \approx \Phi(z) + e_3\varphi^{(3)}(z) + e_5\varphi^{(5)}(z) + e_7\varphi^{(7)}(z)$$

where Φ denotes the standard $\underline{\text{Additive}}$ (Normal) CDF.

The correction coefficients depend on standardized moments:

$$e_3 = \frac{1}{24}\left(\frac{\mu_4}{\mu_2^2} - 3\right), \quad e_5 = \frac{1}{720}\left(\frac{\mu_6}{\mu_2^3} - 15\frac{\mu_4}{\mu_2^2} + 30\right), \quad e_7 = \frac{35}{40320}\left(\frac{\mu_4}{\mu_2^2} - 3\right)^2$$

The moments μ_2, μ_4, μ_6 are computed from sample sizes:

$$\mu_2 = \frac{nm(n+m+1)}{12}$$

$$\mu_4 = \frac{nm(n+m+1)}{240}(5nm(n+m) - 2(n^2+m^2) + 3nm - 2(n+m))$$

$$\mu_6 = \frac{nm(n+m+1)}{4032}(35n^2m^2(n^2+m^2) + 70n^3m^3 - 42nm(n^3+m^3)$$
$$-14n^2m^2(n+m) + 16(n^4+m^4) - 52nm(n^2+m^2)$$
$$-43n^2m^2 + 32(n^3+m^3) + 14nm(n+m)$$
$$+8(n^2+m^2) + 16nm - 8(n+m))$$

The correction terms use Hermite polynomials:

$$\varphi^{(k)}(z) = -\varphi(z)H_{k(z)}$$

$$H_3(z) = z^3 - 3z, \quad H_5(z) = z^5 - 10z^3 + 15z, \quad H_7(z) = z^7 - 21z^5 + 105z^3 - 105z$$

Binary search locates the threshold value efficiently. The algorithm maintains a search interval $[a, b]$ initialized to $[0, nm]$. Each iteration computes the midpoint $c = (a + b)/2$ and evaluates the Edgeworth CDF at c. If $\Pr(\text{Dominance} \leq c) < \text{misrate}/2$, the threshold lies above c and the search continues in $[c, b]$. If $\Pr(\text{Dominance} \leq c) \geq \text{misrate}/2$, the threshold lies below c and the search continues in $[a, c]$. The loop terminates when a and b become adjacent, requiring $O(\log(nm))$ CDF evaluations.

This binary search exhibits uniform performance across misrate values. Whether computing bounds for $\text{misrate} = 10^{-6}$ or $\text{misrate} = 0.05$, the algorithm performs the same number of iterations determined solely by the sample sizes. Each CDF evaluation costs constant time regardless of the threshold location, making the approximate method particularly efficient for large samples where exact computation becomes impractical. The logarithmic scaling ensures that doubling the sample size adds only one additional iteration, enabling practical computation for samples in the thousands or tens of thousands.

The toolkit selects between exact and approximate computation based on combined sample size: exact method for $n + m \leq 400$, approximate method for $n + m > 400$. The exact method guarantees correctness but scales as $O(nm)$ memory and $O((nm)^2)$ time. For $n = m = 200$, this requires 40,000 memory locations. The approximate method achieves 1% accuracy with $O(\log(nm))$ constant-time evaluations. For $n = m = 10000$, the approximate method completes in milliseconds versus minutes for exact computation.

Both methods handle discrete data. Repeated measurements produce tied pairwise differences, creating plateaus in the sorted sequence. The exact method counts orderings without assuming continuity. The approximate method's moment-based corrections capture the actual distribution shape regardless of discreteness.

Minimum achievable misrate

The misrate parameter controls how many extreme pairwise differences the bounds exclude. Lower misrate produces narrower bounds with higher confidence but requires excluding fewer extreme values. However, sample size limits how small misrate can meaningfully become.

Consider the most extreme configuration: all measurements from x exceed all measurements from y, giving $x_1, ..., x_n > y_1, ..., y_m$. Under equivalent populations, this arrangement occurs purely by chance.

The probability equals the chance of having all n elements from x occupy the top n positions among $n + m$ total measurements:

$$\text{misrate}_{\text{min}} = \frac{2}{\binom{n+m}{n}} = \frac{2 \cdot n! \cdot m!}{(n+m)!}$$

This represents the minimum probability of the most extreme ordering under random sampling. Setting $\text{misrate} < \text{misrate}_{\text{min}}$ makes bounds construction problematic. The exact distribution of Dominance cannot support misrates smaller than the probability of its most extreme realization. Attempting to construct bounds with $\text{misrate} < \text{misrate}_{\text{min}}$ forces the algorithm to exclude zero pairwise differences from the tails, making $\text{PairwiseMargin} = 0$. The resulting bounds span all nm pairwise differences, returning $\left[z_{(1)}, z_{(nm)}\right]$ regardless of the desired confidence level. These bounds convey no useful information beyond the range of observed pairwise differences.

For small samples, $\text{misrate}_{\text{min}}$ can exceed commonly used values. With $n = m = 6$, the minimum misrate equals $2/\binom{12}{6} \approx 0.00217$, making the typical choice of $\text{misrate} = 10^{-3}$ impossible. With $n = m = 4$, the minimum becomes $2/\binom{8}{4} \approx 0.0286$, exceeding even $\text{misrate} = 0.01$.

The table below shows $\text{misrate}_{\text{min}}$ for small sample sizes:

	1	2	3	4	5	6	7	8	9	10
1	1.000000	0.666667	0.500000	0.400000	0.333333	0.285714	0.250000	0.222222	0.200000	0.181818
2	0.666667	0.333333	0.200000	0.133333	0.095238	0.071429	0.055556	0.044444	0.036364	0.030303
3	0.500000	0.200000	0.100000	0.057143	0.035714	0.023810	0.016667	0.012121	0.009091	0.006993
4	0.400000	0.133333	0.057143	0.028571	0.015873	0.009524	0.006061	0.004040	0.002797	0.001998
5	0.333333	0.095238	0.035714	0.015873	0.007937	0.004329	0.002525	0.001554	0.000999	0.000666
6	0.285714	0.071429	0.023810	0.009524	0.004329	0.002165	0.001166	0.000666	0.000400	0.000250
7	0.250000	0.055556	0.016667	0.006061	0.002525	0.001166	0.000583	0.000311	0.000175	0.000103
8	0.222222	0.044444	0.012121	0.004040	0.001554	0.000666	0.000311	0.000155	0.000082	0.000046
9	0.200000	0.036364	0.009091	0.002797	0.000999	0.000400	0.000175	0.000082	0.000041	0.000022
10	0.181818	0.030303	0.006993	0.001998	0.000666	0.000250	0.000103	0.000046	0.000022	0.000011

For meaningful bounds construction, choose $\text{misrate} > \text{misrate}_{\text{min}}$. This ensures the margin function excludes at least some extreme pairwise differences, producing bounds narrower than the full range. When working with small samples, verify that the desired misrate exceeds $\text{misrate}_{\text{min}}$ for the given sample sizes. With moderate sample sizes ($n, m \geq 15$), $\text{misrate}_{\text{min}}$ drops below 10^{-8}, making standard choices like $\text{misrate} = 10^{-6}$ feasible.

```
using System;
using JetBrains.Annotations;
using Pragmastat.Exceptions;
using Pragmastat.Internal;

namespace Pragmastat.Functions;

/// <summary>
/// PairwiseMargin function
```

```
/// </summary>
/// <param name="threshold">The maximum value for n+m, after which implementation switches
from exact to approx</param>
internal class PairwiseMargin(int threshold = PairwiseMargin.MaxExactSize)
{
  public static readonly PairwiseMargin Instance = new();

  private const int MaxExactSize = 400;

  [PublicAPI]
  public int Calc(int n, int m, double misrate)
  {
    if (n <= 0)
      throw AssumptionException.Domain(Subject.X);
    if (m <= 0)
      throw AssumptionException.Domain(Subject.Y);
    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
      throw AssumptionException.Domain(Subject.Misrate);

    double minMisrate = MinAchievableMisrate.TwoSample(n, m);
    if (misrate < minMisrate)
      throw AssumptionException.Domain(Subject.Misrate);

    return n + m <= threshold
      ? CalcExact(n, m, misrate)
      : CalcApprox(n, m, misrate);
  }

  internal int CalcExact(int n, int m, double misrate)
  {
    int raw = CalcExactRaw(n, m, misrate / 2);
    return checked(raw * 2);
  }

  internal int CalcApprox(int n, int m, double misrate)
  {
    long raw = CalcApproxRaw(n, m, misrate / 2);
    long margin = raw * 2;
    if (margin > int.MaxValue)
      throw new OverflowException($"Pairwise margin exceeds supported range for n={n},
m={m}");
    return (int)margin;
  }

  // Inversed implementation of Andreas Löffler's (1982) "Über eine Partition der nat.
Zahlen und ihre Anwendung beim U-Test"
  private static int CalcExactRaw(int n, int m, double p)
  {
    double total = n + m < BinomialCoefficientFunction.MaxAcceptableN
      ? BinomialCoefficientFunction.BinomialCoefficient(n + m, m)
      : BinomialCoefficientFunction.BinomialCoefficient(n + m, m * 1.0);

    var pmf = new List<double> { 1 }; // pmf[0] = 1
```

```
    var sigma = new List<double> { 0 }; // sigma[0] is unused

    int u = 0;
    double cdf = 1.0 / total;

    if (cdf >= p)
      return 0;

    while (true)
    {
      u++;
      // Ensure sigma has entry for u
      if (sigma.Count <= u)
      {
        int value = 0;
        for (int d = 1; d <= n; d++)
          if (u % d == 0 && u >= d)
            value += d;
        for (int d = m + 1; d <= m + n; d++)
          if (u % d == 0 && u >= d)
            value -= d;
        sigma.Add(value);
      }

      // Compute pmf[u] using Loeffler recurrence
      double sum = 0;
      for (int i = 0; i < u; i++)
        sum += pmf[i] * sigma[u - i];
      sum /= u;
      pmf.Add(sum);

      cdf += sum / total;
      if (cdf >= p)
        return u;
      if (sum == 0)
        break;
    }

    return pmf.Count - 1;
  }

  // Inverse Edgeworth Approximation
  private static long CalcApproxRaw(int n, int m, double misrate)
  {
    long a = 0;
    long b = (long)n * m;
    while (a < b - 1)
    {
      long c = (a + b) / 2;
      double p = EdgeworthCdf(n, m, c);
      if (p < misrate)
        a = c;
      else
```

```
        b = c;
    }

    return EdgeworthCdf(n, m, b) < misrate ? b : a;
  }

  private static double EdgeworthCdf(int n, int m, long u)
  {
    double nm = (double)n * m;
    double mu = nm / 2.0;
    double su = Sqrt(nm * (n + m + 1) / 12.0);
    // -0.5 continuity correction: computing P(U ≥ u) for a right-tail discrete CDF
    double z = (u - mu - 0.5) / su;
    double phi = Exp(-z.Sqr() / 2) / Sqrt(2 * PI);
    double Phi = AcmAlgorithm209.Gauss(z);

    // Pre-compute powers of n and m for efficiency
    double n2 = (double)n * n;
    double n3 = n2 * n;
    double n4 = n2 * n2;
    double m2 = (double)m * m;
    double m3 = m2 * m;
    double m4 = m2 * m2;

    double mu2 = (double)n * m * (n + m + 1) / 12.0;
    double mu4 =
      (double)n * m * (n + m + 1) *
      (0
       + 5 * m * n * (m + n)
       - 2 * (m2 + n2)
       + 3 * m * n
       - 2 * (n + m)
      ) / 240.0;

    double mu6 =
      (double)n * m * (n + m + 1) *
      (0
       + 35 * m2 * n2 * (m2 + n2)
       + 70 * m3 * n3
       - 42 * m * n * (m3 + n3)
       - 14 * m2 * n2 * (n + m)
       + 16 * (n4 + m4)
       - 52 * n * m * (n2 + m2)
       - 43 * n2 * m2
       + 32 * (m3 + n3)
       + 14 * m * n * (n + m)
       + 8 * (n2 + m2)
       + 16 * n * m
       - 8 * (n + m)
      ) / 4032.0;

    // Pre-compute powers of mu2 and related terms
    double mu2_2 = mu2 * mu2;
```

```
    double mu2_3 = mu2_2 * mu2;
    double mu4_mu2_2 = mu4 / mu2_2;

    // Factorial constants: 4! = 24, 6! = 720, 8! = 40320
    double e3 = (mu4_mu2_2 - 3) / 24.0;
    double e5 = (mu6 / mu2_3 - 15 * mu4_mu2_2 + 30) / 720.0;
    double e7 = 35 * (mu4_mu2_2 - 3) * (mu4_mu2_2 - 3) / 40320.0;

    // Pre-compute powers of z for Hermite polynomials
    double z2 = z * z;
    double z3 = z2 * z;
    double z5 = z3 * z2;
    double z7 = z5 * z2;

    double f3 = -phi * (z3 - 3 * z);
    double f5 = -phi * (z5 - 10 * z3 + 15 * z);
    double f7 = -phi * (z7 - 21 * z5 + 105 * z3 - 105 * z);

    double edgeworth = Phi + e3 * f3 + e5 * f5 + e7 * f7;
    return Min(Max(edgeworth, 0), 1);
  }
}
```

6.5.2. Notes

The Mann-Whitney U test (also known as the Wilcoxon rank-sum test) ranks among the most widely used non-parametric statistical tests, testing whether two independent samples come from the same distribution. Under Additive (Normal) conditions, it achieves nearly the same precision as the Student's t-test, while maintaining reliability under diverse distributional conditions where the t-test fails.

The test operates by comparing all pairs of measurements between the two samples. Given samples $\mathbf{x} = (x_1, ..., x_n)$ and $\mathbf{y} = (y_1, ..., y_m)$, the Mann-Whitney U statistic counts how many pairs satisfy $x_i > y_j$:

$$U = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}(x_i > y_j)$$

If the samples come from the same distribution, U should be near $nm/2$ (roughly half the pairs favor $\mathbf{x}$, half favor $\mathbf{y}$). Large deviations from $nm/2$ suggest the distributions differ.

The test answers: "Could this U value arise by chance if the samples were truly equivalent?" The p-value quantifies this probability. If $p < 0.05$, traditional practice declares the difference "statistically significant".

This approach creates several problems for practitioners:

Binary thinking. The test produces a yes/no answer: reject or fail to reject the null hypothesis. Practitioners typically want to know the magnitude of difference, not just whether one exists.
Arbitrary thresholds. The 0.05 threshold has no universal justification, yet it dominates practice and creates a false dichotomy between $p = 0.049$ and $p = 0.051$.
Hypothesis-centric framework. The test assumes a null hypothesis of "no difference" and evaluates evidence against it. Real questions rarely concern exact equality; practitioners want to know "how different?" rather than "different or not?"
Inverted logic. The natural question is "what shifts are consistent with my data?" The test answers "is this specific shift (zero) consistent with my data?"

The toolkit inverts this framework. Instead of testing whether a hypothesized shift is plausible, we compute which shifts are plausible given the data. This inversion transforms hypothesis testing into bounds estimation.

The mathematical foundation remains the same. The distribution of pairwise comparisons under random sampling determines which order statistics of pairwise differences form reliable bounds. The Mann-Whitney U statistic measures pairwise comparisons ($x_i > y_j$). The Shift estimator uses pairwise differences ($x_i - y_j$). These quantities are mathematically related: a pairwise difference $x_i - y_j$ is positive exactly when $x_i > y_j$. The toolkit renames this comparison count from U to $\mathrm{Dominance}(\mathbf{x}, \mathbf{y})$, clarifying its purpose: measuring how often one sample dominates the other in pairwise comparisons.

The distribution of Dominance determines which order statistics form reliable bounds. Define the margin function:

$$\text{PairwiseMargin}(n, m, \text{misrate}) = \text{number of pairwise differences to exclude from bounds}$$

This function computes how many extreme pairwise differences could occur by chance with probability misrate, based on the distribution of pairwise comparisons.

The PairwiseMargin function requires knowing the distribution of pairwise comparisons under sampling. Two computational approaches exist:

Exact computation (Löffler's algorithm, 1982). Uses a recurrence relation to compute the exact distribution of pairwise comparisons for small samples. Practical for combined sample sizes up to several hundred.
Approximation (Edgeworth expansion, 1955). Refines the normal approximation with correction terms based on higher moments of the distribution. Provides accurate results for large samples where exact computation becomes impractical.

The toolkit automatically selects the appropriate method based on sample sizes, ensuring both accuracy and computational efficiency.

This approach naturally complements Center and Spread:

Center($\mathbf{x}$) uses the median of pairwise averages $(x_i + x_j)/2$
Spread($\mathbf{x}$) uses the median of pairwise differences $|x_i - x_j|$
Shift($\mathbf{x}, \mathbf{y}$) uses the median of pairwise differences $x_i - y_j$
ShiftBounds($\mathbf{x}, \mathbf{y}$, misrate) uses order statistics of the same pairwise differences

All procedures build on pairwise operations. This structural consistency reflects the mathematical unity underlying robust statistics: pairwise operations provide natural robustness while maintaining computational feasibility and statistical efficiency.

The inversion from hypothesis testing to bounds estimation represents a philosophical shift in statistical practice. Traditional methods ask "should I believe this specific hypothesis?" Pragmatic methods ask "what should I believe, given this data?" Bounds provide actionable answers: they tell practitioners which values are plausible, enabling informed decisions without arbitrary significance thresholds.

Traditional Mann-Whitney implementations apply tie correction when samples contain repeated values. This correction modifies variance calculations to account for tied observations, changing p-values and confidence intervals in ways that depend on measurement precision. The toolkit deliberately omits tie correction. Continuous distributions produce theoretically distinct values; observed ties result from finite measurement precision and digital representation. When measurements appear identical, this reflects rounding of underlying continuous variation, not true equality in the measured quantity. Treating ties as artifacts of discretization rather than distributional features simplifies computation while maintaining accuracy. The exact and approximate methods compute comparison distributions without requiring adjustments for tied values, eliminating a source of complexity and potential inconsistency in statistical practice.

Historical Development

The mathematical foundations emerged through decades of refinement. Mann and Whitney (1947) established the distribution of pairwise comparisons under random sampling, creating the theoretical basis for comparing samples through rank-based methods. Their work demonstrated that comparison counts follow predictable patterns regardless of the underlying population distributions.

The original computational approaches suffered from severe limitations. Mann and Whitney proposed a slow exact method requiring exponential resources and a normal approximation that proved grossly inaccurate for practical use. The approximation works reasonably in distribution centers but fails catastrophically in the tails where practitioners most need accuracy. For moderate sample sizes, approximation errors can exceed factors of 10^{11}.

Fix and Hodges (1955) addressed the approximation problem through higher-order corrections. Their expansion adds terms based on the distribution's actual moments rather than assuming perfect normality. This refinement reduces tail probability errors from orders of magnitude to roughly 1%, making approximation practical for large samples where exact computation becomes infeasible.

Löffler (1982) solved the exact computation problem through algorithmic innovation. The naive recurrence requires quadratic memory— infeasible for samples beyond a few dozen measurements. Löffler discovered a reformulation that reduces memory to linear scale, making exact computation practical for combined sample sizes up to several hundred.

Despite these advances, most statistical software continues using the 1947 approximation. The computational literature contains the solutions, but software implementations lag decades behind theoretical developments. This toolkit implements both the exact method for small samples and the refined approximation for large samples, automatically selecting the appropriate approach based on sample sizes.

The shift from hypothesis testing to bounds estimation requires no new mathematics. The same comparison distributions that enable hypothesis tests also determine which order statistics form reliable bounds. Traditional applications ask "is zero plausible?" and answer yes or no. This toolkit asks "which values are plausible?" and answers with an interval. The perspective inverts while the mathematical foundation remains identical.

6.5.3. Tests

$$\mathrm{PairwiseMargin}(n, m, \mathrm{misrate})$$

The PairwiseMargin test suite contains 178 test cases (4 demo + 4 natural + 10 edge + 12 small grid + 148 large grid). The domain constraint $\mathrm{misrate} \geq \frac{2}{\binom{n+m}{n}}$ is enforced; inputs violating this return a domain error. Combinations where the requested misrate falls below the minimum achievable misrate are excluded from the grid.

Demo examples ($n = m = 30$) — from manual introduction:

`demo-1`: $n = 30$, $m = 30$, $\mathrm{misrate} = 10^{-6}$, expected output: 276
`demo-2`: $n = 30$, $m = 30$, $\mathrm{misrate} = 10^{-5}$, expected output: 328
`demo-3`: $n = 30$, $m = 30$, $\mathrm{misrate} = 10^{-4}$, expected output: 390
`demo-4`: $n = 30$, $m = 30$, $\mathrm{misrate} = 10^{-3}$, expected output: 464

These demo cases match the reference values used throughout the manual to illustrate ShiftBounds construction.

Natural sequences ($[n, m] \in \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \times 2$ misrates, filtered by min misrate) — 4 tests:

Misrate values: $\mathrm{misrate} \in \{10^{-1}, 10^{-2}\}$
After filtering by $\mathrm{misrate} \geq \frac{2}{\binom{n+m}{n}}$, only 4 combinations survive:
`natural-3-3-mr1`: $n = 3$, $m = 3$, $\mathrm{misrate} = 0.1$, expected output: 0
`natural-3-4-mr1`: $n = 3$, $m = 4$, $\mathrm{misrate} = 0.1$
`natural-4-3-mr1`: $n = 4$, $m = 3$, $\mathrm{misrate} = 0.1$
`natural-4-4-mr1`: $n = 4$, $m = 4$, $\mathrm{misrate} = 0.1$

Edge cases — boundary condition validation (10 tests):

`boundary-min`: $n = 1$, $m = 1$, $\mathrm{misrate} = 1.0$ (minimum samples with maximum misrate, expected output: 0)
`boundary-zero-margin-small`: $n = 20$, $m = 20$, $\mathrm{misrate} = 10^{-6}$ (strict misrate with sufficient samples)
`boundary-loose`: $n = 5$, $m = 5$, $\mathrm{misrate} = 0.9$ (very permissive misrate)
`symmetry-2-5`: $n = 2$, $m = 5$, $\mathrm{misrate} = 0.1$ (tests symmetry property)
`symmetry-5-2`: $n = 5$, $m = 2$, $\mathrm{misrate} = 0.1$ (symmetric counterpart, same output as above)
`symmetry-3-7`: $n = 3$, $m = 7$, $\mathrm{misrate} = 0.05$ (asymmetric sizes)
`symmetry-7-3`: $n = 7$, $m = 3$, $\mathrm{misrate} = 0.05$ (symmetric counterpart)
`asymmetry-extreme-1-100`: $n = 1$, $m = 100$, $\mathrm{misrate} = 0.1$ (extreme size difference)
`asymmetry-extreme-100-1`: $n = 100$, $m = 1$, $\mathrm{misrate} = 0.1$ (reversed extreme)
`asymmetry-extreme-2-50`: $n = 2$, $m = 50$, $\mathrm{misrate} = 0.05$ (highly unbalanced)

These edge cases validate correct handling of boundary conditions, the symmetry property $\mathrm{PairwiseMargin}(n, m, \mathrm{misrate}) = \mathrm{PairwiseMargin}(m, n, \mathrm{misrate})$, and extreme asymmetry in sample sizes.

Comprehensive grid — systematic coverage for thorough validation:

Small sample combinations ($[n, m] \in \{1, 2, 3, 4, 5\} \times \{1, 2, 3, 4, 5\} \times 6$ misrates, filtered) — 12 tests:

Misrate values: $\text{misrate} \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$
Combinations where $\text{misrate} < \frac{2}{\binom{n+m}{n}}$ are excluded
Test naming: `n{n}_m{m}_mr{k}` where k is the negative log10 of misrate
Examples:
`n5_m5_mr1`: $n = 5, m = 5, \text{misrate} = 0.1$, expected output: 10
`n5_m5_mr2`: $n = 5, m = 5, \text{misrate} = 0.01$

Large sample combinations ($[n, m] \in \{10, 20, 30, 50, 100\} \times \{10, 20, 30, 50, 100\} \times 6$ misrates, filtered) — 148 tests:

Misrate values: same as small samples
Combinations where $\text{misrate} < \frac{2}{\binom{n+m}{n}}$ are excluded (affects $n = m = 10$ at misrates 10^{-5} and 10^{-6})
Test naming: `n{n}_m{m}_r{k}` where k is the negative log10 of misrate
Examples:
`n10_m10_r1`: $n = 10, m = 10, \text{misrate} = 0.1$, expected output: 56
`n50_m50_r3`: $n = 50, m = 50, \text{misrate} = 0.001$, expected output: 1556
`n100_m100_r6`: $n = 100, m = 100, \text{misrate} = 10^{-6}$, expected output: 6060

The comprehensive grid validates both symmetric ($n = m$) and asymmetric sample size combinations across six orders of magnitude in misrate, ensuring robust coverage of the parameter space.

6.5.4. References

Löffler, A. (1982). *On the calculation of the exact null-distribution of the Wilcoxon-Mann-Whitney U-statistic.*

Fix, E., & Hodges, J. L. (1955). Significance probabilities of the Wilcoxon test. *The Annals of Mathematical Statistics*, *26*(2), 301–312. https://projecteuclid.org/euclid.aoms/1177728547

6.6. SignedRankMargin

$$\mathrm{SignedRankMargin}(n, \mathrm{misrate})$$

Exclusion count for one-sample signed-rank based bounds.

Purpose — determines extreme pairwise averages to exclude when constructing bounds
Based on — Wilcoxon signed-rank distribution under weak symmetry
Returns — total margin split evenly between lower and upper tails
Used by — CenterBounds to select appropriate order statistics
Complexity — exact for $n \leq 63$, approximated for larger
Domain — $n \geq 2$, $\mathrm{misrate} \geq 2^{1-n}$
Unit — count
Note — assumes weak symmetry and weak continuity

Properties

Bounds $0 \leq \mathrm{SignedRankMargin}(n, \mathrm{misrate}) \leq n(n+1)/2$
Monotonicity lower misrate $\to$ smaller margin $\to$ wider bounds

Example

```
SignedRankMargin(10, 0.05) = 18
SignedRankMargin(30, 1e-4) = 112
SignedRankMargin(100, 1e-6) = 706
```

This is a supporting function that CenterBounds uses internally, so most users do not need to call it directly. It calculates how many extreme pairwise averages should be excluded when constructing bounds, based on sample size and the desired error rate. A lower misrate (higher confidence) results in a smaller margin, which produces wider bounds. The function automatically chooses between exact computation for small samples and a fast approximation for large samples.

6.6.1. Algorithm

The $\mathrm{SignedRankMargin}$ function determines how many extreme pairwise averages to exclude when constructing bounds around $\mathrm{Center}(\mathbf{x})$. Given a sample $\mathbf{x} = (x_1, ..., x_n)$, the $\mathrm{CenterBounds}$ estimator computes all $N = \frac{n(n+1)}{2}$ pairwise averages $w_{ij} = \frac{x_i + x_j}{2}$ for $i \leq j$ and sorts them. The bounds select specific order statistics from this sorted sequence: $\left[w_{(k_{\text{left}})}, w_{(k_{\text{right}})}\right]$. The challenge lies in determining which order statistics produce bounds that contain the true center with probability $1 - \mathrm{misrate}$.

The margin function is the one-sample analog of $\mathrm{PairwiseMargin}$. While $\mathrm{PairwiseMargin}$ uses the Mann-Whitney distribution for two-sample comparisons, $\mathrm{SignedRankMargin}$ uses the Wilcoxon signed-rank distribution for one-sample inference. Under the weak symmetry assumption, the signed-rank statistic has a known distribution that enables exact computation of bounds coverage.

For symmetric distributions, consider the signs of deviations from the center. The Wilcoxon signed-rank statistic W sums the ranks of positive deviations:

$$W = \sum_{i=1}^{n} R_i \cdot \mathbb{1}(x_i > \theta)$$

where R_i is the rank of $|x_i - \theta|$ among all $\left|x_j - \theta\right|$, and θ is the true center. Under symmetry, each deviation is equally likely to be positive or negative, giving W a discrete distribution over $\left[0, \frac{n(n+1)}{2}\right]$.

The connection to pairwise averages is fundamental: the k-th order statistic of sorted pairwise averages corresponds to a specific threshold of the signed-rank statistic. By computing the distribution of W, we determine which order statistics provide bounds with the desired coverage.

Two computational approaches provide the distribution of W: exact calculation for small samples and approximation for large samples.

Exact method

Small sample sizes allow exact computation without approximation. The Wilcoxon signed-rank distribution has 2^n equally likely outcomes under symmetry, corresponding to all possible sign patterns for deviations from the center.

Dynamic programming builds the probability mass function efficiently. Define $p(w)$ as the number of sign patterns producing signed-rank statistic equal to w. The recurrence considers whether to include rank i in the positive sum:

$$p_{i(w)} = p_{i-1}(w - i) + p_{i-1}(w)$$

with base case $p_0(0) = 1$. This builds the distribution incrementally, rank by rank.

The algorithm computes cumulative probabilities $\Pr(W \leq w)$ sequentially until the threshold $\mathrm{misrate}/2$ is exceeded. For symmetric two-tailed bounds, the margin becomes $\mathrm{SignedRankMargin} = 2w$. Memory is $O(n^2)$ for storing the probability array, and time is $O(n^3)$ for the full computation.

The sequential computation performs well for small misrates. For $\mathrm{misrate} = 10^{-6}$, the threshold w typically remains small, requiring only iterations through the lower tail regardless of sample size.

Approximate method

Large samples make exact computation impractical. For $n > 63$, the Wilcoxon distribution is approximated using an Edgeworth expansion.

Under symmetry, the signed-rank statistic W has:

$$\mathbb{E}[W] = \frac{n(n+1)}{4}, \quad \mathrm{Var}(W) = n(n+1)\frac{2n+1}{24}$$

The basic normal approximation uses these moments directly, but underestimates tail probabilities for moderate sample sizes.

The Edgeworth expansion refines this through moment-based corrections. The fourth central moment of W is:

$$\mu_4 = \frac{9n^5 + 45n^4 + 65n^3 + 15n^2 - 14n}{480}$$

This enables kurtosis correction:

$$e_3 = \frac{\mu_4 - 3\sigma^4}{24\sigma^4}$$

The approximated CDF becomes:

$$\Pr(W \le w) \approx \Phi(z) + e_3\varphi(z)(z^3 - 3z)$$

where $z = \frac{w-\mu+0.5}{\sigma}$ includes a continuity correction.

Binary search locates the threshold efficiently. Each CDF evaluation costs $O(1)$, and $O(\log N)$ evaluations suffice. The approximate method completes in constant time regardless of sample size.

The toolkit uses exact computation for $n \le 63$ and approximation for $n > 63$. At $n = 63$, the exact method requires arrays of size $2,016$ ($= 63 \times \frac{64}{2}$), which remains practical on modern hardware. The transition at $n = 63$ is determined by the requirement that 2^n fits in a 64-bit integer. The approximation achieves sub-1% accuracy for $n > 100$, making the transition smooth.

Minimum achievable misrate

The $\mathrm{misrate}$ parameter controls how many extreme pairwise averages the bounds exclude. However, sample size limits how small misrate can meaningfully become.

The most extreme configuration occurs when all signs are positive (or all negative): $W = \frac{n(n+1)}{2}$ or $W = 0$. Under symmetry, this extreme occurs with probability:

$$\mathrm{misrate}_{\min} = \frac{2}{2^n} = 2^{1-n}$$

Setting $\mathrm{misrate} < \mathrm{misrate}_{\min}$ makes bounds construction problematic. Pragmastat rejects such requests with a `domain` error.

The table below shows $\text{misrate}_{\text{min}}$ for small sample sizes:

n	2	3	5	7	10
$\text{misrate}_{\text{min}}$	0.5	0.25	0.0625	0.0156	0.00195
max conf	50%	75%	93.75%	98.4%	99.8%

For meaningful bounds construction, choose $\text{misrate} > \text{misrate}_{\text{min}}$. With $n \geq 10$, standard choices like $\text{misrate} = 10^{-3}$ become feasible. With $n \geq 20$, even $\text{misrate} = 10^{-6}$ is achievable.

```
using JetBrains.Annotations;
using Pragmastat.Exceptions;

namespace Pragmastat.Functions;

/// <summary>
/// SignedRankMargin function for one-sample bounds.
/// One-sample analog of PairwiseMargin using Wilcoxon signed-rank distribution.
/// </summary>
/// <param name="threshold">Maximum n for exact computation; larger n uses approximation</
param>
internal class SignedRankMargin(int threshold = SignedRankMargin.MaxExactSize)
{
  public static readonly SignedRankMargin Instance = new();

  private const int MaxExactSize = 63;

  [PublicAPI]
  public int Calc(int n, double misrate)
  {
    if (n <= 0)
      throw AssumptionException.Domain(Subject.X);
    if (double.IsNaN(misrate) || misrate < 0 || misrate > 1)
      throw AssumptionException.Domain(Subject.Misrate);

    double minMisrate = MinAchievableMisrate.OneSample(n);
    if (misrate < minMisrate)
      throw AssumptionException.Domain(Subject.Misrate);

    return n <= threshold
      ? CalcExact(n, misrate)
      : CalcApprox(n, misrate);
  }

  internal int CalcExact(int n, double misrate)
  {
    int raw = CalcExactRaw(n, misrate / 2);
    return checked(raw * 2);
  }

  internal int CalcApprox(int n, double misrate)
  {
    long raw = CalcApproxRaw(n, misrate / 2);
```

```
    long margin = raw * 2;
    if (margin > int.MaxValue)
      throw new OverflowException($"Signed-rank margin exceeds supported range for
n={n}");
    return (int)margin;
  }

  /// <summary>
  /// Compute one-sided margin using exact Wilcoxon signed-rank distribution.
  /// Uses dynamic programming to compute the CDF.
  /// </summary>
  private static int CalcExactRaw(int n, double p)
  {
    ulong total = 1UL << n;
    long maxW = (long)n * (n + 1) / 2;

    var count = new ulong[maxW + 1];
    count[0] = 1;

    for (int i = 1; i <= n; i++)
    {
      for (long w = Min(maxW, (long)i * (i + 1) / 2); w >= i; w--)
        count[w] += count[w - i];
    }

    ulong cumulative = 0;
    for (int w = 0; w <= maxW; w++)
    {
      cumulative += count[w];
      double cdf = (double)cumulative / total;
      if (cdf >= p)
        return w;
    }

    return (int)maxW;
  }

  /// <summary>
  /// Compute one-sided margin using Edgeworth approximation for large n.
  /// </summary>
  private static long CalcApproxRaw(int n, double misrate)
  {
    long maxW = (long)n * (n + 1) / 2;
    long a = 0;
    long b = maxW;

    while (a < b - 1)
    {
      long c = (a + b) / 2;
      double cdf = EdgeworthCdf(n, c);
      if (cdf < misrate)
        a = c;
      else
```

```
        b = c;
      }

      return EdgeworthCdf(n, b) < misrate ? b : a;
    }

    /// <summary>
    /// Edgeworth expansion for Wilcoxon signed-rank distribution CDF.
    /// </summary>
    private static double EdgeworthCdf(int n, long w)
    {
      double mu = (double)n * (n + 1) / 4.0;
      double sigma2 = n * (n + 1.0) * (2 * n + 1) / 24.0;
      double sigma = Sqrt(sigma2);

      // +0.5 continuity correction: computing P(W ≤ w) for a left-tail discrete CDF
      double z = (w - mu + 0.5) / sigma;
      double phi = Exp(-z * z / 2) / Sqrt(2 * PI);
      double Phi = AcmAlgorithm209.Gauss(z);

      double nd = n;
      double kappa4 = -nd * (nd + 1) * (2 * nd + 1) * (3 * nd * nd + 3 * nd - 1) / 240.0;

      double e3 = kappa4 / (24 * sigma2 * sigma2);

      double z2 = z * z;
      double z3 = z2 * z;
      double f3 = -phi * (z3 - 3 * z);

      double edgeworth = Phi + e3 * f3;
      return Min(Max(edgeworth, 0), 1);
    }

}
```

6.6.2. Tests

$$\mathrm{SignedRankMargin}(n, \mathrm{misrate})$$

The $\mathrm{SignedRankMargin}$ test suite contains 39 correctness test cases (4 demo + 6 boundary + 7 exact + 20 medium + 2 error).

Demo examples ($n = 30$) — from manual introduction:

`demo-1`: $n = 30$, $\mathrm{misrate} = 10^{-6}$, expected output: 46
`demo-2`: $n = 30$, $\mathrm{misrate} = 10^{-5}$, expected output: 74
`demo-3`: $n = 30$, $\mathrm{misrate} = 10^{-4}$, expected output: 112
`demo-4`: $n = 30$, $\mathrm{misrate} = 10^{-3}$, expected output: 158

These demo cases match the reference values used throughout the manual to illustrate $\mathrm{CenterBounds}$ construction.

Boundary cases — minimum achievable misrate validation:

`boundary-n2-min`: $n = 2$, $\mathrm{misrate} = 0.5$ (minimum misrate for $n = 2$, expected output: 0)
`boundary-n3-min`: $n = 3$, $\mathrm{misrate} = 0.25$ (minimum misrate for $n = 3$)
`boundary-n4-min`: $n = 4$, $\mathrm{misrate} = 0.125$ (minimum misrate for $n = 4$)
`boundary-loose`: $n = 5$, $\mathrm{misrate} = 0.5$ (permissive misrate)
`boundary-tight`: $n = 10$, $\mathrm{misrate} = 0.01$ (strict misrate)
`boundary-very-tight`: $n = 20$, $\mathrm{misrate} = 0.001$ (very strict misrate)

These boundary cases validate correct handling of minimum achievable misrate (formula: 2^{1-n}) and edge conditions.

Exact computation ($n \leq 10$) — validates dynamic programming path:

`exact-n5-mr1e1`: $n = 5$, $\mathrm{misrate} = 0.1$
`exact-n6-mr1e1`: $n = 6$, $\mathrm{misrate} = 0.1$
`exact-n6-mr5e2`: $n = 6$, $\mathrm{misrate} = 0.05$
`exact-n10-mr1e1`: $n = 10$, $\mathrm{misrate} = 0.1$, expected output: 22
`exact-n10-mr1e2`: $n = 10$, $\mathrm{misrate} = 0.01$
`exact-n10-mr5e2`: $n = 10$, $\mathrm{misrate} = 0.05$
`exact-n10-mr5e3`: $n = 10$, $\mathrm{misrate} = 0.005$

These cases exercise the exact Wilcoxon signed-rank CDF computation for small samples where dynamic programming is used.

Medium samples ($n \in \{15, 20, 30, 50, 100\} \times 4$ misrates) — 20 tests:

Misrate values: $\mathrm{misrate} \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$
Test naming: `medium-n{n}-mr{k}` where k encodes the misrate
Examples:
- `medium-n15-mr1e1`: $n = 15$, $\mathrm{misrate} = 0.1$
- `medium-n30-mr1e2`: $n = 30$, $\mathrm{misrate} = 0.01$, expected output: 220

`medium-n50-mr1e3`: $n = 50$, $\text{misrate} = 0.001$
`medium-n100-mr1e4`: $n = 100$, $\text{misrate} = 0.0001$

The medium sample tests validate the transition region between exact computation ($n \leq 63$) and approximate computation, ensuring consistent results across sample sizes and misrate values.

Error case — domain violation:

`error-n1`: $n = 1$, $\text{misrate} = 0.5$ (invalid: misrate below minimum achievable $2^{1-1} = 1.0$)
`error-n0`: $n = 0$, $\text{misrate} = 0.05$ (invalid: n must be positive)

This error case verifies that implementations correctly reject $n = 1$ with $\text{misrate} = 0.5$ as invalid input, since the minimum achievable misrate for $n = 1$ is $2^0 = 1.0$.

7. Appendix

7.1. Assumptions

This chapter defines the **domain assumptions** that govern Pragmastat functions. Unlike parametric assumptions (Normal, LogNormal, Pareto), domain assumptions describe which values are *meaningful inputs* in the first place.

Domain over parametric — assumptions define valid inputs, not distributional shape
Formal system — each function declares required assumptions; a function is applicable iff all hold
Structured errors — violations report assumption ID and subject, not ad-hoc strings

Implicit Validity Assumption

All functions implicitly require **valid samples**: non-empty, with finite defined real values (no NaN, +Inf, -Inf). This shared constraint has formal ID `validity` but is not listed per function.

Hard vs. Weak Assumptions

Hard assumptions (`validity`, `domain`, `positivity`, `sparity`) are enforced constraints — violating them makes the function inapplicable and triggers a structured error.

Weak assumptions (e.g., weak continuity, weak distribution) are performance expectations — estimators are designed to work well when they hold, but violations are never reported. Weak assumptions are assessed through drift tables and simulation studies, not input validation.

7.1.1. Weak Distribution Assumption

Definition

We assume that real data are *often close* to common generative families such as Additive, Multiplic, Power, and Uniform. This is not a requirement for validity. It is a pragmatic **performance expectation**: estimators should work well on these frequent real-world cases.

Why it matters

Pragmastat does not commit to a single "main" distribution. Instead, it requires robust behavior across the distributions practitioners actually see. This is consistent with procedure-first empiricism: we select estimators by properties and validate them across typical generative families.

Implication

Weak distribution assumptions are not enforced by validation and never produce errors. They are assessed through drift tables, tests, and simulation studies, not by checking inputs.

7.1.2. Positivity Assumption

Definition

All values must be strictly positive: $x_i > 0$ for every element.

Why positivity is fundamental

Most physical measurements are positive by construction: time, duration, length, mass, energy, concentration, price, latency. Even "zero" is rarely a physical reality; processes do not occur instantly. Zero and negative values are mathematical constructs used for convenience, not typical states of the physical world.

This makes positivity a **pragmatic** assumption: it reflects how real measurements are produced. When zero or negative values appear in a positive-only domain, they usually indicate measurement error, recording error, or preprocessing artifacts.

Asymmetry of extremes

Positivity implies a hard left boundary at zero, so extremes are typically right-tailed. This asymmetry is structural, not a violation. It motivates ratio-based and log-space estimators that respect multiplicative scales.

Workflow guidance

Strictly negative data — multiply by -1, then apply positivity-based tools.
Mixed signs — use sign-agnostic estimators such as Shift, Center, Spread.

Functions requiring positivity

`Ratio(x, y)` — both samples must be strictly positive.
`RatioBounds(x, y, misrate)` — both samples must be strictly positive.

7.1.3. Weak Continuity Assumption

Definition

The data-generating process is assumed to be continuous, meaning the probability of exact ties is zero in the underlying distribution.

In practice, ties occur because of measurement resolution and finite precision in computer arithmetic. Pragmastat treats ties as **artifacts of rounding**, not as meaningful properties of the data.

Design implication

Pragmastat does **not** introduce special tie-correction hacks. All estimators handle ties naturally without adjustment. This is pragmatic: device limitations belong in data cleaning, not in the estimator definition.

Why this matters for bounds

Functions like ShiftBounds, SpreadBounds, and PairwiseMargin compute distribution-free bounds based on the assumption that pairwise comparisons have no ties in expectation. When ties are present, bounds remain valid but may be slightly conservative. Weak continuity is a weak assumption and is never reported as a violation.

Ties are tolerated as artifacts, but **tie dominance is a hard boundary**. If ties dominate pairwise differences, the median pairwise distance collapses to zero, and spread-based estimators are no longer meaningful. This is enforced by the sparity assumption.

7.1.4. Weak Symmetry Assumption

Definition

The data-generating distribution is assumed to be approximately symmetric around its unknown center. This means deviations above and below the center are equally likely in magnitude.

Properties

Weak symmetry is a performance expectation, not an enforced constraint:

Approximate symmetry — The distribution need not be exactly symmetric; mild asymmetry produces mild coverage drift.
Ties tolerated — Symmetry refers to the underlying process, not observed ties.
Not validated — This is a modeling assumption, never checked computationally.
Violation behavior — Asymmetric distributions cause coverage to drift from nominal; bounds remain valid but may be wider or narrower than requested.

When symmetry is plausible

Measurement error around a true value (additive errors)
Physical quantities that can deviate equally in both directions
Log-transformed multiplicative data (often more symmetric than raw)

When symmetry is doubtful

Durations, latencies, response times (right-skewed)
Counts, concentrations, monetary values (often right-skewed)
Data with natural lower bounds but no upper bounds

Functions requiring weak symmetry

$\mathrm{CenterBounds}(\mathbf{x}, \mathrm{misrate})$ — requires weak symmetry for exact coverage.

Relationship to weak continuity

$\mathrm{CenterBounds}$ requires both weak continuity and weak symmetry for exact coverage.

7.1.5. Sparity Assumption

Definition

The sample must be **non tie-dominant**: the median of pairwise absolute differences must be strictly positive. Equivalently, `Spread(x) > 0`.

Tie-dominant samples are those where at least half of all pairwise differences are zero, which collapses the typical pairwise distance to zero. This can happen even when `min(x) < max(x)`, so a min/max check is not sufficient.

Why it matters

Spread is defined as the median of pairwise differences. If the median is zero, variability is not just small — it is not identifiable at the toolkit's scale. The sparity assumption captures this and prevents tie-dominant samples from entering spread-based estimators.

Implication

Sparity automatically implies the sample has at least two elements. For functions requiring sparity, this is the primary assumption to check. A sample with $n = 1$ fails sparity because `Spread(x) = 0`, not because of a separate size requirement.

Functions requiring sparity

`Spread(x)` — one-sample spread requires sparity.
`SpreadBounds(x, misrate)` — sample must have sparity.
`AvgSpreadBounds(x, y, misrate)` — both samples must have sparity.
`Disparity(x, y)` — both samples must have sparity.

Naming

Sparity is a Pragmastat term combining "spread" and "-ity" (the property suffix): the property of having positive spread. It can also be read as evoking "sparse" — pairwise differences are not dominated by zeros. This follows the toolkit's generative naming principle: the name encodes what the assumption checks (`Spread > 0`), not who defined it.

7.1.6. Assumption IDs and Violation Reporting

Assumption validation must be **structured** across all languages. Errors should report *which assumption failed*, not ad-hoc strings like "values must be positive".

Only one violation is reported per error. The violation is selected using a canonical priority order, and for two-sample functions the first failing sample (`x` before `y`) is reported.

Canonical priority order

1. `validity`
2. `domain`
3. `positivity`
4. `sparity`

Assumption ID registry

validity — non-empty input with finite defined real values.
domain — parameter value outside achievable range (e.g., misrate below minimum for given sample size).
positivity — values must be strictly positive.
sparity — requires `Spread(x) > 0`; sample must be non tie-dominant.

IDs are stable across languages and are the primary contract for error handling and localization. Weak assumptions (e.g., weak continuity) are documented in the chapter but are not part of this registry.

Violation schema (language-agnostic)

```
{
  "id": "positivity",
  "subject": "x"
}
```

```
{
  "id": "domain",
  "subject": "misrate"
}
```

Recommendations (generated from IDs)

validity — provide at least one finite real value; remove NaN/Inf before analysis.
domain — parameter value exceeds data resolution; for misrate violations, increase sample size or use a larger misrate value (see minimum achievable misrate tables).
positivity — if all values are strictly negative, multiply by -1; if mixed-sign, use sign-agnostic estimators or split by sign.
sparity — tie-dominant samples have no usable spread; increase measurement resolution, use a discrete/ordinal framework, or preprocess before applying spread-based tools.

Example minimal error string

```
positivity(x)
```

7.2. Foundations

This chapter introduces the key concepts that underpin the toolkit's estimator definitions and comparisons. Unlike the main estimator definitions, the foundations require knowledge of classic statistical methods. Well-known facts and commonly accepted notation are used without special introduction.

7.2.1. From Statistical Efficiency to Drift

Statistical efficiency measures estimator precision (Serfling (2009)). When multiple estimators target the same quantity, efficiency determines which provides more reliable results.

Efficiency measures how tightly estimates cluster around the true value across repeated samples. For an estimator T applied to samples from distribution X, absolute efficiency is defined relative to the optimal estimator T^*:

$$\text{Efficiency}(T, X) = \frac{\text{Var}[T^*(X_1, ..., X_n)]}{\text{Var}[T(X_1, ..., X_n)]}$$

Relative efficiency compares two estimators by taking the ratio of their variances:

$$\text{RelativeEfficiency}(T_1, T_2, X) = \frac{\text{Var}[T_2(X_1, ..., X_n)]}{\text{Var}[T_1(X_1, ..., X_n)]}$$

Under Additive (Normal) distributions, this approach works well. The sample mean achieves optimal efficiency, while the median operates at roughly 64% efficiency.

However, this variance-based definition creates four critical limitations:

Absolute efficiency requires knowing the optimal estimator, which is difficult to determine. For many distributions, deriving the minimum-variance unbiased estimator requires complex mathematical analysis. Without this reference point, absolute efficiency cannot be computed.

Relative efficiency only compares estimator pairs, preventing systematic evaluation. This limits understanding of how multiple estimators perform relative to each other. Practitioners cannot rank estimators comprehensively or evaluate individual performance in isolation.

The approach depends on variance calculations that break down when variance becomes infinite or when distributions have heavy tails. Many real-world distributions, such as those with power-law tails, exhibit infinite variance. When the variance is undefined, efficiency comparisons become impossible.

Variance is not robust to outliers, which can corrupt efficiency calculations. A single extreme observation can greatly inflate variance estimates. This sensitivity can make efficient estimators look inefficient and vice versa.

The Drift concept provides a robust alternative. Drift measures estimator precision using Spread instead of variance, providing reliable comparisons across a wide range of distributions.

For an average estimator T, random variable X, and sample size n:

$$\text{AvgDrift}(T, X, n) = \frac{\sqrt{n} \cdot \text{Spread}[T(X_1, ..., X_n)]}{\text{Spread}[X]}$$

This formula measures estimator variability compared to data variability. $\mathrm{Spread}[T(X_1, ..., X_n)]$ captures the median absolute difference between estimates across repeated samples. Multiplying by $\sqrt{n}$ removes sample size dependency, making drift values comparable across different sample sizes. Dividing by $\mathrm{Spread}[X]$ creates a scale-free measure that provides consistent drift values across different distribution parameters and measurement units.

Dispersion estimators use a parallel formulation:

$$\mathrm{DispDrift}(T, X, n) = \sqrt{n} \cdot \mathrm{RelSpread}[T(X_1, ..., X_n)]$$

Here $\mathrm{RelSpread}$ (where $\mathrm{RelSpread}[Y] = \frac{\mathrm{Spread}[Y]}{|\mathrm{Center}[Y]|}$) normalizes by the estimator's typical value for fair comparison.

Drift offers four key advantages:

For estimators with $\sqrt{n}$ convergence rates, drift remains finite and comparable across distributions; for heavier tails, drift may diverge, flagging estimator instability.
It provides absolute precision measures rather than only pairwise comparisons.
The robust Spread foundation resists outlier distortion that corrupts variance-based calculations.
The $\sqrt{n}$ normalization makes drift values comparable across different sample sizes, enabling direct comparison of estimator performance regardless of sample size.

Under Additive (Normal) conditions, drift matches traditional efficiency. The sample mean achieves drift near 1.0; the median achieves drift around 1.25. This consistency validates drift as a proper generalization of efficiency that extends to realistic data conditions where traditional efficiency fails.

When switching from one estimator to another while maintaining the same precision, the required sample size adjustment follows:

$$n_{\text{new}} = n_{\text{original}} \cdot \mathrm{Drift}^2 \, \frac{T_2, X}{\mathrm{Drift}^2}(T_1, X)$$

This applies when estimator T_1 has lower drift than T_2.

The ratio of squared drifts determines the data requirement change. If T_2 has drift 1.5 times higher than T_1, then T_2 requires $(1.5)^2 = 2.25$ times more data to match T_1's precision. Conversely, switching to a more precise estimator allows smaller sample sizes.

For asymptotic analysis, $\mathrm{Drift}(T, X)$ denotes the limiting value as $n \to \infty$. With a baseline estimator, rescaled drift values enable direct comparisons:

$$\mathrm{Drift}_{\text{baseline}(T,X)} = \frac{\mathrm{Drift}(T, X)}{\mathrm{Drift}(T_{\text{baseline}}, X)}$$

The standard drift definition assumes $\sqrt{n}$ convergence rates typical under Additive (Normal) conditions. For broader applicability, drift generalizes to:

$$\mathrm{AvgDrift}(T, X, n) = \frac{n^{\text{instability}} \cdot \mathrm{Spread}[T(X_1, ..., X_n)]}{\mathrm{Spread}[X]}$$

$$\text{DispDrift}(T, X, n) = n^{\text{instability}} \cdot \text{RelSpread}[T(X_1, ..., X_n)]$$

The instability parameter adapts to estimator convergence rates. The toolkit uses $\text{instability} = 1/2$ throughout because this choice provides natural intuition and mental representation for the Additive (Normal) distribution. Rather than introduce additional complexity through variable instability parameters, the fixed $\sqrt{n}$ scaling offers practical convenience while maintaining theoretical rigor for the distribution classes most common in applications.

7.2.2. From Confidence Level to Misrate

Traditional statistics expresses uncertainty through confidence levels: "95% confidence interval", "99% confidence", "99.9% confidence". This convention emerged from early statistical practice when tables printed confidence intervals for common levels like 90%, 95%, and 99%.

The confidence level approach creates practical problems:

Cognitive difficulty with high confidence. Distinguishing between 99.999% and 99.9999% confidence requires mental effort. The difference matters — one represents a 1-in-100,000 error rate, the other 1-in-1,000,000 — but the representation obscures this distinction.

Asymmetric scale. The confidence level scale compresses near 100%, where most practical values cluster. Moving from 90% to 95% represents a 2× change in error rate, while moving from 99% to 99.9% represents a 10× change, despite similar visual spacing.

Indirect interpretation. Practitioners care about error rates, not success rates. "What's the chance I'm wrong?" matters more than "What's the chance I'm right?" Confidence level forces mental subtraction to answer the natural question.

Unclear defaults. Traditional practice offers no clear default confidence level. Different fields use different conventions (95%, 99%, 99.9%), creating inconsistency and requiring arbitrary choices.

The misrate parameter provides a more natural representation. Misrate expresses the probability that computed bounds fail to contain the true value:

$$\text{misrate} = 1 - \text{confidence level}$$

This simple inversion provides several advantages:

Direct interpretation. $\text{misrate} = 0.01$ means "1% chance of error" or "wrong 1 time in 100". $\text{misrate} = 10^{-6}$ means "wrong 1 time in a million". No mental arithmetic required.

Linear scale for practical values. $\text{misrate} = 0.1$ (10%), $\text{misrate} = 0.01$ (1%), $\text{misrate} = 0.001$ (0.1%) form a natural sequence. Scientific notation handles extreme values cleanly: 10^{-3}, 10^{-6}, 10^{-9}.

Clear comparisons. 10^{-5} versus 10^{-6} immediately shows a 10× difference in error tolerance. 99.999% versus 99.9999% confidence obscures this same relationship.

Pragmatic default. The toolkit recommends $\text{misrate} = 10^{-3}$ (one-in-a-thousand error rate) as a reasonable default for everyday analysis. For critical decisions where errors are costly, use $\text{misrate} = 10^{-6}$ (one-in-a-million).

The terminology shift from "confidence level" to "misrate" parallels other clarifying renames in this toolkit. Just as Additive better describes the distribution's formation than 'Normal', and Center better

describes the estimator's purpose than 'Hodges-Lehmann', $\mathrm{misrate}$ better describes the quantity practitioners actually reason about: the probability of error.

Traditional confidence intervals become "bounds" in this framework, eliminating statistical jargon in favor of descriptive terminology. $\mathrm{ShiftBounds}(\mathbf{x}, \mathbf{y}, \mathrm{misrate})$ clearly indicates: it provides bounds on the shift, with a specified error rate. No background in classical statistics required to understand the concept.

7.2.3. Invariance

Invariance properties determine how estimators respond to data transformations. These properties are crucial for analysis design and interpretation:

Location-invariant estimators are invariant to additive shifts: $T(\mathbf{x} + k) = T(\mathbf{x})$
Scale-invariant estimators are invariant to positive rescaling: $T(k \cdot \mathbf{x}) = T(\mathbf{x})$ for $k > 0$
Equivariant estimators change predictably with transformations, maintaining relative relationships

Choosing estimators with appropriate invariance properties ensures that results remain meaningful across different measurement scales, units, and data transformations. For example, when comparing datasets collected with different instruments or protocols, location-invariant estimators eliminate the need for data centering, while scale-invariant estimators eliminate the need for normalization.

Location-invariance: An estimator T is location-invariant if adding a constant to the measurements leaves the result unchanged:

$$T(\mathbf{x} + k) = T(\mathbf{x})$$

$$T(\mathbf{x} + k, \mathbf{y} + k) = T(\mathbf{x}, \mathbf{y})$$

Location-equivariance: An estimator T is location-equivariant if it shifts with the data:

$$T(\mathbf{x} + k) = T(\mathbf{x}) + k$$

$$T(\mathbf{x} + k_1, \mathbf{y} + k_2) = T(\mathbf{x}, \mathbf{y}) + f(k_1, k_2)$$

Scale-invariance: An estimator T is scale-invariant if multiplying by a positive constant leaves the result unchanged:

$$T(k \cdot \mathbf{x}) = T(\mathbf{x}) \quad \text{for } k > 0$$

$$T(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = T(\mathbf{x}, \mathbf{y}) \quad \text{for } k > 0$$

Scale-equivariance: An estimator T is scale-equivariant if it scales proportionally with the data:

$$T(k \cdot \mathbf{x}) = k \cdot T(\mathbf{x}) \text{ or } |k| \cdot T(\mathbf{x}) \quad \text{for } k \neq 0$$

$$T(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = k \cdot T(\mathbf{x}, \mathbf{y}) \text{ or } |k| \cdot T(\mathbf{x}, \mathbf{y}) \quad \text{for } k \neq 0$$

	Location	Scale
Center	Equivariant	Equivariant
Spread	Invariant	Equivariant
Shift	Invariant	Equivariant
Ratio	–	Invariant
Disparity	Invariant	Invariant

7.3. Methodology

This chapter examines the methodological principles that guide Pragmastat's design and application.

7.3.1. Pragmatic Philosophy

The toolkit's foundations rest on pragmatist epistemology: truth is determined by practical consequences, not abstract correspondence with reality.

Truth is what works — An estimator is "correct" if it produces useful results across realistic conditions

Meaning from consequences — The value of a statistical method lies in what it enables, not its theoretical elegance

Theory serves practice — Mathematical analysis provides insight, but empirical validation determines adoption

Utility as criterion — When methods conflict, prefer the one that solves more real problems

This stance inverts the traditional relationship between theory and practice. Rather than deriving methods from first principles and hoping they apply, we evaluate methods by their performance and seek theoretical understanding afterward.

7.3.2. Procedure-First Empiricism

Traditional statistical practice follows an assumptions-first methodology:

1. Assume a data-generating model (e.g., "observations are normally distributed")
2. Derive the optimal procedure under those assumptions
3. Apply the procedure to data, hoping assumptions approximately hold

This toolkit inverts the process:

1. Select procedures based on desired properties (robustness, equivariance, interpretability)
2. Empirically measure performance across a wide range of conditions
3. Use theory to explain and predict observed behavior

Monte Carlo simulation serves as the primary instrument of knowledge. Rather than deriving asymptotic formulas for estimator variance, we measure actual variance across thousands of simulated samples. Drift tables in this manual are empirically measured, not analytically derived.

This approach has practical advantages: simulations can explore conditions that resist closed-form analysis, and empirical results are self-validating — they show what actually happens, not what theory predicts should happen.

For the formal treatment of domain assumptions that govern valid inputs, see the Assumptions chapter.

7.3.3. Epistemic Humility

No perfectly Gaussian, log-normal, or Pareto distributions exist in real data. Every distribution we name is a useful fiction — a model we employ because it approximates reality well enough for our purposes, while knowing it cannot be exactly correct.

Models are approximations — They capture essential structure while ignoring irrelevant details

Approximations fail at boundaries — Edge cases, extreme values, and distribution tails often violate assumptions

Graceful degradation — Methods should produce sensible (if less precise) results when assumptions weaken

The toolkit embodies this humility by choosing estimators that remain interpretable and bounded even when distributional assumptions break down. A robust estimator may sacrifice some efficiency under ideal conditions in exchange for reliable behavior when conditions degrade.

7.3.4. The Pairwise Principle

A structural insight unifies all primary robust estimators in this toolkit: they are medians of pairwise operations.

Estimator	Pairwise Operation	Result
Center	$(x_i + x_j)/2$	Median of pairwise averages
Spread	$\lvert x_i - x_j \rvert$	Median of pairwise differences
Shift	$x_i - y_j$	Median of cross-sample differences
Ratio	$\log(x_i) - \log(y_j)$	exp(median of log-differences)
Dominance	$\mathbf{1}(x_i > y_j)$	Proportion of pairwise comparisons

For multiplicative quantities like Ratio, the pairwise operation is defined in log-space, aggregated with the median, then mapped back with exp. This canonical-scale approach preserves the "median of pairwise operations" principle while ensuring exact multiplicative antisymmetry: $\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) \times \mathrm{Ratio}(\mathbf{y}, \mathbf{x}) = 1$.

This pairwise structure provides three benefits:

Natural robustness — Comparing measurements to each other, not to external references, limits outlier influence
Self-calibration — The sample serves as its own reference distribution, requiring no external assumptions
Algebraic closure — Pairwise operations preserve symmetry and equivariance properties

The pairwise principle also enables efficient computation. Matrices of pairwise operations have structural properties (sorted rows and columns) that fast algorithms exploit to achieve $O(n \log n)$ complexity.

7.3.5. Median as Universal Aggregator

The median is the final step in each pairwise estimator. Why median specifically?

The median achieves the maximum possible breakdown point (50%) among all translation-equivariant location estimators. Up to half the data can be arbitrarily corrupted before the median becomes unbounded.

However, Center and Spread achieve only 29% breakdown — not 50%. This is deliberate: a tradeoff between robustness and precision.

Breakdown	Robustness	Precision	Estimators
0%	None	Optimal under assumptions	Mean, StdDev
29%	Substantial	Near-optimal	Center, Spread
50%	Maximum	Reduced	Median, MAD

The 29% breakdown point survives approximately one corrupted measurement in four while maintaining roughly 95% asymptotic efficiency under ideal Gaussian conditions. This represents the practical optimum: enough robustness for realistic contamination levels, enough efficiency to compete with traditional methods when data is clean.

7.3.6. Convergence Conventions

Drift normalizes estimator variability by $\sqrt{n}$, making precision comparable across sample sizes:

$$\text{Drift} = \text{Spread}(\text{estimates}) \times \sqrt{n}$$

This normalization embeds a deliberate assumption: most useful estimators converge at the $\sqrt{n}$ rate. The Central Limit Theorem guarantees this rate for means under mild conditions, and median-based estimators inherit similar convergence behavior.

> **Common case default** — $\sqrt{n}$ convergence covers the vast majority of practical estimators
> **Intuitive interpretation** — Drift represents "effective standard deviation at $n = 1$"
> **Mental calculation** — Expected precision at any n is simply $\text{Drift} / \sqrt{n}$

For estimators with non-standard convergence (e.g., extreme value statistics), drift generalizes to $n^{\text{instability}}$ where instability differs from 0.5. But the toolkit deliberately uses $\sqrt{n}$ throughout because it matches the common case and provides intuitive interpretation without complicating the universal mechanism.

This is pragmatic universalism: adopt the common case as default, acknowledge exceptions exist, and handle them explicitly rather than burdening the common case with unnecessary generality.

7.3.7. Structural Unity

All robust estimators in this toolkit share a common mathematical structure:

$$\text{Estimator} = \text{InvTransform}(\text{Median}(\text{PairwiseOperation}(\text{Transform}(x), \text{Transform}(y))))$$

For additive estimators (Center, Spread, Shift), Transform is identity. For multiplicative estimators (Ratio), $\text{Transform} = \log$ and $\text{InvTransform} = \exp$.

This structural unity is not merely aesthetic — it enables unified algorithmic optimization.

Sorted structure — Matrices of pairwise operations have sorted rows and columns
Monahan's algorithm — Exploits sorted structure for $O(n \log n)$ Center/Spread
Fast shift — Exploits cross-sample matrix structure for efficient two-sample comparison

Because all estimators share the same "median of pairwise" form, insights that accelerate one can often be adapted to accelerate others. A single theoretical framework covers all primary estimators.

7.3.8. Generative Naming

Names in this toolkit encode operational knowledge rather than historical provenance.

Traditional	Pragmastat	What's Encoded
Gaussian / Normal	Additive	Formation: sum of independent factors (CLT)
Log-normal / Galton	Multiplic	Formation: product of independent factors
Pareto	Power	Behavior: power-law relationship
Hodges-Lehmann	Center	Function: measures central tendency
Shamos	Spread	Function: measures variability
(none)	sparity	Assumption: property of having positive spread

Reading "Additive" activates a generative model: this distribution arises when many independent factors add together. Reading "Gaussian" requires recalling an association with Carl Friedrich Gauss, then remembering what properties that name implies.

Generative names create immediate intuition about when a model applies. Additive distributions arise from additive processes. Multiplic distributions arise from multiplicative processes. The name itself encodes the formation mechanism.

7.3.9. The Inversion Principle

Traditional statistical outputs often require mental transformation before use. This toolkit inverts such framings to present information in directly actionable form, following principles of user-centered design (Norman (2013)).

Traditional	Pragmastat	Reason for Inversion
Confidence level (95%)	misrate (0.05)	Direct error interpretation
Confidence interval	Bounds	Plain language, no jargon
Hypothesis test (p-value)	Bounds estimation	"What's plausible?" not "Is zero plausible?"
Efficiency (variance ratio)	Drift (spread-based)	Works with heavy tails

Consider the confidence level vs. misrate inversion. A "95% confidence interval" requires understanding: "If I repeated this procedure infinitely, 95% of intervals would contain the true value." A "5% misrate" states directly: "This procedure errs about 5% of the time."

The shift from confidence intervals to bounds, and from hypothesis testing to interval estimation, moves from frequentist theology toward decision-relevant inference. The practitioner asks "What values are plausible for this parameter?" rather than "Can I reject the hypothesis that this parameter equals zero?"

7.3.10. Multi-Audience Design

This manual serves readers with diverse backgrounds and conflicting preferences:

Audience	Priorities	Challenges
Experienced academics	Rigor, derivation, formalism, citations	May find practical focus too shallow
Professional developers	Examples, APIs, searchability, minimalism	May find theory intimidating
Students and beginners	Clarity, intuition, progressive disclosure	Need both theory and practice
Large language models	Structure, consistency, unambiguous definitions	Need form-independent content

These audiences have conflicting needs. Academics want complete derivations; developers want quick answers. Beginners need gentle introductions; experts need dense references. LLMs need predictable structure; humans appreciate variety.

The manual targets a "neutral zone" where all audiences find acceptable content:

Signature first — Mathematical definition immediately visible
Example second — Concrete computation before abstract explanation
Detail optional — Properties, corner cases, and theory follow for those who need them
Every sentence earns its place — No filler prose, no redundant explanation

Structural Principles

Concrete over abstract — Numbers and examples before symbols and theory
Precision without verbosity — Mathematical rigor in minimal words
Consistent layout — Same structure across all toolkit items enables scanning
Self-contained sections — Each section readable independently

LLM-Friendliness

The manual's structure also serves machine readers:

Predictable patterns — Consistent section ordering aids extraction
Explicit definitions — No implicit knowledge assumed
Tabular data — Structured information in tables, not prose
Short paragraphs — Content chunks cleanly for context windows

This multi-audience optimization forces elimination of audience-specific conventions, revealing form-independent essential content that serves everyone adequately rather than serving one group perfectly and others poorly.

7.3.11. Reference Tests as Specification

The toolkit maintains seven implementations across different programming languages: Python, TypeScript, R, C#, Kotlin, Rust, and Go. Each implementation must produce identical numerical results for all estimators.

This cross-language consistency is achieved through executable specifications:

```
Manual (definitions) ↔ C# (reference) → JSON (tests) → All languages (validation)
```

The specification IS the test suite. Reference tests serve three critical purposes:

Cross-language validation — All implementations pass identical test cases
Regression prevention — Changes validated against known outputs
Implementation guidance — Concrete examples for porting to new languages

Test Design Principles

Minimal sufficiency — Smallest test set providing high confidence in correctness
Comprehensive coverage — Both typical cases and edge cases that expose errors
Deterministic reproducibility — Fixed seeds for all random tests

Test Categories

Canonical cases — Deterministic inputs like natural number sequences where outputs are easily verified
Edge cases — Boundary conditions: single element, zeros, minimum viable sample sizes
Fuzzy tests — Controlled random exploration beyond hand-crafted examples

The C# implementation serves as the reference generator. All test cases are defined programmatically, executed to produce expected outputs, and serialized to JSON. Other implementations load these JSON files and verify their outputs match within numerical tolerance.

7.3.12. Cross-Language Determinism

Reproducibility requires determinism at every layer. When a simulation in Python produces a result, the same simulation in Rust, Go, or any other supported language must produce the identical result.

Portable RNG — $\mathrm{Rng}(\text{experiment-}1)$ produces identical sequences in all languages
Specified algorithms — xoshiro256++ for generation, SplitMix64 for seeding, FNV-1a for string hashing
No implementation-dependent behavior — Floating-point operations follow IEEE 754

Unified API

Beyond numerical determinism, the toolkit maintains a consistent API across all implementations. Function names, parameter orders, and return types follow the same conventions in every language.

Same vocabulary — Center, Spread, Shift mean the same thing everywhere
Same signatures — `Center(x)` in Python, `Center(x)` in Rust, `Center(x)` in Go
Same behavior — Edge cases, error conditions, and defaults are identical

This unified API enables frictionless language switching. A practitioner prototyping in Python can port to Rust for production without learning new abstractions or revalidating statistical assumptions. The mental model transfers directly; only syntax changes.

Benefits of Unification

Debugging across languages — A failing test in TypeScript can be debugged in C#
Verified ports — New implementations can be validated against existing ones
Reproducible research — Results can be reproduced in any supported language
Team flexibility — Different team members can use preferred languages on the same analysis
Migration paths — Move from prototype to production without statistical revalidation

7.3.13. Summary Principles

The methodology of this toolkit can be distilled into twelve guiding principles:

1. **Name things by what they do, not who discovered them** — Generative names encode operational knowledge
2. **All models are wrong; design for graceful degradation** — Robust methods fail gently
3. **Evaluate empirically, organize theoretically** — Simulation before derivation
4. **Self-reference provides robustness** — Pairwise operations compare data to itself
5. **29% breakdown is the practical optimum** — Balance robustness and precision
6. **Invert framings that require mental transformation** — Present directly actionable information
7. **Default to the common case** — Use $\sqrt{n}$ convergence; handle exceptions explicitly
8. **Multi-audience optimization reveals essential content** — Serve everyone adequately, not one group perfectly
9. **Executable specifications are reliable specifications** — Tests define correctness
10. **Reproducibility requires portable determinism** — Same seeds, same results, any language
11. **Structural unity enables unified optimization** — "Median of pairwise" admits fast algorithms
12. **Utility is the ultimate criterion** — Methods that solve real problems are correct methods

7.3.14. Strict Domains Principle

For each function parameter, Pragmastat enforces the strictest domain that:

1. Supports virtually all legitimate real-world use cases
2. Rejects pathological cases that would produce misleading results
3. Fails immediately with actionable guidance rather than silently degrading

Rationale: Learning from NHST Problems

Traditional tools accept arbitrary confidence levels without warning when the requested precision exceeds data resolution. This leads to misleading results: a practitioner requests 99.99% confidence with $n = 5$ and receives bounds that look like valid statistical inference but actually have much lower coverage.

Strict validation approach

Making impossible requests impossible — If $n = 5$ cannot achieve 99% confidence, the function rejects $\text{misrate} = 0.01$ rather than returning meaningless bounds.
Actionable errors — Messages explain WHY the request failed and HOW to fix it.
Explicit tradeoffs — Practitioners learn their data's actual resolution limits.

Minimum achievable misrate

For one-sample bounds, minimum achievable misrate $= 2^{1-n}$:

n	$\text{misrate}_{\text{min}}$	max confidence	notes
2	0.5	50%	only trivial bounds possible
5	0.0625	93.75%	cannot achieve 95%
7	0.0156	98.4%	cannot achieve 99%
10	0.00195	99.8%	most practical misrates achievable
20	1.9×10^{-6}	99.9998%	$\text{misrate} = 10^{-6}$ is achievable

Practical implications

$n < 5$: Cannot achieve 95% confidence ($\text{misrate} = 0.05$)
$n < 7$: Cannot achieve 99% confidence ($\text{misrate} = 0.01$)
$n \geq 20$: $\text{misrate} = 10^{-6}$ is achievable

This principle ensures that Pragmastat functions never silently produce misleading results when the requested precision exceeds what the data can support.

7.3.15. Test Framework

The reference test framework consists of three components:

Test generation — The C# implementation defines test inputs programmatically using builder patterns. For deterministic cases, inputs are explicitly specified. For random cases, the framework uses controlled seeds with `System.Random` to ensure reproducibility across all platforms.

The random generation mechanism works as follows:

> Each test suite builder maintains a seed counter initialized to zero.
> For one-sample estimators, each distribution type receives the next available seed. The same random generator produces all samples for all sizes within that distribution.
> For two-sample estimators, each pair of distributions receives two consecutive seeds: one for the x sample generator and one for the y sample generator.
> The seed counter increments with each random generator creation, ensuring deterministic test data generation.

For Additive distributions, random values are generated using the Box-Müller transform, which converts pairs of uniform random values into normally distributed values. The transform applies the formula:

$$X = \mu + \sigma\sqrt{-2\ln(U_1)}\sin(2\pi U_2)$$

where U_1, U_2 are uniform random values from $\text{Uniform}(0, 1)$, μ is the mean, and σ is the standard deviation.

For Uniform distributions, random values are generated directly using the quantile function:

$$X = \min + U \cdot (\max - \min)$$

where U is a uniform random value from $\text{Uniform}(0, 1)$.

The framework executes the reference implementation on all generated inputs and serializes input-output pairs to JSON format.

Test validation — Each language implementation loads the JSON test cases and executes them against its local estimator implementation. Assertions verify that outputs match expected values within a given numerical tolerance (typically 10^{-10} for relative error).

Test data format — Each test case is a JSON file containing `input` and `output` fields. For one-sample estimators, the input contains array `x` and optional `parameters`. For two-sample estimators, input contains arrays `x` and `y`. For bounds estimators (ShiftBounds, RatioBounds), input additionally contains `misrate`. Output is a single numeric value for point estimators, or an object with `lower` and `upper` fields for bounds estimators.

Performance testing — The toolkit provides $O(n \log n)$ fast algorithms for Center, Spread, and Shift estimators, dramatically more efficient than naive implementations that materialize all pairwise combinations. Performance tests use sample size $n = 100,000$ (for one-sample) or $n = m = 100,000$

(for two-sample). This specific size creates a clear performance distinction: fast implementations ($O(n \log n)$ or $O((m+n) \log L)$) complete in under 5 seconds on modern hardware across all supported languages, while naive implementations ($O(n^2 \log n)$ or $O(mn \log(mn))$) would be prohibitively slow (taking hours or failing due to memory exhaustion). With $n = 100,000$, naive approaches would need to materialize approximately 5 billion pairwise values for Center/Spread or 10 billion for Shift, whereas fast algorithms require only $O(n)$ additional memory. Performance tests serve dual purposes: correctness validation at scale and performance regression detection, ensuring implementations use the efficient algorithms and remain practical for real-world datasets with hundreds of thousands of observations. Performance test specifications are provided in the respective estimator sections above.

This framework ensures that all seven language implementations maintain strict numerical agreement across the full test suite.

Bibliography

Blackman, D., & Vigna, S. (2021). Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software*, *47*(4), 1–32. https://dl.acm.org/doi/10.1145/3460772

Box, G. E. P., & Muller, M. E. (1958). A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, *29*(2), 610–611. https://projecteuclid.org/euclid.aoms/1177706645

Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Lawrence Erlbaum Associates.

Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, *7*(1), 1–26. https://projecteuclid.org/euclid.aos/1176344552

Fan, C. T., Muller, M. E., & Rezucha, I. (1962). Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers. *Journal of the American Statistical Association*, *57*(298), 387–402. https://www.tandfonline.com/doi/abs/10.1080/01621459.1962.10480667

Fisher, R. A., & Yates, F. (1938). *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd.

Fix, E., & Hodges, J. L. (1955). Significance probabilities of the Wilcoxon test. *The Annals of Mathematical Statistics*, *26*(2), 301–312. https://projecteuclid.org/euclid.aoms/1177728547

Fowler, G., Noll, L. C., & Vo, K.-P. (1991,). *FNV Hash*. http://www.isthe.com/chongo/tech/comp/fnv/

Hodges, J. L., & Lehmann, E. L. (1963). Estimates of Location Based on Rank Tests. *The Annals of Mathematical Statistics*, *34*(2), 598–611. http://projecteuclid.org/euclid.aoms/1177704172

Huber, P. J. (2009). Robust statistics. In *International encyclopedia of statistical science* (pp. 1248–1251). Springer.

Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. *The American Statistician*, *50*(4), 361. https://www.jstor.org/stable/2684934

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (3rd ed.). Addison-Wesley.

Löffler, A. (1982). *On the calculation of the exact null-distribution of the Wilcoxon-Mann-Whitney U-statistic*.

Monahan, J. F. (1984). Algorithm 616: fast computation of the Hodges-Lehmann location estimator. *ACM Transactions on Mathematical Software*, *10*(3), 265–270. https://dl.acm.org/doi/10.1145/1271.319414

Norman, D. A. (2013). *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books.

Sen, P. K. (1963). On the Estimation of Relative Potency in Dilution (-Direct) Assays by Distribution-Free Methods. *Biometrics*, *19*(4), 532. https://www.jstor.org/stable/2527532

Serfling, R. J. (2009). *Approximation theorems of mathematical statistics*. John Wiley & Sons.

Shamos, M. I. (1976). *Geometry and Statistics: Problems at the Interface*.

Sidak, Z., Sen, P. K., & Hajek, J. (1999). *Theory of rank tests*. Elsevier.

Steele, G. L., Lea, D., & Flood, C. H. (2014). *Fast Splittable Pseudorandom Number Generators* (pp. 453–472). ACM. https://dl.acm.org/doi/10.1145/2660193.2660195