Pragmastat: Pragmatic Statistical Toolkit

Andrey Akinshin
andrey.akinshin@gmail.com

Version 4.0.1
DOI: 10.5281/zenodo.17236778

Abstract

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. Written for software developers, mathematicians, and LLMs.

Contents

1 Introduction

1.1 Primer

Given two numeric samples $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_m)$, the toolkit provides the following primary procedures:

$\text{Center}(\mathbf{x}) = \underset{1 \le i \le j \le n}{\text{Median}} \left((x_i + x_j)/2 \right)$ — robust average of $\mathbf{x}$

For $\mathbf{x} = (0, 2, 4, 6, 8)$:

$$\begin{aligned} \text{Center}(\mathbf{x}) &= 4 \\ \text{Center}(\mathbf{x} + 10) &= 14 \\ \text{Center}(3\mathbf{x}) &= 12 \end{aligned}$$

$\text{Spread}(\mathbf{x}) = \underset{1 \le i < j \le n}{\text{Median}} |x_i - x_j|$ — robust dispersion of $\mathbf{x}$

For $\mathbf{x} = (0, 2, 4, 6, 8)$:

$$\begin{aligned} \text{Spread}(\mathbf{x}) &= 4 \\ \text{Spread}(\mathbf{x} + 10) &= 4 \\ \text{Spread}(2\mathbf{x}) &= 8 \end{aligned}$$

$\text{RelSpread}(\mathbf{x}) = \text{Spread}(\mathbf{x}) / |\text{Center}(\mathbf{x})|$ — robust relative dispersion of $\mathbf{x}$

For $\mathbf{x} = (0, 2, 4, 6, 8)$:

$$\begin{aligned} \text{RelSpread}(\mathbf{x}) &= 1 \\ \text{RelSpread}(5\mathbf{x}) &= 1 \end{aligned}$$

$\text{Shift}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n, 1 \le j \le m}{\text{Median}} \left(x_i - y_j \right)$ — robust signed difference $(\mathbf{x} - \mathbf{y})$

For $\mathbf{x} = (0, 2, 4, 6, 8)$ and $\mathbf{y} = (10, 12, 14, 16, 18)$:

$$\begin{aligned} \text{Shift}(\mathbf{x}, \mathbf{y}) &= -10 \\ \text{Shift}(\mathbf{x}, \mathbf{x}) &= 0 \\ \text{Shift}(\mathbf{x} + 7, \mathbf{y} + 3) &= -6 \\ \text{Shift}(2\mathbf{x}, 2\mathbf{y}) &= -20 \\ \text{Shift}(\mathbf{y}, \mathbf{x}) &= 10 \end{aligned}$$

$\text{Ratio}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n, 1 \le j \le m}{\text{Median}} \left(x_i / y_j \right)$ — robust ratio $(\mathbf{x}/\mathbf{y})$

For $\mathbf{x} = (1, 2, 4, 8, 16)$ and $\mathbf{y} = (2, 4, 8, 16, 32)$:

$$\begin{aligned} \text{Ratio}(\mathbf{x}, \mathbf{y}) &= 0.5 \\ \text{Ratio}(\mathbf{x}, \mathbf{x}) &= 1 \\ \text{Ratio}(2\mathbf{x}, 5\mathbf{y}) &= 0.2 \end{aligned}$$

$\text{AvgSpread}(\mathbf{x}, \mathbf{y}) = (n\,\text{Spread}(\mathbf{x}) + m\,\text{Spread}(\mathbf{y}))/(n+m)$ — robust average spread of $\mathbf{x}$ and $\mathbf{y}$

For $\mathbf{x} = (0, 3, 6, 9, 12)$ and $\mathbf{y} = (0, 2, 4, 6, 8)$:

$$\begin{aligned}
\text{Spread}(\mathbf{x}) &= 6 \\
\text{Spread}(\mathbf{y}) &= 4 \\
\text{AvgSpread}(\mathbf{x}, \mathbf{y}) &= 5 \\
\text{AvgSpread}(\mathbf{x}, \mathbf{x}) &= 6 \\
\text{AvgSpread}(2\mathbf{x}, 3\mathbf{x}) &= 15 \\
\text{AvgSpread}(\mathbf{y}, \mathbf{x}) &= 5 \\
\text{AvgSpread}(2\mathbf{x}, 2\mathbf{y}) &= 10
\end{aligned}$$

$\text{Disparity}(\mathbf{x}, \mathbf{y}) = \text{Shift}(\mathbf{x}, \mathbf{y})/\,\text{AvgSpread}(\mathbf{x}, \mathbf{y})$ — robust effect size between $\mathbf{x}$ and $\mathbf{y}$

For $\mathbf{x} = (0, 3, 6, 9, 12)$ and $\mathbf{y} = (0, 2, 4, 6, 8)$:

$$\begin{aligned}
\text{Shift}(\mathbf{x}, \mathbf{y}) &= 2 \\
\text{AvgSpread}(\mathbf{x}, \mathbf{y}) &= 5 \\
\text{Disparity}(\mathbf{x}, \mathbf{y}) &= 0.4 \\
\text{Disparity}(\mathbf{x} + 5, \mathbf{y} + 5) &= 0.4 \\
\text{Disparity}(2\mathbf{x}, 2\mathbf{y}) &= 0.4 \\
\text{Disparity}(\mathbf{y}, \mathbf{x}) &= -0.4
\end{aligned}$$

$\text{PairwiseMargin}(n, m, \text{misrate})$ — determines how many extreme pairwise differences to exclude when constructing bounds based on the distribution of dominance statistics

For $n = 30, m = 30$:

$$\begin{aligned}
\text{PairwiseMargin}(30, 30, 10^{-6}) &= 276 \\
\text{PairwiseMargin}(30, 30, 10^{-5}) &= 328 \\
\text{PairwiseMargin}(30, 30, 10^{-4}) &= 390 \\
\text{PairwiseMargin}(30, 30, 10^{-3}) &= 464
\end{aligned}$$

$\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$ — bounds on $\text{Shift}(\mathbf{x}, \mathbf{y})$ with specified misrate; these bounds fail to cover the true value of shift in misrate probability in the long run

For $\mathbf{x} = (1, 2, \ldots, 30)$ and $\mathbf{y} = (21, 22, \ldots, 50)$:

$$\begin{aligned}
\text{Shift}(\mathbf{x}, \mathbf{y}) &= -20 \\
\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, 10^{-6}) &= [-33, -7] \\
\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, 10^{-5}) &= [-32, -8] \\
\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, 10^{-4}) &= [-30, -10] \\
\text{ShiftBounds}(\mathbf{x}, \mathbf{y}, 10^{-3}) &= [-28, -12]
\end{aligned}$$

These procedures are designed to serve as default choices for routine analysis and comparison tasks in engineering contexts. The toolkit has ready-to-use implementations for Python, TypeScript/JavaScript, R, C#, Kotlin, Rust, and Go.

1.2 Breaking changes

Statistical practice has evolved through decades of research and teaching, creating a system where historical naming conventions became embedded in textbooks and standard practice. Traditional statistics often names procedures after their discoverers or uses arbitrary symbols that reveal nothing about their actual purpose or application context. This approach forces practitioners to memorize meaningless mappings between historical figures and mathematical concepts.

The result is unnecessary friction for anyone learning or applying statistical methods. Beginners face an inconsistent landscape of confusing names, fragile defaults, and incompatible approaches with little guidance on selection or interpretation. Modern practitioners would benefit from a more consistent system, which requires some renaming and redefining. This manual offers a coherent system designed for clarity and practical use, breaking from tradition. The following concepts were adopted from traditional textbooks via renaming or reworking:

- Estimators
  - Average: Center (former 'Hodges-Lehmann location estimator')
  - Dispersion: Spread (former 'Shamos scale estimator')
  - Effect Size: Disparity (a robust alternative to 'Cohen's d')
- Estimator properties
  - Precision: Drift (a robust alternative to statistical efficiency)
- Distributions
  - Additive (former 'Normal' or 'Gaussian')
  - Multiplic (former 'Log-Normal' or 'Galton')
  - Power (former 'Pareto')
- Terms
  - Bounds (former 'Confidence Interval')
  - Misrate (former 1 - 'Confidence Level')

1.3 Definitions

- X, Y: random variables, can be treated as generators of random real measurements
  - $X \sim$ Distribution defines a distribution from which this variable comes
- x_i, y_j: specific individual measurements
- $\mathbf{x} = (x_1, x_2, \dots, x_n)$, $\mathbf{y} = (y_1, y_2, \dots, y_m)$: samples of measurements of a given size
  - Samples are non-empty: $n, m \geq 1$
- $x_{(1)}, x_{(2)}, \dots, x_{(n)}$: sorted measurements of the sample ('order statistics')
- Asymptotic case: the sample size goes to infinity $n, m \to \infty$
  - Can typically be treated as an approximation for large samples
- Estimator($\mathbf{x}$): a function that estimates the property of a distribution from given measurements
  - Estimator[X] shows the true property value of the distribution (asymptotic value)
- Median: an estimator that finds the value splitting the distribution into two equal parts

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

2 Summary Estimators

The following sections introduce definitions of one-sample and two-sample summary estimators. Later sections will evaluate properties of these estimators and applicability to different conditions.

2.1 Center

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

- Measures average (central tendency, measure of location)
- Equals the *Hodges-Lehmann estimator* ((Hodges and Lehmann 1963), (Sen 1963)), renamed to Center for clarity
- Also known as 'pseudomedian' because it is consistent with Median for symmetric distributions
- Pragmatic alternative to Mean and Median
- Asymptotically, Center$[X]$ is the Median of the arithmetic average of two random measurements from X
- Straightforward implementations have $O(n^2 \log n)$ complexity; a fast $O(n \log n)$ version is provided in the Algorithms section.
- Domain: any real numbers
- Unit: the same as measurements

$$\text{Center}(\mathbf{x} + k) = \text{Center}(\mathbf{x}) + k$$

$$\text{Center}(k \cdot \mathbf{x}) = k \cdot \text{Center}(\mathbf{x})$$

2.2 Spread

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

- Measures dispersion (variability, scatter)
- Corner case: for $n = 1$, Spread$(\mathbf{x}) = 0$
- Equals the *Shamos scale estimator* ((Shamos 1976)), renamed to Spread for clarity
- Pragmatic alternative to the standard deviation and the median absolute deviation
- Asymptotically, Spread$[X]$ is the median of the absolute difference between two random measurements from distribution X
- Straightforward implementations have $O(n^2 \log n)$ complexity; a fast $O(n \log n)$ version is provided in the Algorithms section.
- Domain: any real numbers
- Unit: the same as measurements

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

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

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

2.3 RelSpread

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

- Measures the relative dispersion of a sample to Center($\mathbf{x}$)
- Pragmatic alternative to the *coefficient of variation*
- Domain: Center($\mathbf{x}$) $\neq 0$
- Unit: relative

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

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

2.4 Shift

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

- Measures the median of pairwise differences between elements of two samples
- Equals the *Hodges-Lehmann estimator* for two samples ((Hodges and Lehmann 1963))
- Asymptotically, Shift$[X, Y]$ is the median of the difference of random measurements from X and Y
- Straightforward implementations have $O(mn \log(mn))$ complexity; a fast $O((m+n) \log L)$ version is provided in the Algorithms section.
- Domain: any real numbers
- Unit: the same as measurements

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

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

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

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

2.5 Ratio

$$\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n, 1 \le j \le m}{\mathrm{Median}} \left(\frac{x_i}{y_j} \right)$$

- Measures the median of pairwise ratios between elements of two samples
- Asymptotically, $\mathrm{Ratio}[X, Y]$ is the median of the ratio of random measurements from X and Y
- Note: $\mathrm{Ratio}(\mathbf{x}, \mathbf{y}) \neq 1/\mathrm{Ratio}(\mathbf{y}, \mathbf{x})$ in general (example: $x = (1, 100)$, $y = (1, 10)$)
- Practical Domain: $x_i, y_j > 0$ or $x_i, y_j < 0$. In practice, exclude values with $|y_j|$ near zero.
- Unit: relative

$$\mathrm{Ratio}(\mathbf{x}, \mathbf{x}) = 1$$

$$\mathrm{Ratio}(k_x \cdot \mathbf{x}, k_y \cdot \mathbf{y}) = \frac{k_x}{k_y} \cdot \mathrm{Ratio}(\mathbf{x}, \mathbf{y})$$

2.6 AvgSpread

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

- Measures average dispersion across two samples
- Pragmatic alternative to the 'pooled standard deviation'
- Note: $\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) \neq \mathrm{Spread}(\mathbf{x} \cup \mathbf{y})$ in general (defines a pooled scale, not the spread of the concatenated sample)
- Domain: any real numbers
- Unit: the same as measurements

$$\mathrm{AvgSpread}(\mathbf{x}, \mathbf{x}) = \mathrm{Spread}(\mathbf{x})$$

$$\mathrm{AvgSpread}(k_1 \cdot \mathbf{x}, k_2 \cdot \mathbf{x}) = \frac{|k_1| + |k_2|}{2} \cdot \mathrm{Spread}(\mathbf{x})$$

$$\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) = \mathrm{AvgSpread}(\mathbf{y}, \mathbf{x})$$

$$\mathrm{AvgSpread}(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = |k| \cdot \mathrm{AvgSpread}(\mathbf{x}, \mathbf{y})$$

2.7 Disparity ('robust effect size')

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

- Measures a normalized Shift between $\mathbf{x}$ and $\mathbf{y}$ expressed in spread units
- Expresses the 'effect size', renamed to Disparity for clarity
- Pragmatic alternative to Cohen's d (note: exact estimates differ due to robust construction)
- Domain: $\mathrm{AvgSpread}(\mathbf{x}, \mathbf{y}) > 0$

- Unit: spread unit

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

$$\text{Disparity}(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = \text{sign}(k) \cdot \text{Disparity}(\mathbf{x}, \mathbf{y})$$

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

2.8 PairwiseMargin

$$\text{PairwiseMargin}(n, m, \text{misrate})$$

- Determines how many extreme pairwise differences to exclude when constructing bounds
- Based on the 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 the total margin split evenly between lower and upper tails
- Used by ShiftBounds to select appropriate order statistics
- Can be computed exactly for small samples or approximated for large samples (see Algorithms section)
- Domain: $n, m \geq 1$, misrate $\in (0; 1)$
- Unit: count (number of pairwise differences)

$$\text{PairwiseMargin}(n, m, \text{misrate}) = \text{PairwiseMargin}(m, n, \text{misrate})$$

$$\text{PairwiseMargin}(n, m, \text{misrate}) \geq 0$$

$$\text{PairwiseMargin}(n, m, \text{misrate}) \leq nm$$

2.9 ShiftBounds

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

where

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

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

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

- Provides bounds on $\text{Shift}(\mathbf{x}, \mathbf{y})$ with specified misrate
- The misrate represents the probability that the true shift falls outside the computed bounds

- Pragmatic alternative to traditional confidence intervals for the Hodges-Lehmann estimator
- Domain: any real numbers
- Unit: the same as measurements

$$\text{ShiftBounds}(\mathbf{x} + k, \mathbf{y} + k, \text{misrate}) = \text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$$

$$\text{ShiftBounds}(k \cdot \mathbf{x}, k \cdot \mathbf{y}, \text{misrate}) = k \cdot \text{ShiftBounds}(\mathbf{x}, \mathbf{y}, \text{misrate})$$

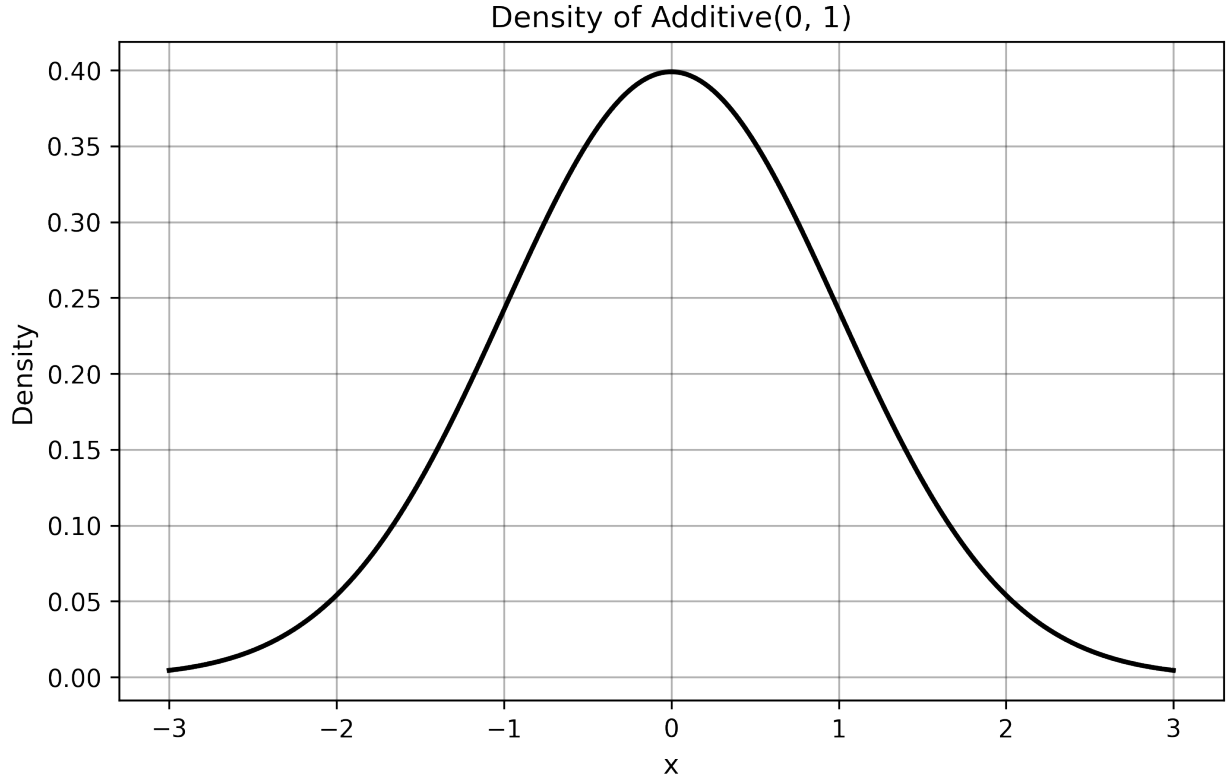
3 Distributions

This section defines the distributions used throughout the manual.

3.1 Additive ('Normal')

$$\underline{\text{Additive}}(\text{mean}, \text{stdDev})$$

- mean: location parameter (center of the distribution), consistent with Center
- stdDev: scale parameter (standard deviation), can be rescaled to Spread



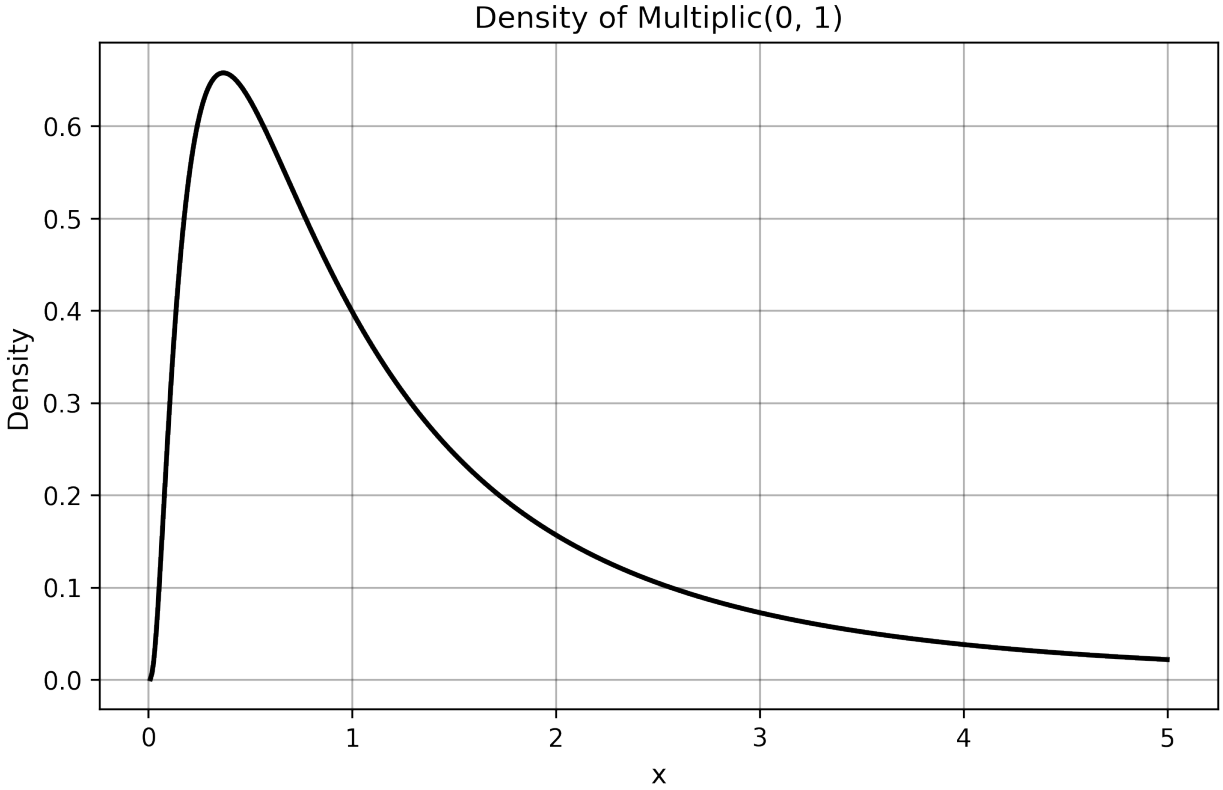
- **Formation:** the sum of many variables $X_1 + X_2 + ... + X_n$ under mild CLT (Central Limit Theorem) conditions (e.g., Lindeberg-Feller).
- **Origin:** historically called 'Normal' or 'Gaussian' distribution after Carl Friedrich Gauss and others.

- **Rename Motivation:** renamed to Additive to reflect its formation mechanism through addition.
- **Properties:** symmetric, bell-shaped, characterized by central limit theorem convergence.
- **Applications:** measurement errors, heights and weights in populations, test scores, temperature variations.
- **Characteristics:** symmetric around the mean, light tails, finite variance.
- **Caution:** no perfectly additive distributions exist in real data; all real-world measurements contain some deviations. Traditional estimators like Mean and StdDev lack robustness to outliers; use them only when strong evidence supports small deviations from additivity with no extreme measurements.

3.2 Multiplic ('LogNormal')

$$\text{Multiplic}(\text{logMean}, \text{logStdDev})$$

- logMean: mean of log values (location parameter; e^{logMean} equals the geometric mean)
- logStdDev: standard deviation of log values (scale parameter; controls multiplicative spread)



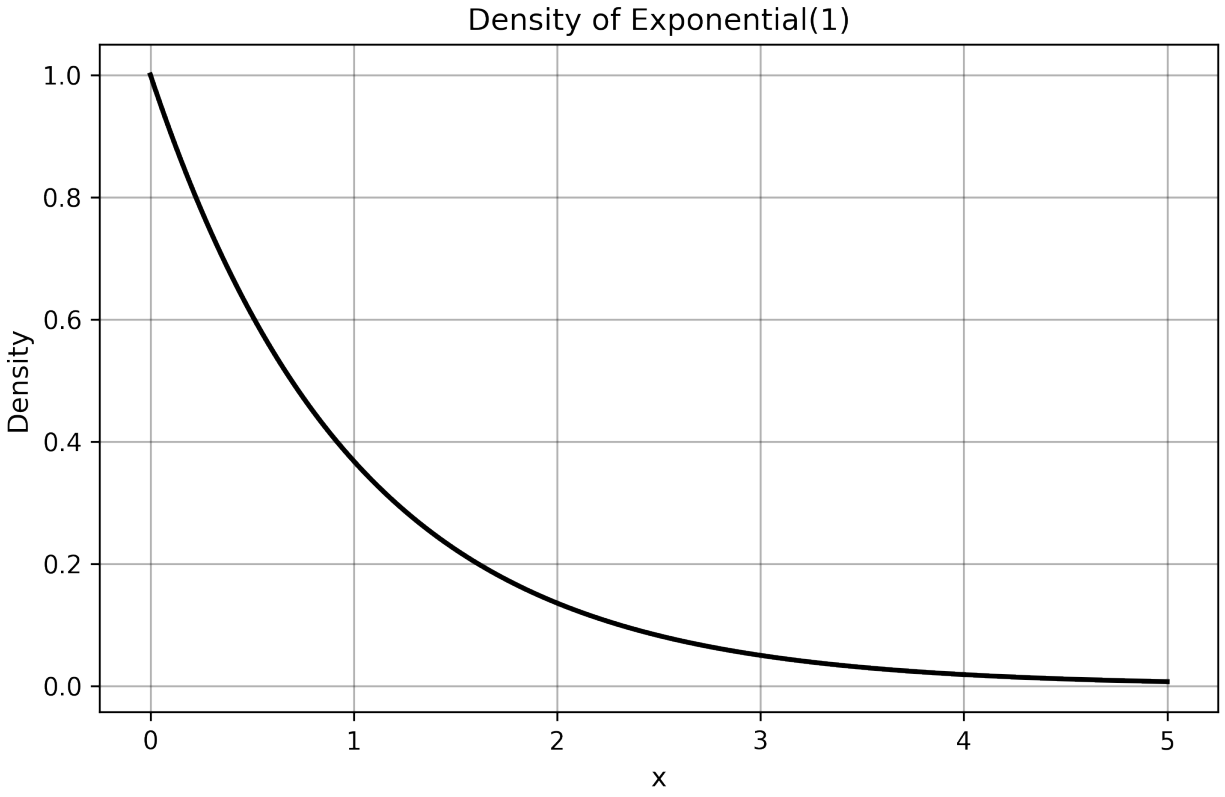
- **Formation:** the product of many positive variables $X_1 \cdot X_2 \cdot ... \cdot X_n$ with mild conditions (e.g., finite variance of $\log X$).
- **Origin:** historically called 'Log-Normal' or 'Galton' distribution after Francis Galton.
- **Rename Motivation:** renamed to Multiplic to reflect its formation mechanism through multiplication.
- **Properties:** logarithm of a Multiplic ('LogNormal') variable follows an Additive ('Normal') distribution.

- **Applications:** stock prices, file sizes, reaction times, income distributions, biological growth rates.
- **Caution:** no perfectly multiplic distributions exist in real data; all real-world measurements contain some deviations. Traditional estimators may struggle with the inherent skewness and heavy right tail.

3.3 Exponential

$$\underline{\text{Exp}}(\text{rate})$$

- rate: rate parameter ($\lambda > 0$, controls decay speed; mean = 1/rate)

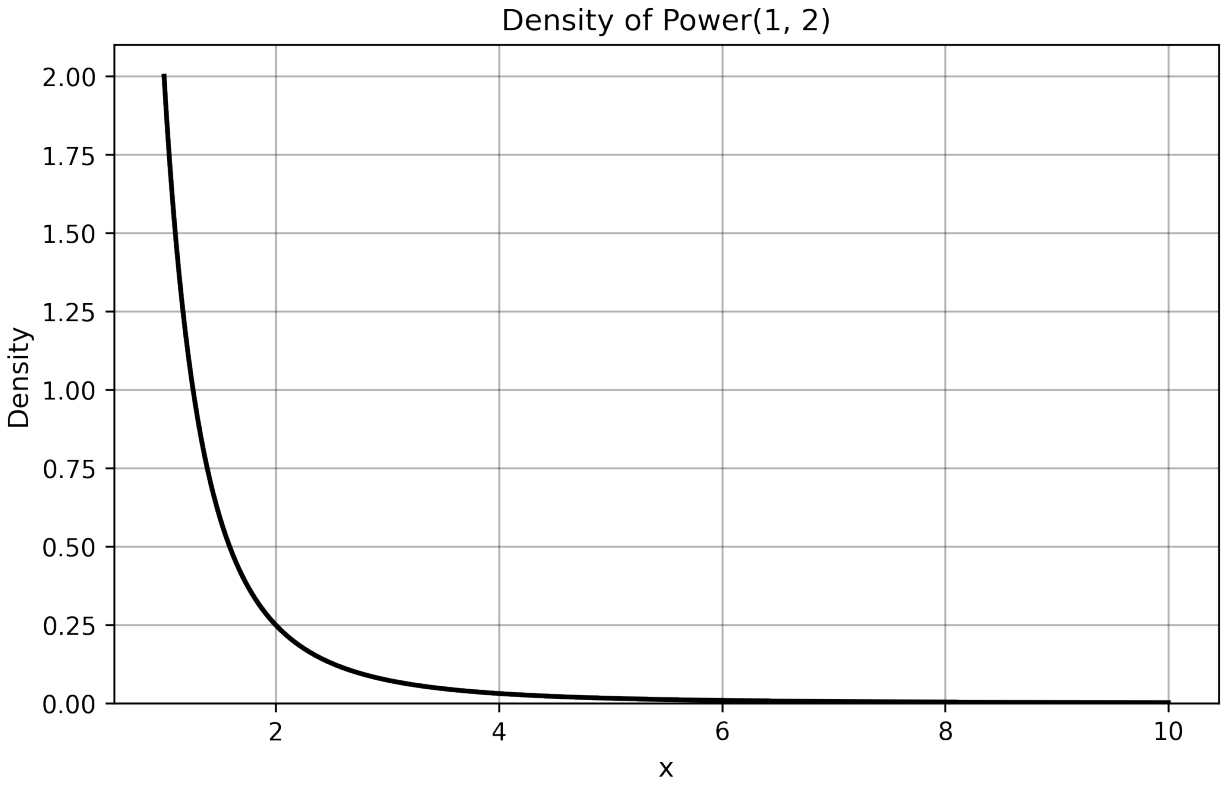


- **Formation:** the waiting time between events in a Poisson process.
- **Origin:** naturally arises from memoryless processes where the probability of an event occurring is constant over time.
- **Properties:** memoryless (past events do not affect future probabilities).
- **Applications:** time between 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.

3.4 Power ('Pareto')

$$\underline{\text{Power}}(\text{min}, \text{shape})$$

- min: minimum value (lower bound, min > 0)
- shape: shape parameter ($\alpha > 0$, controls tail heaviness; smaller values = heavier tails)

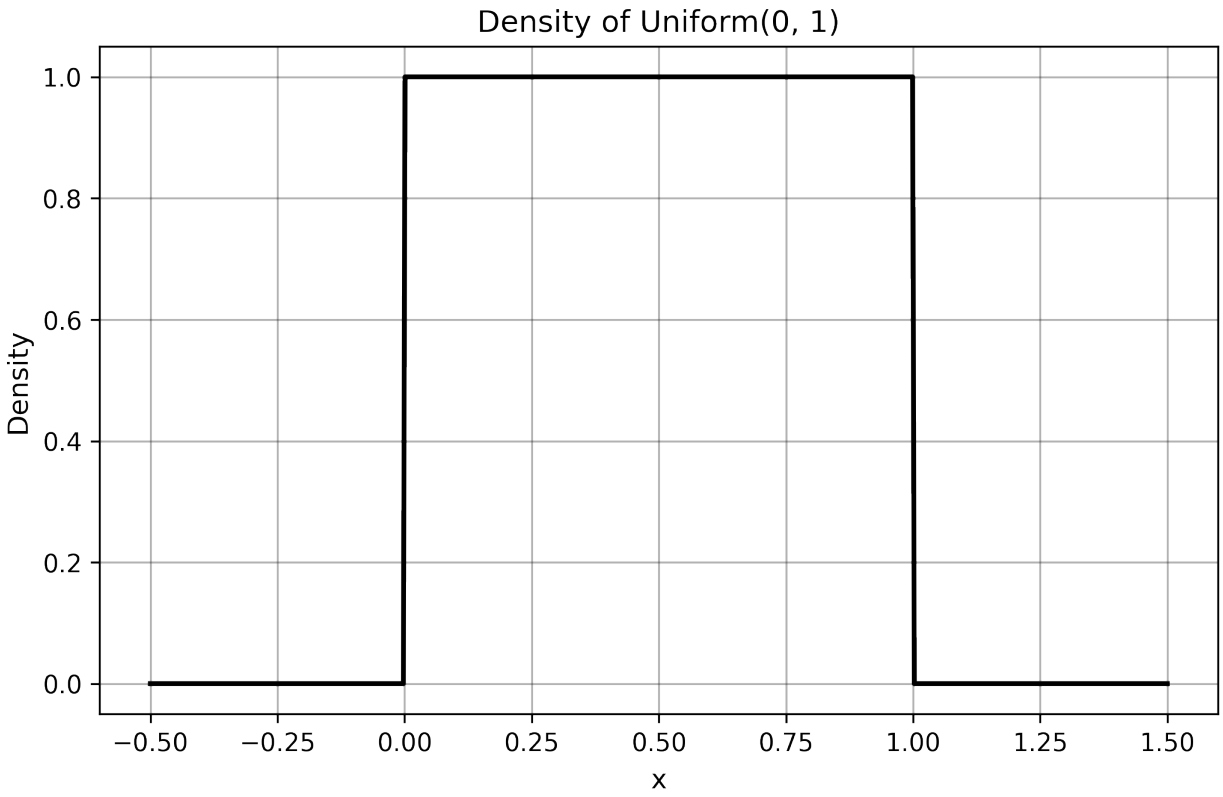


- **Formation:** follows a power-law relationship where large values are rare but possible.
- **Origin:** historically called 'Pareto' distribution after Vilfredo Pareto's work on wealth distribution.
- **Rename Motivation:** renamed to Power to reflect its connection with power-law.
- **Properties:** 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.

3.5 Uniform

$$\text{Uniform}(\text{min}, \text{max})$$

- min: lower bound of the support interval
- max: upper bound of the support interval (max $>$ min)



- **Formation:** all values within a bounded interval have equal probability.
- **Origin:** represents complete uncertainty within known bounds.
- **Properties:** rectangular probability density, finite support with hard boundaries.
- **Applications:** random number generation, round-off errors, arrival times within known intervals.
- **Characteristics:** symmetric, bounded, no tail behavior.
- **Note:** traditional estimators work reasonably well due to symmetry and bounded nature.

4 Summary Estimator Properties

This section compares the toolkit's robust estimators against traditional statistical methods to demonstrate their advantages and universally good properties. 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):

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

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):

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

4.1 Breakdown

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

The breakdown point 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%

4.2 Drift

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

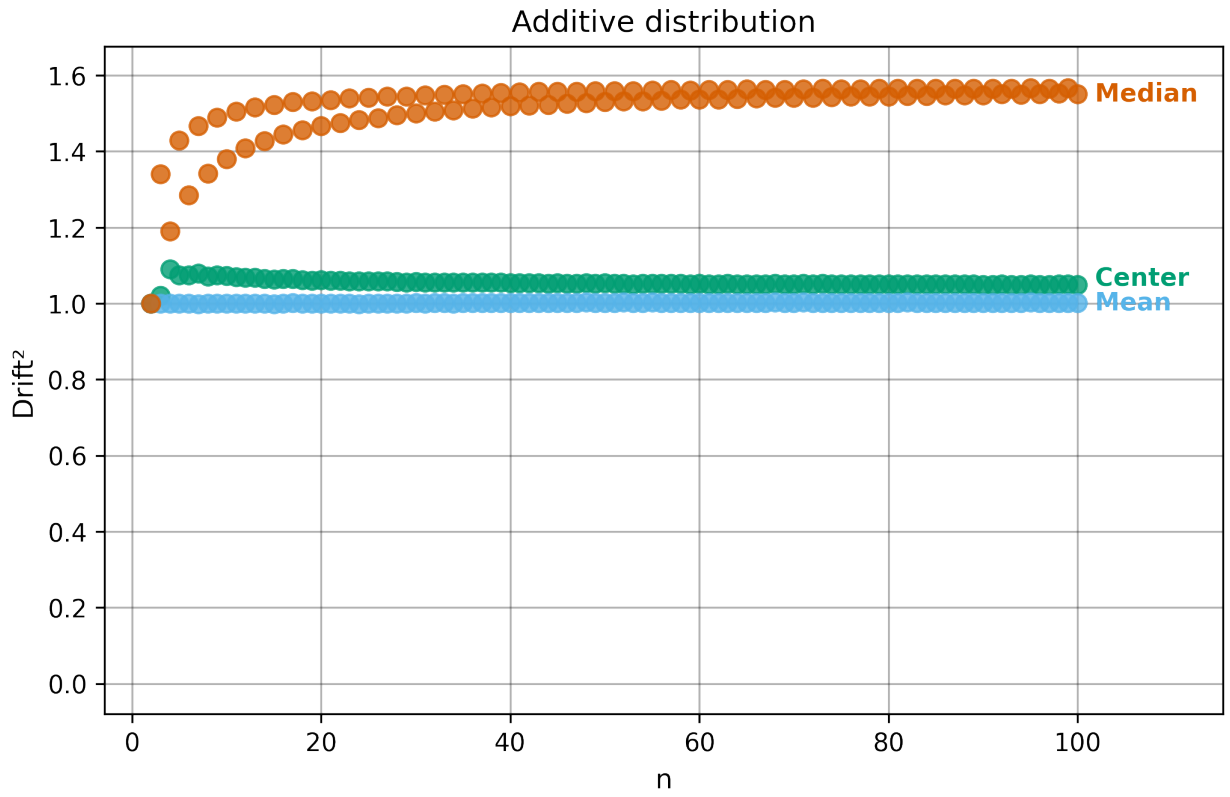
Drift is useful for comparing the precision of several estimators. To simplify the comparison, one of the estimators can be chosen as a baseline. A table of squared drift values, normalized by the baseline, shows the required sample size adjustment factor for switching from the baseline to another estimator.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

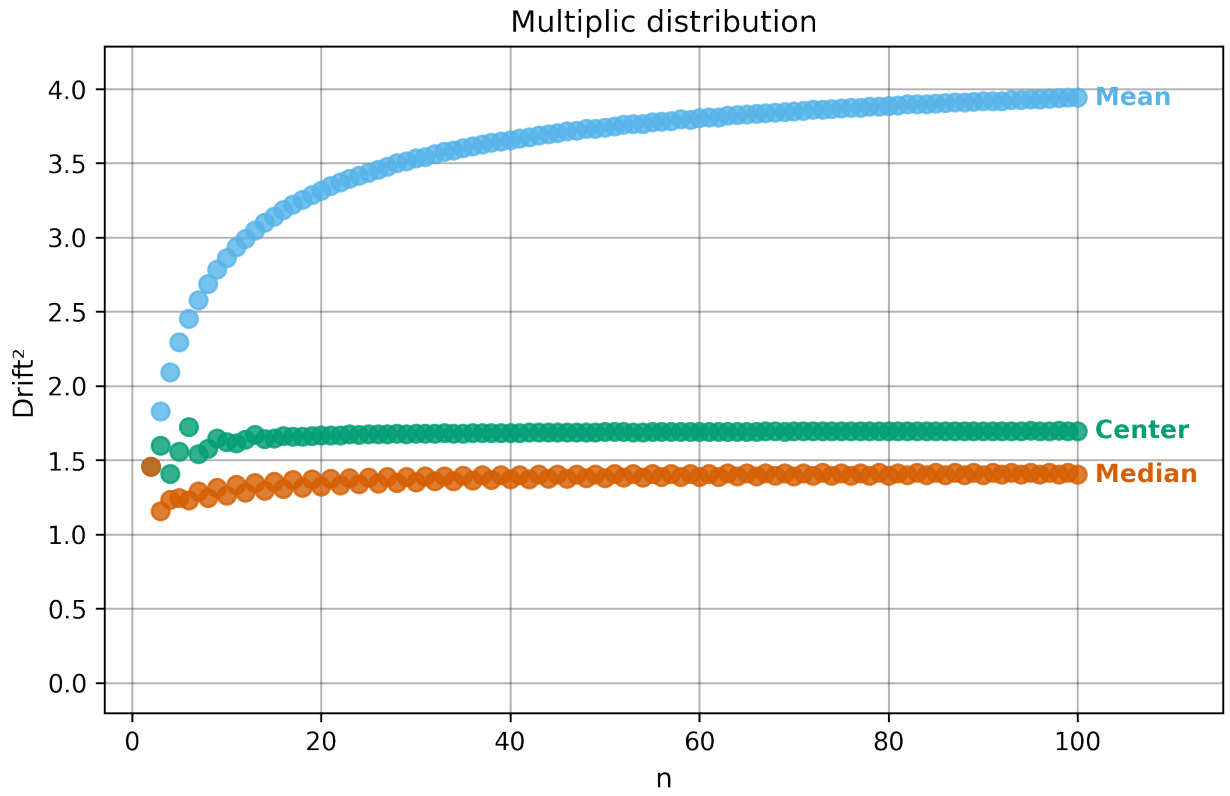
Squared Asymptotic Drift of Average Estimators (values are approximated):

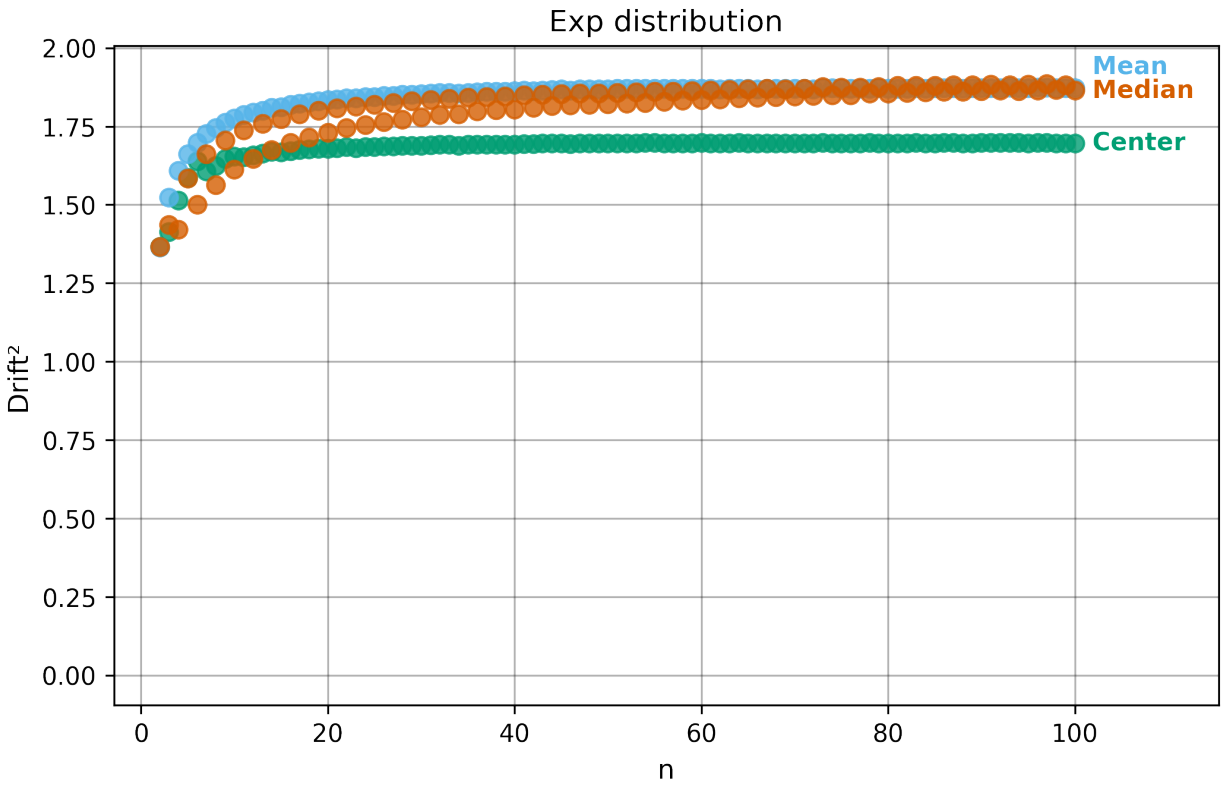
	Mean	Median	Center
Additive	1.0	1.571	1.047
Multiplic	3.95	1.40	1.7
Exp	1.88	1.88	1.69
Power	∞	0.9	2.1
Uniform	0.88	2.60	0.94

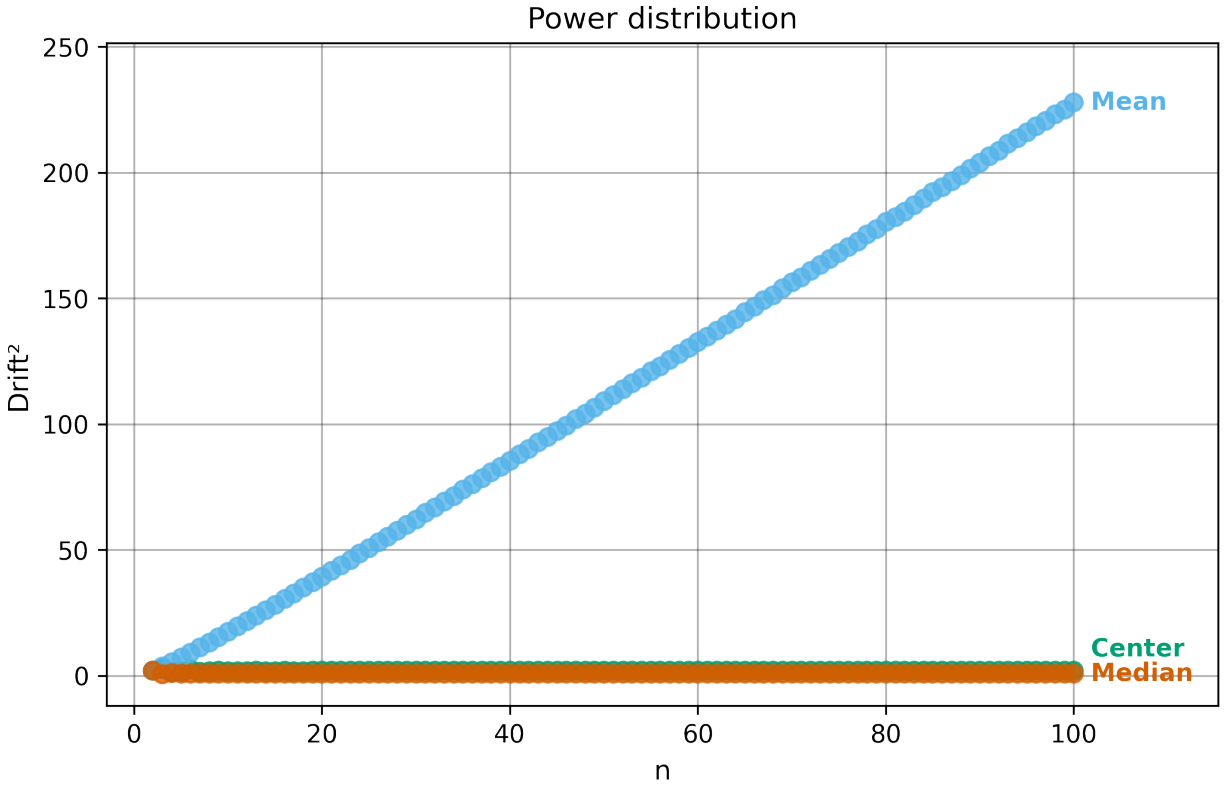
Rescaled to Center (sample size adjustment factors):

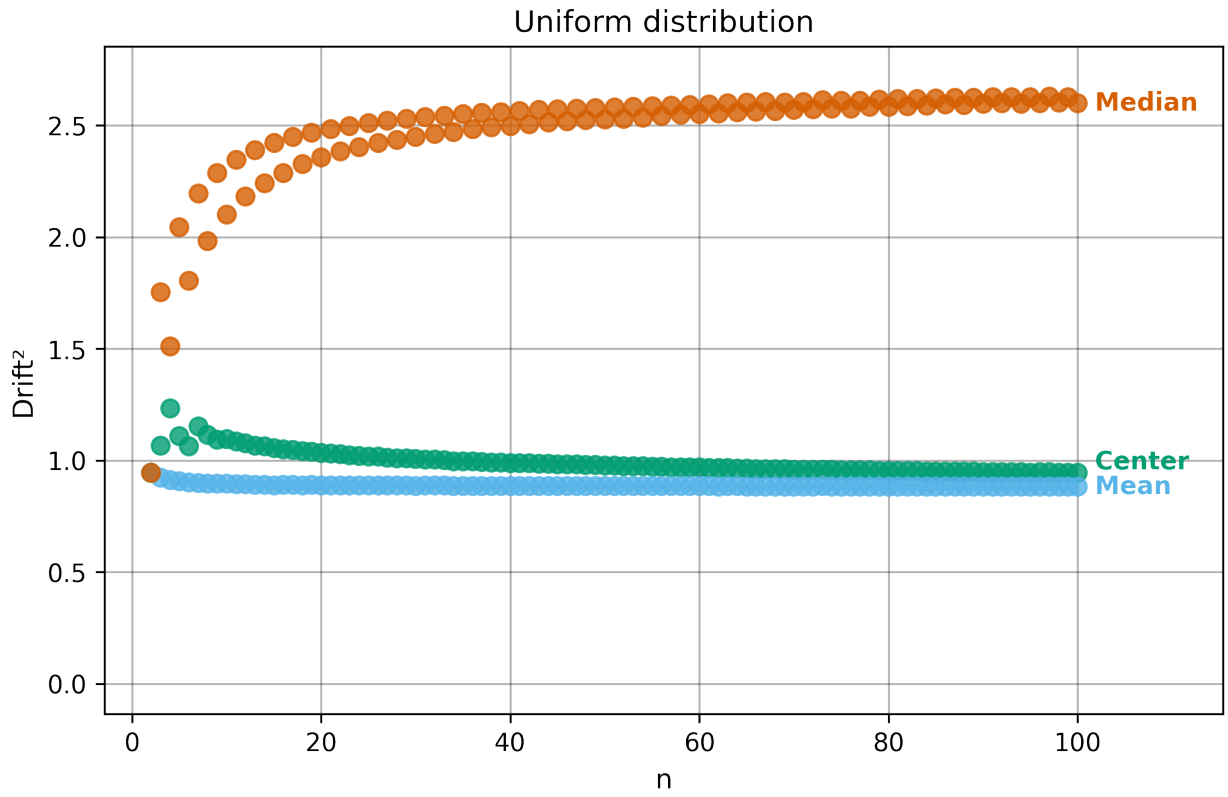
	Mean	Median	Center
Additive	0.96	1.50	1.0
Multiplic	2.32	0.82	1.0
Exp	1.11	1.11	1.0
Power	∞	0.43	1.0
Uniform	0.936	2.77	1.0


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


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


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


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

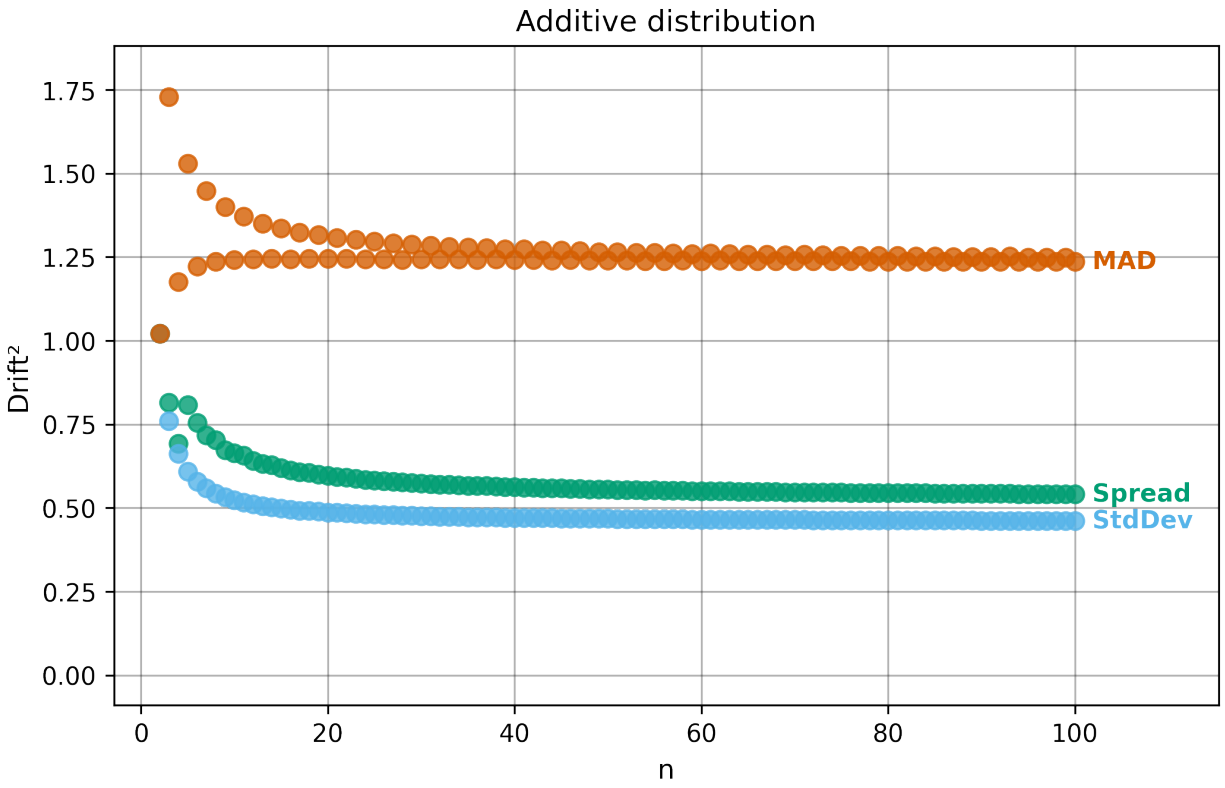


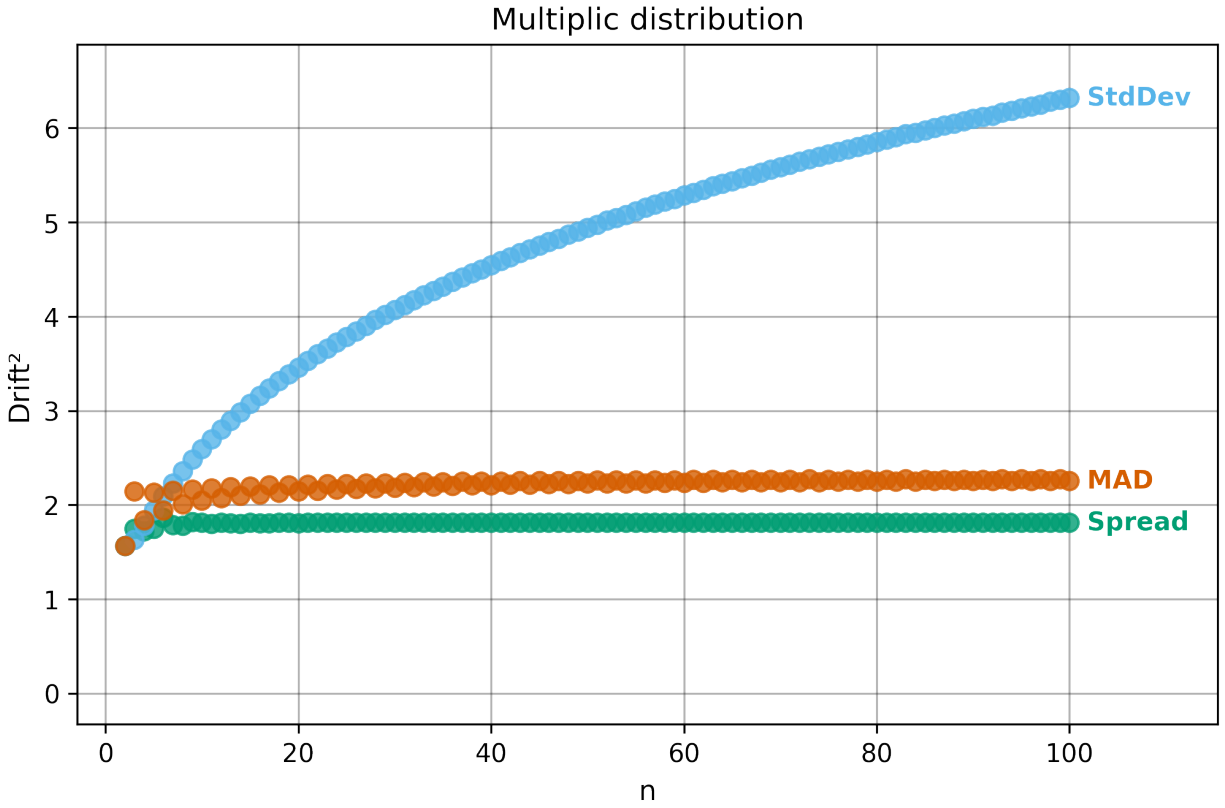
Squared Asymptotic Drift of Dispersion Estimators (values are approximated):

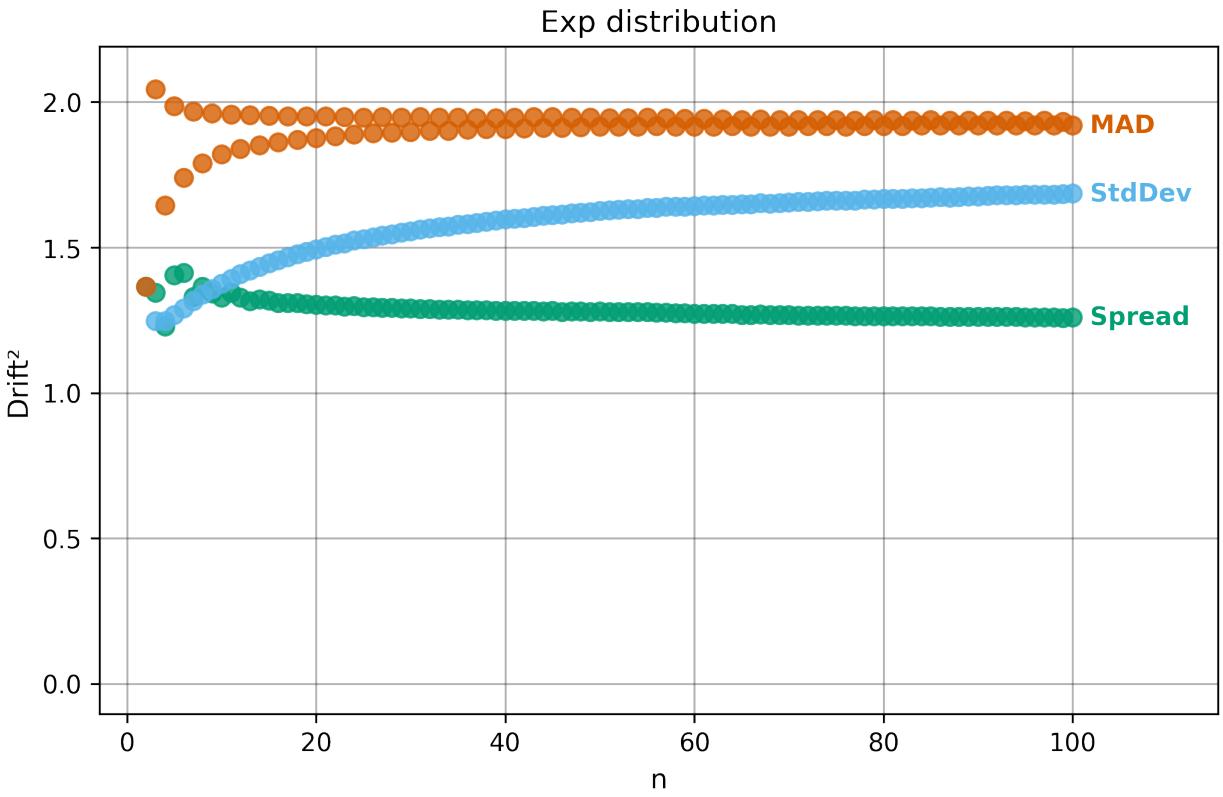
	StdDev	MAD	Spread
Additive	0.45	1.22	0.52
Multiplic	∞	2.26	1.81
Exp	1.69	1.92	1.26
Power	∞	3.5	4.4
Uniform	0.18	0.90	0.43

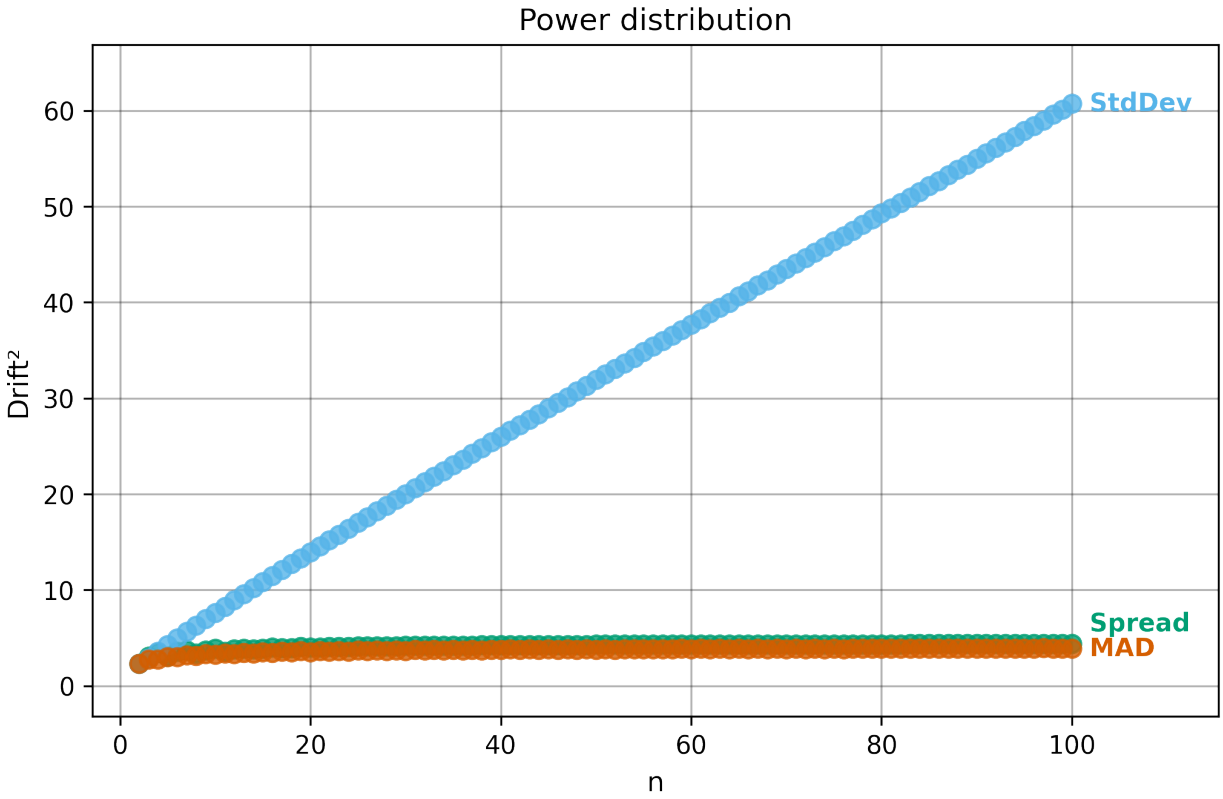
Rescaled to Spread (sample size adjustment factors):

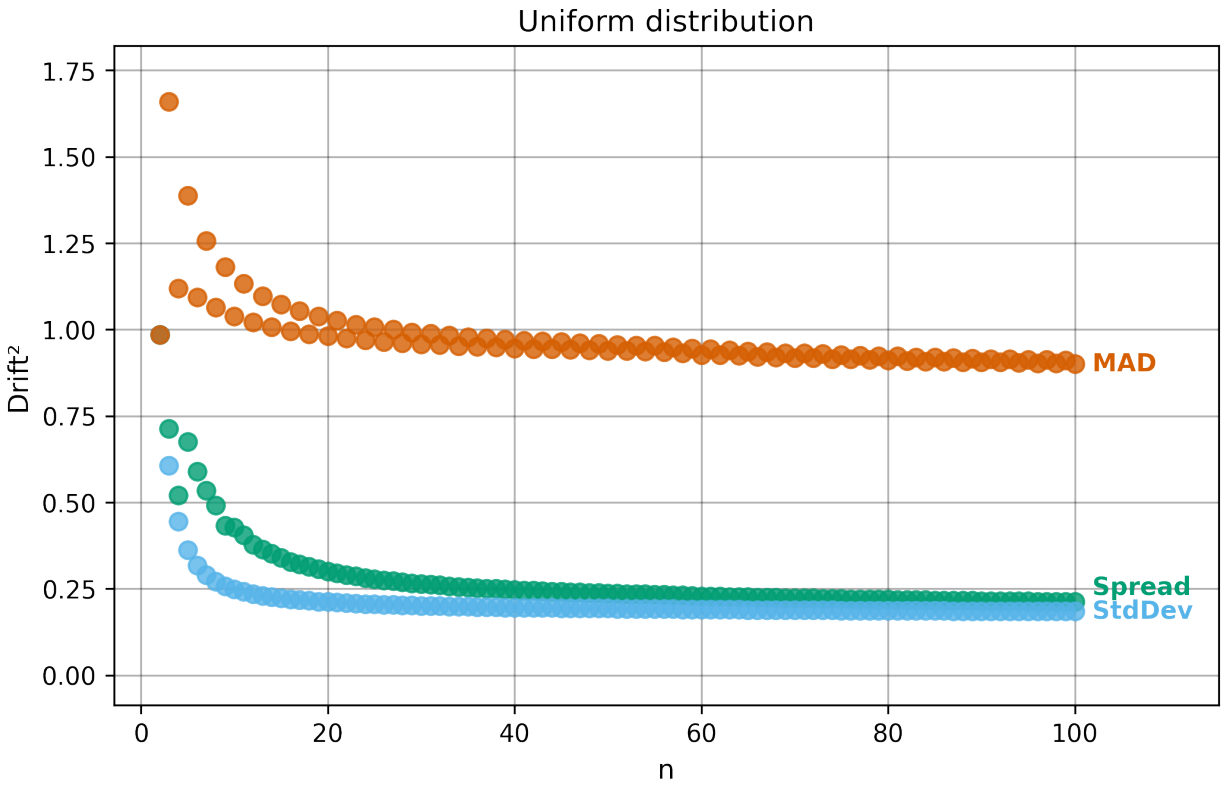
	StdDev	MAD	Spread
Additive	0.87	2.35	1.0
Multiplic	∞	1.25	1.0
Exp	1.34	1.52	1.0
Power	∞	0.80	1.0
Uniform	0.42	2.09	1.0


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


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


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


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



4.3 Invariance

Invariance properties determine how estimators respond to data transformations. These properties are crucial for analysis design and interpretation:

- **Location-invariant** estimators are invariant to additive shifts: $T(\mathbf{x} + k) = T(\mathbf{x})$
- **Scale-invariant** estimators are invariant to positive rescaling: $T(k \cdot \mathbf{x}) = T(\mathbf{x})$ for $k > 0$
- **Equivariant** estimators change predictably with transformations, maintaining relative relationships

Choosing estimators with appropriate invariance properties ensures that results remain meaningful across different measurement scales, units, and data transformations. For example, when comparing datasets collected with different instruments or protocols, location-invariant estimators eliminate the need for data centering, while scale-invariant estimators eliminate the need for normalization.

Location-invariance: An estimator T is location-invariant if adding a constant to the measurements leaves the result unchanged:

$$T(\mathbf{x} + k) = T(\mathbf{x})$$

$$T(\mathbf{x} + k, \mathbf{y} + k) = T(\mathbf{x}, \mathbf{y})$$

Location-equivariance: An estimator T is location-equivariant if it shifts with the data:

$$T(\mathbf{x} + k) = T(\mathbf{x}) + k$$

$$T(\mathbf{x} + k_1, \mathbf{y} + k_2) = T(\mathbf{x}, \mathbf{y}) + f(k_1, k_2)$$

Scale-invariance: An estimator T is scale-invariant if multiplying by a positive constant leaves the result unchanged:

$$T(k \cdot \mathbf{x}) = T(\mathbf{x}) \quad \text{for } k > 0$$

$$T(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = T(\mathbf{x}, \mathbf{y}) \quad \text{for } k > 0$$

Scale-equivariance: An estimator T is scale-equivariant if it scales proportionally with the data:

$$T(k \cdot \mathbf{x}) = k \cdot T(\mathbf{x}) \text{ or } |k| \cdot T(\mathbf{x}) \quad \text{for } k \neq 0$$

$$T(k \cdot \mathbf{x}, k \cdot \mathbf{y}) = k \cdot T(\mathbf{x}, \mathbf{y}) \text{ or } |k| \cdot T(\mathbf{x}, \mathbf{y}) \quad \text{for } k \neq 0$$

	Location	Scale
Center	Equivariant	Equivariant
Spread	Invariant	Equivariant
RelSpread	–	Invariant
Shift	Invariant	Equivariant
Ratio	–	Invariant
AvgSpread	Invariant	Equivariant
Disparity	Invariant	Invariant

5 Methodology

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

5.1 Desiderata

The toolkit consists of statistical *procedures* — practical methods that transform raw measurements into actionable insights and decisions. When practitioners face real-world problems involving data analysis, their success depends on selecting the right procedure for each specific situation. Convenient and efficient procedures have the following *essential attributes*:

- **Usability.** Procedures should feel natural to practitioners and minimize opportunities for misuse. They should be mathematically elegant yet accessible to readers with standard mathematical backgrounds. Implementation should be straightforward across programming languages. Like well-designed APIs, these procedures should follow intuitive design principles that reduce cognitive load.

- **Reliability.** Procedures should deliver consistent, trustworthy results, even in the presence of noise, data corruption, and extreme outliers.
- **Applicability.** Procedures should perform well across diverse contexts and sample sizes. They should handle the full spectrum of distributions commonly encountered in practice, from ideal theoretical models to data that deviates significantly from any assumed distribution.

This manual introduces a unified toolkit that aims to satisfy these properties and provide reliable rule-of-thumb procedures for everyday analytical tasks.

5.2 From Assumptions to Conditions

Traditional statistical practice starts with model assumptions, then derives optimal procedures under those assumptions. This approach prioritizes mathematical convenience over practical application. Practitioners don't know the distribution in advance, so they lack clear guidance on which procedure to choose by default.

Most traditional statistical procedures rely heavily on the Additive ('Normal') distribution and fail on real data because actual measurements contain outliers, exhibit skewness, or follow unknown distributions. When assumptions fail, procedures designed for those assumptions also fail.

This toolkit starts with procedures and tests how they perform under different distributional conditions. This approach reverses the traditional workflow: instead of deriving procedures from assumptions, we evaluate how each procedure performs across various distributions. This enables direct comparison and provides clear guidance on procedure selection based on known characteristics of the data source.

This procedure-first approach eliminates the need for complex mathematical derivations. All evaluations can be done numerically through Monte Carlo simulation. Generate samples from each distribution, apply each procedure, and measure the results. The numerical evidence directly shows which procedures work best under which conditions.

5.3 From Statistical Efficiency to Drift

Statistical efficiency measures estimator precision. 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^*:

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

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

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

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 often 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:

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

This formula measures estimator variability compared to data variability. $\text{Spread}\left[T(X_1, \ldots, X_n)\right]$ 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:

$$\text{DispDrift}(T, X, n) = \sqrt{n}\ \text{RelSpread}\left[T(X_1, \ldots, X_n)\right]$$

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:

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

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 $n \to \infty$. With a baseline estimator, rescaled drift values enable direct comparisons:

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

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

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

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

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.

5.4 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 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**. misrate = 0.01 means "1% chance of error" or "wrong 1 time in 100". misrate = 10^{-6} means "wrong 1 time in a million". No mental arithmetic required.
- **Linear scale for practical values**. misrate = 0.1 (10%), misrate = 0.01 (1%), 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 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($\mathbf{x}, \mathbf{y}$, misrate) clearly indicates: it provides bounds on the shift, with a specified error rate. No background in classical statistics required to understand the concept.

5.5 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. Developed in the 1940s, it tests 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 $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_m)$, 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($\mathbf{x}, \mathbf{y}, \text{misrate}$) uses order statistics of the same pairwise differences

All procedures build on pairwise operations. This structural consistency reflects the mathematical unity underlying robust statistics: pairwise operations provide natural robustness while maintaining computational feasibility and statistical efficiency.

The inversion from hypothesis testing to bounds estimation represents a philosophical shift in statistical practice. Traditional methods ask "should I believe this specific hypothesis?" Pragmatic methods ask "what should I believe, given this data?" Bounds provide actionable answers: they tell practitioners which values are plausible, enabling informed decisions without arbitrary significance thresholds.

Traditional Mann-Whitney implementations apply tie correction when samples contain repeated values. This correction modifies variance calculations to account for tied observations, changing p-values and confidence intervals in ways that depend on measurement precision. The toolkit deliberately omits tie correction. Continuous distributions produce theoretically distinct values; observed ties result from finite measurement precision and digital representation. When measurements appear identical, this reflects rounding of underlying continuous variation, not true equality in the measured quantity. Treating ties as artifacts of discretization rather than distributional features simplifies computation while maintaining accuracy. The exact and approximate methods compute comparison distributions without requiring adjustments for tied values, eliminating a source of complexity and potential inconsistency in statistical practice.

Historical Development

The mathematical foundations emerged through decades of refinement. Mann and Whitney (1947) established the distribution of pairwise comparisons under random sampling, creating the theoretical basis for comparing samples through rank-based methods. Their work demonstrated that comparison counts follow predictable patterns regardless of the underlying population distributions.

The original computational approaches suffered from severe limitations. Mann and Whitney proposed a slow exact method requiring exponential resources and a normal approximation that proved grossly inaccurate for practical use. The approximation works reasonably in distribution centers but fails catastrophically in the tails where practitioners most need accuracy. For moderate sample sizes, approximation errors can exceed factors of 10^{11}.

Fix and Hodges (1955) addressed the approximation problem through higher-order corrections. Their expansion adds terms based on the distribution's actual moments rather than assuming perfect normality. This refinement reduces tail probability errors from orders of magnitude to roughly 1%, making approximation practical for large samples where exact computation becomes infeasible.

Löffler (1982) solved the exact computation problem through algorithmic innovation. The naive recurrence requires quadratic memory— infeasible for samples beyond a few dozen measurements. Löffler discovered a reformulation that reduces memory to linear scale, making exact computation practical for combined sample sizes up to several hundred.

Despite these advances, most statistical software continues using the 1947 approximation. The computational literature contains the solutions, but software implementations lag decades behind theoretical developments. This toolkit implements both the exact method for small samples and the refined approximation for large samples, automatically selecting the appropriate approach based on sample sizes.

The shift from hypothesis testing to bounds estimation requires no new mathematics. The same comparison distributions that enable hypothesis tests also determine which order statistics form reliable bounds. Traditional applications ask "is zero plausible?" and answer yes or no. This toolkit asks "which values are plausible?" and answers with an interval. The perspective inverts while the mathematical foundation remains identical.

6 Algorithms

This chapter describes the core algorithms that power the robust estimators in the toolkit. Both algorithms solve a fundamental computational challenge: how to efficiently find medians within large collections of derived values without materializing the entire collection in memory.

6.1 Fast Center

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

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

A direct implementation would generate all $\frac{n(n+1)}{2}$ pairwise averages and sort them. With $n = 10,000$, this approach creates approximately 50 million values, requiring quadratic memory and $O(n^2 \log n)$ time.

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

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

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

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

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

“‘cs { title = “FastCenter.cs” } namespace Pragmastat.Algorithms;

internal static class FastCenter { ///

/// ACM Algorithm 616: fast computation of the Hodges-Lehmann location estimator ///

/// /// Computes the median of all pairwise averages (xi + xj)/2 efficiently. /// See: John F Monahan, “Algorithm 616: fast computation of the Hodges-Lehmann location estimator” /// (1984) DOI: 10.1145/1271.319414 /// /// A sorted sample of values /// Random number generator /// If values are sorted /// Exact center value (Hodges-Lehmann estimator) public static double Estimate(IReadOnlyList 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 medianRankH

// 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;
  }
}
```

} }

```
## Fast Spread

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

$$
```

```
\Spread(\x) = \underset{1 \leq i < j \leq n}{\Median} |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 di
  yielding an exact $O(n \log n)$ algorithm.
After sorting the input to obtain $y_1 \leq y_2 \leq \cdots \leq y_n$,
  all pairwise absolute differences $|x_i - x_j|$ with $i < j$ become positive differences $y_
This allows considering the implicit upper triangular matrix $D$ where $D_{i,j} = y_j - y_i$ fo
This matrix has a crucial structural property: for a fixed row $i$, differences increase monoto
  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, adapted for differences rather than sums.
For each row $i$, it tracks active column indices representing differences still under consider
  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
  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 $p$ ar

  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 l
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 differe
  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 difference
  and pivots to their midrange.
This midrange strategy ensures convergence even with highly discrete data or datasets with many

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.

```cs { title = "FastSpread.cs" }
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 isSo
  {
    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 pa

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

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

    long prevCountBelow = -1;

    while (true)
```
```

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

  int j = 1; // global two-pointer (non-decreasing across rows)
  for (int i = 0; i < n - 1; i++)
  {
    if (j < i + 1) j = i + 1;
    while (j < n && a[j] - a[i] < pivot) j++;

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

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

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

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

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

```
    bool needLargest = (countBelow == kLow);
    return needLargest ? largestBelow : smallestAtOrAbove;
  }
}

// === STALL HANDLING (ties / no progress) ===
if (countBelow == prevCountBelow)
{
  // Compute min/max remaining difference in the ACTIVE set and pivot to their midrange.
  double minActive = double.PositiveInfinity;
  double maxActive = double.NegativeInfinity;
  long active = 0;

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

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

  if (active <= 0)
  {
    // No active candidates left: the only consistent answer is the boundary implied by c
    // 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;
    }
  }
}
```

6.2 Fast Shift

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

$$\text{Shift}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n,\ 1 \le j \le m}{\text{Median}} \left(x_i - y_j\right)$$

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 $n = m = 10,000$, this approach creates 100 million values, requiring quadratic memory and $O(nm \log(nm))$ time.

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

The algorithm operates through a value-space search rather than index-space selection. It maintains a search interval [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, which interpolates 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.

“‘cs { title = “FastShift.cs” } namespace Pragmastat.Algorithms;

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

public static class FastShift { ///

/// Computes quantiles of all pairwise differences { x_i - y_j }. /// Time: O((m + n) * log(precision)) per quantile. Space: O(1). ///

/// Probabilities in [0, 1]. /// If false, collections will be sorted. public static double[] Estimate(IReadOnlyList x, IReadOnlyList y, double[] p, bool assumeSorted = false) { if (x == null || y == null || p == null) throw new ArgumentNullException(); if (x.Count == 0 || y.Count == 0) throw new ArgumentException("x and y must be non-empty.");

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

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

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

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

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

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

var result = new double[p.Length];
for (int i = 0; i < p.Length; i++)
```

```
{
  var (lowerRank, upperRank, weight) = interpolationParams[i];
  double lower = rankValues[lowerRank];
  double upper = rankValues[upperRank];
  result[i] = weight == 0.0 ? lower : (1.0 - weight) * lower + weight * upper;
}

return result;
```

}

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

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

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

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

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

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

  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; }

```
## Fast PairwiseMargin
```

```
The $\PairwiseMargin$ function determines how many extreme pairwise differences to exclude
  when constructing bounds around $\Shift(\x, \y)$.
Given samples $\x = (x_1, \ldots, x_n)$ and $\y = (y_1, \ldots, y_m)$,
  the $\ShiftBounds$ estimator computes all $nm$ pairwise differences $z_{ij} = x_i - y_j$ and
The bounds select specific order statistics from this sorted sequence:
  $[z_{(k_{\mathrm{left}})}, z_{(k_{\mathrm{right}})}]$.
The challenge lies in determining which order statistics produce bounds
  that contain the true shift $\Shift[X, Y]$ with probability $1 - \misrate$.

Random sampling creates natural variation in pairwise differences.
Even when populations have identical distributions, sampling variation produces both positive a
The margin function quantifies this sampling variability:
  it specifies how many extreme pairwise differences could occur by chance with probability $\r
For symmetric bounds, this margin splits evenly between the tails,
  giving $k_{\mathrm{left}} = \lfloor \PairwiseMargin(n, m, \misrate) / 2 \rfloor + 1$
  and $k_{\mathrm{right}} = nm - \lfloor \PairwiseMargin(n, m, \misrate) / 2 \rfloor$.

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:

$$
\Dominance(\x, \y) = \sum_{i=1}^n \sum_{j=1}^m \mathbb{1}(x_i > y_j)
$$

The dominance function counts how many pairwise comparisons favor $\x$ over $\y$.
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 symmet
The distribution of $\Dominance$ across all possible sample orderings determines reliable bound
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 $1 - \misrate$.
Translation invariance extends this relationship to arbitrary shifts:
  the margin computed from the comparison distribution applies regardless of the true shift val

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 $C_{n+m}^n$ 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 memory and computat
For samples beyond a few dozen measurements, naive implementation becomes impractical.

Löffler's recurrence relation ([@loeffler1982]) 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:

$$
\Pr(\Dominance = c) = \frac{p_{n,m}(c)}{C_{n+m}^n}
$$

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 ([@loeffler1982]) 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:

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

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

The algorithm computes cumulative probabilities $\Pr(\Dominance \leq c)$ sequentially
  until the threshold $\misrate/2$ is exceeded.
By symmetry, the lower and upper thresholds determine the total margin $\PairwiseMargin = 2c$.

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 $\Pr(\Dominance \leq u)$ 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 size
  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:

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

This approximation underestimates tail probabilities for moderate sample sizes.
The $\Additive$ ('Normal') approximation provides a convenient 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 ([@fix1955]
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
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 co
  substantially improving tail probability estimates compared to the basic approximation.

The standardized comparison count:
```

```
$$
z = \frac{c - nm/2}{\sqrt{nm(n+m+1)/12}}
$$

produces the approximated cumulative distribution:

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

where $\Phi$ 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 $\mu_2$, $\mu_4$, $\mu_6$ are computed from sample sizes:

$$
\mu_2 = \frac{nm(n+m+1)}{12}
$$

$$
\mu_4 = \frac{nm(n+m+1)}{240} \left(5nm(n+m) - 2(n^2 + m^2) + 3nm - 2(n+m)\right)
$$

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

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 $\Pr(\Dominance \leq c) < \misrate/2$, the threshold lies above $c$ and the search continues
If $\Pr(\Dominance \leq c) \geq \misrate/2$, the threshold lies below $c$ and the search contin
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 computatio
The logarithmic scaling ensures that doubling the sample size adds only one additional iteratio
  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 $n = m = 10,000$, the approximate method completes in milliseconds versus minutes for exact

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 reasonable 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 extr
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, \ldots, x_n > y_1
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:

$$
\misrate_{\min} = \frac{1}{C_{n+m}^n} = \frac{n! \cdot m!}{(n+m)!}
$$
```

```
This represents the minimum probability of the most extreme ordering under random sampling.
Setting $\misrate < \misrate_{\min}$ 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 $\misrate < \misrate_{\min}$ forces the algorithm
  to exclude zero pairwise differences from the tails, making $\PairwiseMargin = 0$.
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 $1/C_{12}^6 \approx 0.00108$,
  making the typical choice of $\misrate = 10^{-3}$ impossible.
With $n = m = 4$, the minimum becomes $1/C_8^4 \approx 0.0143$,
  exceeding even $\misrate = 0.01$.
The table below shows $\misrate_{\min}$ for small sample sizes:

|   |        1|        2|        3|        4|        5|        6|        7|        8|        9
|:--|--------:|--------:|--------:|--------:|--------:|--------:|--------:|--------:|--------:
|1  | 0.500000| 0.333333| 0.250000| 0.200000| 0.166667| 0.142857| 0.125000| 0.111111| 0.100000
|2  | 0.333333| 0.166667| 0.100000| 0.066667| 0.047619| 0.035714| 0.027778| 0.022222| 0.018182
|3  | 0.250000| 0.100000| 0.050000| 0.028571| 0.017857| 0.011905| 0.008333| 0.006061| 0.004545
|4  | 0.200000| 0.066667| 0.028571| 0.014286| 0.007937| 0.004762| 0.003030| 0.002020| 0.001399
|5  | 0.166667| 0.047619| 0.017857| 0.007937| 0.003968| 0.002165| 0.001263| 0.000777| 0.000500
|6  | 0.142857| 0.035714| 0.011905| 0.004762| 0.002165| 0.001082| 0.000583| 0.000333| 0.000200
|7  | 0.125000| 0.027778| 0.008333| 0.003030| 0.001263| 0.000583| 0.000291| 0.000155| 0.000087
|8  | 0.111111| 0.022222| 0.006061| 0.002020| 0.000777| 0.000333| 0.000155| 0.000078| 0.000041
|9  | 0.100000| 0.018182| 0.004545| 0.001399| 0.000500| 0.000200| 0.000087| 0.000041| 0.000021
|10 | 0.090909| 0.015152| 0.003497| 0.000999| 0.000333| 0.000125| 0.000051| 0.000023| 0.000011

For meaningful bounds construction, choose $\misrate > \misrate_{\min}$.
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 $\misrate = 10^{-6}$ feasible.

```cs { title = "PairwiseMargin.cs" }
using System;
using JetBrains.Annotations;
using Pragmastat.Internal;

namespace Pragmastat.Functions;

/// <summary>
/// PairwiseMargin function
/// </summary>
```
```

```
/// <param name="threshold">The maximum value for n+m, after which implementation switches fro
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)
  {
    Assertion.MoreThan(nameof(n), n, 0);
    Assertion.MoreThan(nameof(m), m, 0);
    Assertion.InRangeInclusive(nameof(misrate), misrate, 0, 1);

    return n + m <= threshold
      ? CalcExact(n, m, misrate)
      : CalcApprox(n, m, misrate);
  }

  [PublicAPI]
  public int CalcExact(int n, int m, double misrate)
  {
    int raw = CalcExactRaw(n, m, misrate / 2);
    return checked(raw * 2);
  }

  [PublicAPI]
  public 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 u
  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);
  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 = n * n;
  double n3 = n2 * n;
  double n4 = n2 * n2;
  double m2 = m * m;
  double m3 = m2 * m;
  double m4 = m2 * m2;

  double mu2 = n * m * (n + m + 1) / 12.0;
  double mu4 =
    n * m * (n + m + 1) *
    (0
     + 5 * m * n * (m + n)
     - 2 * (m2 + n2)
     + 3 * m * n
     - 2 * (n + m)
    ) / 240.0;

  double mu6 =
    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);
  }
}
```

7 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.

7.1 Additive ('Normal') Distribution

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

7.1.1 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, $E[D] = 0$. 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

$$F_{|D|}(z) = \Pr(|D| \leq z) = 2\Phi\Big(\frac{z}{\tau}\Big) - 1,$$

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

$$2\Phi\Big(\frac{m}{\tau}\Big) - 1 = \tfrac{1}{2} \quad \Longrightarrow \quad \Phi\Big(\frac{m}{\tau}\Big) = \tfrac{3}{4}.$$

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.

7.1.2 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 distribution $T_n \sim \text{approx Additive}(\mu, (a \cdot \text{stdDev})^2/n)$, 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: $\text{RelSpread}[T_n] = a/\sqrt{n}$
- The asymptotic drift: $\text{Drift}(T, X) = a$

7.1.3 Asymptotic Mean Drift

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

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

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

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.

7.1.4 Asymptotic Median Drift

For the sample median 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 $a = \sqrt{\pi/2}$:

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

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

7.1.5 Asymptotic Center Drift

For the sample center $\text{Center}(\mathbf{x}) = \underset{1 \le i \le j \le n}{\text{Median}} \left(\frac{x_i + x_j}{2} \right)$ 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:

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

This gives an asymptotic standard deviation of:

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

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

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

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.

7.1.6 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 $T_n \sim \text{approx } \underline{\text{Additive}}(b \cdot \text{stdDev}, (a \cdot \text{stdDev})^2/n)$, 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.

7.1.7 Asymptotic StdDev Drift

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

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

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

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

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

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

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

Therefore:

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

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

7.1.8 Asymptotic MAD Drift

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

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

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

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

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

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

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

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

Therefore:

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

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

7.1.9 Asymptotic Spread Drift

For the sample spread $\text{Spread}(\mathbf{x}) = \underset{1\le i<j\le n}{\text{Median}}|x_i - x_j|$ 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:

$$\text{StdDev}[\text{Spread}(X_{1:n})] = c_{\text{spr}} \cdot \frac{\text{stdDev}}{\sqrt{n}}$$

where $c_{\text{spr}} \approx 0.72$.

Applying the lemma with $a = c_{\text{spr}}$ and $b = \sqrt{2} \cdot z_{0.75}$:

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

Since $\text{Center}[\text{Spread}(X_{1:n})] \approx \sqrt{2} \cdot z_{0.75} \cdot \text{stdDev}$ asymptotically:

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

Therefore:

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

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

7.1.10 Summary

Summary for average estimators:

Estimator	$\mathrm{Drift}(E, X)$	$\mathrm{Drift}^2(E, X)$	$1/\mathrm{Drift}^2(E, X)$
Mean	1	1	1
Median	≈ 1.253	$\pi/2 \approx 1.571$	$2/\pi \approx 0.637$
Center	≈ 1.023	$\pi/3 \approx 1.047$	$3/\pi \approx 0.955$

The squared drift values indicate the sample size adjustment needed when switching estimators. For instance, switching from Mean to Median while maintaining the same precision requires increasing the sample size by a factor of $\pi/2 \approx 1.571$ (about 57% more observations). Similarly, switching from Mean to Center requires only about 5% more observations.

The inverse squared drift (rightmost column) equals the classical statistical efficiency relative to the Mean. The Mean achieves optimal performance (unit efficiency) for the Additive ('Normal') distribution, as expected from classical theory. The Center maintains 95.5% efficiency while offering greater robustness to outliers, making it an attractive alternative when some contamination is possible. The Median, while most robust, operates at only 63.7% efficiency under purely Additive ('Normal') conditions.

Summary for dispersion estimators:

For the Additive ('Normal') distribution, the asymptotic drift values reveal the relative precision of different dispersion estimators:

Estimator	$\mathrm{Drift}(E, X)$	$\mathrm{Drift}^2(E, X)$	$1/\mathrm{Drift}^2(E, X)$
StdDev	≈ 0.67	≈ 0.45	≈ 2.22
MAD	≈ 1.10	≈ 1.22	≈ 0.82
Spread	≈ 0.72	≈ 0.52	≈ 1.92

The squared drift values indicate the sample size adjustment needed when switching estimators. For instance, switching from StdDev to MAD while maintaining the same precision requires increasing the sample size by a factor of $1.22/0.45 \approx 2.71$ (more than doubling the observations). Similarly, switching from StdDev to Spread requires a factor of $0.52/0.45 \approx 1.16$.

The StdDev achieves optimal performance for the Additive ('Normal') distribution. The MAD requires about 2.7 times more data to match the StdDev precision while offering greater robustness to outliers. The Spread requires about 1.16 times more data to match the StdDev precision under purely Additive ('Normal') conditions while maintaining robustness.

8 Reference Implementations

8.1 Python

Install from PyPI:

```
pip install pragmastat==4.0.1
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/py

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

Demo:

```
from pragmastat import (
    center,
    spread,
    rel_spread,
    shift,
    ratio,
    avg_spread,
    disparity,
    pairwise_margin,
    shift_bounds,
)


def main():
    x = [0, 2, 4, 6, 8]
    print(center(x))  # 4
    print(center([v + 10 for v in x]))  # 14
    print(center([v * 3 for v in x]))  # 12

    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))  # 1
    print(rel_spread([v * 5 for v in x]))  # 1

    y = [10, 12, 14, 16, 18]
    print(shift(x, y))  # -10
    print(shift(x, x))  # 0
    print(shift([v + 7 for v in x], [v + 3 for v in y]))  # -6
    print(shift([v * 2 for v in x], [v * 2 for v in y]))  # -20
    print(shift(y, x))  # 10

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

```
    print(ratio([v * 2 for v in x], [v * 5 for v in y]))  # 0.2

    x = [0, 3, 6, 9, 12]
    y = [0, 2, 4, 6, 8]
    print(spread(x))  # 6
    print(spread(y))  # 4

    print(avg_spread(x, y))  # 5
    print(avg_spread(x, x))  # 6
    print(avg_spread([v * 2 for v in x], [v * 3 for v in x]))  # 15
    print(avg_spread(y, x))  # 5
    print(avg_spread([v * 2 for v in x], [v * 2 for v in y]))  # 10

    print(shift(x, y))  # 2
    print(avg_spread(x, y))  # 5

    print(disparity(x, y))  # 0.4
    print(disparity([v + 5 for v in x], [v + 5 for v in y]))  # 0.4
    print(disparity([v * 2 for v in x], [v * 2 for v in y]))  # 0.4
    print(disparity(y, x))  # -0.4

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

    print(pairwise_margin(30, 30, 1e-6))  # 276
    print(pairwise_margin(30, 30, 1e-5))  # 328
    print(pairwise_margin(30, 30, 1e-4))  # 390
    print(pairwise_margin(30, 30, 1e-3))  # 464

    print(shift(x, y))  # -20

    bounds = shift_bounds(x, y, 1e-6)  # [-33, -7]
    print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")
    bounds = shift_bounds(x, y, 1e-5)  # [-32, -8]
    print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")
    bounds = shift_bounds(x, y, 1e-4)  # [-30, -10]
    print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")
    bounds = shift_bounds(x, y, 1e-3)  # [-28, -12]
    print(f"Bounds(lower={bounds.lower}, upper={bounds.upper})")


if __name__ == "__main__":
    main()
```

8.2 TypeScript

Install from npm:

```
npm i pragmastat@4.0.1
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/ts

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

Demo:

```
import { center, spread, relSpread, shift, ratio, avgSpread, disparity,
 ↪  shiftBounds, pairwiseMargin } from '../src';

function main() {
    let x = [0, 2, 4, 6, 8];
    console.log(center(x)); // 4
    console.log(center(x.map(v => v + 10))); // 14
    console.log(center(x.map(v => v * 3))); // 12

    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)); // 1
    console.log(relSpread(x.map(v => v * 5))); // 1

    let y = [10, 12, 14, 16, 18];
    console.log(shift(x, y)); // -10
    console.log(shift(x, x)); // 0
    console.log(shift(x.map(v => v + 7), y.map(v => v + 3))); // -6
    console.log(shift(x.map(v => v * 2), y.map(v => v * 2))); // -20
    console.log(shift(y, x)); // 10

    x = [1, 2, 4, 8, 16];
    y = [2, 4, 8, 16, 32];
    console.log(ratio(x, y)); // 0.5
    console.log(ratio(x, x)); // 1
    console.log(ratio(x.map(v => v * 2), y.map(v => v * 5))); // 0.2

    x = [0, 3, 6, 9, 12];
    y = [0, 2, 4, 6, 8];
    console.log(spread(x)); // 6
    console.log(spread(y)); // 4

    console.log(avgSpread(x, y)); // 5
    console.log(avgSpread(x, x)); // 6
    console.log(avgSpread(x.map(v => v * 2), x.map(v => v * 3))); // 15
    console.log(avgSpread(y, x)); // 5
    console.log(avgSpread(x.map(v => v * 2), y.map(v => v * 2))); // 10

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

    console.log(disparity(x, y)); // 0.4
```

```
    console.log(disparity(x.map(v => v + 5), y.map(v => v + 5))); // 0.4
    console.log(disparity(x.map(v => v * 2), y.map(v => v * 2))); // 0.4
    console.log(disparity(y, x)); // -0.4

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

    console.log(pairwiseMargin(30, 30, 1e-6)); // 276
    console.log(pairwiseMargin(30, 30, 1e-5)); // 328
    console.log(pairwiseMargin(30, 30, 1e-4)); // 390
    console.log(pairwiseMargin(30, 30, 1e-3)); // 464

    console.log(shift(x, y)); // -20

    console.log(shiftBounds(x, y, 1e-6)); // [-33, -7]
    console.log(shiftBounds(x, y, 1e-5)); // [-32, -8]
    console.log(shiftBounds(x, y, 1e-4)); // [-30, -10]
    console.log(shiftBounds(x, y, 1e-3)); // [-28, -12]
}

main();
```

8.3 R

Install from GitHub:

```
install.packages("remotes") # If 'remotes' is not installed
remotes::install_github("AndreyAkinshin/pragmastat",
                        subdir = "r/pragmastat", ref = "v4.0.1")
library(pragmastat)
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/r

Demo:

```
library(pragmastat)

x <- c(0, 2, 4, 6, 8)
print(center(x)) # 4
print(center(x + 10)) # 14
print(center(x * 3)) # 12

print(spread(x)) # 4
print(spread(x + 10)) # 4
print(spread(x * 2)) # 8

print(rel_spread(x)) # 1
print(rel_spread(x * 5)) # 1

y <- c(10, 12, 14, 16, 18)
print(shift(x, y)) # -10
```

```
print(shift(x, x)) # 0
print(shift(x + 7, y + 3)) # -6
print(shift(x * 2, y * 2)) # -20
print(shift(y, x)) # 10

x <- c(1, 2, 4, 8, 16)
y <- c(2, 4, 8, 16, 32)
print(ratio(x, y)) # 0.5
print(ratio(x, x)) # 1
print(ratio(x * 2, y * 5)) # 0.2

x <- c(0, 3, 6, 9, 12)
y <- c(0, 2, 4, 6, 8)
print(spread(x)) # 6
print(spread(y)) # 4

print(avg_spread(x, y)) # 5
print(avg_spread(x, x)) # 6
print(avg_spread(x * 2, x * 3)) # 15
print(avg_spread(y, x)) # 5
print(avg_spread(x * 2, y * 2)) # 10

print(shift(x, y)) # 2
print(avg_spread(x, y)) # 5

print(disparity(x, y)) # 0.4
print(disparity(x + 5, y + 5)) # 0.4
print(disparity(x * 2, y * 2)) # 0.4
print(disparity(y, x)) # -0.4

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

print(pairwise_margin(30, 30, 1e-6)) # 276
print(pairwise_margin(30, 30, 1e-5)) # 328
print(pairwise_margin(30, 30, 1e-4)) # 390
print(pairwise_margin(30, 30, 1e-3)) # 464

print(shift(x, y)) # -20

bounds <- shift_bounds(x, y, 1e-6) # [-33, -7]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))
bounds <- shift_bounds(x, y, 1e-5) # [-32, -8]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))
bounds <- shift_bounds(x, y, 1e-4) # [-30, -10]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))
bounds <- shift_bounds(x, y, 1e-3) # [-28, -12]
print(paste("[", bounds$lower, ", ", bounds$upper, "]", sep=""))
```

8.4 C#

Install from NuGet via .NET CLI:

```
dotnet add package Pragmastat --version 4.0.1
```

Install from NuGet via Package Manager Console:

```
NuGet\Install-Package Pragmastat -Version 4.0.1
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/cs

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

Demo:

```cs
using static System.Console;
using Pragmastat.Functions;

namespace Pragmastat.Demo;

class Program
{
  static void Main()
  {
    var x = new Sample(0, 2, 4, 6, 8);
    WriteLine(x.Center()); // 4
    WriteLine((x + 10).Center()); // 14
    WriteLine((x * 3).Center()); // 12

    WriteLine(x.Spread()); // 4
    WriteLine((x + 10).Spread()); // 4
    WriteLine((x * 2).Spread()); // 8

    WriteLine(x.RelSpread()); // 1
    WriteLine((x * 5).RelSpread()); // 1

    var y = new Sample(10, 12, 14, 16, 18);
    WriteLine(Toolkit.Shift(x, y)); // -10
    WriteLine(Toolkit.Shift(x, x)); // 0
    WriteLine(Toolkit.Shift(x + 7, y + 3)); // -6
    WriteLine(Toolkit.Shift(x * 2, y * 2)); // -20
    WriteLine(Toolkit.Shift(y, x)); // 10

    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(x, x)); // 1
    WriteLine(Toolkit.Ratio(x * 2, y * 5)); // 0.2

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

```
    WriteLine(x.Spread()); // 6
    WriteLine(y.Spread()); // 4

    WriteLine(Toolkit.AvgSpread(x, y)); // 5
    WriteLine(Toolkit.AvgSpread(x, x)); // 6
    WriteLine(Toolkit.AvgSpread(x * 2, x * 3)); // 15
    WriteLine(Toolkit.AvgSpread(y, x)); // 5
    WriteLine(Toolkit.AvgSpread(x * 2, y * 2)); // 10

    WriteLine(Toolkit.Shift(x, y)); // 2
    WriteLine(Toolkit.AvgSpread(x, y)); // 5

    WriteLine(Toolkit.Disparity(x, y)); // 0.4
    WriteLine(Toolkit.Disparity(x + 5, y + 5)); // 0.4
    WriteLine(Toolkit.Disparity(x * 2, y * 2)); // 0.4
    WriteLine(Toolkit.Disparity(y, x)); // -0.4

    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-6)); // 276
    WriteLine(PairwiseMargin.Instance.Calc(30, 30, 1e-5)); // 328
    WriteLine(PairwiseMargin.Instance.Calc(30, 30, 1e-4)); // 390
    WriteLine(PairwiseMargin.Instance.Calc(30, 30, 1e-3)); // 464

    WriteLine(Toolkit.Shift(x, y)); // -20

    WriteLine(Toolkit.ShiftBounds(x, y, 1e-6)); // [-33, -7]
    WriteLine(Toolkit.ShiftBounds(x, y, 1e-5)); // [-32, -8]
    WriteLine(Toolkit.ShiftBounds(x, y, 1e-4)); // [-30, -10]
    WriteLine(Toolkit.ShiftBounds(x, y, 1e-3)); // [-28, -12]
  }
}
```

8.5 Kotlin

Install from Maven Central Repository via Apache Maven:

```
<dependency>
    <groupId>dev.pragmastat</groupId>
    <artifactId>pragmastat</artifactId>
    <version>4.0.1</version>
</dependency>
```

Install from Maven Central Repository via Gradle:

```
implementation 'dev.pragmastat:pragmastat:4.0.1'
```

Install from Maven Central Repository via Gradle (Kotlin):

```
implementation("dev.pragmastat:pragmastat:4.0.1")
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/kt

Pragmastat on Maven Central Repository: https://central.sonatype.com/artifact/dev.pragmastat/pragmastat/ove

Demo:

```
package dev.pragmastat.demo

import dev.pragmastat.*

fun main() {
    var x = listOf(0.0, 2.0, 4.0, 6.0, 8.0)
    println(center(x)) // 4
    println(center(x.map { it + 10 })) // 14
    println(center(x.map { it * 3 })) // 12

    println(spread(x)) // 4
    println(spread(x.map { it + 10 })) // 4
    println(spread(x.map { it * 2 })) // 8

    println(relSpread(x)) // 1
    println(relSpread(x.map { it * 5 })) // 1

    var y = listOf(10.0, 12.0, 14.0, 16.0, 18.0)
    println(shift(x, y)) // -10
    println(shift(x, x)) // 0
    println(shift(x.map { it + 7 }, y.map { it + 3 })) // -6
    println(shift(x.map { it * 2 }, y.map { it * 2 })) // -20
    println(shift(y, x)) // 10

    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(x, x)) // 1
    println(ratio(x.map { it * 2 }, y.map { it * 5 })) // 0.2

    x = listOf(0.0, 3.0, 6.0, 9.0, 12.0)
    y = listOf(0.0, 2.0, 4.0, 6.0, 8.0)
    println(spread(x)) // 6
    println(spread(y)) // 4

    println(avgSpread(x, y)) // 5
    println(avgSpread(x, x)) // 6
    println(avgSpread(x.map { it * 2 }, x.map { it * 3 })) // 15
    println(avgSpread(y, x)) // 5
```

```
    println(avgSpread(x.map { it * 2 }, y.map { it * 2 })) // 10

    println(shift(x, y)) // 2
    println(avgSpread(x, y)) // 5

    println(disparity(x, y)) // 0.4
    println(disparity(x.map { it + 5 }, y.map { it + 5 })) // 0.4
    println(disparity(x.map { it * 2 }, y.map { it * 2 })) // 0.4
    println(disparity(y, x)) // -0.4

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

    println(pairwiseMargin(30, 30, 1e-6)) // 276
    println(pairwiseMargin(30, 30, 1e-5)) // 328
    println(pairwiseMargin(30, 30, 1e-4)) // 390
    println(pairwiseMargin(30, 30, 1e-3)) // 464

    println(shift(x, y)) // -20

    println(shiftBounds(x, y, 1e-6)) // [-33, -7]
    println(shiftBounds(x, y, 1e-5)) // [-32, -8]
    println(shiftBounds(x, y, 1e-4)) // [-30, -10]
    println(shiftBounds(x, y, 1e-3)) // [-28, -12]
}
```

8.6 Rust

Install from crates.io via cargo:

```
cargo add pragmastat@4.0.1
```

Install from crates.io via `Cargo.toml`:

```
[dependencies]
pragmastat = "4.0.1"
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/rs

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

Demo:

```
use pragmastat::*;

fn print(result: Result<f64, &str>) {
    println!("{}", result.unwrap());
}

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

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

fn main() {
    let x = vec![0.0, 2.0, 4.0, 6.0, 8.0];
    print(center(&x)); // 4
    print(center(&add(&x, 10.0))); // 14
    print(center(&multiply(&x, 3.0))); // 12

    print(spread(&x)); // 4
    print(spread(&add(&x, 10.0))); // 4
    print(spread(&multiply(&x, 2.0))); // 8

    print(rel_spread(&x)); // 1
    print(rel_spread(&multiply(&x, 5.0))); // 1

    let y = vec![10.0, 12.0, 14.0, 16.0, 18.0];
    print(shift(&x, &y)); // -10
    print(shift(&x, &x)); // 0
    print(shift(&add(&x, 7.0), &add(&y, 3.0))); // -6
    print(shift(&multiply(&x, 2.0), &multiply(&y, 2.0))); // -20
    print(shift(&y, &x)); // 10

    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(&x, &x)); // 1
    print(ratio(&multiply(&x, 2.0), &multiply(&y, 5.0))); // 0.2

    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(spread(&x)); // 6
    print(spread(&y)); // 4

    print(avg_spread(&x, &y)); // 5
    print(avg_spread(&x, &x)); // 6
    print(avg_spread(&multiply(&x, 2.0), &multiply(&x, 3.0))); // 15
    print(avg_spread(&y, &x)); // 5
    print(avg_spread(&multiply(&x, 2.0), &multiply(&y, 2.0))); // 10

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

    print(disparity(&x, &y)); // 0.4
    print(disparity(&add(&x, 5.0), &add(&y, 5.0))); // 0.4
    print(disparity(&multiply(&x, 2.0), &multiply(&y, 2.0))); // 0.4
```

```
    print(disparity(&y, &x)); // -0.4

    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-6)); // 276
    println!("{}", pairwise_margin(30, 30, 1e-5)); // 328
    println!("{}", pairwise_margin(30, 30, 1e-4)); // 390
    println!("{}", pairwise_margin(30, 30, 1e-3)); // 464

    print(shift(&x, &y)); // -20

    let bounds = shift_bounds(&x, &y, 1e-6).unwrap(); // [-33, -7]
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);

    let bounds = shift_bounds(&x, &y, 1e-5).unwrap(); // [-32, -8]
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);

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

    let bounds = shift_bounds(&x, &y, 1e-3).unwrap(); // [-28, -12]
    println!("{{lower: {}, upper: {}}}", bounds.lower, bounds.upper);
}
```

8.7 Go

Install from GitHub:

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

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v4.0.1/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() {
    x := []float64{0, 2, 4, 6, 8}
    print(pragmastat.Center(x))              // 4
    print(pragmastat.Center(add(x, 10)))     // 14
    print(pragmastat.Center(multiply(x, 3))) // 12

    print(pragmastat.Spread(x))              // 4
    print(pragmastat.Spread(add(x, 10)))     // 4
    print(pragmastat.Spread(multiply(x, 2))) // 8

    print(pragmastat.RelSpread(x))              // 1
    print(pragmastat.RelSpread(multiply(x, 5))) // 1

    y := []float64{10, 12, 14, 16, 18}
    print(pragmastat.Shift(x, y))                           // -10
    print(pragmastat.Shift(x, x))                           // 0
    print(pragmastat.Shift(add(x, 7), add(y, 3)))           // -6
    print(pragmastat.Shift(multiply(x, 2), multiply(y, 2))) // -20
    print(pragmastat.Shift(y, x))                           // 10

    x = []float64{1, 2, 4, 8, 16}
    y = []float64{2, 4, 8, 16, 32}
    print(pragmastat.Ratio(x, y))                           // 0.5
    print(pragmastat.Ratio(x, x))                           // 1
    print(pragmastat.Ratio(multiply(x, 2), multiply(y, 5))) // 0.2

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

```
    print(pragmastat.Spread(x)) // 6
    print(pragmastat.Spread(y)) // 4

    print(pragmastat.AvgSpread(x, y))                              // 5
    print(pragmastat.AvgSpread(x, x))                              // 6
    print(pragmastat.AvgSpread(multiply(x, 2), multiply(x, 3))) // 15
    print(pragmastat.AvgSpread(y, x))                              // 5
    print(pragmastat.AvgSpread(multiply(x, 2), multiply(y, 2))) // 10

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

    print(pragmastat.Disparity(x, y))                              // 0.4
    print(pragmastat.Disparity(add(x, 5), add(y, 5)))              // 0.4
    print(pragmastat.Disparity(multiply(x, 2), multiply(y, 2))) // 0.4
    print(pragmastat.Disparity(y, x))                              // -0.4

    x = []float64{
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
    y = []float64{
        21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50}

    fmt.Println(pragmastat.PairwiseMargin(30, 30, 1e-6)) // 276
    fmt.Println(pragmastat.PairwiseMargin(30, 30, 1e-5)) // 328
    fmt.Println(pragmastat.PairwiseMargin(30, 30, 1e-4)) // 390
    fmt.Println(pragmastat.PairwiseMargin(30, 30, 1e-3)) // 464

    print(pragmastat.Shift(x, y)) // -20

    fmt.Println(must(pragmastat.ShiftBounds(x, y, 1e-6))) // [-33, -7]
    fmt.Println(must(pragmastat.ShiftBounds(x, y, 1e-5))) // [-32, -8]
    fmt.Println(must(pragmastat.ShiftBounds(x, y, 1e-4))) // [-30, -10]
    fmt.Println(must(pragmastat.ShiftBounds(x, y, 1e-3))) // [-28, -12]
}
```

9 Reference Tests

9.1 Motivation

The toolkit maintains seven implementations across different programming languages: Python, TypeScript, R, C#, Kotlin, Rust, and Go. Each implementation must produce identical numerical results for all estimators. Maintaining correctness across this diverse set of languages requires a rigorous reference test suite.

Reference tests serve three critical purposes:

- **Cross-language validation.** All implementations must pass identical test cases, ensuring

consistent behavior regardless of language choice.

- **Regression prevention.** Changes to any implementation can be validated against the reference outputs to detect unintended modifications.
- **Implementation guidance.** The test cases provide concrete examples that guide developers in implementing the toolkit in new languages.

The test design follows established quality assurance principles:

- **Minimal sufficiency.** The test set should be as small as possible while still providing high confidence in correctness. Smaller test suites reduce CI execution time and simplify maintenance.
- **Comprehensive coverage.** Tests must cover both typical use cases and edge cases that expose potential implementation errors.
- **Deterministic reproducibility.** All random test cases use fixed seeds to ensure identical results across all platforms and implementations.

The test suite balances three categories:

- **Canonical cases** use deterministic, easily verified inputs like natural number sequences. These provide intuitive examples where correct outputs can be validated by inspection.
- **Edge cases** test boundary conditions such as single-element samples, zero values, and minimum viable sample sizes. These expose off-by-one errors, division by zero, and other common implementation mistakes.
- **Fuzzy tests** use controlled random number generation to explore the input space beyond hand-crafted examples. Random tests catch issues that might not be apparent from simple deterministic cases.

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 that their estimators produce matching results within a given numerical tolerance.

9.2 Center Tests

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm stress tests — edge cases for fast algorithm implementation:

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

Extreme values — numerical stability and range tests:

- `extreme-large-5`: $\mathbf{x} = (1e8, 2e8, 3e8, 4e8, 5e8)$ (very large values)
- `extreme-small-5`: $\mathbf{x} = (1e{-}8, 2e{-}8, 3e{-}8, 4e{-}8, 5e{-}8)$ (very small positive values)
- `extreme-wide-5`: $\mathbf{x} = (0.001, 1, 100, 1000, 1000000)$ (wide range, tests precision)

Unsorted tests — verify sorting correctness (14 tests):

- `unsorted-reverse-{n}` for $n \in \{2, 3, 4, 5, 7\}$: reverse sorted natural sequences (5 tests)
- `unsorted-shuffle-3`: $\mathbf{x} = (2, 1, 3)$ (middle element first)
- `unsorted-shuffle-4`: $\mathbf{x} = (3, 1, 4, 2)$ (interleaved)
- `unsorted-shuffle-5`: $\mathbf{x} = (5, 2, 4, 1, 3)$ (complex shuffle)
- `unsorted-last-first-5`: $\mathbf{x} = (5, 1, 2, 3, 4)$ (last moved to first)
- `unsorted-first-last-5`: $\mathbf{x} = (2, 3, 4, 5, 1)$ (first moved to last)
- `unsorted-duplicates-mixed-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$ (all identical, any order)
- `unsorted-duplicates-unsorted-10`: $\mathbf{x} = (3, 1, 2, 3, 1, 3, 2, 1, 3, 2)$ (duplicates mixed)
- `unsorted-extreme-large-unsorted-5`: $\mathbf{x} = (5e8, 1e8, 4e8, 2e8, 3e8)$ (large values unsorted)
- `unsorted-parity-odd-reverse-7`: $\mathbf{x} = (7, 6, 5, 4, 3, 2, 1)$ (odd size reverse)

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

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

- **Input**: $\mathbf{x} = (1, 2, 3, \ldots, 100000)$

- **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.

9.3 Spread Tests

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

The Spread 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: 4 (location invariance)
- `demo-3`: $\mathbf{x} = (0, 4, 8, 12, 16)$ (= 2 × demo-1), expected output: 8 (scale equivariance)

Natural sequences ($n = 1, 2, 3, 4$):

- `natural-1`: $\mathbf{x} = (1)$, expected output: 0 (single element has zero dispersion)
- `natural-2`: $\mathbf{x} = (1, 2)$, expected output: 1
- `natural-3`: $\mathbf{x} = (1, 2, 3)$, expected output: 1
- `natural-4`: $\mathbf{x} = (1, 2, 3, 4)$, expected output: 1.5 (smallest even size with rich structure)

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

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

Zero values ($n = 1, 2$):

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

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

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

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

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

The natural sequence cases validate the basic pairwise difference calculation. The zero cases confirm that constant samples correctly produce zero spread.

Algorithm stress tests — edge cases for fast algorithm implementation:

- `duplicates-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$ (all identical, expected output: 0)
- `duplicates-10`: $\mathbf{x} = (1, 1, 1, 2, 2, 2, 3, 3, 3, 3)$ (many duplicates, stress tie-breaking)
- `parity-odd-7`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$ (odd sample size, 21 differences)
- `parity-even-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6)$ (even sample size, 15 differences)
- `parity-odd-49`: 49-element sequence $(1, 2, \ldots, 49)$ (large odd, 1176 differences)

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

Extreme values — numerical stability and range tests:

- `extreme-large-5`: $\mathbf{x} = (1e8, 2e8, 3e8, 4e8, 5e8)$ (very large values)
- `extreme-small-5`: $\mathbf{x} = (1e{-}8, 2e{-}8, 3e{-}8, 4e{-}8, 5e{-}8)$ (very small positive values)
- `extreme-wide-5`: $\mathbf{x} = (0.001, 1, 100, 1000, 1000000)$ (wide range, tests precision)

Unsorted tests — verify sorting correctness (14 tests):

- `unsorted-reverse-{n}` for $n \in \{2, 3, 4, 5, 7\}$: reverse sorted natural sequences (5 tests)
- `unsorted-shuffle-3`: $\mathbf{x} = (3, 1, 2)$ (rotated)
- `unsorted-shuffle-4`: $\mathbf{x} = (4, 2, 1, 3)$ (mixed order)
- `unsorted-shuffle-5`: $\mathbf{x} = (5, 1, 3, 2, 4)$ (partial shuffle)
- `unsorted-last-first-5`: $\mathbf{x} = (5, 1, 2, 3, 4)$ (last moved to first)
- `unsorted-first-last-5`: $\mathbf{x} = (2, 3, 4, 5, 1)$ (first moved to last)
- `unsorted-duplicates-mixed-5`: $\mathbf{x} = (3, 3, 3, 3, 3)$ (all identical)
- `unsorted-duplicates-unsorted-10`: $\mathbf{x} = (2, 3, 1, 3, 2, 1, 2, 3, 1, 3)$ (duplicates mixed)
- `unsorted-extreme-wide-unsorted-5`: $\mathbf{x} = (1000, 0.001, 1000000, 100, 1)$ (wide range unsorted)
- `unsorted-negative-unsorted-5`: $\mathbf{x} = (-1, -5, -2, -4, -3)$ (negative unsorted)

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

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

- **Input**: $\mathbf{x} = (1, 2, 3, \ldots, 100000)$
- **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.

9.4 RelSpread Tests

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

The RelSpread test suite contains 25 test cases (15 original + 10 unsorted) focusing on relative dispersion.

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

- `demo-1`: $\mathbf{x} = (0, 2, 4, 6, 8)$, expected output: 1 (base case)
- `demo-2`: $\mathbf{x} = (0, 10, 20, 30, 40)$ ($= 5 \times$ demo-1), expected output: 1 (scale invariance)

Natural sequences ($n = 1, 2, 3, 4$):

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

- `natural-2`: $\mathbf{x} = (1, 2)$, expected output: ≈ 0.667
- `natural-3`: $\mathbf{x} = (1, 2, 3)$, expected output: 0.5
- `natural-4`: $\mathbf{x} = (1, 2, 3, 4)$, expected output: 0.6 (validates composite with even size)

Negative values ($n = 3$) — validates absolute value in denominator:

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

Uniform distribution ($n = 5, 10, 20, 30, 100$) — $\underline{\text{Uniform}}(0, 1)$:

- `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. The absence of zero-value tests reflects the domain constraint requiring $\text{Center}(\mathbf{x}) \neq 0$.

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

- `composite-small-center`: $\mathbf{x} = (0.001, 0.002, 0.003, 0.004, 0.005)$ (small center, tests division stability)
- `composite-large-spread`: $\mathbf{x} = (1, 100, 200, 300, 1000)$ (large spread relative to center)
- `composite-extreme-ratio`: $\mathbf{x} = (1, 1.0001, 1.0002, 1.0003, 1.0004)$ (tiny spread, tests precision)

Unsorted tests — verify sorting for composite estimator (10 tests):

- `unsorted-reverse-{n}` for $n \in \{3, 4, 5\}$: reverse sorted natural sequences (3 tests)
- `unsorted-shuffle-4`: $\mathbf{x} = (4, 1, 3, 2)$ (mixed order)
- `unsorted-shuffle-5`: $\mathbf{x} = (5, 3, 1, 4, 2)$ (complex shuffle)
- `unsorted-negative-unsorted-3`: $\mathbf{x} = (-1, -3, -2)$ (negative unsorted)
- `unsorted-demo-unsorted-5`: $\mathbf{x} = (8, 0, 4, 2, 6)$ (demo case unsorted)
- `unsorted-composite-small-unsorted`: $\mathbf{x} = (0.005, 0.001, 0.003, 0.002, 0.004)$ (small center unsorted)
- `unsorted-composite-large-unsorted`: $\mathbf{x} = (1000, 1, 300, 100, 200)$ (large spread unsorted)
- `unsorted-extreme-ratio-unsorted-4`: $\mathbf{x} = (1.0003, 1, 1.0002, 1.0001)$ (extreme ratio unsorted)

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

9.5 Shift Tests

$$\text{Shift}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n,\ 1 \le j \le m}{\text{Median}} \left(x_i - y_j\right)$$

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

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

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

- `demo-4`: $\mathbf{x} = (0, 4, 8, 12, 16)$, $\mathbf{y} = (20, 24, 28, 32, 36)$ (= 2 × demo-1), expected output: -20 (scale equivariance)
- `demo-5`: $\mathbf{x} = (10, 12, 14, 16, 18)$, $\mathbf{y} = (0, 2, 4, 6, 8)$ (= reversed demo-1), expected output: 10 (anti-symmetry)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `parity-asymmetric-7-6`: $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6)$ (mixed parity, 42 differences)
- `parity-large-49-50`: $\mathbf{x} = (1, 2, \ldots, 49)$, $\mathbf{y} = (1, 2, \ldots, 50)$ (large asymmetric, 2450 differences)

Extreme asymmetry — tests with very unbalanced sample sizes:

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

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

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

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

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

- **Input**: $\mathbf{x} = (1, 2, 3, \ldots, 100000)$, $\mathbf{y} = (1, 2, 3, \ldots, 100000)$
- **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.

9.6 Ratio Tests

$$\text{Ratio}(\mathbf{x}, \mathbf{y}) = \underset{1 \le i \le n, 1 \le j \le m}{\text{Median}} \left(\frac{x_i}{y_j} \right)$$

The Ratio test suite contains 37 test cases (25 original + 12 unsorted), excluding zero values due to division constraints.

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

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

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

- `natural-1-1`: $\mathbf{x} = (1)$, $\mathbf{y} = (1)$, expected output: 1
- `natural-1-2`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2)$, expected output: ≈ 0.667
- `natural-1-3`: $\mathbf{x} = (1)$, $\mathbf{y} = (1, 2, 3)$, expected output: 0.5
- `natural-2-1`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1)$, expected output: 1.5
- `natural-2-2`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2)$, expected output: 1
- `natural-2-3`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (1, 2, 3)$, expected output: ≈ 0.833
- `natural-3-1`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1)$, expected output: 2
- `natural-3-2`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2)$, expected output: 1.5
- `natural-3-3`: $\mathbf{x} = (1, 2, 3)$, $\mathbf{y} = (1, 2, 3)$, expected output: 1

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

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

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

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

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

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

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

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

9.7 AvgSpread Tests

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

The AvgSpread test suite contains 49 test cases (35 original + 14 unsorted). Since AvgSpread computes Spread($\mathbf{x}$) and Spread($\mathbf{y}$) 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`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (0, 2, 4, 6, 8)$, expected output: 5 (base case: $(5 \cdot 6 + 5 \cdot 4)/10$)
- `demo-2`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (0, 3, 6, 9, 12)$, expected output: 6 (identity case)
- `demo-3`: $\mathbf{x} = (0, 6, 12, 18, 24)$, $\mathbf{y} = (0, 9, 18, 27, 36)$ (= [2×demo-1.x, 3×demo-1.y]), expected output: 15 (scale equivariance)
- `demo-4`: $\mathbf{x} = (0, 2, 4, 6, 8)$, $\mathbf{y} = (0, 3, 6, 9, 12)$ (= reversed demo-1), expected output: 5 (symmetry)
- `demo-5`: $\mathbf{x} = (0, 6, 12, 18, 24)$, $\mathbf{y} = (0, 4, 8, 12, 16)$ (= 2 × demo-1), expected output: 10 (uniform scaling)

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

- All combinations from single-element to three-element samples, validating the weighted average calculation

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

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

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

- All produce output 0 since Spread of constant samples is zero

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

- Tests pooled dispersion across different sample size combinations
- Random generation: $\mathbf{x}$ uses seed 0, $\mathbf{y}$ uses seed 1

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

- Validates correct weighting when sample sizes differ substantially
- Random generation: $\mathbf{x}$ uses seed 2, $\mathbf{y}$ 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`: $\mathbf{x} = (1, 2)$, $\mathbf{y} = (3, 4, 5, 6, 7, 8, 9, 10)$ (2 vs 8, tests weighting formula)
- `composite-zero-spread-one`: $\mathbf{x} = (5, 5, 5)$, $\mathbf{y} = (1, 2, 3, 4, 5)$ (one zero spread, tests edge case)
- `composite-extreme-sizes`: $\mathbf{x} = (10)$, $\mathbf{y} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ (1 vs 10, extreme weighting)

Unsorted tests — critical for verifying independent sorting (14 tests):

- `unsorted-x-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X unsorted (reversed), Y sorted (2 tests)
- `unsorted-y-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: X sorted, Y unsorted (reversed) (2 tests)
- `unsorted-both-natural-{n}-{m}` for $(n, m) \in \{(3, 3), (4, 4)\}$: both unsorted (reversed) (2 tests)
- `unsorted-demo-unsorted-x`: $\mathbf{x} = (12, 0, 6, 3, 9)$, $\mathbf{y} = (0, 2, 4, 6, 8)$ (demo-1 with X unsorted)
- `unsorted-demo-unsorted-y`: $\mathbf{x} = (0, 3, 6, 9, 12)$, $\mathbf{y} = (8, 0, 4, 2, 6)$ (demo-1 with Y unsorted)
- `unsorted-demo-both-unsorted`: $\mathbf{x} = (9, 0, 12, 3, 6)$, $\mathbf{y} = (6, 0, 8, 2, 4)$ (demo-1 both unsorted)
- `unsorted-identity-unsorted`: $\mathbf{x} = (6, 0, 12, 3, 9)$, $\mathbf{y} = (9, 0, 12, 6, 3)$ (demo-2 unsorted)
- `unsorted-negative-unsorted`: $\mathbf{x} = (-1, -2)$, $\mathbf{y} = (-1, -2)$ (negative unsorted)
- `unsorted-zero-unsorted-2-2`: $\mathbf{x} = (0, 0)$, $\mathbf{y} = (0, 0)$ (zeros, any order)
- `unsorted-asymmetric-weights-unsorted`: $\mathbf{x} = (2, 1)$, $\mathbf{y} = (8, 3, 6, 4, 10, 5, 9, 7)$ (asymmetric unsorted)
- `unsorted-zero-spread-x-unsorted`: $\mathbf{x} = (5, 5, 5)$, $\mathbf{y} = (5, 1, 4, 2, 3)$ (zero spread X, Y unsorted)

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

9.8 Disparity Tests

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

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

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

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

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

- `natural-2-2`, `natural-2-3`, `natural-3-2`, `natural-3-3`

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

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

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

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

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

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

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

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

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

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

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

9.9 PairwiseMargin Tests

$$\text{PairwiseMargin}(n, m, \text{misrate})$$

The PairwiseMargin test suite contains 346 correctness test cases (4 demo + 32 natural + 10 edge + 300 comprehensive 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) — 32 tests:

- Misrate values: misrate $\in \{10^{-1}, 10^{-2}\}$
- Test naming: `natural-{n}-{m}-mr{k}` where k is the negative log10 of misrate
- Examples:
  - `natural-1-1-mr1`: $n = 1$, $m = 1$, misrate $= 0.1$, expected output: 0
  - `natural-2-2-mr1`: $n = 2$, $m = 2$, misrate $= 0.1$, expected output: 0
  - `natural-3-3-mr2`: $n = 3$, $m = 3$, misrate $= 0.01$, expected output: 0
  - `natural-4-4-mr1`: $n = 4$, $m = 4$, misrate $= 0.1$, expected output: 4

The natural sequences provide canonical examples with small, easily verified parameter values.

Edge cases — boundary condition validation:

- `boundary-min`: $n = 1$, $m = 1$, misrate $= 0.5$ (minimum samples, expected output: 0)
- `boundary-zero-margin-small`: $n = 2$, $m = 2$, misrate $= 10^{-6}$ (misrate too strict, expected output: 0)
- `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) — 150 tests:

- Misrate values: misrate $\in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$
- Test naming: `n{n}_m{m}_mr{k}` where k is the negative log10 of misrate
- Examples:
  - `n1_m1_mr1`: $n = 1$, $m = 1$, misrate $= 0.1$, expected output: 0
  - `n5_m5_mr1`: $n = 5$, $m = 5$, misrate $= 0.1$, expected output: 10
  - `n5_m5_mr3`: $n = 5$, $m = 5$, misrate $= 0.001$, expected output: 0

Large sample combinations ($[n, m] \in \{10, 20, 30, 50, 100\} \times \{10, 20, 30, 50, 100\} \times 6$ misrates) — 150 tests:

- Misrate values: same as small samples
- 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
- `n10_m10_r6`: $n = 10$, $m = 10$, misrate $= 10^{-6}$, expected output: 0
- `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.

9.10 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`. Output is a single numeric value.

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.

10 Artifacts

Manual:

- PDF: pragmastat-v4.0.1.pdf
- Markdown: pragmastat-v4.0.1.md
- Website: web-v4.0.1.zip

Implementations:

- Python: py-v4.0.1.zip
- TypeScript: ts-v4.0.1.zip
- R: r-v4.0.1.zip
- C#: cs-v4.0.1.zip
- Kotlin: kt-v4.0.1.zip
- Rust: rs-v4.0.1.zip
- Go: go-v4.0.1.zip

Data:

- Reference tests (json): tests-v4.0.1.zip
- Reference simulations (json): sim-v4.0.1.zip

Source code:

- pragmastat-4.0.1.zip

Hodges, J. L., and E. L. Lehmann. 1963. "Estimates of Location Based on Rank Tests." *The Annals of Mathematical Statistics* 34 (2): 598–611. https://doi.org/10.1214/aoms/1177704172.

Monahan, John F. 1984. "Algorithm 616: Fast Computation of the Hodges-Lehmann Location Estimator." *ACM Transactions on Mathematical Software* 10 (3): 265–70. https://doi.org/10.1145/1271.319414.

Sen, Pranab Kumar. 1963. "On the Estimation of Relative Potency in Dilution (-Direct) Assays by Distribution-Free Methods." *Biometrics* 19 (4): 532. https://doi.org/10.2307/2527532.

Shamos, Michael Ian. 1976. "Geometry and Statistics: Problems at the Interface."