

Technical Manual

Pragmastat: Pragmatic Statistical Toolkit

11.02.2026

Andrey Akinshin · andrey.akinshin@gmail.com

Version 7.0.1 · DOI: [10.5281/zenodo.17236778](https://doi.org/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-v7.0.1.pdf web-v7.0.1.zip
Implementations	py-v7.0.1.zip ts-v7.0.1.zip r-v7.0.1.zip cs-v7.0.1.zip kt-v7.0.1.zip rs-v7.0.1.zip go-v7.0.1.zip
Reference data	tests-v7.0.1.zip sim-v7.0.1.zip
Source code	pragmastat-7.0.1.zip

Andrey Akinshin
andreyakinshin@gmail.com
 DOI: [10.5281/zenodo.17236778](https://doi.org/10.5281/zenodo.17236778)

Copyright © 2025–2026 Andrey Akinshin

This manual is licensed under the **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License** (CC BY-NC-SA 4.0). Sharing and adapting this material for non-commercial purposes is permitted, provided appropriate credit is given, changes are indicated, and contributions are distributed under the same license.

The accompanying source code and software implementations are licensed under the **MIT License**. Use, copying, modification, merging, publishing, distribution, sublicensing, and/or selling copies of the software is permitted, subject to the conditions stated in the license. For complete license terms, see the LICENSE file in the source repository.

While the information in this manual is believed to be accurate at the date of publication, the author makes no warranty, express or implied, with respect to the material contained herein. The author shall not be liable for any errors, omissions, or damages arising from the use of this information.

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

Typeset with Typst. Text refined with LLM assistance.

Contents

1. Toolkit	5
1.1. Rng	5
1.2. Sample	6
1.3. Shuffle	7
1.4. Resample	8
1.5. Median	9
1.6. Center	10
1.7. Spread	11
1.8. RelSpread	12
1.9. Shift	13
1.10. Ratio	14
1.11. AvgSpread	15
1.12. Disparity	16
1.13. PairwiseMargin	17
1.14. SignedRankMargin	18
1.15. ShiftBounds	19
1.16. RatioBounds	20
1.17. CenterBounds	21
2. Distributions	22
2.1. Additive ('Normal')	22
2.2. Multiplic ('LogNormal')	23
2.3. Exponential	24
2.4. Power ('Pareto')	24
2.5. Uniform	25
3. Algorithms	27
3.1. Pseudorandom Number Generation	27
3.2. Shuffle Algorithm	28
3.3. Selection Sampling	28
3.4. Distribution Sampling	28
3.5. Fast Center	29
3.6. Fast Spread	35
3.7. Fast Shift	41
3.8. Fast Ratio	45
3.9. Fast PairwiseMargin	47
3.10. Fast SignedRankMargin	55
3.11. Fast CenterQuantiles	60
4. Assumptions	65
4.1. Weak Distribution Assumption	66
4.2. Positivity Assumption	67

4.3. Weak Continuity Assumption	68
4.4. Weak Symmetry Assumption	69
4.5. Sparsity Assumption	70
4.6. Assumption IDs and Violation Reporting	71
5. Studies	72
5.1. Summary Estimator Properties	72
5.2. Breakdown	73
5.3. Drift	73
5.4. Invariance	80
5.5. From Statistical Efficiency to Drift	81
5.6. From Confidence Level to Misrate	83
5.7. From Mann-Whitney U-test to Pairwise Margin	84
5.8. On Bootstrap for Center Bounds	87
5.9. On Misrate Efficiency of MedianBounds	88
5.10. Additive ('Normal') Distribution	91
6. Reference Implementations	96
6.1. Python	96
6.2. TypeScript	98
6.3. R	100
6.4. C#	102
6.5. Kotlin	104
6.6. Rust	106
6.7. Go	109
7. Reference Tests	112
7.1. Center Tests	112
7.2. Spread Tests	113
7.3. RelSpread Tests	115
7.4. Shift Tests	116
7.5. Ratio Tests	118
7.6. AvgSpread Tests	119
7.7. Disparity Tests	120
7.8. PairwiseMargin Tests	122
7.9. SignedRankMargin Tests	123
7.10. ShiftBounds Tests	124
7.11. RatioBounds Tests	127
7.12. CenterBounds Tests	130
7.13. Test Framework	132
8. Methodology	134
8.1. Pragmatic Philosophy	134
8.2. Procedure-First Empiricism	135
8.3. Epistemic Humility	136

8.4. The Pairwise Principle	137
8.5. Median as Universal Aggregator	138
8.6. Convergence Conventions	139
8.7. Structural Unity	140
8.8. Generative Naming	141
8.9. The Inversion Principle	142
8.10. Multi-Audience Design	143
8.11. Reference Tests as Specification	144
8.12. Cross-Language Determinism	145
8.13. Summary Principles	146
8.14. Strict Domains Principle	147
Bibliography	148

1. Toolkit

This chapter provides formal definitions and properties of each toolkit function.

1.1. Rng

$\text{Rng}(s)$

Deterministic pseudorandom number generator from seed s .

Algorithm — xoshiro256++ with SplitMix64 seeding (see Pseudorandom Number Generation)

Seed types — integer seed or string seed (hashed via FNV-1a)

Determinism — identical sequences across all supported languages

Period — $2^{256} - 1$

Notation

X, Y random variables (generators of real measurements)

Properties

Reproducibility $\text{Rng}(s)$ produces identical sequence for same s

Independence different seeds produce uncorrelated sequences

Example

`Rng("demo-uniform")` — string seed for reproducible demos

`Rng("experiment-1")` — string seed for named experiments

Rng provides reproducible random numbers. The same seed produces exactly the same sequence of values, identical across Python, TypeScript, R, C#, Kotlin, Rust, and Go. Passing a descriptive string like "experiment-1" makes code self-documenting. Each draw advances the generator's internal state, so independent random streams require separate generators with different seeds.

1.2. Sample

$r.\text{Sample}(\mathbf{x}, k)$

Select k elements from sample \mathbf{x} without replacement using generator r .

Algorithm — selection sampling (Fan, Muller, Rezucha 1962), see Pseudorandom Number Generation

Complexity — $O(n)$ time, single pass

Output — preserves original order of selected elements

Domain — $1 \leq k \leq n$

Notation

$\mathbf{x} = (x_1, \dots, x_n), \mathbf{y} = (y_1, \dots, y_m)$ samples ($n, m \geq 1$)
 x_i, y_j 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

`Sample([1, 2, 3, 4, 5], 3, Rng("demo-sample"))` — select 3 elements
`Sample(x, n, r) = x` — selecting all elements returns original order

Sample picks a random subset of data without replacement. Common uses include bootstrap resampling, 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.

1.3. Shuffle

$r.\text{Shuffle}(x)$

Uniformly random permutation of sample x using generator r .

Algorithm — Fisher-Yates (Knuth shuffle), see Pseudorandom Number Generation

Complexity — $O(n)$ time, $O(1)$ additional space

Output — new array (does not modify input)

Properties

Uniformity each of $n!$ permutations has equal probability

Determinism same generator state produces same permutation

Example

`Shuffle([1, 2, 3, 4, 5], Rng("demo-shuffle"))` — shuffled copy

`Shuffle(x, r)` preserves multiset (same elements, different order)

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.

1.4. Resample

$r.$ Resample(\mathbf{x}, k)

Select k elements from sample \mathbf{x} with replacement using generator r .

Algorithm — uniform sampling with replacement

Complexity — $O(k)$ time

Output — new array with k elements (may contain duplicates)

Domain — $k \geq 0$, sample size $n \geq 1$

Notation

$\mathbf{x} = (x_1, \dots, x_n)$ 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

`Resample([1, 2, 3, 4, 5], 3, Rng("demo-resample"))` — select 3 with replacement

`Resample(x, n, r)` — bootstrap sample of same size as original

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"))`.

1.5. 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 — $O(n)$ with selection, $O(n \log n)$ with sorting

Domain — any real numbers

Unit — same as measurements

Notation

$x_{(1)}, \dots, x_{(n)}$ 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

$\text{Median}([1, 2, 3, 4, 5]) = 3$

$\text{Median}([1, 2, 3, 4]) = 2.5$

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.

1.6. Center

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

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 — $O(n^2 \log n)$ naive, $O(n \log n)$ fast (see Fast Center)

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

$\text{Center}([0, 2, 4, 6, 8]) = 4$

$\text{Center}(\mathbf{x} + 10) = 14$ $\text{Center}(3\mathbf{x}) = 12$

References

Hodges & Lehmann (1963)

Sen (1963)

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.

1.7. Spread

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

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 — $O(n^2 \log n)$ naive, $O(n \log n)$ fast (see Fast Spread)

Domain — any real numbers

Assumptions — sparsity(\mathbf{x})

Unit — same as measurements

Properties

Shift invariance $\text{Spread}(\mathbf{x} + k) = \text{Spread}(\mathbf{x})$

Scale equivariance $\text{Spread}(k \cdot \mathbf{x}) = |k| \cdot \text{Spread}(\mathbf{x})$

Non-negativity $\text{Spread}(\mathbf{x}) \geq 0$

Example

$$\text{Spread}([0, 2, 4, 6, 8]) = 4$$

$$\text{Spread}(\mathbf{x} + 10) = 4 \quad \text{Spread}(2\mathbf{x}) = 8$$

References

Shamos (1976)

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.

1.8. RelSpread

$$\text{RelSpread}(\mathbf{x}) = \frac{\text{Spread}(\mathbf{x})}{|\text{Center}(\mathbf{x})|}$$

Relative dispersion normalized by location.

Also known as — robust coefficient of variation

Domain — $\text{Center}(\mathbf{x}) \neq 0$

Assumptions — positivity(\mathbf{x})

Unit — dimensionless

Properties

Scale invariance $\text{RelSpread}(k \cdot \mathbf{x}) = \text{RelSpread}(\mathbf{x})$

Non-negativity $\text{RelSpread}(\mathbf{x}) \geq 0$

Example

$\text{RelSpread}([1, 3, 5, 7, 9]) = 0.8$

$\text{RelSpread}(5\mathbf{x}) = 0.8$

RelSpread compares how “noisy” different datasets are, even if they have completely different scales or units. A dataset centered around 100 with spread of 10 has the same relative variability as one centered around 1000 with spread of 100. Both show 10% relative variation, and RelSpread captures exactly this. This makes it useful for comparing measurement quality across different experiments, instruments, or physical quantities where absolute numbers are not directly comparable.

1.9. Shift

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

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 — $O(mn \log(mn))$ naive, $O((m + n) \log L)$ fast (see Fast Shift)

Domain — any real numbers

Unit — same as measurements

Properties

Self-difference $\text{Shift}(\mathbf{x}, \mathbf{x}) = 0$

Shift equivariance $\text{Shift}(\mathbf{x} + k_x, \mathbf{y} + k_y) = \text{Shift}(\mathbf{x}, \mathbf{y}) + k_x - k_y$

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

Antisymmetry $\text{Shift}(\mathbf{x}, \mathbf{y}) = -\text{Shift}(\mathbf{y}, \mathbf{x})$

Example

$\text{Shift}([0, 2, 4, 6, 8], [10, 12, 14, 16, 18]) = -10$

$\text{Shift}(\mathbf{y}, \mathbf{x}) = -\text{Shift}(\mathbf{x}, \mathbf{y})$

References

Hodges & Lehmann (1963)

Sidak et al. (1999)

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.

1.10. Ratio

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

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(\mathbf{x}), positivity(\mathbf{y})

Unit — dimensionless

Complexity — naive $O(mn \log(mn))$, fast $O((m + n) \log L)$ via FastRatio

Properties

Self-ratio $\text{Ratio}(\mathbf{x}, \mathbf{x}) = 1$

Scale equivariance $\text{Ratio}(k_x \cdot \mathbf{x}, k_y \cdot \mathbf{y}) = \left(\frac{k_x}{k_y}\right) \cdot \text{Ratio}(\mathbf{x}, \mathbf{y})$

Multiplicative antisymmetry $\text{Ratio}(\mathbf{x}, \mathbf{y}) = \frac{1}{\text{Ratio}(\mathbf{y}, \mathbf{x})}$

Example

$$\text{Ratio}([1, 2, 4, 8, 16], [2, 4, 8, 16, 32]) = 0.5$$

$$\text{Ratio}(\mathbf{x}, \mathbf{x}) = 1 \quad \text{Ratio}(2\mathbf{x}, 5\mathbf{y}) = 0.4 \cdot \text{Ratio}(\mathbf{x}, \mathbf{y})$$

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:

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

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.

1.11. AvgSpread

$$\text{AvgSpread}(x, y) = \frac{n \cdot \text{Spread}(x) + m \cdot \text{Spread}(y)}{n + m}$$

Weighted average of dispersions (pooled scale).

Also known as — robust pooled standard deviation

Domain — any real numbers

Assumptions — sparsity(x), sparsity(y)

Unit — same as measurements

Caveat — $\text{AvgSpread}(x, y) \neq \text{Spread}(x \cup y)$ (pooled scale, not concatenated spread)

Properties

Self-average $\text{AvgSpread}(x, x) = \text{Spread}(x)$

Symmetry $\text{AvgSpread}(x, y) = \text{AvgSpread}(y, x)$

Scale equivariance $\text{AvgSpread}(k \cdot x, k \cdot y) = |k| \cdot \text{AvgSpread}(x, y)$

Mixed scaling $\text{AvgSpread}(k_1 \cdot x, k_2 \cdot x) = \frac{|k_1| + |k_2|}{2} \cdot \text{Spread}(x)$

Example

$\text{AvgSpread}(x, y) = 5$ where $\text{Spread}(x) = 6$, $\text{Spread}(y) = 4$, $n = m$

$\text{AvgSpread}(x, y) = \text{AvgSpread}(y, x)$

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.

1.12. Disparity

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

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 — $\text{sparsity}(\mathbf{x}), \text{sparsity}(\mathbf{y})$

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

$\text{Disparity}(\mathbf{x}, \mathbf{y}) = 0.4$ where $\text{Shift} = 2, \text{AvgSpread} = 5$

$\text{Disparity}(\mathbf{x} + c, \mathbf{y} + c) = \text{Disparity}(\mathbf{x}, \mathbf{y})$ $\text{Disparity}(k\mathbf{x}, k\mathbf{y}) = \text{Disparity}(\mathbf{x}, \mathbf{y})$

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.

1.13. PairwiseMargin

`PairwiseMargin(n, m, 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 (see Fast PairwiseMargin)

Domain — $n, m \geq 1$, 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.

1.14. SignedRankMargin

SignedRankMargin(n , 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 (see Fast SignedRankMargin)

Domain — $n \geq 2$, misrate $\geq 2^{1-n}$

Unit — count

Note — assumes weak symmetry and weak continuity

Properties

Bounds $0 \leq \text{SignedRankMargin}(n, \text{misrate}) \leq n(n + 1)/2$

Monotonicity lower misrate \rightarrow smaller margin \rightarrow 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.

1.15. ShiftBounds

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

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, 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^{-6} for critical decisions where errors are costly, or 10^{-3} for everyday analysis. 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.

1.16. RatioBounds

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

Robust bounds on $\text{Ratio}(x, y)$ 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 $\text{RatioBounds}(k \cdot x, k \cdot y, \text{misrate}) = \text{RatioBounds}(x, y, \text{misrate})$

Scale equivariance $\text{RatioBounds}(k_x \cdot x, k_y \cdot y, \text{misrate}) = \left(\frac{k_x}{k_y}\right) \cdot \text{RatioBounds}(x, y, \text{misrate})$

Multiplicative antisymmetry $\text{RatioBounds}(x, y, \text{misrate}) = \frac{1}{\text{RatioBounds}(y, x, \text{misrate})}$ (bounds reversed)

Example

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

Relationship to ShiftBounds

RatioBounds is computed via log-transformation:

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

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^{-6} for critical decisions where errors are costly, or 10^{-3} for everyday analysis. 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.

1.17. CenterBounds

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

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

Unit — same as measurements

Note — assumes weak symmetry and weak continuity; exact for $n \leq 63$, 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^{-6} for critical decisions where errors are costly, or 10^{-3} for everyday analysis. 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.

2. Distributions

Distributions are parametrized random generators with well-defined statistical properties. Each distribution describes a family of random variables characterized by specific parameters.

Notation

$X \sim \text{Additive}(0, 1)$ — X is distributed as standard normal

$\text{Estimator}(x)$ — estimate computed from sample

$\text{Estimator}[X]$ — true value (asymptotic limit)

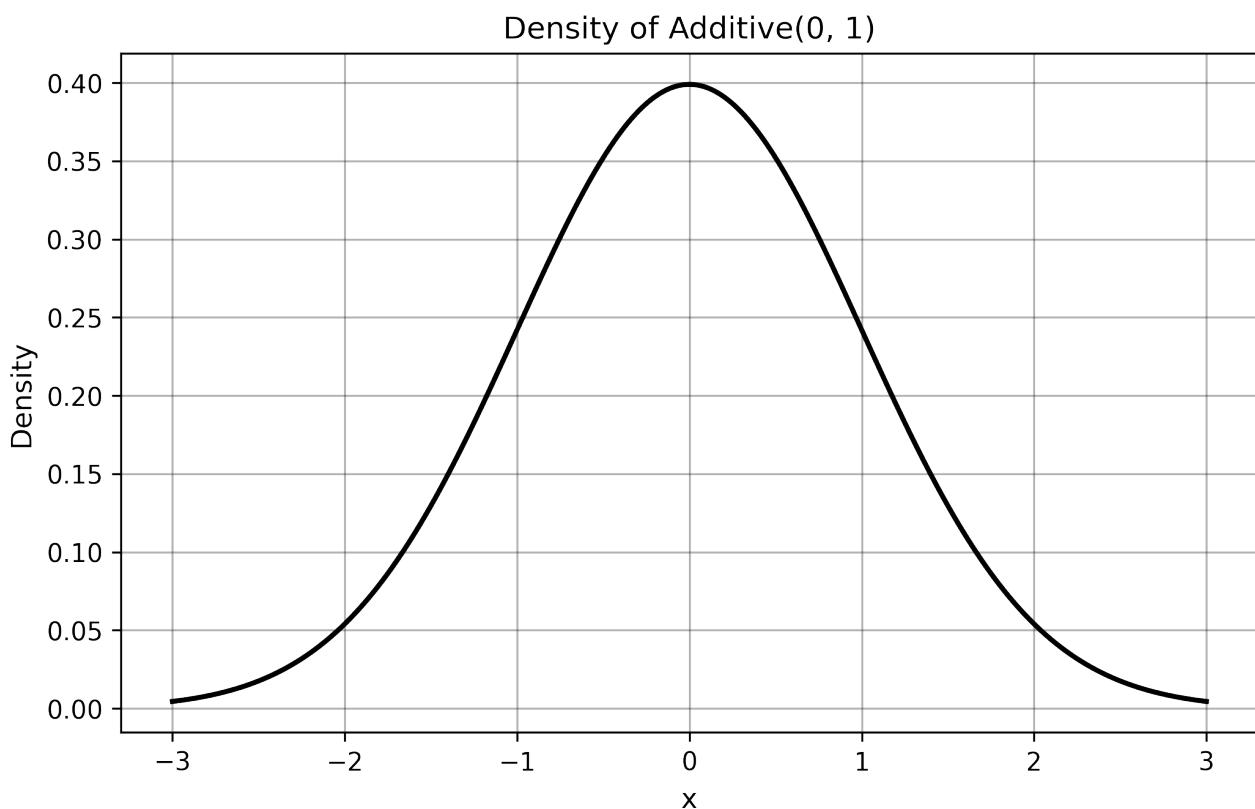
$n \rightarrow \infty$ — asymptotic case (large sample approximation)

2.1. Additive ('Normal')

Additive(mean, 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 $X_1 + X_2 + \dots + X_n$ 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.

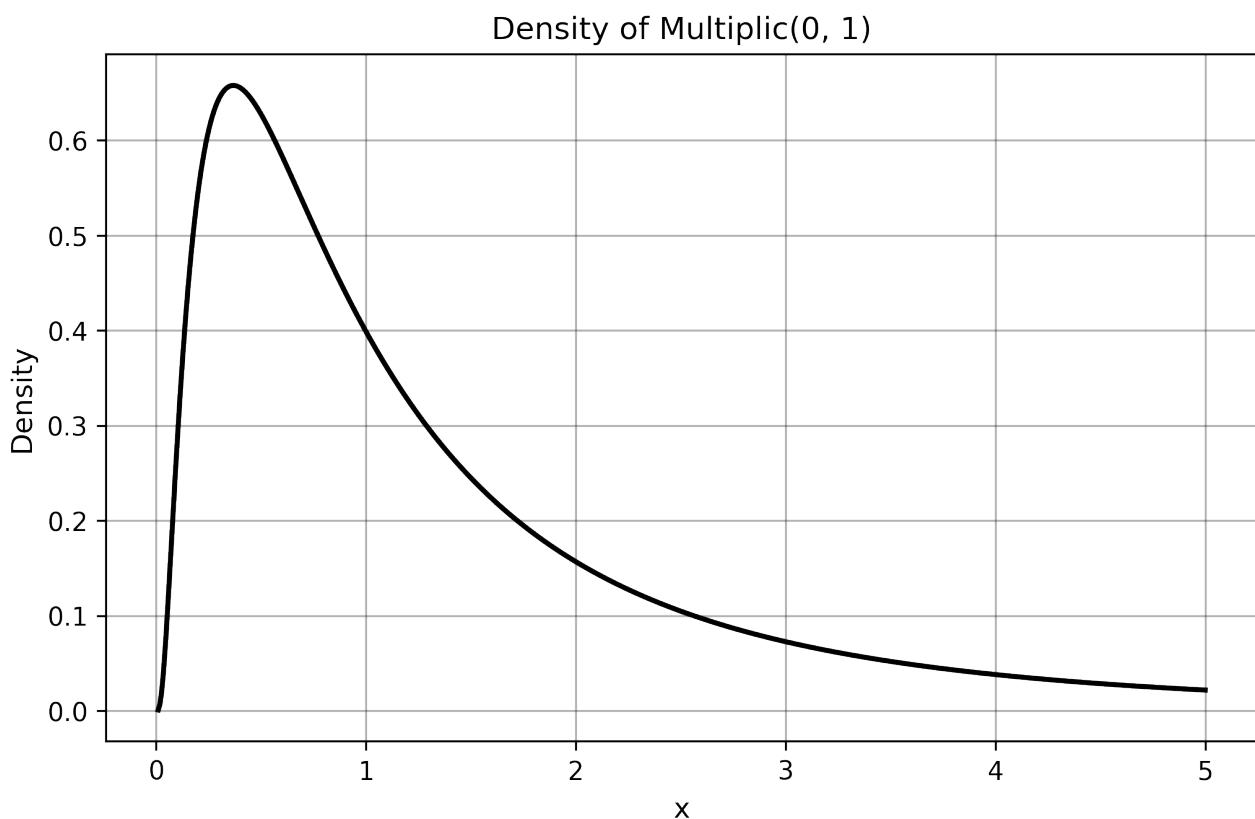
Caution: no perfectly additive distributions exist in real data; all real-world measurements contain deviations. Traditional estimators like Mean and StdDev lack robustness to outliers; use them only when strong evidence supports approximate additivity with no extreme measurements.

2.2. Multiplic ('LogNormal')

Multiplic(logMean, 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 $X_1 \cdot X_2 \cdot \dots \cdot X_n$ 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.

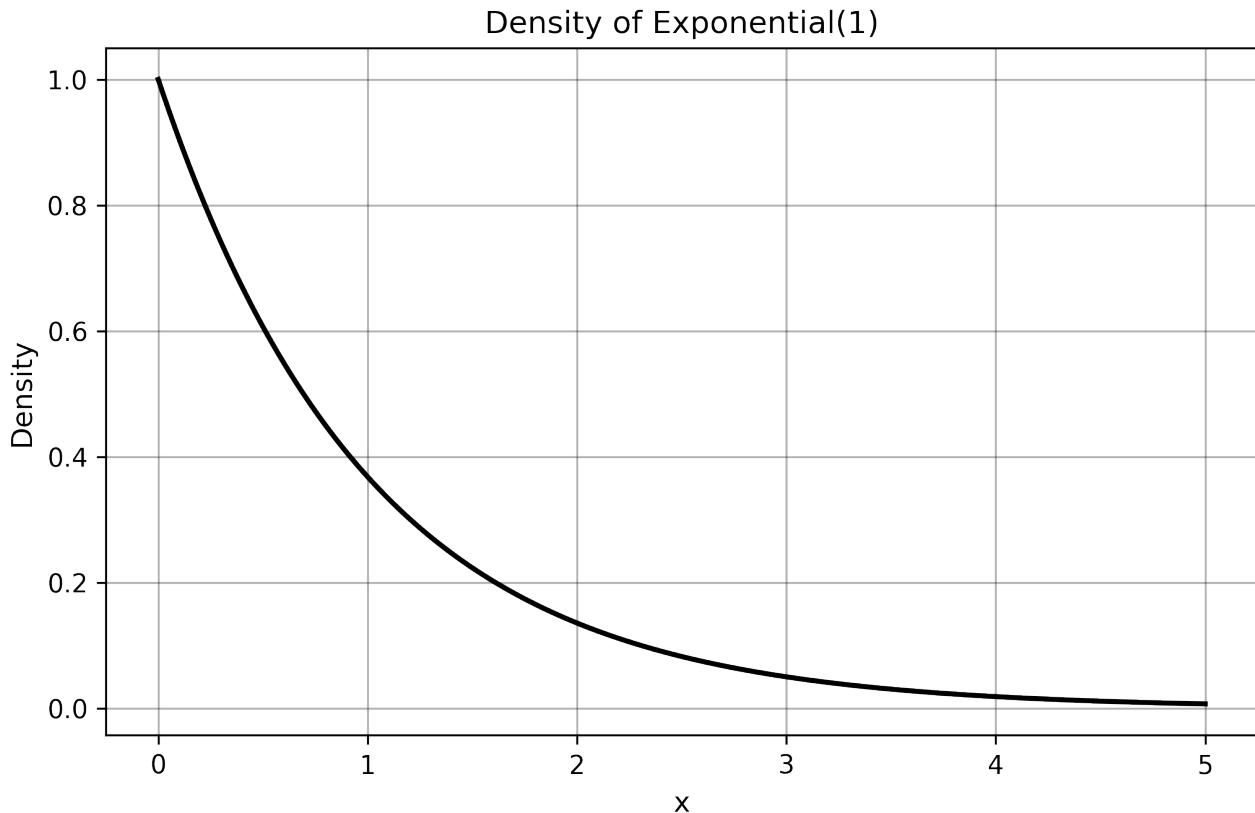
Applications: stock prices, file sizes, reaction times, income distributions, biological growth rates.

Caution: no perfectly multiplicative distributions exist in real data; all real-world measurements contain deviations. Traditional estimators struggle with the inherent skewness and heavy right tail.

2.3. Exponential

Exp(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.

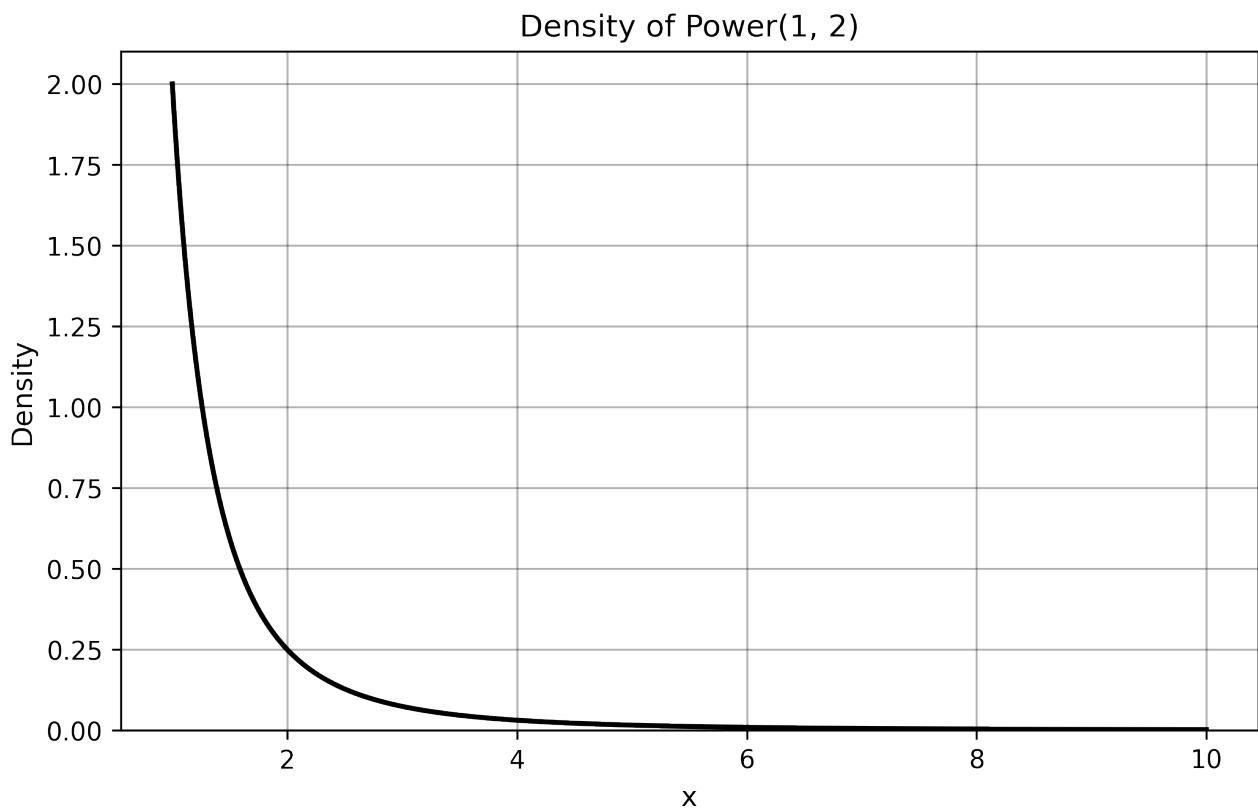
Caution: extreme skewness makes traditional location estimators like Mean unreliable; robust estimators provide more stable results.

2.4. Power ('Pareto')

Power(min, shape)

min: minimum value (lower bound, 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.

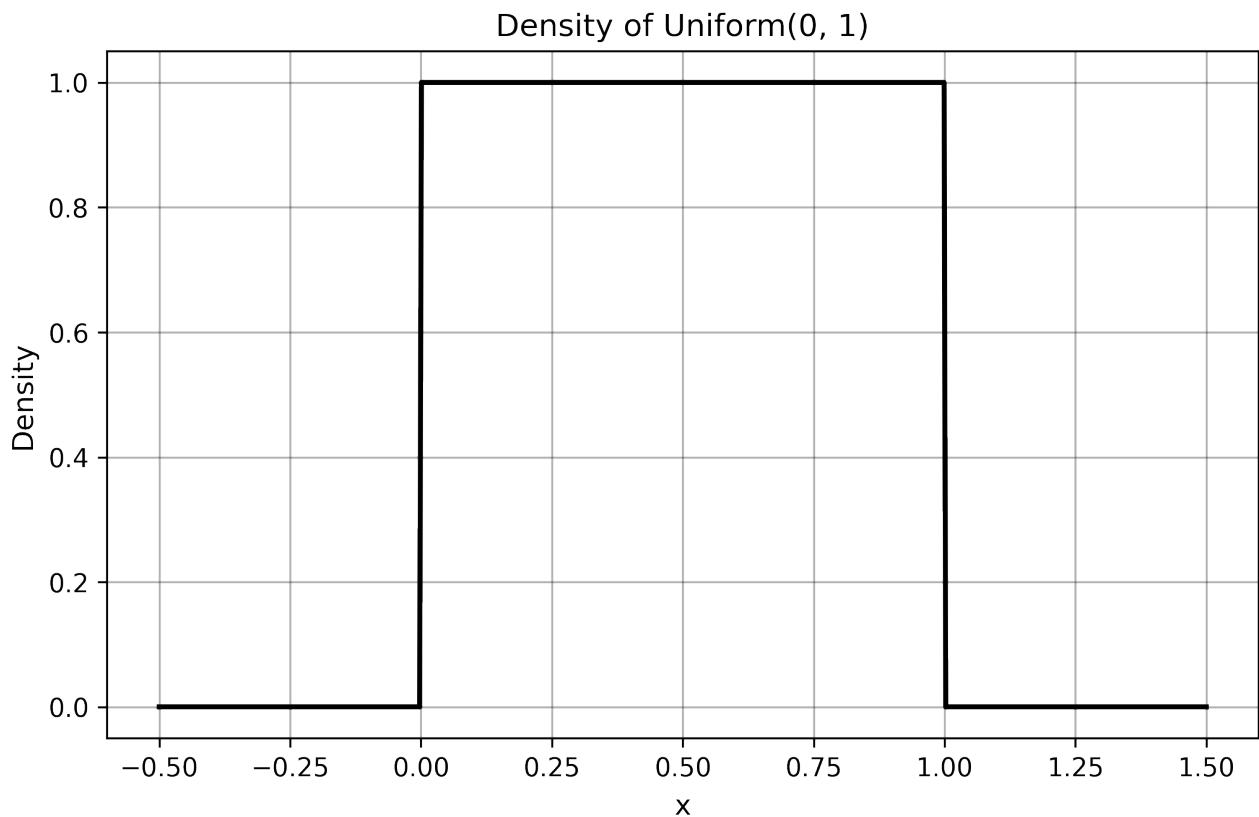
Caution: traditional variance-based estimators completely fail; robust estimators are essential for reliable analysis.

2.5. Uniform

Uniform(min, max)

min: lower bound of the support interval

max: upper bound of the support interval ($\text{max} > \text{min}$)



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.

3. Algorithms

This chapter describes the core algorithms that power the toolkit.

3.1. Pseudorandom Number Generation

The toolkit provides deterministic randomization utilities that produce identical sequences across all supported programming languages. This cross-language reproducibility enables deterministic experiments and cross-validation of statistical analyses.

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 algorithm 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:

```

 $x \leftarrow x + 0x9e3779b97f4a7c15$ 
 $z \leftarrow (x \oplus (x \gg 30)) \times 0xbff58476d1ce4e5b9$ 
 $z \leftarrow (z \oplus (z \gg 27)) \times 0x94d049bb133111eb$ 
output  $\leftarrow z \oplus (z \gg 31)$ 

```

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., `Rng(experiment-1)`), string seeds are converted to integers using FNV-1a hash (see Fowler et al. (1991)):

```

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

```

This enables meaningful experiment identifiers while maintaining determinism.

Uniform Floating-Point Generation

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

$$\text{uniform}() = (\text{next}() \gg 11) \times 2^{-53}$$

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

3.2. Shuffle Algorithm

The Shuffle function uses the Fisher-Yates algorithm (see Fisher & Yates (1938), Knuth (1997)), also known as the Knuth shuffle:

```
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 $O(1)$ additional space. The algorithm is unbiased: each of the $n!$ permutations has equal probability.

3.3. Selection Sampling

The Sample function uses selection sampling (see Fan et al. (1962)) to select k elements from n without replacement:

```
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. Distribution Sampling

Box-Muller Transform

The Additive (normal) distribution uses the Box-Muller transform (see Box & Muller (1958)) to convert uniform random values to normally distributed values. Given two independent uniform values $U_1, U_2 \in [0, 1]$:

$$\begin{aligned} Z_0 &= \sqrt{-2 \ln(U_1)} \cos(2\pi U_2) \\ Z_1 &= \sqrt{-2 \ln(U_1)} \sin(2\pi U_2) \end{aligned}$$

Both Z_0 and Z_1 are independent standard normal values. The implementation uses only Z_0 to maintain cross-language determinism.

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 the Box-Muller transform.

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.

Inverse Transform Sampling

Other distributions use inverse transform sampling. Given a uniform value $U \in [0, 1]$ and the inverse CDF F^{-1} :

Exponential: $X = -\ln(1 - U)/\lambda$

Pareto (Power): $X = x_{\min}/(1 - U)^{1/\alpha}$

Uniform: $X = a + U(b - a)$

The log-normal (Multiplic) distribution samples from Additive and applies the exponential transform: $X = \exp(\text{Additive}(\mu, \sigma))$.

3.5. Fast Center

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

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

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 $x_1 \leq x_2 \leq \dots \leq x_n$, the algorithm considers the implicit upper triangular matrix M where $M_{i,j} = x_i + x_j$ for $i \leq j$. 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  $(x_i + x_j)/2$  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;
}
}
```

3.6. Fast Spread

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

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

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 $y_1 \leq y_2 \leq \dots \leq y_n$, all pairwise absolute differences $|x_i - x_j|$ 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 candAt0rAbove = a[j] - a[i];
        if (candAt0rAbove < smallestAt0rAbove) smallestAt0rAbove = candAt0rAbove;
    }
}

// === 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 + smallestAt0rAbove);
    }
    else
    {
        // Odd N: pick the single middle depending on which side we hit.
        bool needLargest = (countBelow == kLow);
        return needLargest ? largestBelow : smallestAt0rAbove;
    }
}

// === 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;  
}  
}  
}
```

3.7. Fast Shift

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

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

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 $x_1 \leq x_2 \leq \dots \leq x_n$ and $y_1 \leq y_2 \leq \dots \leq y_m$, 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 $[searchMin, searchMax]$, 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 $x_i - y_j \leq t$ is counted. This is equivalent to counting pairs where $y_j \geq x_i - t$. 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 $x_i - y_j \leq t$. 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 inter-

polates between order statistics, the algorithm collects the necessary boundary values in a single pass and performs linear interpolation: $(1 - w) \cdot \text{lower} + w \cdot \text{upper}$.

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(pk), "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;
}

```

3.8. Fast Ratio

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

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

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 x was sorted, $\log x$ remains sorted (\log is monotonically increasing for positive values).
2. **Delegate to FastShift** — Use the Fast Shift algorithm to compute the desired quantile of pairwise differences in log-space. This leverages the $O((m + n) \log L)$ complexity of FastShift.
3. **Exp-transform** — Apply \exp to convert the result back to ratio-space. If the log-space median is d , then $\exp(d) = \exp(\log(x_i) - \log(y_j)) = \frac{x_i}{y_j}$ 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.

This algorithm naturally extends to computing bounds on ratios: RatioBounds uses the same log-exp transformation, delegating to ShiftBounds in log-space.

```
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();
    }
}

```

3.9. Fast PairwiseMargin

The PairwiseMargin function determines how many extreme pairwise differences to exclude when constructing bounds around Shift(\mathbf{x}, \mathbf{y}). Given samples $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_m)$, 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: $[z_{(k_{\text{left}})}, z_{(k_{\text{right}})}]$. The challenge lies in determining which order statistics produce bounds that contain the true shift 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 $k_{\text{left}} = \lfloor \frac{\text{PairwiseMargin}(n, m, \text{misrate})}{2} \rfloor + 1$ and $k_{\text{right}} = nm - \lfloor \frac{\text{PairwiseMargin}(n, m, \text{misrate})}{2} \rfloor$.

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:

$$\sigma_{n,m}(d) = \sum_{k|d} \varepsilon_k \cdot k, \quad \varepsilon_k = \begin{cases} 1 & \text{if } 1 \leq k \leq n \\ -1 & \text{if } m+1 \leq k \leq m+n \\ 0 & \text{otherwise} \end{cases}$$

This reduces memory to $O(nm)$ and enables efficient computation through $c = nm$.

The algorithm computes cumulative probabilities $\Pr(\text{Dominance} \leq c)$ sequentially until the threshold 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 $\Pr(\text{Dominance} \leq u)$ reaches misrate/2, returning the threshold value u without computing further probabilities.

This sequential approach performs particularly well for small misrates. For 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 probabilities 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 Dominance concentrates near $nm/2$ with variance $nm(n + m + 1)/12$. A basic Additive (Normal) approximation suffices asymptotically:

$$\text{Dominance} \approx \text{Additive}\left(nm/2, \sqrt{nm(n + m + 1)/12}\right)$$

This approximation underestimates tail probabilities for moderate sample sizes. The 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 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 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:

$$\Pr(\text{Dominance} \leq c) \approx \Phi(z) + e_3\varphi^{(3)}(z) + e_5\varphi^{(5)}(z) + e_7\varphi^{(7)}(z)$$

where Φ denotes the standard 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:

$$\begin{aligned} \mu_2 &= \frac{nm(n+m+1)}{12} \\ \mu_4 &= \frac{nm(n+m+1)}{240} (5nm(n+m) - 2(n^2 + m^2) + 3nm - 2(n+m)) \\ \mu_6 &= \frac{nm(n+m+1)}{4032} (35n^2m^2(n^2 + m^2) + 70n^3m^3 - 42nm(n^3 + m^3) \\ &\quad - 14n^2m^2(n+m) + 16(n^4 + m^4) - 52nm(n^2 + m^2) \\ &\quad - 43n^2m^2 + 32(n^3 + m^3) + 14nm(n+m) \\ &\quad + 8(n^2 + m^2) + 16nm - 8(n+m)) \end{aligned}$$

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 misrate = 10^{-6} or 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 $x_1, \dots, x_n > y_1, \dots, y_m$. 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:

$$\text{misrate}_{\min} = \frac{2}{\binom{n+m}{n}} = \frac{2 \cdot n! \cdot m!}{(n+m)!}$$

This represents the minimum probability of the most extreme ordering under random sampling. Setting $\text{misrate} < \text{misrate}_{\min}$ 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 $\text{misrate} < \text{misrate}_{\min}$ forces the algorithm to exclude zero pairwise differences from the tails, making $\text{PairwiseMargin} = 0$. The resulting bounds span all nm pairwise differences, returning $[z_{(1)}, z_{(nm)}]$ regardless of the desired confidence level. These bounds convey no useful information beyond the range of observed pairwise differences.

For small samples, misrate_{\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 misrate_{\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 $\text{misrate} > \text{misrate}_{\min}$. 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 misrate_{\min} for the given sample sizes. With moderate sample sizes ($n, m \geq 15$), misrate_{\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>
public 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);
}
}

```

3.10. Fast SignedRankMargin

The SignedRankMargin function determines how many extreme pairwise averages to exclude when constructing bounds around Center(\mathbf{x}). Given a sample $\mathbf{x} = (x_1, \dots, x_n)$, the 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: $[w_{(k_{\text{left}})}, w_{(k_{\text{right}})}]$. The challenge lies in determining which order statistics produce bounds that contain the true center with probability $1 - \text{misrate}$.

The margin function is the one-sample analog of PairwiseMargin. While PairwiseMargin uses the Mann-Whitney distribution for two-sample comparisons, 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 $|x_j - \theta|$, and θ is the true center. Under symmetry, each deviation is equally likely to be positive or negative, giving W a discrete distribution over $[0, \frac{n(n+1)}{2}]$.

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 misrate/2 is exceeded. For symmetric two-tailed bounds, the margin becomes $\text{SignedRankMargin} = 2w$. 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 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:

$$\mathbb{E}[W] = \frac{n(n+1)}{4}, \quad \text{Var}(W) = n(n+1) \frac{2n+1}{24}$$

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:

$$\Pr(W \leq w) \approx \Phi(z) + e_3 \varphi(z)(z^3 - 3z)$$

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 $n \leq 63$ 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 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:

$$\text{misrate}_{\min} = \frac{2}{2^n} = 2^{1-n}$$

Setting $\text{misrate} < \text{misrate}_{\min}$ makes bounds construction problematic. Pragmastat rejects such requests with a domain error.

The table below shows misrate_{\min} for small sample sizes:

n	2	3	5	7	10
misrate_{\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 $\text{misrate} > \text{misrate}_{\min}$. 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>
public 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);
    }

}

```

3.11. Fast CenterQuantiles

The Center estimator computes the median of all pairwise averages $\frac{x_i+x_j}{2}$ for $i \leq j$. For 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 $x_1 \leq x_2 \leq \dots \leq x_n$, 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 $[lo, hi]$, 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 $\text{count} \geq k$, the target is at or below the threshold and the search continues in the lower half; if $\text{count} < k$, 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 $j \geq i$ satisfy $\frac{x_i+x_j}{2} \leq t$, equivalently $x_j \leq 2t - x_i$. Because the array is sorted, a single pass through the array suffices. For each i , a pointer j tracks the largest index where $x_j \leq 2t - x_i$. 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.

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

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: $hi - lo \leq 10^{-14} \cdot \max(1, |lo|, |hi|)$. 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  $i \leq j$ .
/// 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;  
}  
}
```

4. 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, sparsity) 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.

4.1. 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.

4.2. 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.

`RelSpread(x)` — sample must be strictly positive (ensures `Center > 0`).

4.3. 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 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 sparsity assumption.

4.4. 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

`CenterBounds(x, misrate)` — requires weak symmetry for exact coverage.

Relationship to weak continuity

`CenterBounds` requires both weak continuity and weak symmetry for exact coverage.

4.5. Sparsity Assumption

Definition

The sample must be **non tie-dominant**: the median of pairwise absolute differences must be strictly positive. Equivalently, $\text{Spread}(x) > 0$.

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 sparsity assumption captures this and prevents tie-dominant samples from entering spread-based estimators.

Implication

Sparsity automatically implies the sample has at least two elements. For functions requiring sparsity, this is the primary assumption to check. A sample with $n = 1$ fails sparsity because $\text{Spread}(x) = 0$, not because of a separate size requirement.

Functions requiring sparsity

$\text{Spread}(x)$ — one-sample spread requires sparsity.

$\text{AvgSpread}(x, y)$ — both samples must have sparsity.

$\text{Disparity}(x, y)$ — both samples must have sparsity.

Naming

Sparsity 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 ($\text{Spread} > 0$), not who defined it.

4.6. 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. sparsity

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.

sparsity — requires $\text{Spread}(x) > 0$; 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.

sparsity — 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)

5. Studies

This section analyzes the estimators' properties using mathematical proofs. Most proofs are adapted from various textbooks and research papers, but only essential references are provided.

Unlike the main part of the manual, the studies require knowledge of classic statistical methods. Well-known facts and commonly accepted notation are used without special introduction. The studies provide detailed analyses of estimator properties for practitioners interested in rigorous proofs and numerical simulation results.

5.1. Summary Estimator Properties

This section compares the toolkit's robust estimators against traditional statistical methods to demonstrate their advantages across diverse conditions. While traditional estimators often work well under ideal conditions, the toolkit's estimators maintain reliable performance across diverse real-world scenarios.

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):

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

Dispersion Estimators:

Standard Deviation:

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

Median Absolute Deviation (around the median):

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

Spread (Shamos scale estimator):

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

5.2. 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 and Spread estimators achieve 29% breakdown points, providing substantial protection against realistic contamination levels while maintaining good precision. Below is a comparison with traditional estimators.

Asymptotic breakdown points for average estimators:

Mean	Median	Center
0%	50%	29%

Asymptotic breakdown points for dispersion estimators:

StdDev	MAD	Spread
0%	50%	29%

5.3. 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%.

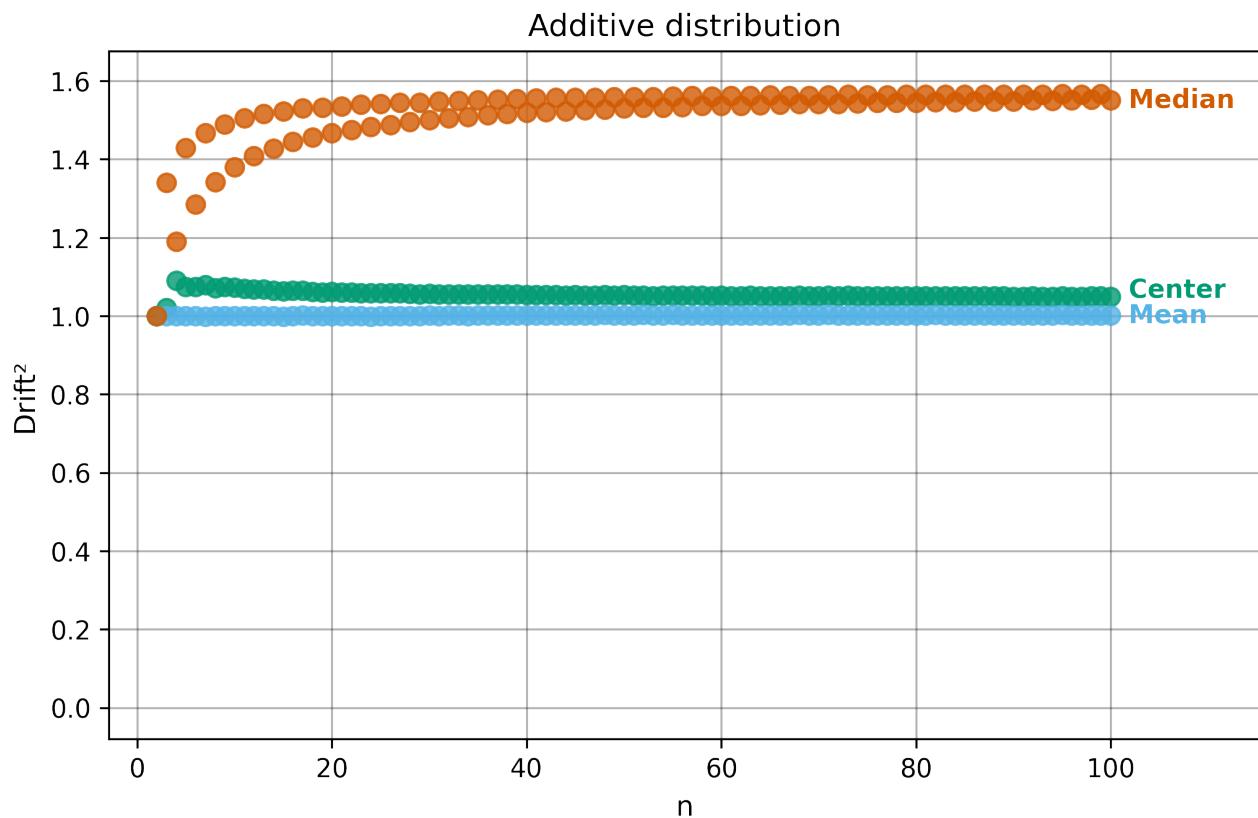
Drift is useful for comparing the precision of several estimators. To simplify the comparison, one of the estimators can be chosen as a baseline. A table of squared drift values, normalized by the baseline, shows the required sample size adjustment factor for switching from the baseline to another estimator. 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 the “From Statistical Efficiency to Drift” section for details.

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

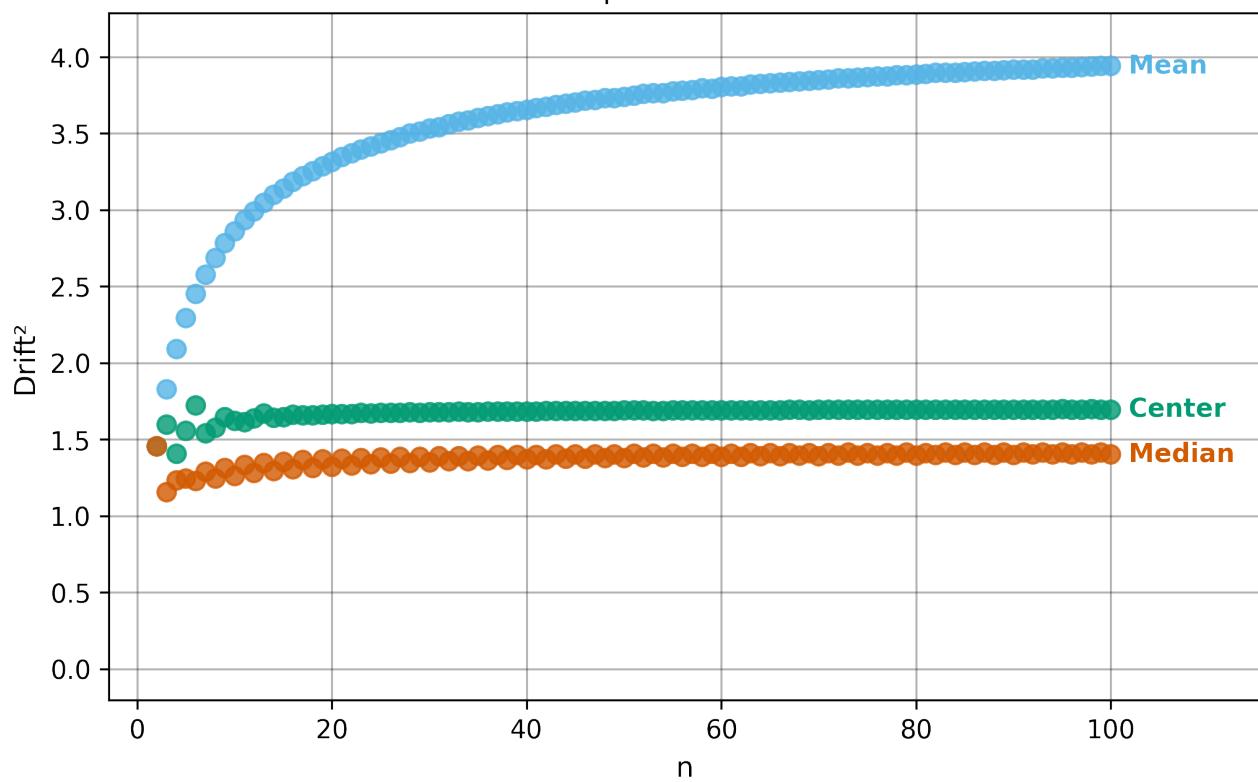
	Mean	Median	Center
Additive	1.0	1.571	1.047
Multiplic	3.95	1.40	1.7
Exp	1.88	1.88	1.69
Power	∞	0.9	2.1
Uniform	0.88	2.60	0.94

Rescaled to Center (sample size adjustment factors):

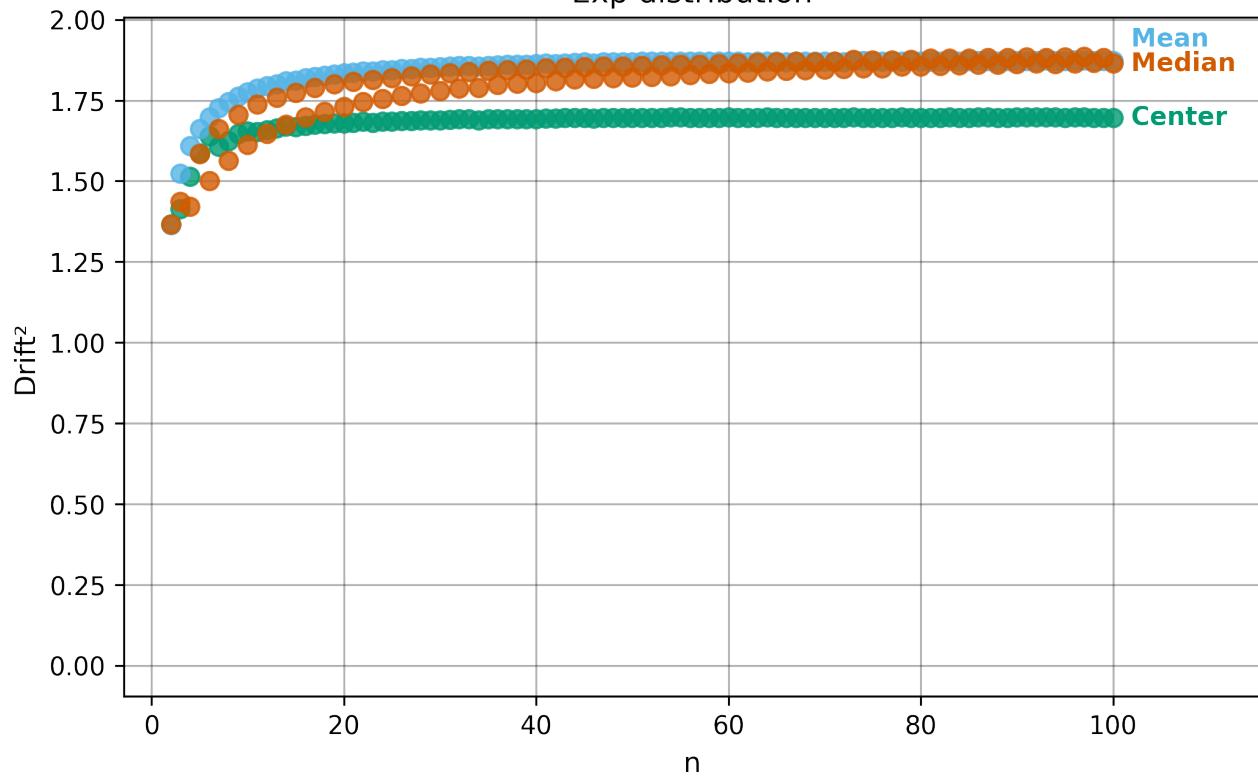
	Mean	Median	Center
Additive	0.96	1.50	1.0
Multiplic	2.32	0.82	1.0
Exp	1.11	1.11	1.0
Power	∞	0.43	1.0
Uniform	0.936	2.77	1.0

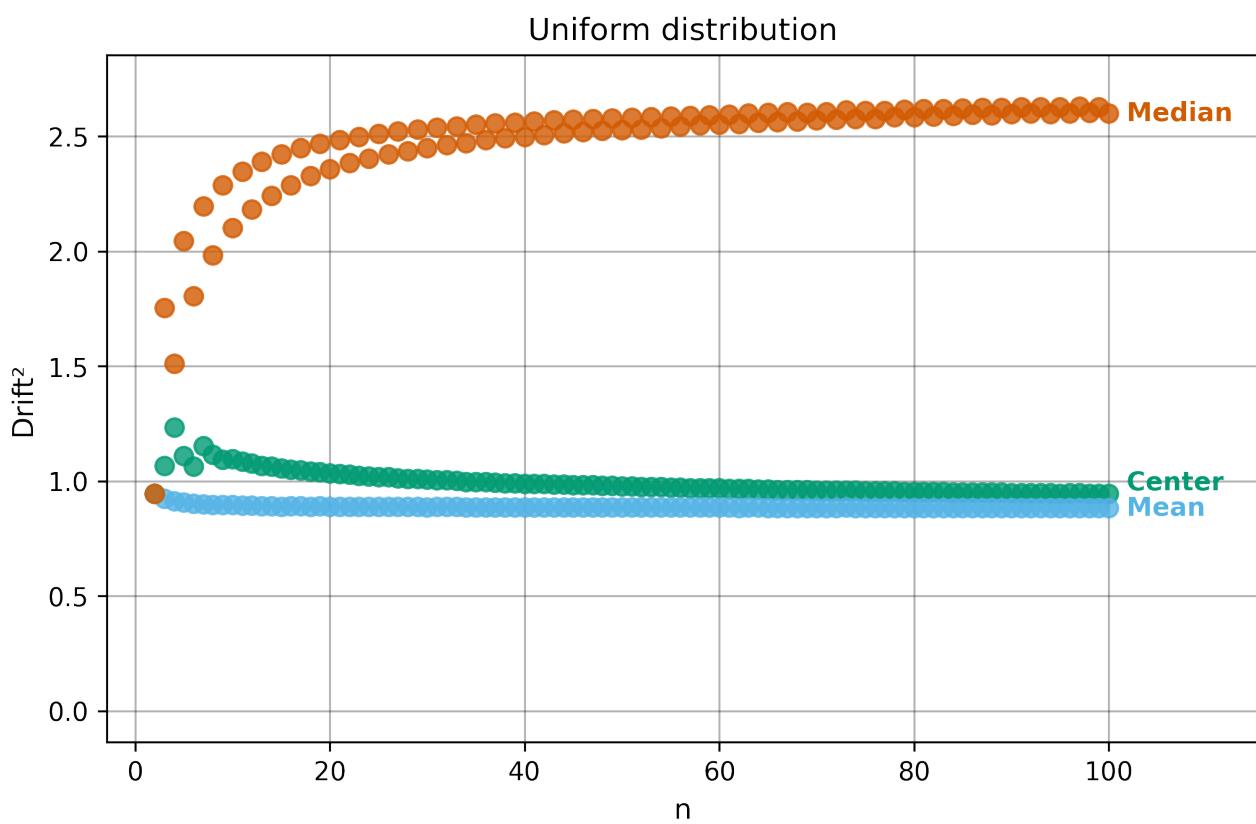
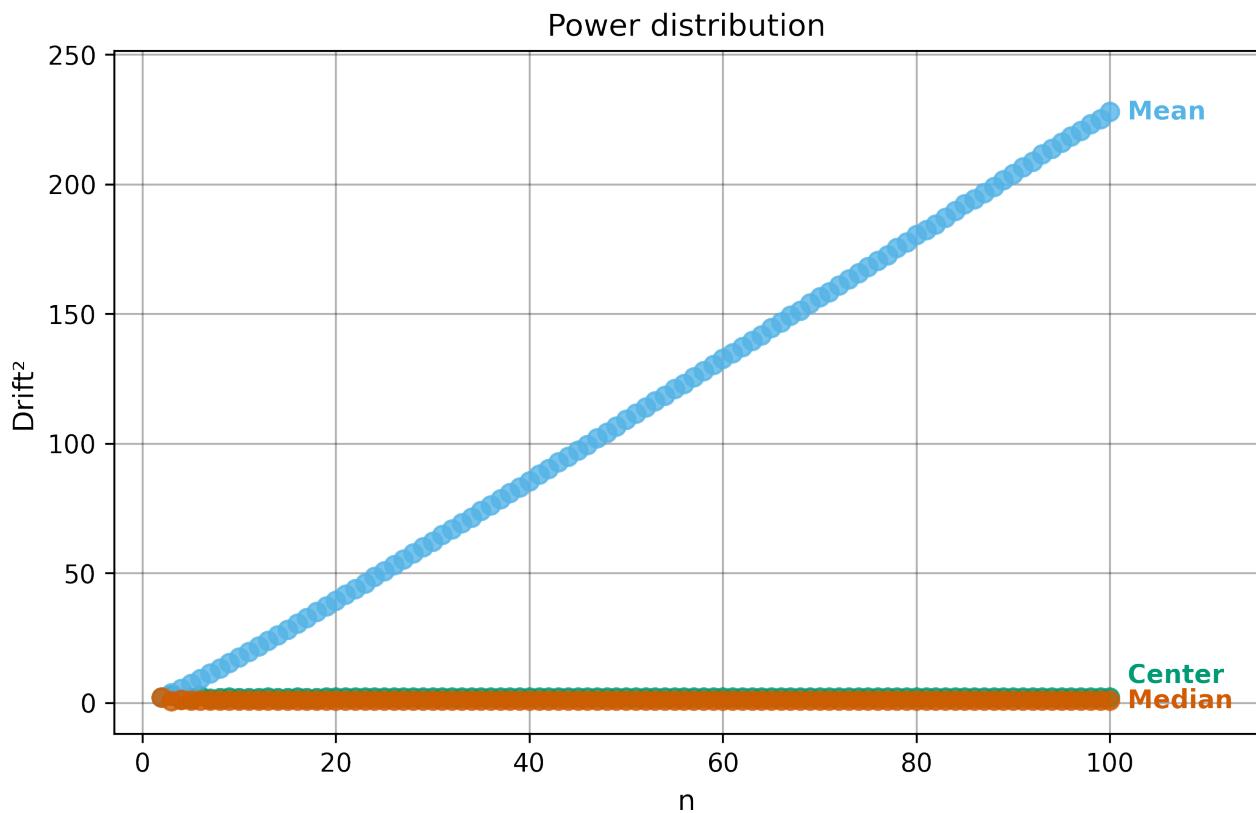


Multiplic distribution



Exp distribution



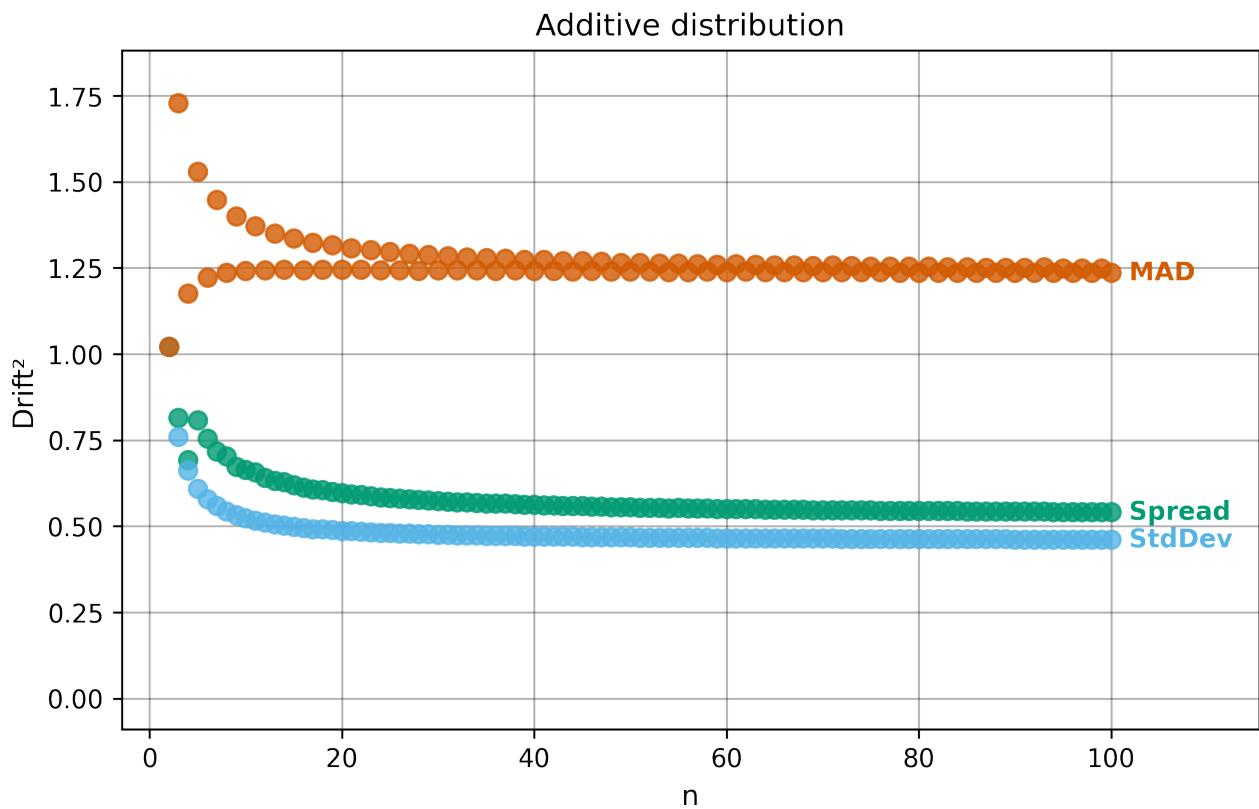


Squared Asymptotic Drift of Dispersion Estimators (values are approximated):

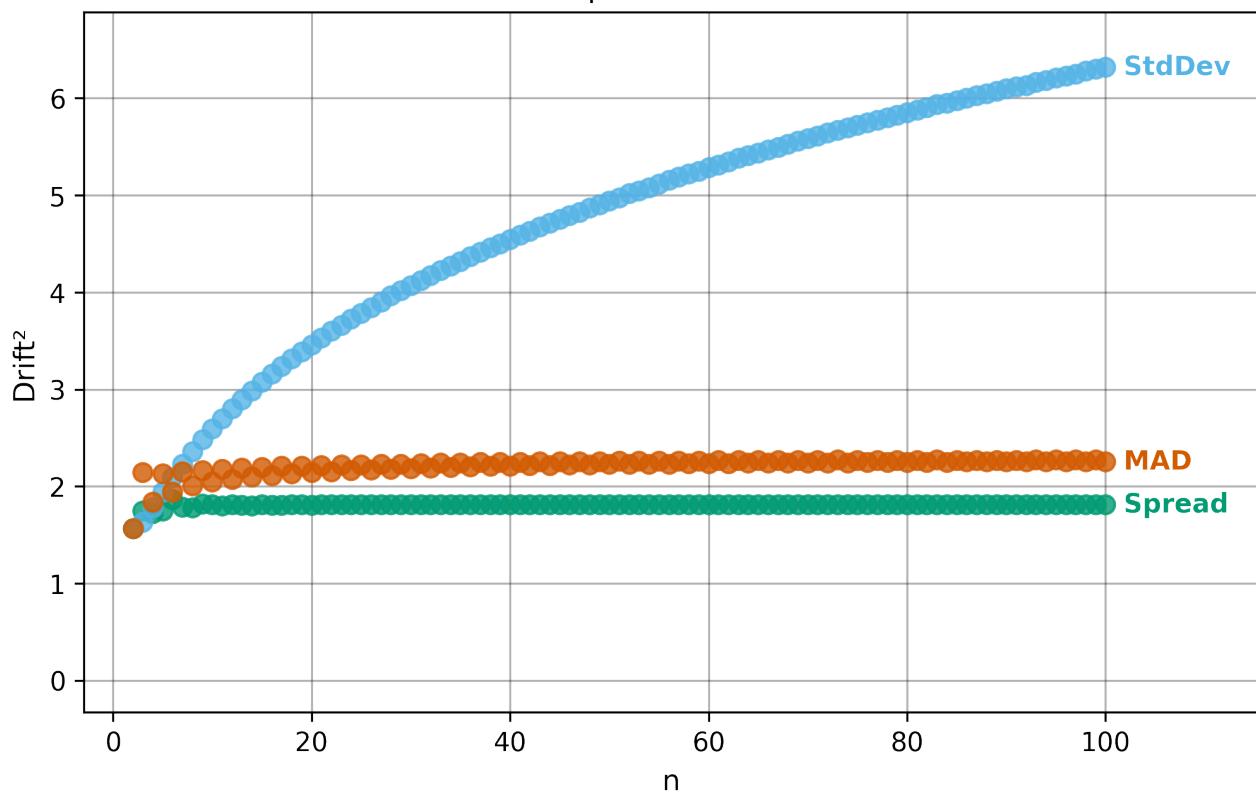
	StdDev	MAD	Spread
Additive	0.45	1.22	0.52
Multiplic	∞	2.26	1.81
Exp	1.69	1.92	1.26
Power	∞	3.5	4.4
Uniform	0.18	0.90	0.43

Rescaled to Spread (sample size adjustment factors):

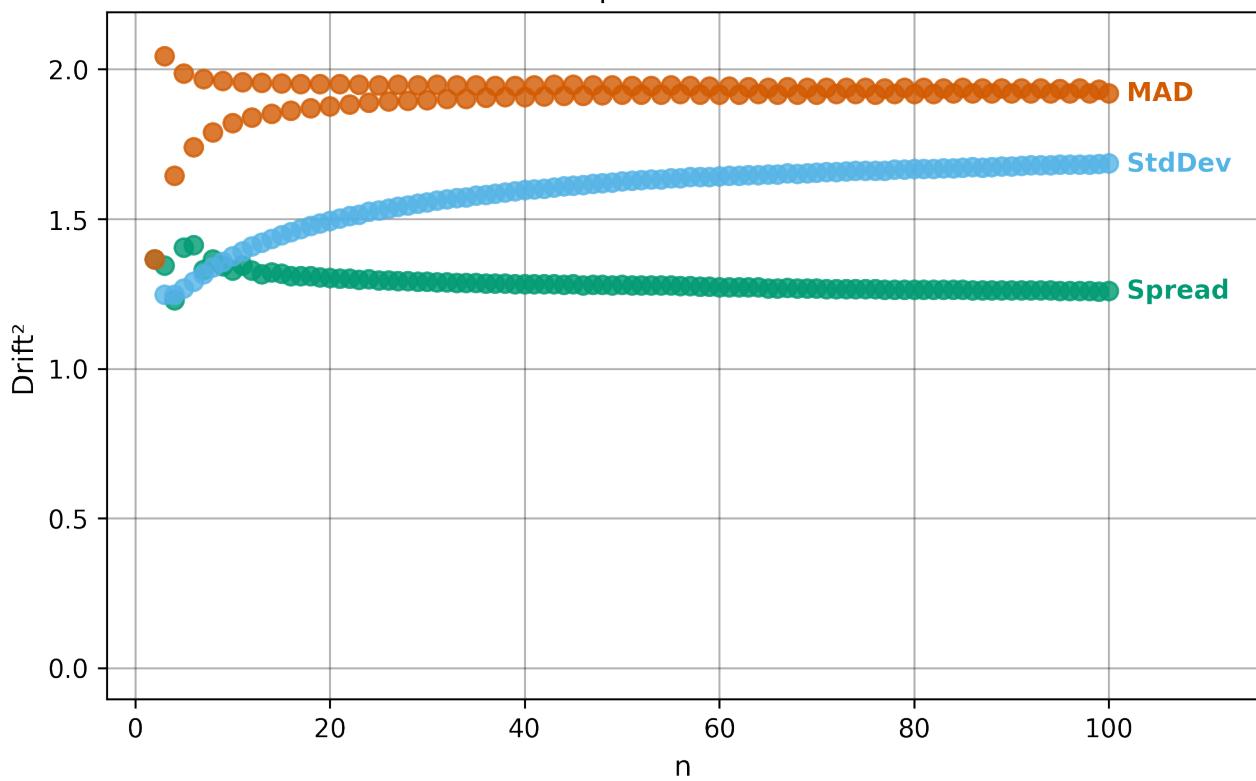
	StdDev	MAD	Spread
Additive	0.87	2.35	1.0
Multiplic	∞	1.25	1.0
Exp	1.34	1.52	1.0
Power	∞	0.80	1.0
Uniform	0.42	2.09	1.0



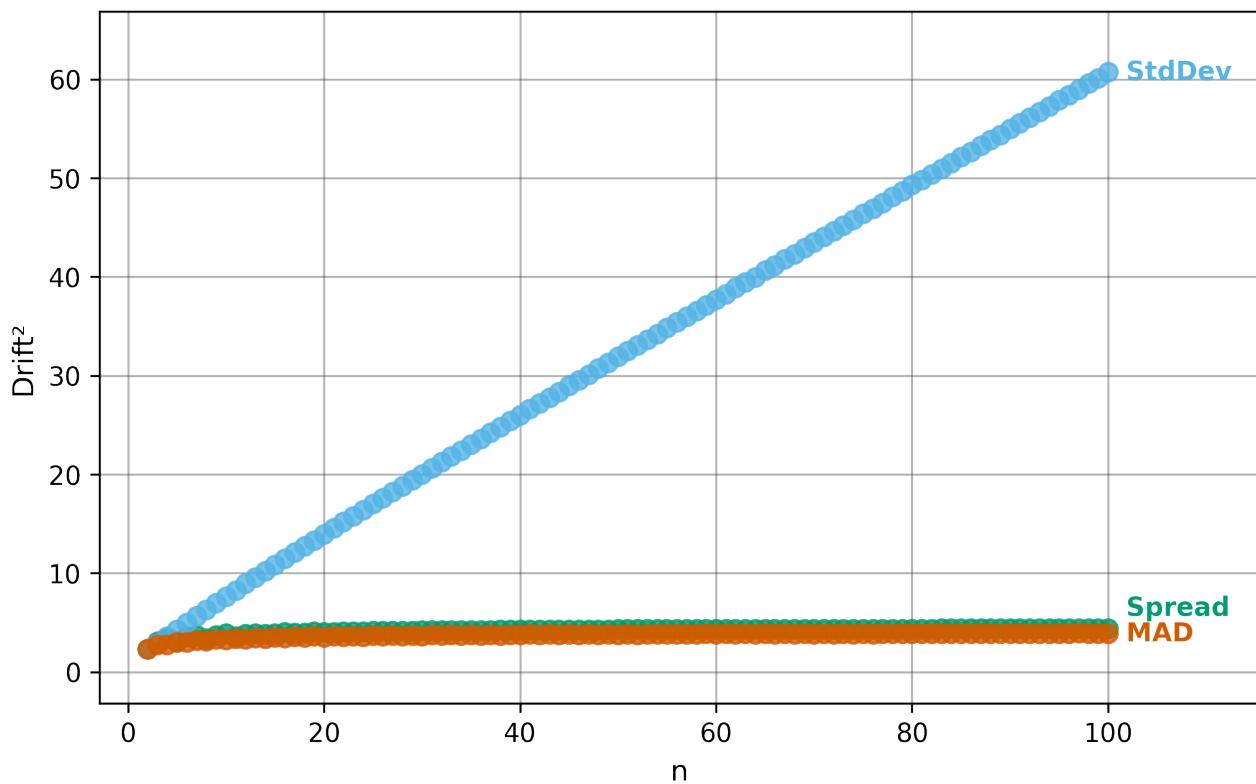
Multiplic distribution



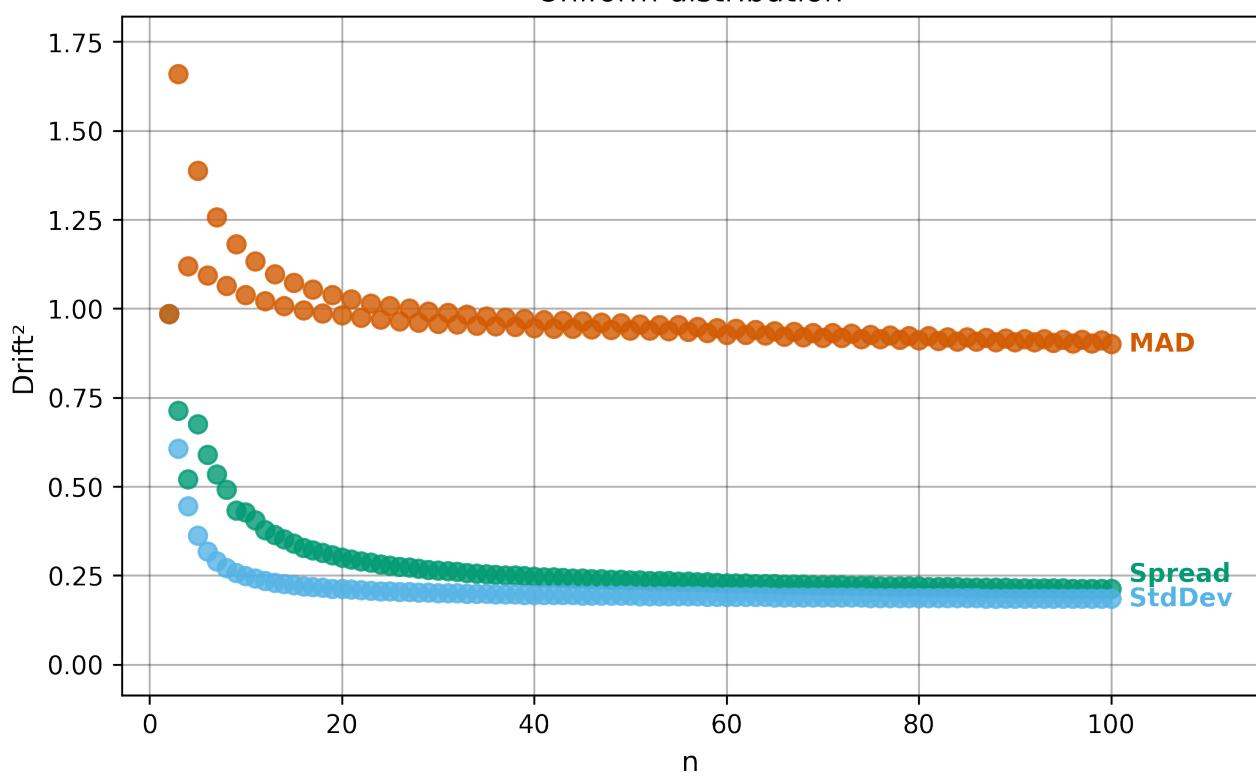
Exp distribution



Power distribution



Uniform distribution



5.4. 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
RelSpread	–	Invariant
Shift	Invariant	Equivariant
Ratio	–	Invariant
AvgSpread	Invariant	Equivariant
Disparity	Invariant	Invariant

5.5. 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^* :

$$\text{Efficiency}(T, X) = \frac{\text{Var}[T^*(X_1, \dots, X_n)]}{\text{Var}[T(X_1, \dots, X_n)]}$$

Relative efficiency compares two estimators by taking the ratio of their variances:

$$\text{RelativeEfficiency}(T_1, T_2, X) = \frac{\text{Var}[T_2(X_1, \dots, X_n)]}{\text{Var}[T_1(X_1, \dots, X_n)]}$$

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 :

$$\text{AvgDrift}(T, X, n) = \frac{\sqrt{n} \cdot \text{Spread}[T(X_1, \dots, X_n)]}{\text{Spread}[X]}$$

This formula measures estimator variability compared to data variability. $\text{Spread}[T(X_1, \dots, X_n)]$ 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 $\text{Spread}[X]$ creates a scale-free measure that provides consistent drift values across different distribution parameters and measurement units.

Dispersion estimators use a parallel formulation:

$$\text{DispDrift}(T, X, n) = \sqrt{n} \cdot \text{RelSpread}[T(X_1, \dots, X_n)]$$

Here RelSpread 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:

$$n_{\text{new}} = n_{\text{original}} \cdot \text{Drift}^2 \frac{T_2, X}{\text{Drift}^2}(T_1, X)$$

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, $\text{Drift}(T, X)$ denotes the limiting value as $n \rightarrow \infty$. With a baseline estimator, rescaled drift values enable direct comparisons:

$$\text{Drift}_{\text{baseline}(T, X)} = \frac{\text{Drift}(T, X)}{\text{Drift}(T_{\text{baseline}}, X)}$$

The standard drift definition assumes \sqrt{n} convergence rates typical under Additive (Normal) conditions. For broader applicability, drift generalizes to:

$$\text{AvgDrift}(T, X, n) = \frac{n^{\text{instability}} \cdot \text{Spread}[T(X_1, \dots, X_n)]}{\text{Spread}[X]}$$

$$\text{DispDrift}(T, X, n) = n^{\text{instability}} \cdot \text{RelSpread}[T(X_1, \dots, X_n)]$$

The instability parameter adapts to estimator convergence rates. The toolkit uses 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.

5.6. 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 $\text{misrate} = 10^{-6}$ (one-in-a-million error rate) as a reasonable default for most applications. This represents extremely high confidence (99.9999%) while remaining computationally practical and conceptually clear.

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’, 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. `ShiftBounds(x, y, misrate)` clearly indicates: it provides bounds on the shift, with a specified error rate. No background in classical statistics required to understand the concept.

5.7. From Mann-Whitney U-test to Pairwise Margin

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 $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_m)$, 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 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.

5.8. On Bootstrap for Center Bounds

A natural question arises: can bootstrap resampling improve `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 `CenterBounds`. The reasons are both computational and statistical.

Computational cost

`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 (`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, `CenterBounds` provides exact distribution-free coverage under symmetry. For $n = 5$ requesting `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 the study on misrate efficiency of [MedianBounds](#)), but its $O(n^{-1/2})$ 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.

5.9. On Misrate Efficiency of [MedianBounds](#)

This study 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

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

where k is the largest integer satisfying $2 \cdot \Pr(B \leq k - 1) \leq \text{misrate}$ 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 study derives the resulting efficiency loss and its convergence rate.

Achievable misrate levels

The achievable misrate values for sample size n are:

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

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 $O(n^{-1/2})$: 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

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 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 $O(n^{-3/2})$ — three polynomial orders faster than 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 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 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.

CenterBounds with its $O(n^{-3/2})$ 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 CenterBounds as the single bounds estimator in the toolkit. When symmetry is severely violated, the coverage drift of CenterBounds is gradual — mild asymmetry produces mild drift — making it a robust default without the efficiency penalty inherent to the sign test approach.

5.10. Additive ('Normal') Distribution

The Additive('Normal') distribution has two parameters: the mean and the standard deviation, written as Additive(mean, stdDev).

Asymptotic Spread Value

Consider two independent draws X and Y from the Additive(mean, 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, $\text{Var}[D] = 2 \cdot \text{stdDev}^2$. Thus D has distribution Additive(0, $\sqrt{2} \cdot \text{stdDev}$), 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 $\tau = \sqrt{2} \cdot \text{stdDev}$, so that $D \sim \text{Additive}(0, \tau)$. 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 $\tau = \sqrt{2} \cdot \text{stdDev}$ produces

$$\text{Median}(|X - Y|) = \sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}$$

Define $z_{0.75} := \Phi^{-1}(0.75) \approx 0.6744897502$. Numerically, the median absolute difference is approximately $\sqrt{2} \cdot z_{0.75} \cdot \text{stdDev} \approx 0.9538725524 \cdot \text{stdDev}$. This expression depends only on the scale parameter stdDev, not on the mean, reflecting the translation invariance of the problem.

Lemma: Average Estimator Drift Formula

For average estimators T_n with asymptotic standard deviation $a \cdot \text{stdDev} / \sqrt{n}$ around the mean μ , define $\text{RelSpread}[T_n] := \text{Spread}[T_n] / \text{Spread}[X]$. In the Additive('Normal') case, $\text{Spread}[X] = \sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}$.

For any average estimator T_n with asymptotic standard deviation $a \cdot \text{stdDev} / \sqrt{n}$ around the mean μ , the drift calculation follows:

The spread of two independent estimates: $\text{Spread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a \cdot \text{stdDev} / \sqrt{n}$

The relative spread: $\text{RelSpread}[T_n] = a / \sqrt{n}$

The asymptotic drift: $\text{Drift}(T, X) = a$

Asymptotic Mean Drift

For the sample mean $\text{Mean}(\mathbf{x}) = 1/n \sum_{i=1}^n x_i$ applied to samples from Additive(mean, stdDev), the sampling distribution of Mean is also additive with mean mean and standard deviation stdDev/\sqrt{n} .

Using the lemma with $a = 1$ (since the standard deviation is stdDev/\sqrt{n}):

$$\text{Drift}(\text{Mean}, X) = 1$$

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.

Asymptotic Median Drift

For the sample median $\text{Median}(\mathbf{x})$ applied to samples from Additive(mean, stdDev), the asymptotic sampling distribution of Median is approximately Additive ('Normal') with mean mean and standard deviation $\sqrt{\pi/2} \cdot \text{stdDev}/\sqrt{n}$.

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 $1/(4n[f(F^{-1}(0.5))]^2)$. For the Additive ('Normal') distribution with standard deviation stdDev, the density at the median (which equals the mean) is $1/(\text{stdDev} \sqrt{2\pi})$. Thus the asymptotic variance becomes $\pi \cdot \text{stdDev}^2/(2n)$.

Using the lemma with $a = \sqrt{\pi/2}$:

$$\text{Drift}(\text{Median}, X) = \sqrt{\pi/2}$$

Numerically, $\sqrt{\pi/2} \approx 1.2533$, so the median has approximately 25% higher drift than the mean under the Additive ('Normal') distribution.

Asymptotic Center Drift

For the sample center $\text{Center}(\mathbf{x}) = \text{Median}_{1 \leq i \leq j \leq n}(x_i + x_j)/2$ applied to samples from Additive(mean, 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:

$$\text{Var}[\text{Center}(X_{1:n})] = (\pi \cdot \text{stdDev}^2)/(3n)$$

This gives an asymptotic standard deviation of:

$$\text{StdDev}[\text{Center}(X_{1:n})] = \sqrt{\pi/3} \cdot \text{stdDev}/\sqrt{n}$$

Using the lemma with $a = \sqrt{\pi/3}$:

$$\text{Drift}(\text{Center}, X) = \sqrt{\pi/3}$$

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.

Lemma: Dispersion Estimator Drift Formula

For dispersion estimators T_n with asymptotic center $b \cdot \text{stdDev}$ and standard deviation $a \cdot \text{stdDev} / \sqrt{n}$, define $\text{RelSpread}[T_n] := \text{Spread}[T_n] / (b \cdot \text{stdDev})$.

For any dispersion estimator T_n with asymptotic distribution $T_n \sim \text{approx Additive}(b \cdot \text{stdDev}, (a \cdot \text{stdDev})^2/n)$, the drift calculation follows:

The spread of two independent estimates: $\text{Spread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a \cdot \text{stdDev} / \sqrt{n}$

The relative spread: $\text{RelSpread}[T_n] = \sqrt{2} \cdot z_{0.75} \cdot a / (b \sqrt{n})$

The asymptotic drift: $\text{Drift}(T, X) = \sqrt{2} \cdot z_{0.75} \cdot a / b$

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.

Asymptotic StdDev Drift

For the sample standard deviation $\text{StdDev}(\mathbf{x}) = \sqrt{1/(n-1) \sum_{i=1}^n (x_i - \text{Mean}(\mathbf{x}))^2}$ applied to samples from Additive(mean, stdDev), the sampling distribution of StdDev is approximately Additive ('Normal') for large n with mean stdDev and standard deviation $\text{stdDev}/\sqrt{2n}$.

Applying the lemma with $a = 1/\sqrt{2}$ and $b = 1$:

$$\text{Spread}[\text{StdDev}(X_{1:n})] = \sqrt{2} \cdot z_{0.75} \cdot 1/\sqrt{2} \cdot \text{stdDev}/\sqrt{n} = z_{0.75} \cdot \text{stdDev}/\sqrt{n}$$

For the dispersion drift, we use the relative spread formula:

$$\text{RelSpread}[\text{StdDev}(X_{1:n})] = \text{Spread}[\text{StdDev}(X_{1:n})] / \text{Center}[\text{StdDev}(X_{1:n})]$$

Since $\text{Center}[\text{StdDev}(X_{1:n})] \approx \text{stdDev}$ asymptotically:

$$\text{RelSpread}[\text{StdDev}(X_{1:n})] = (z_{0.75} \cdot \text{stdDev}/\sqrt{n}) / \text{stdDev} = z_{0.75}/\sqrt{n}$$

Therefore:

$$\text{Drift}(\text{StdDev}, X) = \lim_{n \rightarrow \infty} \sqrt{n} \cdot \text{RelSpread}[\text{StdDev}(X_{1:n})] = z_{0.75}$$

Numerically, $z_{0.75} \approx 0.67449$.

Asymptotic MAD Drift

For the median absolute deviation $MAD(\mathbf{x}) = \text{Median}(|x_i - \text{Median}(\mathbf{x})|)$ applied to samples from Additive(mean, stdDev), the asymptotic distribution is approximately Additive('Normal').

For the Additive ('Normal') distribution, the population MAD equals $z_{0.75} \cdot \text{stdDev}$. The asymptotic standard deviation of the sample MAD is:

$$\text{StdDev}[MAD(X_{1:n})] = c_{\text{mad}} \cdot \text{stdDev} / \sqrt{n}$$

where $c_{\text{mad}} \approx 0.78$.

Applying the lemma with $a = c_{\text{mad}}$ and $b = z_{0.75}$:

$$\text{Spread}[MAD(X_{1:n})] = \sqrt{2} \cdot z_{0.75} \cdot c_{\text{mad}} \cdot \text{stdDev} / \sqrt{n}$$

Since $\text{Center}[MAD(X_{1:n})] \approx z_{0.75} \cdot \text{stdDev}$ asymptotically:

$$\text{RelSpread}[MAD(X_{1:n})] = (\sqrt{2} \cdot z_{0.75} \cdot c_{\text{mad}} \cdot \text{stdDev} / \sqrt{n}) / (z_{0.75} \cdot \text{stdDev}) = (\sqrt{2} \cdot c_{\text{mad}}) / \sqrt{n}$$

Therefore:

$$\text{Drift}(MAD, X) = \lim_{n \rightarrow \infty} \sqrt{n} \cdot \text{RelSpread}[MAD(X_{1:n})] = \sqrt{2} \cdot c_{\text{mad}}$$

Numerically, $\sqrt{2} \cdot c_{\text{mad}} \approx \sqrt{2} \cdot 0.78 \approx 1.10$.

Asymptotic Spread Drift

For the sample spread $\text{Spread}(\mathbf{x}) = \text{Median}_{1 \leq i < j \leq n} |x_i - x_j|$ applied to samples from Additive(mean, stdDev), the asymptotic distribution is approximately Additive('Normal').

The spread estimator computes all pairwise absolute differences and takes their median. For the 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 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:

$$\text{RelSpread}[\text{Spread}(X_{1:n})] = (\sqrt{2} \cdot z_{0.75} \cdot c_{\text{spr}} \cdot \text{stdDev} / \sqrt{n}) / (\sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}) = c_{\text{spr}} / \sqrt{n}$$

Therefore:

$$\text{Drift}(\text{Spread}, X) = \lim_{n \rightarrow \infty} \sqrt{n} \cdot \text{RelSpread}[\text{Spread}(X_{1:n})] = c_{\text{spr}}$$

Numerically, $c_{\text{spr}} \approx 0.72$.

Summary

Summary for average estimators:

Estimator	$\text{Drift}(E, X)$	$\text{Drift}^2(E, X)$	$1/\text{Drift}^2(E, X)$
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	$\text{Drift}(E, X)$	$\text{Drift}^2(E, X)$	$1/\text{Drift}^2(E, X)$
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.

6. Reference Implementations

6.1. Python

Install from PyPI:

```
pip install pragmastat==7.0.1
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/py>

Pragmastat on PyPI: <https://pypi.org/project/pragmastat/>

Demo:

```
from pragmastat import (
    Rng,
    median,
    center,
    spread,
    rel_spread,
    shift,
    ratio,
    avg_spread,
    disparity,
    signed_rank_margin,
    pairwise_margin,
    center_bounds,
    shift_bounds,
    ratio_bounds,
)
from pragmastat.distributions import Additive, Exp, Multiplic, Power, Uniform

def main():
    # --- Randomization ---

    rng = Rng("demo-uniform")
    print(rng.uniform()) # 0.2640554428629759
    print(rng.uniform()) # 0.9348534835582796

    rng = Rng("demo-sample")
    print(rng.sample([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 3)) # [3, 8, 9]

    rng = Rng("demo-shuffle")
    print(rng.shuffle([1, 2, 3, 4, 5])) # [4, 2, 3, 5, 1]

    rng = Rng("demo-resample")
    print(rng.resample([1, 2, 3, 4, 5], 7)) # [5, 1, 1, 3, 3, 4, 5]

    # --- Distribution Sampling ---

    rng = Rng("demo-dist-uniform")
    dist = Uniform(0, 10)
```

```
print(dist.sample(rng)) # 6.54043657816832

rng = Rng("demo-dist-additive")
dist = Additive(0, 1)
print(dist.sample(rng)) # 0.17410448679568188

rng = Rng("demo-dist-exp")
dist = Exp(1)
print(dist.sample(rng)) # 0.6589065267276553

rng = Rng("demo-dist-power")
dist = Power(1, 2)
print(dist.sample(rng)) # 1.023677535537084

rng = Rng("demo-dist-multiplic")
dist = Multiplic(0, 1)
print(dist.sample(rng)) # 1.1273244602673853

# --- Single-Sample Statistics ---

x = [1, 3, 5, 7, 9]

print(median(x)) # 5
print(center(x)) # 5
print(spread(x)) # 4
print(spread([v + 10 for v in x])) # 4
print(spread([v * 2 for v in x])) # 8
print(rel_spread(x)) # 0.8

# --- Two-Sample Comparison ---

x = [0, 3, 6, 9, 12]
y = [0, 2, 4, 6, 8]

print(shift(x, y)) # 2
print(shift(y, x)) # -2
print(avg_spread(x, y)) # 5
print(disparity(x, y)) # 0.4
print(disparity(y, x)) # -0.4

x = [1, 2, 4, 8, 16]
y = [2, 4, 8, 16, 32]
print(ratio(x, y)) # 0.5
print(ratio(y, x)) # 2

# --- One-Sample Bounds ---

x = list(range(1, 11))

print(signed_rank_margin(10, 0.05)) # 18
print(center(x)) # 5.5
bounds = center_bounds(x, 0.05) # Bounds(lower=3.5, upper=7.5)
print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")
```

```

# --- Two-Sample Bounds ---

x = list(range(1, 31))
y = list(range(21, 51))

print(pairwise_margin(30, 30, 1e-4)) # 390
print(shift(x, y)) # -20
bounds = shift_bounds(x, y, 1e-4) # Bounds(lower=-30, upper=-10)
print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")

x = [1, 2, 3, 4, 5]
y = [2, 3, 4, 5, 6]
bounds = ratio_bounds(x, y, 0.05) # Bounds(lower=0.333..., upper=1.5)
print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")

if __name__ == "__main__":
    main()

```

6.2. TypeScript

Install from npm:

```
npm i pragmastat@7.0.1
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/ts>

Pragmastat on npm: <https://www.npmjs.com/package/pragmastat>

Demo:

```

import {
    median, center, spread, relSpread, shift, ratio, avgSpread, disparity,
    signedRankMargin, pairwiseMargin,
    centerBounds, shiftBounds, ratioBounds,
    Rng, Uniform, Additive, Exp, Power, Multiplic
} from '../src';

function main() {
    // --- Randomization ---

    let rng = new Rng("demo-uniform");
    console.log(rng.uniform()); // 0.2640554428629759
    console.log(rng.uniform()); // 0.9348534835582796

    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-shuffle");
    console.log(rng.shuffle([1, 2, 3, 4, 5])); // [4, 2, 3, 5, 1]

    rng = new Rng("demo-resample");

```

```
console.log(rng.resample([1, 2, 3, 4, 5], 7)); // [5, 1, 1, 3, 3, 4, 5]

// --- Distribution Sampling ---

rng = new Rng("demo-dist-uniform");
let dist = new Uniform(0, 10);
console.log(dist.sample(rng)); // 6.54043657816832

rng = new Rng("demo-dist-additive");
let addDist = new Additive(0, 1);
console.log(addDist.sample(rng)); // 0.17410448679568188

rng = new Rng("demo-dist-exp");
let expDist = new Exp(1);
console.log(expDist.sample(rng)); // 0.6589065267276553

rng = new Rng("demo-dist-power");
let powDist = new Power(1, 2);
console.log(powDist.sample(rng)); // 1.023677535537084

rng = new Rng("demo-dist-multiplic");
let mulDist = new Multiplic(0, 1);
console.log(mulDist.sample(rng)); // 1.1273244602673853

// --- Single-Sample Statistics ---

let x = [1, 3, 5, 7, 9];

console.log(median(x)); // 5
console.log(center(x)); // 5
console.log(spread(x)); // 4
console.log(spread(x.map(v => v + 10))); // 4
console.log(spread(x.map(v => v * 2))); // 8
console.log(relSpread(x)); // 0.8

// --- Two-Sample Comparison ---

x = [0, 3, 6, 9, 12];
let y = [0, 2, 4, 6, 8];

console.log(shift(x, y)); // 2
console.log(shift(y, x)); // -2
console.log(avgSpread(x, y)); // 5
console.log(disparity(x, y)); // 0.4
console.log(disparity(y, x)); // -0.4

x = [1, 2, 4, 8, 16];
y = [2, 4, 8, 16, 32];
console.log(ratio(x, y)); // 0.5
console.log(ratio(y, x)); // 2

// --- One-Sample Bounds ---
```

```

x = Array.from({ length: 10 }, (_, i) => i + 1);

console.log(signedRankMargin(10, 0.05)); // 18
console.log(center(x)); // 5.5
console.log(centerBounds(x, 0.05)); // { lower: 3.5, upper: 7.5 }

// --- Two-Sample Bounds ---

x = Array.from({ length: 30 }, (_, i) => i + 1);
y = Array.from({ length: 30 }, (_, i) => i + 21);

console.log(pairwiseMargin(30, 30, 1e-4)); // 390
console.log(shift(x, y)); // -20
console.log(shiftBounds(x, y, 1e-4)); // { lower: -30, upper: -10 }

x = [1, 2, 3, 4, 5];
y = [2, 3, 4, 5, 6];
console.log(ratioBounds(x, y, 0.05)); // { lower: 0.333..., upper: 1.5 }

}

main();

```

6.3. R

Install from GitHub:

```

install.packages("remotes") # If 'remotes' is not installed
remotes::install_github("AndreyAkinshin/pragmastat",
                      subdir = "r/pragmastat", ref = "v7.0.1")
library(pragmastat)

```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/r>

Demo:

```

library(pragmastat)

# --- Randomization ---

r <- rng("demo-uniform")
print(r$uniform()) # 0.2640554428629759
print(r$uniform()) # 0.9348534835582796

r <- rng("demo-sample")
print(r$sample(0:9, 3)) # [3, 8, 9]

r <- rng("demo-shuffle")
print(r$shuffle(c(1, 2, 3, 4, 5))) # [4, 2, 3, 5, 1]

r <- rng("demo-resample")
print(r$resample(c(1, 2, 3, 4, 5), 7)) # [5, 1, 1, 3, 3, 4, 5]

# --- Distribution Sampling ---

```

```
r <- rng("demo-dist-uniform")
dist <- dist_uniform(0, 10)
print(dist$sample(r)) # 6.54043657816832

r <- rng("demo-dist-additive")
dist <- dist_additive(0, 1)
print(dist$sample(r)) # 0.17410448679568188

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-multiplic")
dist <- dist_multiplic(0, 1)
print(dist$sample(r)) # 1.1273244602673853

# --- Single-Sample Statistics ---
x <- c(1, 3, 5, 7, 9)

print(median(x)) # 5
print(center(x)) # 5
print(spread(x)) # 4
print(spread(x + 10)) # 4
print(spread(x * 2)) # 8
print(rel_spread(x)) # 0.8

# --- Two-Sample Comparison ---
x <- c(0, 3, 6, 9, 12)
y <- c(0, 2, 4, 6, 8)

print(shift(x, y)) # 2
print(shift(y, x)) # -2
print(avg_spread(x, y)) # 5
print(disparity(x, y)) # 0.4
print(disparity(y, x)) # -0.4

x <- c(1, 2, 4, 8, 16)
y <- c(2, 4, 8, 16, 32)
print(ratio(x, y)) # 0.5
print(ratio(y, x)) # 2

# --- One-Sample Bounds ---
x <- 1:10

print(signed_rank_margin(10, 0.05)) # 18
```

```

print(center(x)) # 5.5
bounds <- center_bounds(x, 0.05) # [lower=3.5, upper=7.5]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))

# --- Two-Sample Bounds ---

x <- 1:30
y <- 21:50

print(pairwise_margin(30, 30, 1e-4)) # 390
print(shift(x, y)) # -20
bounds <- shift_bounds(x, y, 1e-4) # [lower=-30, upper=-10]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))

x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 4, 5, 6)
bounds <- ratio_bounds(x, y, 0.05) # [lower=0.333..., upper=1.5]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))

```

6.4. C#

Install from NuGet via .NET CLI:

```
dotnet add package Pragmastat --version 7.0.1
```

Install from NuGet via Package Manager Console:

```
NuGet\Install-Package Pragmastat -Version 7.0.1
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/cs>

Pragmastat on NuGet: <https://www.nuget.org/packages/Pragmastat/>

Demo:

```

using static System.Console;
using Pragmastat.Distributions;
using Pragmastat.Functions;
using Pragmastat.Randomization;

namespace Pragmastat.Demo;

class Program
{
    static void Main()
    {
        // --- Randomization ---

        var rng = new Rng("demo-uniform");
        WriteLine(rng.Uniform()); // 0.2640554428629759
        WriteLine(rng.Uniform()); // 0.9348534835582796

        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-shuffle");
    WriteLine(string.Join(", ", rng.Shuffle([1.0, 2, 3, 4, 5]))); // 4, 2, 3, 5, 1

    rng = new Rng("demo-resample");
    WriteLine(string.Join(", ", rng.Resample([1.0, 2, 3, 4, 5], 7))); // 5, 1, 1, 3, 3, 4,
5

// --- Distribution Sampling ---

rng = new Rng("demo-dist-uniform");
IDistribution dist = new Uniform(0, 10);
WriteLine(dist.Sample(rng)); // 6.54043657816832

rng = new Rng("demo-dist-additive");
dist = new Additive(0, 1);
WriteLine(dist.Sample(rng)); // 0.17410448679568188

rng = new Rng("demo-dist-exp");
dist = new Exp(1);
WriteLine(dist.Sample(rng)); // 0.6589065267276553

rng = new Rng("demo-dist-power");
dist = new Power(1, 2);
WriteLine(dist.Sample(rng)); // 1.023677535537084

rng = new Rng("demo-dist-multiplic");
dist = new Multiplic(0, 1);
WriteLine(dist.Sample(rng)); // 1.1273244602673853

// --- Single-Sample Statistics ---

var x = new Sample(1, 3, 5, 7, 9);

WriteLine(Toolkit.Median(x)); // 5
WriteLine(x.Center()); // 5
WriteLine(x.Spread()); // 4
WriteLine((x + 10).Spread()); // 4
WriteLine((x * 2).Spread()); // 8
WriteLine(x.RelSpread()); // 0.8

// --- Two-Sample Comparison ---

x = new Sample(0, 3, 6, 9, 12);
var y = new Sample(0, 2, 4, 6, 8);

WriteLine(Toolkit.Shift(x, y)); // 2
WriteLine(Toolkit.Shift(y, x)); // -2
WriteLine(Toolkit.AvgSpread(x, y)); // 5
WriteLine(Toolkit.Disparity(x, y)); // 0.4
WriteLine(Toolkit.Disparity(y, x)); // -0.4
```

```

x = new Sample(1, 2, 4, 8, 16);
y = new Sample(2, 4, 8, 16, 32);
WriteLine(Toolkit.Ratio(x, y)); // 0.5
WriteLine(Toolkit.Ratio(y, x)); // 2

// --- One-Sample Bounds ---

x = new Sample(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

WriteLine(SignedRankMargin.Instance.Calc(10, 0.05)); // 18
WriteLine(Toolkit.Center(x)); // 5.5
WriteLine(Toolkit.CenterBounds(x, 0.05)); // [3.5, 7.5]

// --- Two-Sample Bounds ---

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);
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(PairwiseMargin.Instance.Calc(30, 30, 1e-4)); // 390
WriteLine(Toolkit.Shift(x, y)); // -20
WriteLine(Toolkit.ShiftBounds(x, y, 1e-4)); // [-30, -10]

x = new Sample(1, 2, 3, 4, 5);
y = new Sample(2, 3, 4, 5, 6);
WriteLine(Toolkit.RatioBounds(x, y, 0.05)); // [0.333..., 1.5]
}

}
}

```

6.5. Kotlin

Install from Maven Central Repository via Apache Maven:

```
<dependency>
    <groupId>dev.pragmastat</groupId>
    <artifactId>pragmastat</artifactId>
    <version>7.0.1</version>
</dependency>
```

Install from Maven Central Repository via Gradle:

```
implementation 'dev.pragmastat:pragmastat:7.0.1'
```

Install from Maven Central Repository via Gradle (Kotlin):

```
implementation("dev.pragmastat:pragmastat:7.0.1")
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/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() {
    // --- Randomization ---

    var rng = Rng("demo-uniform")
    println(rng.uniform()) // 0.2640554428629759
    println(rng.uniform()) // 0.9348534835582796

    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, 8, 9]

    rng = Rng("demo-shuffle")
    println(rng.shuffle(listOf(1.0, 2.0, 3.0, 4.0, 5.0))) // [4, 2, 3, 5, 1]

    rng = Rng("demo-resample")
    println(rng.resample(listOf(1.0, 2.0, 3.0, 4.0, 5.0), 7)) // [5, 1, 1, 3, 3, 4, 5]

    // --- Distribution Sampling ---

    rng = Rng("demo-dist-uniform")
    var dist: Distribution = Uniform(0.0, 10.0)
    println(dist.sample(rng)) // 6.54043657816832

    rng = Rng("demo-dist-additive")
    dist = Additive(0.0, 1.0)
    println(dist.sample(rng)) // 0.17410448679568188

    rng = Rng("demo-dist-exp")
    dist = Exp(1.0)
    println(dist.sample(rng)) // 0.6589065267276553

    rng = Rng("demo-dist-power")
    dist = Power(1.0, 2.0)
    println(dist.sample(rng)) // 1.023677535537084

    rng = Rng("demo-dist-multiplic")
    dist = Multiplic(0.0, 1.0)
    println(dist.sample(rng)) // 1.1273244602673853

    // --- Single-Sample Statistics ---

    var x = listOf(1.0, 3.0, 5.0, 7.0, 9.0)
```

```

println(median(x)) // 5
println(center(x)) // 5
println(spread(x)) // 4
println(spread(x.map { it + 10 })) // 4
println(spread(x.map { it * 2 })) // 8
println(relSpread(x)) // 0.8

// --- Two-Sample Comparison ---

x = listOf(0.0, 3.0, 6.0, 9.0, 12.0)
var y = listOf(0.0, 2.0, 4.0, 6.0, 8.0)

println(shift(x, y)) // 2
println(shift(y, x)) // -2
println(avgSpread(x, y)) // 5
println(disparity(x, y)) // 0.4
println(disparity(y, x)) // -0.4

x = listOf(1.0, 2.0, 4.0, 8.0, 16.0)
y = listOf(2.0, 4.0, 8.0, 16.0, 32.0)
println(ratio(x, y)) // 0.5
println(ratio(y, x)) // 2

// --- One-Sample Bounds ---

x = (1..10).map { it.toDouble() }

println(signedRankMargin(10, 0.05)) // 18
println(center(x)) // 5.5
println(centerBounds(x, 0.05)) // Bounds(lower=3.5, upper=7.5)

// --- Two-Sample Bounds ---

x = (1..30).map { it.toDouble() }
y = (21..50).map { it.toDouble() }

println(pairwiseMargin(30, 30, 1e-4)) // 390
println(shift(x, y)) // -20
println(shiftBounds(x, y, 1e-4)) // Bounds(lower=-30.0, upper=-10.0)

x = listOf(1.0, 2.0, 3.0, 4.0, 5.0)
y = listOf(2.0, 3.0, 4.0, 5.0, 6.0)
println(ratioBounds(x, y, 0.05)) // Bounds(lower=0.333..., upper=1.5)
}

```

6.6. Rust

Install from crates.io via cargo:

```
cargo add pragmastat@7.0.1
```

Install from crates.io via Cargo.toml:

```
[dependencies]
pragmastat = "7.0.1"
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/rs>

Pragmastat on crates.io: <https://crates.io/crates/pragmastat>

Demo:

```
use pragmastat::distributions::{Additive, Distribution, Exp, Multiplic, Power, Uniform};
use pragmastat::*;

fn print<E: std::fmt::Debug>(result: Result<f64, E>) {
    println!("{}", result.unwrap());
}

fn add(x: &[f64], val: f64) -> Vec<f64> {
    x.iter().map(|v| v + val).collect()
}

fn multiply(x: &[f64], val: f64) -> Vec<f64> {
    x.iter().map(|v| v * val).collect()
}

fn main() {
    // --- Randomization ---

    let mut rng = Rng::from_string("demo-uniform");
    println!("{}", rng.uniform()); // 0.2640554428629759
    println!("{}", rng.uniform()); // 0.9348534835582796

    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-shuffle");
    println!("{}", rng.shuffle(&[1.0, 2.0, 3.0, 4.0, 5.0])); // [4, 2, 3, 5, 1]

    let mut rng = Rng::from_string("demo-resample");
    println!("{}", rng.resample(&[1.0, 2.0, 3.0, 4.0, 5.0], 7)); // [5, 1, 1, 3, 3, 4,
5]

    // --- Distribution Sampling ---

    let mut rng = Rng::from_string("demo-dist-uniform");
    let dist = Uniform::new(0.0, 10.0);
    println!("{}", dist.sample(&mut rng)); // 6.54043657816832

    let mut rng = Rng::from_string("demo-dist-additive");
    let dist = Additive::new(0.0, 1.0);
    println!("{}", dist.sample(&mut rng)); // 0.17410448679568188
```

```
let mut rng = Rng::from_string("demo-dist-exp");
let dist = Exp::new(1.0);
println!("{}", dist.sample(&mut rng)); // 0.6589065267276553

let mut rng = Rng::from_string("demo-dist-power");
let dist = Power::new(1.0, 2.0);
println!("{}", dist.sample(&mut rng)); // 1.023677535537084

let mut rng = Rng::from_string("demo-dist-multiplic");
let dist = Multiplic::new(0.0, 1.0);
println!("{}", dist.sample(&mut rng)); // 1.1273244602673853

// --- Single-Sample Statistics ---

let x = vec![1.0, 3.0, 5.0, 7.0, 9.0];

print(median(&x)); // 5
print(center(&x)); // 5
print(spread(&x)); // 4
print(spread(&add(&x, 10.0))); // 4
print(spread(&multiply(&x, 2.0))); // 8
print(rel_spread(&x)); // 0.8

// --- Two-Sample Comparison ---

let x = vec![0.0, 3.0, 6.0, 9.0, 12.0];
let y = vec![0.0, 2.0, 4.0, 6.0, 8.0];

print(shift(&x, &y)); // 2
print(shift(&y, &x)); // -2
print(avg_spread(&x, &y)); // 5
print(disparity(&x, &y)); // 0.4
print(disparity(&y, &x)); // -0.4

let x = vec![1.0, 2.0, 4.0, 8.0, 16.0];
let y = vec![2.0, 4.0, 8.0, 16.0, 32.0];
print(ratio(&x, &y)); // 0.5
print(ratio(&y, &x)); // 2

// --- One-Sample Bounds ---

let x: Vec<f64> = (1..=10).map(|i| i as f64).collect();

println!("{}", signed_rank_margin(10, 0.05).unwrap()); // 18
print(center(&x)); // 5.5
let bounds = center_bounds(&x, 0.05).unwrap(); // {lower: 3.5, upper: 7.5}
println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);

// --- Two-Sample Bounds ---

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!("{}", pairwise_margin(30, 30, 1e-4).unwrap()); // 390
    print(shift(&x, &y)); // -20

    let bounds = shift_bounds(&x, &y, 1e-4).unwrap(); // {lower: -30, upper: -10}
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);

    let x = vec![1.0, 2.0, 3.0, 4.0, 5.0];
    let y = vec![2.0, 3.0, 4.0, 5.0, 6.0];
    let bounds = ratio_bounds(&x, &y, 0.05).unwrap(); // {lower: 0.333..., upper: 1.5}
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);
}

```

6.7. Go

Install from GitHub:

```
go get github.com/AndreyAkinshin/pragmastat/go/v4@v7.0.1
```

Source code: <https://github.com/AndreyAkinshin/pragmastat/tree/v7.0.1/go>

Demo:

```

package main

import (
    "fmt"
    "log"

    pragmastat "github.com/AndreyAkinshin/pragmastat/go/v4"
)

func must[T any](val T, err error) T {
    if err != nil {
        log.Fatal(err)
    }
    return val
}

func print(val float64, err error) {
    fmt.Println(must(val, err))
}

func add(x []float64, val float64) []float64 {
    result := make([]float64, len(x))
    for i, v := range x {
        result[i] = v + val
    }
    return result
}

func multiply(x []float64, val float64) []float64 {
    result := make([]float64, len(x))
}

```

```
for i, v := range x {
    result[i] = v * val
}
return result
}

func main() {
    // --- Randomization ---

    rng := pragmastat.NewRngFromString("demo-uniform")
    fmt.Println(rng.Uniform()) // 0.2640554428629759
    fmt.Println(rng.Uniform()) // 0.9348534835582796

    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-shuffle")
    fmt.Println(pragmastat.Shuffle(rng, []float64{1, 2, 3, 4, 5})) // [4 2 3 5 1]

    rng = pragmastat.NewRngFromString("demo-resample")
    fmt.Println(pragmastat.Resample(rng, []float64{1, 2, 3, 4, 5}, 7)) // [5 1 1 3 3 4 5]

    // --- Distribution Sampling ---

    rng = pragmastat.NewRngFromString("demo-dist-uniform")
    dist := pragmastat.NewUniform(0, 10)
    fmt.Println(dist.Sample(rng)) // 6.54043657816832

    rng = pragmastat.NewRngFromString("demo-dist-additive")
    addDist := pragmastat.NewAdditive(0, 1)
    fmt.Println(addDist.Sample(rng)) // 0.17410448679568188

    rng = pragmastat.NewRngFromString("demo-dist-exp")
    expDist := pragmastat.NewExp(1)
    fmt.Println(expDist.Sample(rng)) // 0.6589065267276553

    rng = pragmastat.NewRngFromString("demo-dist-power")
    powDist := pragmastat.NewPower(1, 2)
    fmt.Println(powDist.Sample(rng)) // 1.023677535537084

    rng = pragmastat.NewRngFromString("demo-dist-multiplic")
    mulDist := pragmastat.NewMultiplic(0, 1)
    fmt.Println(mulDist.Sample(rng)) // 1.1273244602673853

    // --- Single-Sample Statistics ---

    x := []float64{1, 3, 5, 7, 9}

    print(pragmastat.Median(x))           // 5
    print(pragmastat.Center(x))          // 5
    print(pragmastat.Spread(x))          // 4
    print(pragmastat.Spread(add(x, 10))) // 4
```

```
print(pragmastat.Spread(multiply(x, 2))) // 8
print(pragmastat.RelSpread(x))           // 0.8

// --- Two-Sample Comparison ---

x = []float64{0, 3, 6, 9, 12}
y := []float64{0, 2, 4, 6, 8}

print(pragmastat.Shift(x, y))      // 2
print(pragmastat.Shift(y, x))      // -2
print(pragmastat.AvgSpread(x, y)) // 5
print(pragmastat.Disparity(x, y)) // 0.4
print(pragmastat.Disparity(y, x)) // -0.4

x = []float64{1, 2, 4, 8, 16}
y = []float64{2, 4, 8, 16, 32}
print(pragmastat.Ratio(x, y)) // 0.5
print(pragmastat.Ratio(y, x)) // 2

// --- One-Sample Bounds ---

x = []float64{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

srMargin, _ := pragmastat.SignedRankMargin(10, 0.05)
fmt.Println(srMargin)                  // 18
print(pragmastat.Center(x))          // 5.5
fmt.Println(must(pragmastat.CenterBounds(x, 0.05))) // {Lower: 3.5, Upper: 7.5}

// --- Two-Sample Bounds ---

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}

pwMargin, _ := pragmastat.PairwiseMargin(30, 30, 1e-4)
fmt.Println(pwMargin)                  // 390
print(pragmastat.Shift(x, y))        // -20
fmt.Println(must(pragmastat.ShiftBounds(x, y, 1e-4))) // {Lower: -30, Upper: -10}

x = []float64{1, 2, 3, 4, 5}
y = []float64{2, 3, 4, 5, 6}
fmt.Println(must(pragmastat.RatioBounds(x, y, 0.05))) // {Lower: 0.333..., Upper: 1.5}
}
```

7. Reference Tests

7.1. Center Tests

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

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 Additive(10, 1):

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

Uniform distribution ($n = 5, 100$) — fuzzy testing with 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, ..., 49) (large odd, 1225 pairs)

parity-even-50: 50-element sequence (1, 2, ..., 50) (large even, 1275 pairs)

Extreme values — numerical stability and range tests:

extreme-large-5: $x = (10^8, 2 \cdot 10^8, 3 \cdot 10^8, 4 \cdot 10^8, 5 \cdot 10^8)$ (very large values)

extreme-small-5: $x = (10^{-8}, 2 \cdot 10^{-8}, 3 \cdot 10^{-8}, 4 \cdot 10^{-8}, 5 \cdot 10^{-8})$ (very small positive values)

extreme-wide-5: $x = (0.001, 1, 100, 1000, 1000000)$ (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: $x = (2, 1, 3)$ (middle element first)

unsorted-shuffle-4: $x = (3, 1, 4, 2)$ (interleaved)

unsorted-shuffle-5: $x = (5, 2, 4, 1, 3)$ (complex shuffle)

unsorted-last-first-5: $x = (5, 1, 2, 3, 4)$ (last moved to first)

unsorted-first-last-5: $x = (2, 3, 4, 5, 1)$ (first moved to last)

unsorted-duplicates-mixed-5: $x = (3, 3, 3, 3, 3)$ (all identical, any order)

unsorted-duplicates-unsorted-10: $x = (3, 1, 2, 3, 1, 3, 2, 1, 3, 2)$ (duplicates mixed)

unsorted-extreme-large-unsorted-5: $x = (5 \cdot 10^8, 10^8, 4 \cdot 10^8, 2 \cdot 10^8, 3 \cdot 10^8)$ (large values unsorted)

unsorted-parity-odd-reverse-7: $x = (7, 6, 5, 4, 3, 2, 1)$ (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: $x = (1, 2, 3, \dots, 100000)$

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.

7.2. Spread Tests

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

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: $x = (0, 2, 4, 6, 8)$, expected output: 4 (base case)

demo-2: $x = (10, 12, 14, 16, 18)$ (= demo-1 + 10), expected output: 4 (location invariance)

demo-3: $x = (0, 4, 8, 12, 16)$ ($= 2 \times$ demo-1), expected output: 8 (scale equivariance)

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

natural-2: $x = (1, 2)$, expected output: 1

natural-3: $x = (1, 2, 3)$, expected output: 1

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

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

negative-3: $x = (-3, -2, -1)$, expected output: 1

Additive distribution ($n = 5, 10, 30$) — Additive(10, 1):

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

Uniform distribution ($n = 5, 100$) — 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 $\text{Spread}(x) > 0$.

Algorithm stress tests — edge cases for fast algorithm implementation:

duplicates-10: $x = (1, 1, 1, 2, 2, 2, 3, 3, 3, 3)$ (many duplicates, stress tie-breaking)

parity-odd-7: $x = (1, 2, 3, 4, 5, 6, 7)$ (odd sample size, 21 differences)

parity-even-6: $x = (1, 2, 3, 4, 5, 6)$ (even sample size, 15 differences)

parity-odd-49: 49-element sequence $(1, 2, \dots, 49)$ (large odd, 1176 differences)

parity-even-50: 50-element sequence $(1, 2, \dots, 50)$ (large even, 1225 differences)

Extreme values — numerical stability and range tests:

extreme-large-5: $x = (10^8, 2 \cdot 10^8, 3 \cdot 10^8, 4 \cdot 10^8, 5 \cdot 10^8)$ (very large values)

extreme-small-5: $x = (10^{-8}, 2 \cdot 10^{-8}, 3 \cdot 10^{-8}, 4 \cdot 10^{-8}, 5 \cdot 10^{-8})$ (very small positive values)

extreme-wide-5: $x = (0.001, 1, 100, 1000, 1000000)$ (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: $x = (3, 1, 2)$ (rotated)

unsorted-shuffle-4: $x = (4, 2, 1, 3)$ (mixed order)

unsorted-shuffle-5: $x = (5, 1, 3, 2, 4)$ (partial shuffle)

unsorted-duplicates-unsorted-10: $x = (2, 3, 1, 3, 2, 1, 2, 3, 1, 3)$ (duplicates mixed)

unsorted-extreme-wide-unsorted-5: $x = (1000, 0.001, 1000000, 100, 1)$ (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: $x = (1, 2, 3, \dots, 100000)$

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.

7.3. RelSpread Tests

$$\text{RelSpread}(x) = \frac{\text{Spread}(x)}{|\text{Center}(x)|}$$

The RelSpread test suite contains 18 test cases (13 original + 5 unsorted) focusing on relative dispersion.

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

demo-1: $x = (1, 3, 5, 7, 9)$, expected output: 0.8 (base case)

demo-2: $x = (5, 15, 25, 35, 45)$ ($= 5 \times$ demo-1), expected output: 0.8 (scale invariance)

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

natural-2: $x = (1, 2)$, expected output: ≈ 0.667

natural-3: $x = (1, 2, 3)$, expected output: 0.5

natural-4: $x = (1, 2, 3, 4)$, expected output: 0.6 (validates composite with even size)

Uniform distribution ($n = 5, 10, 20, 30, 100$) — Uniform(0, 1):

uniform-5, uniform-10, uniform-20, uniform-30, uniform-100: random samples generated with seed 0

The uniform distribution tests span multiple sample sizes to verify that RelSpread correctly normalizes dispersion. Zero and negative values are excluded because RelSpread requires strictly positive samples.

Composite estimator stress tests — edge cases specific to division operation:

composite-small-center: $x = (0.001, 0.002, 0.003, 0.004, 0.005)$ (small center, tests division stability)

composite-large-spread: $x = (1, 100, 200, 300, 1000)$ (large spread relative to center)

composite-extreme-ratio: $x = (1, 1.0001, 1.0002, 1.0003, 1.0004)$ (tiny spread, tests precision)

Unsorted tests — verify sorting for composite estimator (5 tests):

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

unsorted-composite-small-unsorted: $x = (0.005, 0.001, 0.003, 0.002, 0.004)$ (small center unsorted)

unsorted-composite-large-unsorted: $x = (1000, 1, 300, 100, 200)$ (large spread unsorted)

Since RelSpread combines both Center and Spread, these tests verify that sorting works correctly for composite estimators.

7.4. Shift Tests

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

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: x = (0, 2, 4, 6, 8), y = (10, 12, 14, 16, 18), expected output: -10 (base case)
demo-2: x = (0, 2, 4, 6, 8), y = (0, 2, 4, 6, 8), expected output: 0 (identity property)
demo-3: x = (7, 9, 11, 13, 15), y = (13, 15, 17, 19, 21) (= demo-1+[7,3]), expected output: -6 (location equivariance)
demo-4: x = (0, 4, 8, 12, 16), y = (20, 24, 28, 32, 36) (= 2 × demo-1), expected output: -20 (scale equivariance)
demo-5: x = (10, 12, 14, 16, 18), 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: x = (1), y = (1), expected output: 0
natural-1-2: x = (1), y = (1, 2), expected output: -0.5
natural-1-3: x = (1), y = (1, 2, 3), expected output: -1
natural-2-1: x = (1, 2), y = (1), expected output: 0.5
natural-2-2: x = (1, 2), y = (1, 2), expected output: 0
natural-2-3: x = (1, 2), y = (1, 2, 3), expected output: -0.5
natural-3-1: x = (1, 2, 3), y = (1), expected output: 1
natural-3-2: x = (1, 2, 3), y = (1, 2), expected output: 0.5
natural-3-3: x = (1, 2, 3), y = (1, 2, 3), expected output: 0
```

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

```
negative-2-2: x = (-2, -1), y = (-2, -1), expected output: 0
```

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

```
mixed-2-2: x = (-1, 1), 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 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 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

The natural sequences validate anti-symmetry ($\text{Shift}(x, y) = -\text{Shift}(y, x)$) and the identity property ($\text{Shift}(x, x) = 0$). 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: $x = (3, 3, 3, 3, 3)$, $y = (3, 3, 3, 3, 3)$ (all identical, expected output: 0)

duplicates-10-10: $x = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5)$, $y = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5)$ (many duplicates)

parity-odd-7-7: $x = (1, 2, 3, 4, 5, 6, 7)$, $y = (1, 2, 3, 4, 5, 6, 7)$ (odd sizes, 49 differences, expected output: 0)

parity-even-6-6: $x = (1, 2, 3, 4, 5, 6)$, $y = (1, 2, 3, 4, 5, 6)$ (even sizes, 36 differences, expected output: 0)

parity-asymmetric-7-6: $x = (1, 2, 3, 4, 5, 6, 7)$, $y = (1, 2, 3, 4, 5, 6)$ (mixed parity, 42 differences)

parity-large-49-50: $x = (1, 2, \dots, 49)$, $y = (1, 2, \dots, 50)$ (large asymmetric, 2450 differences)

Extreme asymmetry — tests with very unbalanced sample sizes:

asymmetry-1-100: $x = (50)$, $y = (1, 2, \dots, 100)$ (single vs many, 100 differences)

asymmetry-2-50: $x = (10, 20)$, $y = (1, 2, \dots, 50)$ (tiny vs medium, 100 differences)

asymmetry-constant-varied: $x = (5, 5, 5, 5, 5)$, $y = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ (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: $x = (3, 2, 1)$, $y = (3, 2, 1)$ (both reversed)

unsorted-x-shuffle-3-3: $x = (2, 1, 3)$, $y = (1, 2, 3)$ (X shuffled, Y sorted)

unsorted-y-shuffle-3-3: $x = (1, 2, 3)$, $y = (3, 1, 2)$ (X sorted, Y shuffled)

unsorted-both-shuffle-4-4: $x = (3, 1, 4, 2)$, $y = (4, 2, 1, 3)$ (both shuffled)

unsorted-duplicates-mixed-5-5: $x = (3, 3, 3, 3, 3)$, $y = (3, 3, 3, 3, 3)$ (all identical)

unsorted-x-unsorted-duplicates: $x = (2, 1, 3, 2, 1)$, $y = (1, 1, 2, 2, 3)$ (X has unsorted duplicates)

unsorted-y-unsorted-duplicates: $x = (1, 1, 2, 2, 3)$, $y = (3, 2, 1, 3, 2)$ (Y has unsorted duplicates)

unsorted-asymmetric-unsorted-2-5: $x = (2, 1)$, $y = (5, 2, 4, 1, 3)$ (asymmetric sizes, both unsorted)

unsorted-negative-unsorted-3-3: $x = (-1, -3, -2)$, $y = (-2, -3, -1)$ (negative unsorted)

These tests are critical for two-sample estimators because they verify that x and 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: $x = (1, 2, 3, \dots, 100000)$, $y = (1, 2, 3, \dots, 100000)$

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.

7.5. Ratio Tests

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

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: $x = (1, 2, 4, 8, 16)$, $y = (2, 4, 8, 16, 32)$, expected output: 0.5 (base case, odd $m \times n$)

demo-2: $x = (1, 2, 4, 8, 16)$, $y = (1, 2, 4, 8, 16)$, expected output: 1 (identity property)

demo-3: $x = (2, 4, 8, 16, 32)$, $y = (10, 20, 40, 80, 160)$ ($= [2 \times \text{demo-1.x}, 5 \times \text{demo-1.y}]$), expected output: 0.2 (scale property)

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

natural-1-1: $x = (1)$, $y = (1)$, expected output: 1

natural-1-2: $x = (1)$, $y = (1, 2)$, expected output: ≈ 0.707 ($= \sqrt{0.5}$, geometric interpolation)

natural-1-3: $x = (1)$, $y = (1, 2, 3)$, expected output: 0.5

natural-2-1: $x = (1, 2)$, $y = (1)$, expected output: ≈ 1.414 ($= \sqrt{2}$, geometric interpolation)

natural-2-2: $x = (1, 2)$, $y = (1, 2)$, expected output: 1

natural-2-3: $x = (1, 2)$, $y = (1, 2, 3)$, expected output: ≈ 0.816 (geometric interpolation)

natural-3-1: $x = (1, 2, 3)$, $y = (1)$, expected output: 2

natural-3-2: $x = (1, 2, 3)$, $y = (1, 2)$, expected output: ≈ 1.225 (geometric interpolation)

natural-3-3: $x = (1, 2, 3)$, $y = (1, 2, 3)$, expected output: 1

Additive distribution ($[n, m] \in \{5, 10, 30\} \times \{5, 10, 30\}$) — 9 combinations with 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 o, y uses seed i

Uniform distribution ($[n, m] \in \{5, 100\} \times \{5, 100\}$) — 4 combinations with 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 ($\text{Ratio}(x, x) = 1$) 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: $x = (16, 1, 8, 2, 4)$, $y = (2, 4, 8, 16, 32)$ (demo-1 with X unsorted)
 unsorted-demo-unsorted-y: $x = (1, 2, 4, 8, 16)$, $y = (32, 2, 16, 4, 8)$ (demo-1 with Y unsorted)
 unsorted-demo-both-unsorted: $x = (8, 1, 16, 4, 2)$, $y = (16, 32, 2, 8, 4)$ (demo-1 both unsorted)
 unsorted-identity-unsorted: $x = (4, 1, 8, 2, 16)$, $y = (16, 1, 8, 4, 2)$ (identity property, both unsorted)
 unsorted-asymmetric-unsorted-2-3: $x = (2, 1)$, $y = (3, 1, 2)$ (asymmetric, both unsorted)
 unsorted-power-unsorted-5: $x = (16, 2, 8, 1, 4)$, $y = (32, 4, 16, 2, 8)$ (powers of 2 unsorted)

7.6. AvgSpread Tests

$$\text{AvgSpread}(x, y) = \frac{n \cdot \text{Spread}(x) + m \cdot \text{Spread}(y)}{n + m}$$

The AvgSpread test suite contains 36 test cases (24 original + 12 unsorted). Since AvgSpread computes $\text{Spread}(x)$ and $\text{Spread}(y)$ independently, unsorted tests are critical to verify that both samples are sorted independently before computing their spreads.

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

demo-1: $x = (0, 3, 6, 9, 12)$, $y = (0, 2, 4, 6, 8)$, expected output: 5 (base case: $(5 \cdot 6 + 5 \cdot 4)/10$)

demo-2: $x = (0, 3, 6, 9, 12)$, $y = (0, 3, 6, 9, 12)$, expected output: 6 (identity case)

demo-3: $x = (0, 6, 12, 18, 24)$, $y = (0, 9, 18, 27, 36)$ ($= [2 \times \text{demo-1.x}, 3 \times \text{demo-1.y}]$), expected output: 15 (scale equivariance)

demo-4: $x = (0, 2, 4, 6, 8)$, $y = (0, 3, 6, 9, 12)$ (= reversed demo-1), expected output: 5 (symmetry)

demo-5: $x = (0, 6, 12, 18, 24)$, $y = (0, 4, 8, 12, 16)$ ($= 2 \times \text{demo-1}$), expected output: 10 (uniform scaling)

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

All combinations for two- and three-element samples, validating the weighted average calculation

Negative values ($[n, m] = [2, 2]$) — validates spread calculation with negative values:

negative-2-2: $x = (-2, -1)$, $y = (-2, -1)$, expected output: 1

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

Tests pooled dispersion across different sample size combinations

Random generation: x uses seed 0, y uses seed 1

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

Validates correct weighting when sample sizes differ substantially

Random generation: x uses seed 2, y uses seed 3

The asymmetric size combinations are particularly important for AvgSpread because the estimator must correctly weight each sample's contribution by its size.

Composite estimator stress tests — edge cases for weighted averaging:

composite-asymmetric-weights: $x = (1, 2)$, $y = (3, 4, 5, 6, 7, 8, 9, 10)$ (2 vs 8, tests weighting formula)

Unsorted tests — critical for verifying independent 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: $x = (12, 0, 6, 3, 9)$, $y = (0, 2, 4, 6, 8)$ (demo-1 with X unsorted)

unsorted-demo-unsorted-y: $x = (0, 3, 6, 9, 12)$, $y = (8, 0, 4, 2, 6)$ (demo-1 with Y unsorted)

unsorted-demo-both-unsorted: $x = (9, 0, 12, 3, 6)$, $y = (6, 0, 8, 2, 4)$ (demo-1 both unsorted)

unsorted-identity-unsorted: $x = (6, 0, 12, 3, 9)$, $y = (9, 0, 12, 6, 3)$ (demo-2 unsorted)

unsorted-negative-unsorted: $x = (-1, -2)$, $y = (-1, -2)$ (negative unsorted)

unsorted-asymmetric-weights-unsorted: $x = (2, 1)$, $y = (8, 3, 6, 4, 10, 5, 9, 7)$ (asymmetric unsorted)

These tests verify that implementations compute $\text{Spread}(x)$ and $\text{Spread}(y)$ with properly sorted samples.

7.7. Disparity Tests

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

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: $x = (0, 3, 6, 9, 12)$, $y = (0, 2, 4, 6, 8)$, expected output: 0.4 (base case: 2/5)

demo-2: $x = (5, 8, 11, 14, 17)$, $y = (5, 7, 9, 11, 13)$ (= demo-1 + 5), expected output: 0.4 (location invariance)

demo-3: $x = (0, 6, 12, 18, 24)$, $y = (0, 4, 8, 12, 16)$ ($= 2 \times$ demo-1), expected output: 0.4 (scale invariance)

demo-4: $x = (0, 2, 4, 6, 8)$, $y = (0, 3, 6, 9, 12)$ ($=$ 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: $x = (-2, -1)$, $y = (-2, -1)$, expected output: 0

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

uniform-5-5, uniform-5-100, uniform-100-5, uniform-100-100

Random generation: x uses seed o, y uses seed i

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: $x = (10.001, 10.002, 10.003)$, $y = (10.004, 10.005, 10.006)$ (tiny spread, large shift)

composite-large-avgspread: $x = (1, 100, 200)$, $y = (50, 150, 250)$ (large spread, small shift)

composite-extreme-disparity: $x = (1, 1.001)$, $y = (100, 100.001)$ (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: $x = (12, 0, 6, 3, 9)$, $y = (0, 2, 4, 6, 8)$ (demo-1 with X unsorted)

unsorted-demo-unsorted-y: $x = (0, 3, 6, 9, 12)$, $y = (8, 0, 4, 2, 6)$ (demo-1 with Y unsorted)

unsorted-demo-both-unsorted: $x = (9, 0, 12, 3, 6)$, $y = (6, 0, 8, 2, 4)$ (demo-1 both unsorted)

unsorted-location-invariance-unsorted: $x = (17, 5, 11, 8, 14)$, $y = (13, 5, 9, 7, 11)$ (demo-2 unsorted)

unsorted-scale-invariance-unsorted: $x = (24, 0, 12, 6, 18)$, $y = (16, 0, 8, 4, 12)$ (demo-3 unsorted)

unsorted-anti-symmetry-unsorted: $x = (8, 0, 4, 2, 6)$, $y = (12, 0, 6, 3, 9)$ (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.

7.8. PairwiseMargin Tests

`PairwiseMargin(n, m, misrate)`

The `PairwiseMargin` test suite contains 178 test cases (4 demo + 4 natural + 10 edge + 12 small grid + 148 large grid). The domain constraint $\text{misrate} \geq \frac{2}{\binom{n+m}{n}}$ 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, misrate = 10-6, expected output: 276
demo-2: n = 30, m = 30, misrate = 10-5, expected output: 328
demo-3: n = 30, m = 30, misrate = 10-4, expected output: 390
demo-4: n = 30, m = 30, 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: $\text{misrate} \in \{10^{-1}, 10^{-2}\}$
 After filtering by $\text{misrate} \geq \frac{2}{\binom{n+m}{n}}$, only 4 combinations survive:

- natural-3-3-mr1: $n = 3, m = 3, \text{misrate} = 0.1$, expected output: 0
- natural-3-4-mr1: $n = 3, m = 4, \text{misrate} = 0.1$
- natural-4-3-mr1: $n = 4, m = 3, \text{misrate} = 0.1$
- natural-4-4-mr1: $n = 4, m = 4, \text{misrate} = 0.1$

Edge cases — boundary condition validation (10 tests):

```
boundary-min: n = 1, m = 1, misrate = 1.0 (minimum samples with maximum misrate, expected output: 0)
boundary-zero-margin-small: n = 20, m = 20, misrate = 10-6 (strict misrate with sufficient samples)
boundary-loose: n = 5, m = 5, misrate = 0.9 (very permissive misrate)
symmetry-2-5: n = 2, m = 5, misrate = 0.1 (tests symmetry property)
symmetry-5-2: n = 5, m = 2, misrate = 0.1 (symmetric counterpart, same output as above)
symmetry-3-7: n = 3, m = 7, misrate = 0.05 (asymmetric sizes)
symmetry-7-3: n = 7, m = 3, misrate = 0.05 (symmetric counterpart)
asymmetry-extreme-1-100: n = 1, m = 100, misrate = 0.1 (extreme size difference)
asymmetry-extreme-100-1: n = 100, m = 1, misrate = 0.1 (reversed extreme)
asymmetry-extreme-2-50: n = 2, m = 50, misrate = 0.05 (highly unbalanced)
```

These edge cases validate correct handling of boundary conditions, the symmetry property $\text{PairwiseMargin}(n, m, \text{misrate}) = \text{PairwiseMargin}(m, n, \text{misrate})$, 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: misrate $\in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$

Combinations where 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`: $n = 5, m = 5$, misrate = 0.1, expected output: 10

`n5_m5_mr2`: $n = 5, m = 5$, 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 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`: $n = 10, m = 10$, misrate = 0.1, expected output: 56

`n50_m50_r3`: $n = 50, m = 50$, misrate = 0.001, expected output: 1556

`n100_m100_r6`: $n = 100, m = 100$, 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.

7.9. SignedRankMargin Tests

`SignedRankMargin(n, misrate)`

The 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`: $n = 30$, misrate = 10^{-6} , expected output: 46

`demo-2`: $n = 30$, misrate = 10^{-5} , expected output: 74

`demo-3`: $n = 30$, misrate = 10^{-4} , expected output: 112

`demo-4`: $n = 30$, misrate = 10^{-3} , expected output: 158

These demo cases match the reference values used throughout the manual to illustrate CenterBounds construction.

Boundary cases — minimum achievable misrate validation:

`boundary-n2-min`: $n = 2$, misrate = 0.5 (minimum misrate for $n = 2$, expected output: 0)

`boundary-n3-min`: $n = 3$, misrate = 0.25 (minimum misrate for $n = 3$)

`boundary-n4-min`: $n = 4$, misrate = 0.125 (minimum misrate for $n = 4$)

`boundary-loose`: $n = 5$, misrate = 0.5 (permissive misrate)

`boundary-tight`: $n = 10$, misrate = 0.01 (strict misrate)

`boundary-very-tight`: $n = 20$, 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, misrate = 0.1
exact-n6-mr1e1: n = 6, misrate = 0.1
exact-n6-mr5e2: n = 6, misrate = 0.05
exact-n10-mr1e1: n = 10, misrate = 0.1, expected output: 22
exact-n10-mr1e2: n = 10, misrate = 0.01
exact-n10-mr5e2: n = 10, misrate = 0.05
exact-n10-mr5e3: n = 10, 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: 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, misrate = 0.1
medium-n30-mr1e2: n = 30, misrate = 0.01, expected output: 220
medium-n50-mr1e3: n = 50, misrate = 0.001
medium-n100-mr1e4: n = 100, 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, misrate = 0.5 (invalid: misrate below minimum achievable  $2^{1-1} = 1.0$ )
error-n0: n = 0, misrate = 0.05 (invalid: n must be positive)
```

This error case verifies that implementations correctly reject $n = 1$ with misrate = 0.5 as invalid input, since the minimum achievable misrate for $n = 1$ is $2^0 = 1.0$.

7.10. ShiftBounds Tests

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

where

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

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

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

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 $\text{misrate} \geq \frac{2}{(n+m)}$ is enforced; inputs violating this return a domain error.

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

```
demo-1: x = (1, 2, 3, 4, 5), y = (3, 4, 5, 6, 7), misrate = 0.05, expected output: [-4, 0]
demo-2: x = (1, 2, 3, 4, 5), y = (3, 4, 5, 6, 7), misrate = 0.01, expected output: [-5, 1]
demo-3: x = (3, 4, 5, 6, 7), y = (3, 4, 5, 6, 7), 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: x = (1, ..., 5), y = (1, ..., 5), expected bounds containing 0
natural-5-8: x = (1, ..., 5), y = (1, ..., 8)
natural-5-10: x = (1, ..., 5), y = (1, ..., 10)
natural-8-5: x = (1, ..., 8), y = (1, ..., 5)
natural-8-8: x = (1, ..., 8), y = (1, ..., 8), expected bounds containing 0
natural-8-10: x = (1, ..., 8), y = (1, ..., 10)
natural-10-5: x = (1, ..., 10), y = (1, ..., 5)
natural-10-8: x = (1, ..., 10), y = (1, ..., 8)
natural-10-10: x = (1, ..., 10), y = (1, ..., 10), expected bounds containing 0
```

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

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

```
property-identity: x = (0, 2, 4, ..., 18), y = (0, 2, 4, ..., 18), bounds must contain 0
property-location-shift: x = (7, 9, 11, ..., 25), y = (13, 15, 17, ..., 31)
    Must produce same bounds as base case (location invariance)
property-scale-2x: x = (2, 4, 6, ..., 20), y = (6, 8, 10, ..., 24)
    Bounds must be 2× the base case bounds (scale equivariance)
property-antisymmetry: x = (3, 4, ..., 12), y = (1, 2, ..., 10)
    Bounds must be negated: if original is  $[a, b]$ , this yields  $[-b, -a]$ 
property-negative: x = (-10, -9, ..., -1), y = (-12, -11, ..., -3)
    Validates sign handling with all negative values
property-mixed-signs: x = (-4, -3, ..., 5), y = (-3, -2, ..., 6)
    Validates bounds crossing zero with mixed-sign samples
```

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

```

edge-min-samples: x = (1, 2, 3, 4, 5), y = (6, 7, 8, 9, 10), misrate = 0.05
edge-permissive-misrate: x = (1, 2, 3, 4, 5), y = (3, 4, 5, 6, 7), misrate = 0.5 (very wide bounds)
edge-strict-misrate: n = m = 20, misrate = 10-6 (very narrow bounds)
edge-zero-shift: n = m = 10, all values = 5, misrate = 10-3 (bounds around 0)
edge-asymmetric-3-100: n = 3, m = 100, misrate = 10-2 (extreme size difference)
edge-asymmetric-5-50: n = 5, m = 50, misrate = 10-3 (highly unbalanced)
edge-duplicates: x = (3, 3, 3, 3, 3), y = (5, 5, 5, 5, 5), misrate = 10-2 (all duplicates, bounds around -2)
edge-wide-range: n = m = 10, values spanning 10-3 to 108, misrate = 10-3 (extreme value range)
edge-tiny-values: n = m = 10, values ≈ 10-8, misrate = 10-3 (numerical precision)
edge-large-values: n = m = 10, values ≈ 108, misrate = 10-3 (large magnitude)

```

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

Additive distribution ($[n, m] \in \{10, 30, 50\} \times \{10, 30, 50\}$, misrate = 10^{-3}) — 9 combinations with 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: x uses seed 0, 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 ($[n, m] \in \{10, 100\} \times \{10, 100\}$, misrate = 10^{-4}) — 4 combinations with Uniform(0, 1):

```

uniform-10-10, uniform-10-100, uniform-100-10, uniform-100-100
Random generation: x uses seed 2, y uses seed 3

```

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

Misrate variation ($n = m = 20$, $x = (0, 2, 4, \dots, 38)$, $y = (10, 12, 14, \dots, 48)$) — 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 x and y (15 tests):

```

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

7.11. RatioBounds Tests

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

The RatioBounds test suite contains 61 correctness test cases (3 demo + 9 natural + 6 property + 10 edge + 9 multiplicative + 4 uniform + 5 misrate + 15 unsorted). Since RatioBounds returns bounds rather than a point estimate, tests validate that the bounds contain $\text{Ratio}(x, y)$ 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 $\text{misrate} \geq \frac{2}{(n+m)}$ is enforced; inputs violating this return a domain error.

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

demo-1: $x = (1, 2, 3, 4, 5)$, $y = (2, 3, 4, 5, 6)$, misrate = 0.05
 demo-2: $x = (1, 2, 3, 4, 5)$, $y = (2, 3, 4, 5, 6)$, misrate = 0.01, expected: wider bounds than demo-1
 demo-3: $x = (2, 3, 4, 5, 6)$, $y = (2, 3, 4, 5, 6)$, 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 ($[n, m] \in \{5, 8, 10\} \times \{5, 8, 10\}$, misrate = 10^{-2}) — 9 combinations:

natural-5-5: $x = (1, \dots, 5)$, $y = (1, \dots, 5)$, expected bounds containing 1
 natural-5-8: $x = (1, \dots, 5)$, $y = (1, \dots, 8)$
 natural-5-10: $x = (1, \dots, 5)$, $y = (1, \dots, 10)$
 natural-8-5: $x = (1, \dots, 8)$, $y = (1, \dots, 5)$
 natural-8-8: $x = (1, \dots, 8)$, $y = (1, \dots, 8)$, expected bounds containing 1
 natural-8-10: $x = (1, \dots, 8)$, $y = (1, \dots, 10)$
 natural-10-5: $x = (1, \dots, 10)$, $y = (1, \dots, 5)$
 natural-10-8: $x = (1, \dots, 10)$, $y = (1, \dots, 8)$
 natural-10-10: $x = (1, \dots, 10)$, $y = (1, \dots, 10)$, expected bounds containing 1

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

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

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

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

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

edge-tiny-values: $n = m = 10$, values $\approx 10^{-6}$, misrate = 10^{-3} (numerical precision)
 edge-large-values: $n = m = 10$, values $\approx 10^8$, misrate = 10^{-3} (large magnitude)

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

Multiplic distribution ($[n, m] \in \{10, 30, 50\} \times \{10, 30, 50\}$, misrate = 10^{-3}) — 9 combinations with 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: x uses seed o, y uses seed i
```

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

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

```
uniform-10-10, uniform-10-100, uniform-100-10, uniform-100-100
Random generation: x uses seed 2, 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 ($n = m = 20$, $x = (1, 2, \dots, 20)$, $y = (2, 4, \dots, 40)$) — 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 x and y (15 tests):

```
unsorted-x-natural-5-5: x = (5, 3, 1, 4, 2), y = (1, 2, 3, 4, 5), misrate = 10-2 (X reversed, Y sorted)
unsorted-y-natural-5-5: x = (1, 2, 3, 4, 5), y = (5, 3, 1, 4, 2), misrate = 10-2 (X sorted, Y reversed)
unsorted-both-natural-5-5: x = (5, 3, 1, 4, 2), y = (5, 3, 1, 4, 2), misrate = 10-2 (both reversed)
unsorted-x-shuffle-5-5: x = (3, 1, 5, 4, 2), y = (1, 2, 3, 4, 5), misrate = 10-2 (X shuffled)
unsorted-y-shuffle-5-5: x = (1, 2, 3, 4, 5), y = (4, 2, 5, 1, 3), misrate = 10-2 (Y shuffled)
unsorted-both-shuffle-5-5: x = (3, 1, 5, 4, 2), y = (2, 4, 1, 5, 3), misrate = 10-2 (both shuffled)
unsorted-demo-unsorted-x: x = (5, 1, 4, 2, 3), y = (2, 3, 4, 5, 6), misrate = 0.05 (demo-1 X unsorted)
```

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

7.12. CenterBounds Tests

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

where $\mathbf{w} = \{(x_i + x_j)/2\}$ (pairwise averages, sorted) for $i \leq j$, $k_{\text{left}} = \lfloor \text{SignedRankMargin}/2 \rfloor + 1$, $k_{\text{right}} = N - \lfloor \text{SignedRankMargin}/2 \rfloor$, 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 Center(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: x = (1, 2, 3, 4, 5), misrate = 0.1, expected output: [1.5, 4.5]
demo-2: x = (1, ..., 10), misrate = 0.01, expected output: [2.5, 8.5]
demo-3: x = (0, 2, 4, 6, 8), misrate = 0.1
  
```

These cases illustrate how tighter misrates produce wider bounds.

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

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

Property validation ($n = 5$, misrate = 0.05) — 5 tests:

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

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

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

Symmetric distributions (misrate = 0.05) — 4 tests with symmetric data:

symmetric-5: $x = (-2, -1, 0, 1, 2)$, bounds centered around 0
 symmetric-7: $x = (-3, -2, -1, 0, 1, 2, 3)$, 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 ($n = 5$, misrate = 0.1) — 4 tests validating bounds with asymmetric data:

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

These tests validate that CenterBounds handles asymmetric data correctly, complementing the symmetric test cases.

Additive distribution (misrate = 0.01) — 2 tests with Additive(10, 1):

additive-10: $n = 10$, seed o
 additive-20: $n = 20$, seed o

Uniform distribution (misrate = 0.01) — 2 tests with Uniform(0, 1):

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

Misrate variation ($x = (1, \dots, 10)$) — 4 tests with varying misrates:

```
misrate-1e-1: misrate = 0.1
misrate-5e-2: misrate = 0.05
misrate-1e-2: misrate = 0.01
misrate-5e-3: misrate = 0.005
```

These tests validate monotonicity: smaller misrates produce wider bounds.

Unsorted tests — verify sorting independence (6 tests):

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

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

Error cases — 2 tests validating input validation:

```
error-single-element: x = (1), misrate = 0.5 (minimum sample size violation)
error-invalid-misrate: x = (1, 2, 3, 4, 5), misrate = 0.001 (misrate below minimum achievable)
```

7.13. 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 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 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.

8. Methodology

This chapter examines the methodological principles that guide Pragmastat's design and application.

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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	$ x_i - x_j $	Median of pairwise differences
Shift	$x_i - y_j$	Median of cross-sample differences
Ratio	$\log(x_i) - \log(y_j)$	$\exp(\text{median of log-differences})$
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: $\text{Ratio}(\mathbf{x}, \mathbf{y}) \times \text{Ratio}(\mathbf{y}, \mathbf{x}) = 1$.

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.

8.5. 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.

8.6. 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 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.

8.7. 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), Transform = log and 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.

8.8. 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.

8.9. 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?”

8.10. 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.

8.11. Reference Tests as Specification

The toolkit maintains seven implementations across different programming languages: Python, Type-Script, 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.

8.12. 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 — `Rng(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

8.13. 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

8.14. 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 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	misrate_{\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%	misrate = 10^{-6} is achievable

Practical implications

$n < 5$: Cannot achieve 95% confidence (misrate = 0.05)

$n < 7$: Cannot achieve 99% confidence (misrate = 0.01)

$n \geq 20$: 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.

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.
- 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. <https://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>