Pragmastat: Pragmatic Statistical Toolkit

Andrey Akinshin
andrey.akinshin@gmail.com

Version 3.1.23
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 **3**

1.1 Primer . . . 3

1.2 Breaking changes . . . 4

1.3 Definitions . . . 5

2 Summary Estimators **5**

2.1 Center . . . 5

2.2 Spread . . . 6

2.3 RelSpread . . . 6

2.4 Shift . . . 6

2.5 Ratio . . . 7

2.6 AvgSpread . . . 7

2.7 Disparity ('robust effect size') . . . 8

3 Distributions **8**

3.1 Additive ('Normal') . . . 8

3.2 Multiplic ('LogNormal') . . . 9

3.3 Exponential . . . 10

3.4 Power ('Pareto') . . . 11

3.5 Uniform . . . 12

4 Summary Estimator Properties **13**

4.1 Breakdown . . . 14

4.2 Drift . . . 14

4.3 Invariance . . . 21

5 Methodology **22**

5.1 Desiderata . . . 22

5.2 From Assumptions to Conditions . . . 22

5.3 From Statistical Efficiency to Drift . . . 23

6 Algorithms **24**

6.1 Fast Center Algorithm . . . 25

6.2 Fast Spread Algorithm . . . 30

7 Studies **36**

7.1 Additive ('Normal') Distribution . . . 36

8 Reference Implementations **41**

8.1 Python . . . 41

8.2 TypeScript . . . 42

8.3 R . . . 43

8.4 .NET . . . 44

8.5 Kotlin . . . 46

8.6 Rust . . . 47

8.7 Go . . . 48

1 Introduction

1.1 Primer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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, .NET, 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 permanently 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 breaks from the traditions, offering a coherent system designed for clarity and practical use.

- Renamed distributions:
  - Additive (former 'Normal' or 'Gaussian')
  - Multiplic (former 'Log-Normal' or 'Galton')
  - Power (former 'Pareto')
- Primary measure of average: Center (instead of Mean)
- Primary measure of dispersion: Spread (instead of StdDev)
- Primary measure of effect size: Disparity (instead of Cohen's d)
- Reworked statistical efficiency (see section "Drift")

1.3 Definitions

- X, Y: random variables, can be treated as generators of random real measurements
  - $X \sim \underline{\text{Distribution}}$ defines a distribution from which this variable comes
- x_i, y_j: specific individual measurements
- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{y} = (y_1, y_2, \ldots, y_m)$: samples of measurements of a given size
  - Samples are non-empty: $n, m \geq 1$
- $x_{(1)}, x_{(2)}, \ldots, x_{(n)}$: 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

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

- Measures average (central tendency, measure of location)
- Equals the *Hodges-Lehmann estimator* ([HL63], [Sen63]), 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 $n = 1$, $\text{Spread}(\mathbf{x}) = 0$
- Equals the *Shamos scale estimator* ([Sha76]), renamed to Spread for clarity
- Pragmatic alternative to the standard deviation and the median absolute deviation
- Asymptotically, $\text{Spread}[X]$ is the median of the absolute difference 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

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

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

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

2.3 RelSpread

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

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

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

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

2.4 Shift

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

- Measures the median of pairwise differences between elements of two samples
- Equals the *Hodges-Lehmann estimator* for two samples ([HL63])
- Asymptotically, $\text{Shift}[X, Y]$ is the median of the difference of random measurements from X and Y
- Domain: any real numbers
- Unit: the same as measurements

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

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

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

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

2.5 Ratio

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

- Measures the median of pairwise ratios between elements of two samples
- Asymptotically, $\operatorname{Ratio}[X, Y]$ is the median of the ratio of random measurements from X and Y
- Note: $\operatorname{Ratio}(\mathbf{x}, \mathbf{y}) \neq 1 / \operatorname{Ratio}(\mathbf{y}, \mathbf{x})$ 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

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

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

2.6 AvgSpread

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

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

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

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

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

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

2.7 Disparity ('robust effect size')

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

- 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: $\text{AvgSpread}(\mathbf{x}, \mathbf{y}) > 0$
- Unit: spread unit

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

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

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

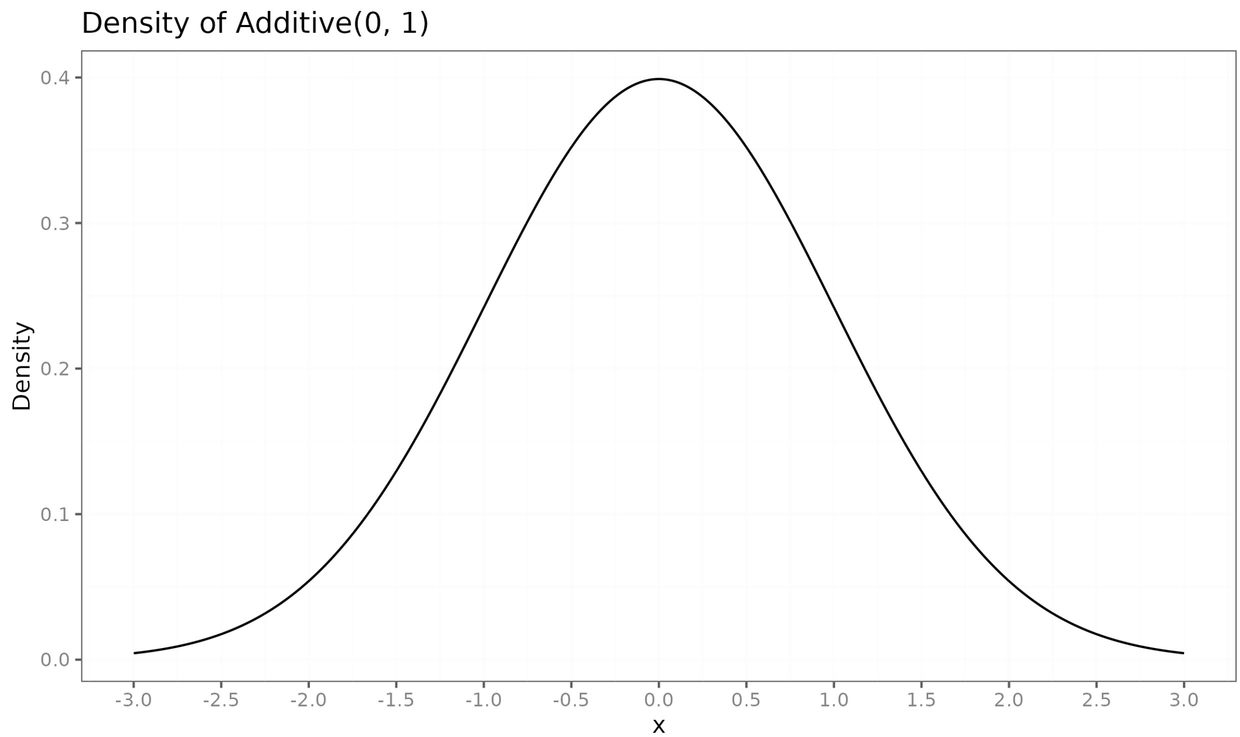
3 Distributions

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

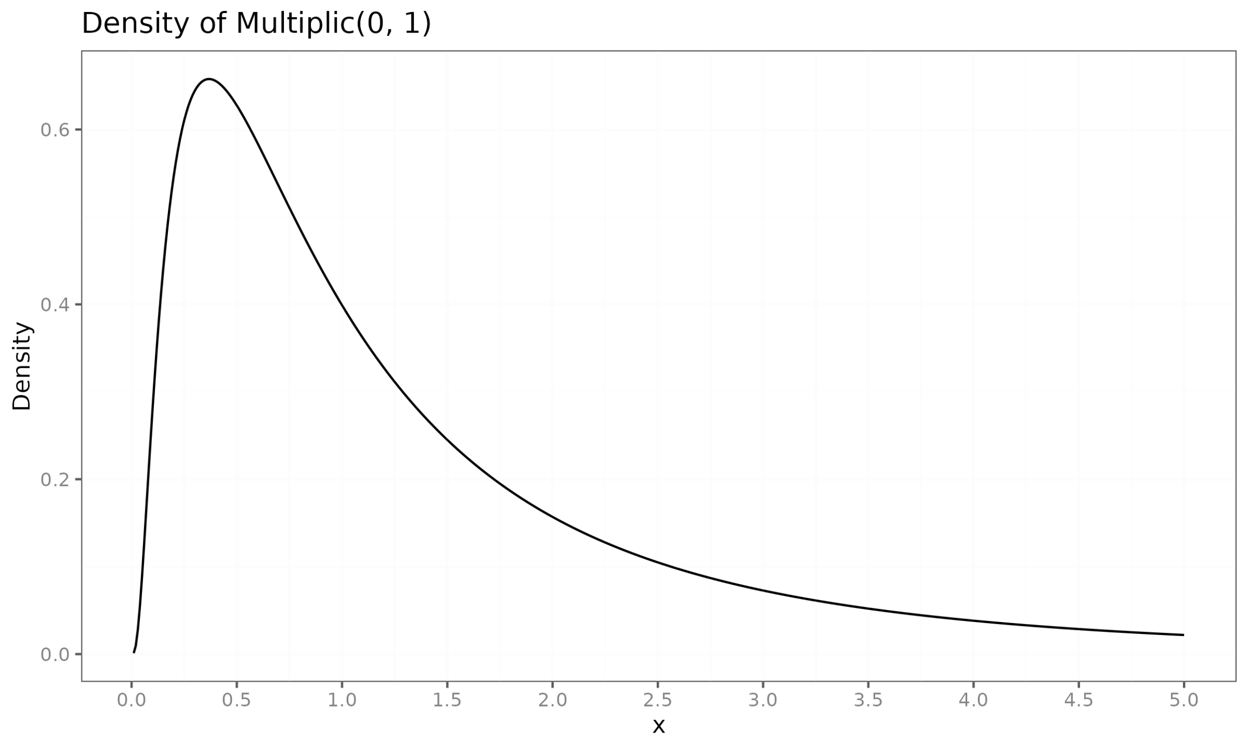


- **Formation:** the sum of many variables $X_1 + X_2 + \ldots + X_n$ under mild CLT (Central Limit Theorem) conditions (e.g., Lindeberg-Feller).
- **Origin:** historically called 'Normal' or 'Gaussian' distribution after Carl Friedrich Gauss and others.
- **Rename Motivation:** renamed to Additive to 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; all real data 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')

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

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

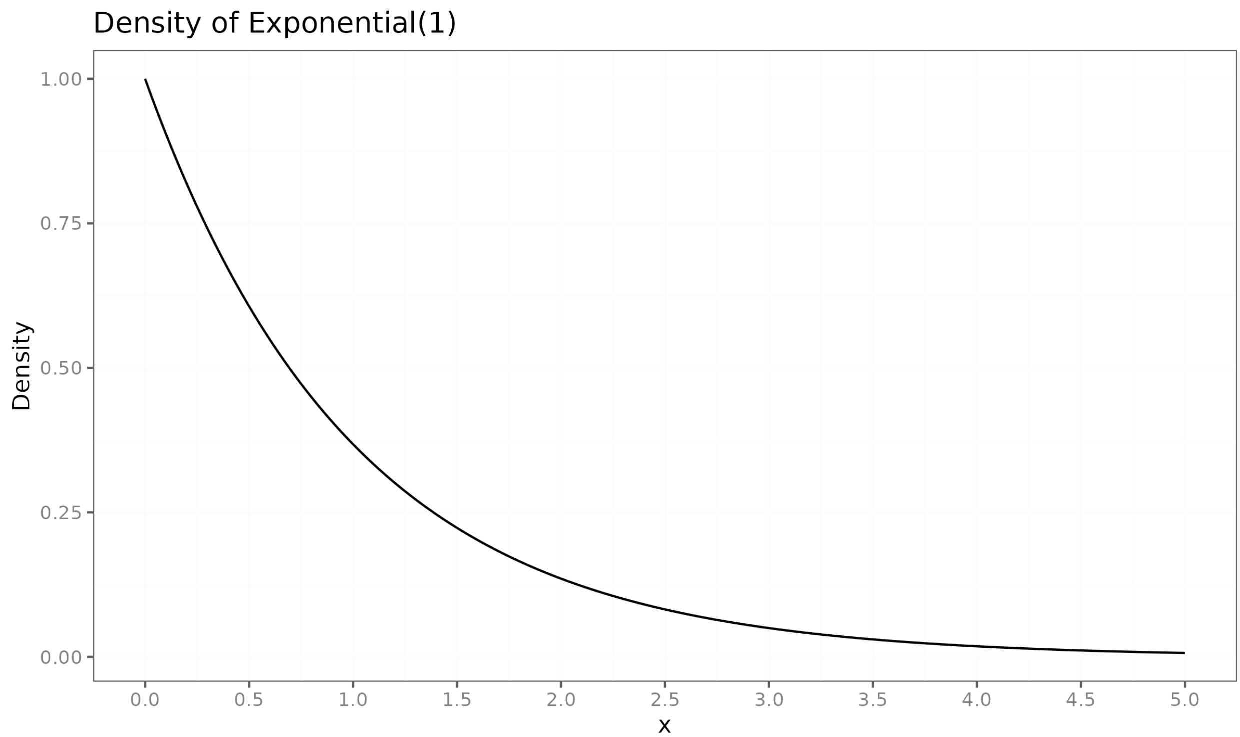


- **Formation:** the product of many positive variables $X_1 \cdot X_2 \cdot \ldots \cdot X_n$ with mild conditions (e.g., finite variance of $\log X$).
- **Origin:** historically called 'Log-Normal' or 'Galton' distribution after Francis Galton.
- **Rename Motivation:** renamed to Multiplic to reflect 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; all real data contain some deviations. Traditional estimators may struggle with the inherent skewness and heavy right tail.

3.3 Exponential

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

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

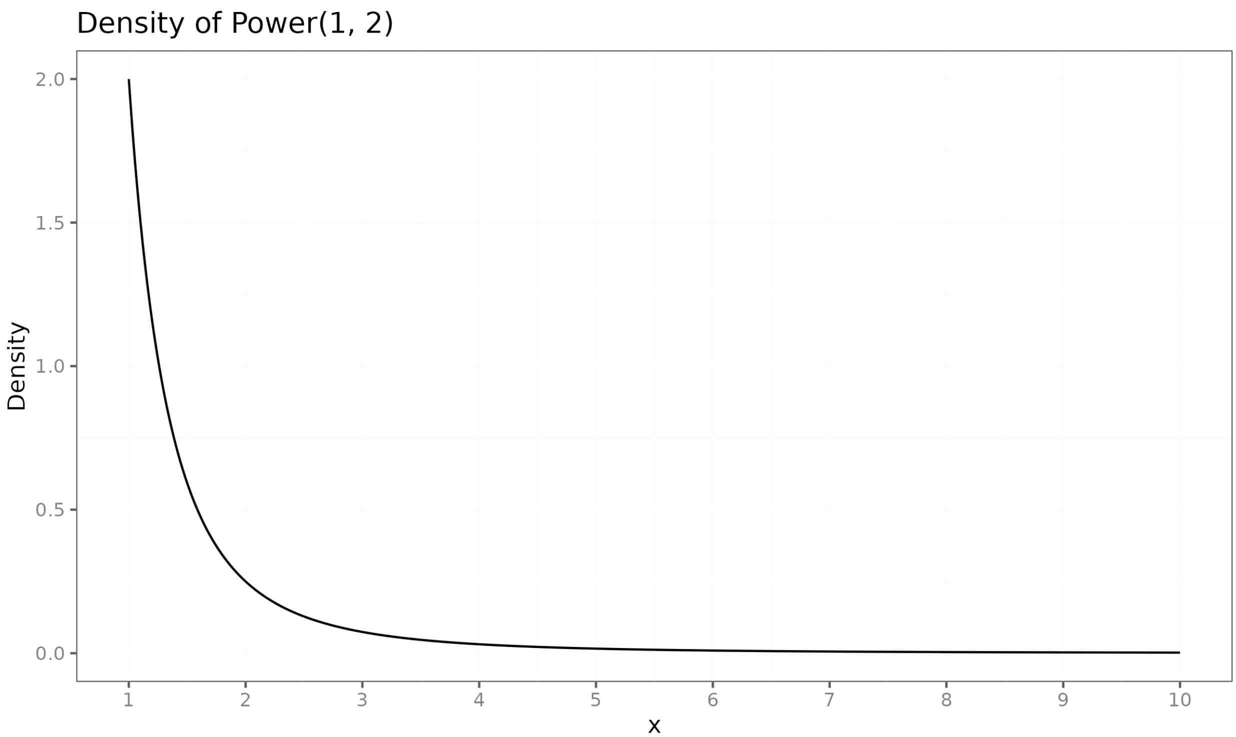


- **Formation:** the waiting time between events in a Poisson process.
- **Origin:** naturally arises from memoryless processes where the probability of an event occurring is constant over time.
- **Properties:** memoryless (past events do not affect future probabilities).
- **Applications:** time between failures, waiting times in queues, radioactive decay, customer service times.
- **Characteristics:** always positive, right-skewed with 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)

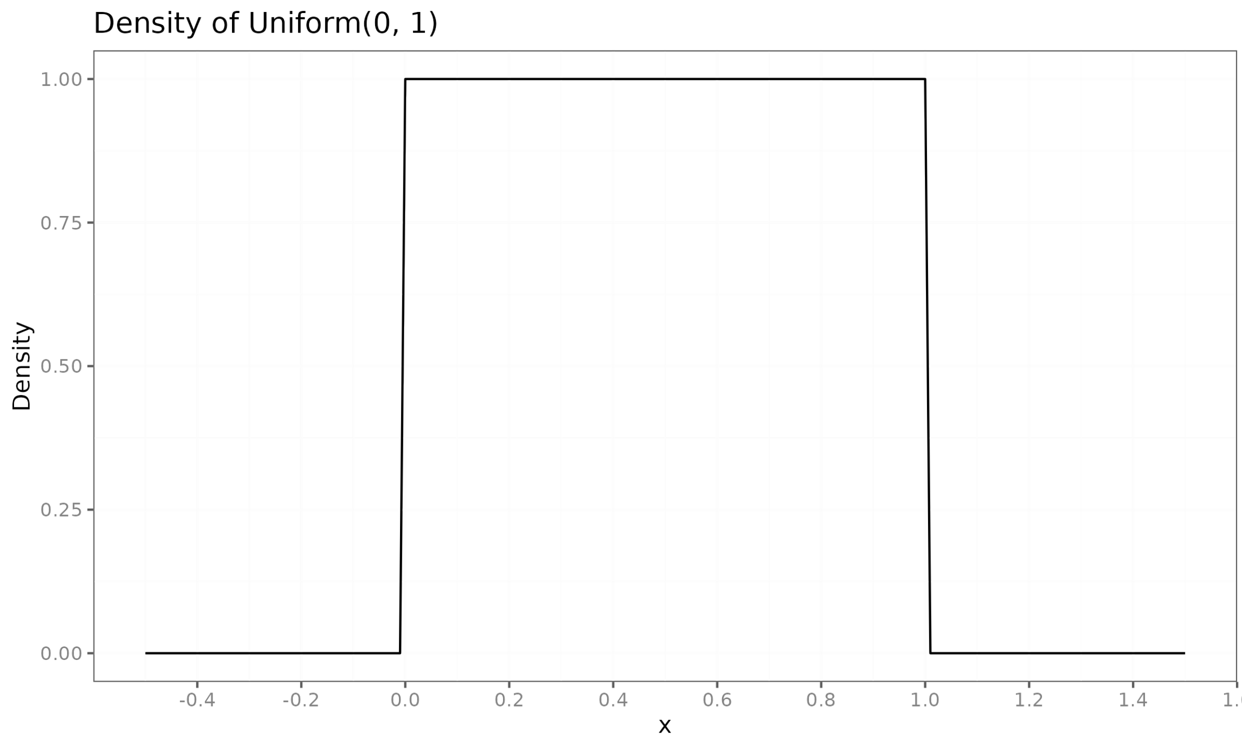


- **Formation:** follows a power-law relationship where large values are rare but possible.
- **Origin:** historically called 'Pareto' distribution after Vilfredo Pareto's work on wealth distribution.
- **Rename Motivation:** renamed to Power to reflect connection with power-law.
- **Properties:** exhibits 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 common.
- **Caution:** traditional variance-based estimators completely fail; robust estimators essential for reliable analysis.

3.5 Uniform

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

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



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

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 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:

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

Median Absolute Deviation (around the median):

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

Spread (Shamos scale estimator):

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

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 also contains corrupted measurements from instrument failures, recording errors, or transmission problems. Both natural extremes and data corruption create the same challenge: how to extract reliable information when some measurements are too influential.

The breakdown point is the fraction of the sample that can be replaced by arbitrarily large values without making the 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; a more sophisticated approach is needed.

Even 50% is rarely needed in practice; more conservative breakdown points also cover practical needs. Also, when the breakdown point is high, the precision is low (we lose information by neglecting part of the data). The optimal practical breakdown point should be somewhere 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 Spread of the estimates, and therefore has a breakdown point of $\approx 29\%$.

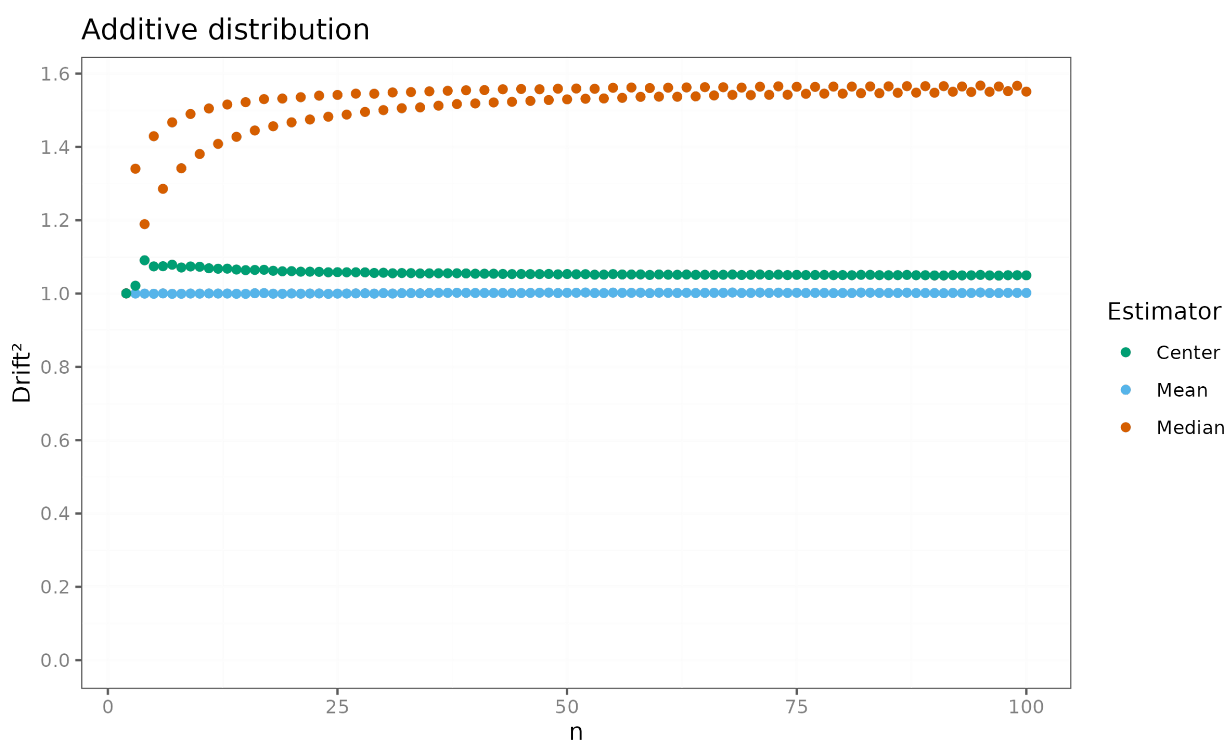
Drift is useful when comparing precisions of several estimators. To simplify the comparison, it is convenient to choose one of the estimators as a baseline. A table with drift squares normalized by the baseline shows the sample adjustment factor for switching from the baseline to another estimator. For example, if Center is the baseline, and the rescaled drift square of Median is 1.5, this means that Median would require 1.5 times more data than Center to match in precision. See the "From Statistical Efficiency to Drift" section for details.

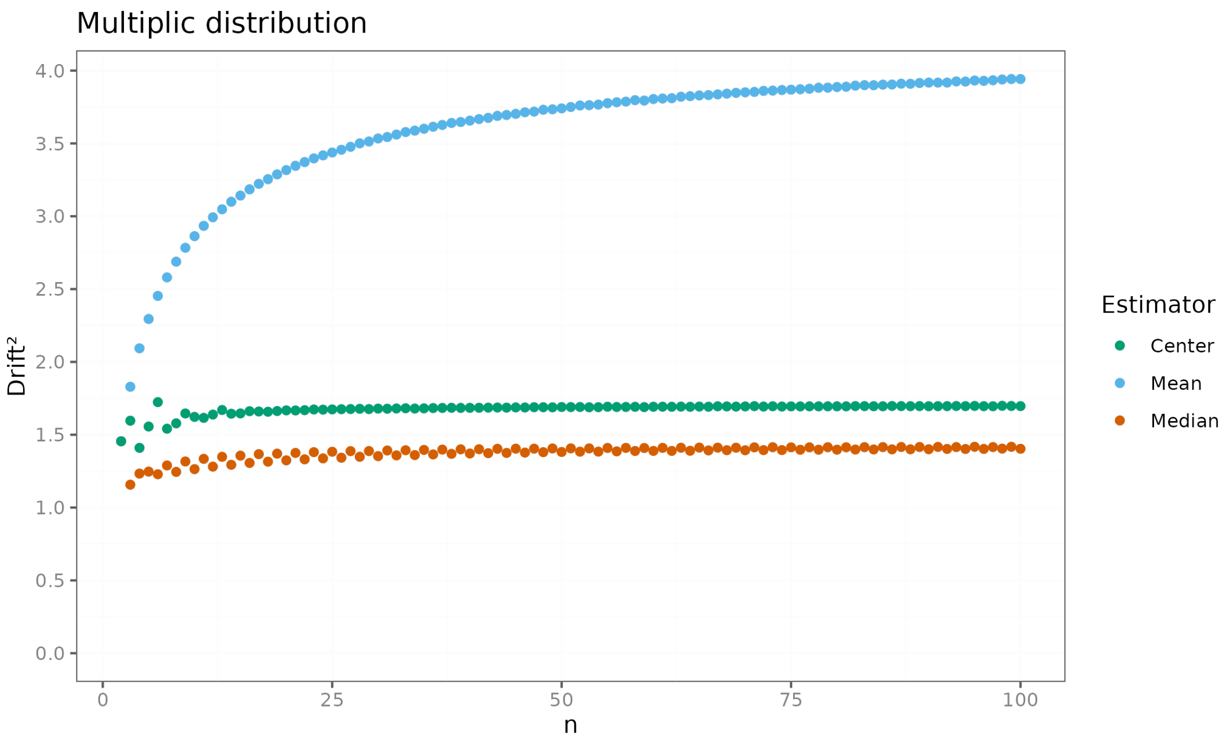
Asymptotic Average estimator drift² (values are approximated):

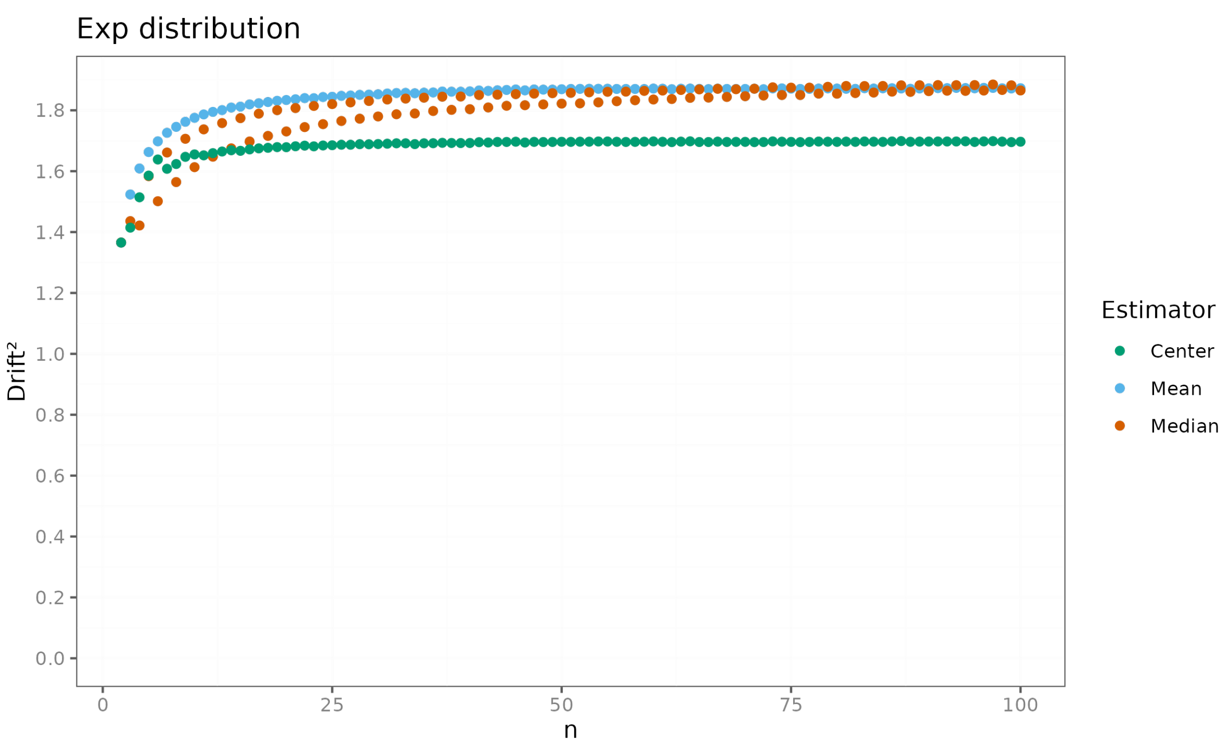
	Mean	Median	Center
Additive	1.0	1.571	1.047
Multiplic	3.95	1.40	1.7
Exp	1.88	1.88	1.69
Power	∞	0.9	2.1
Uniform	0.88	2.60	0.94

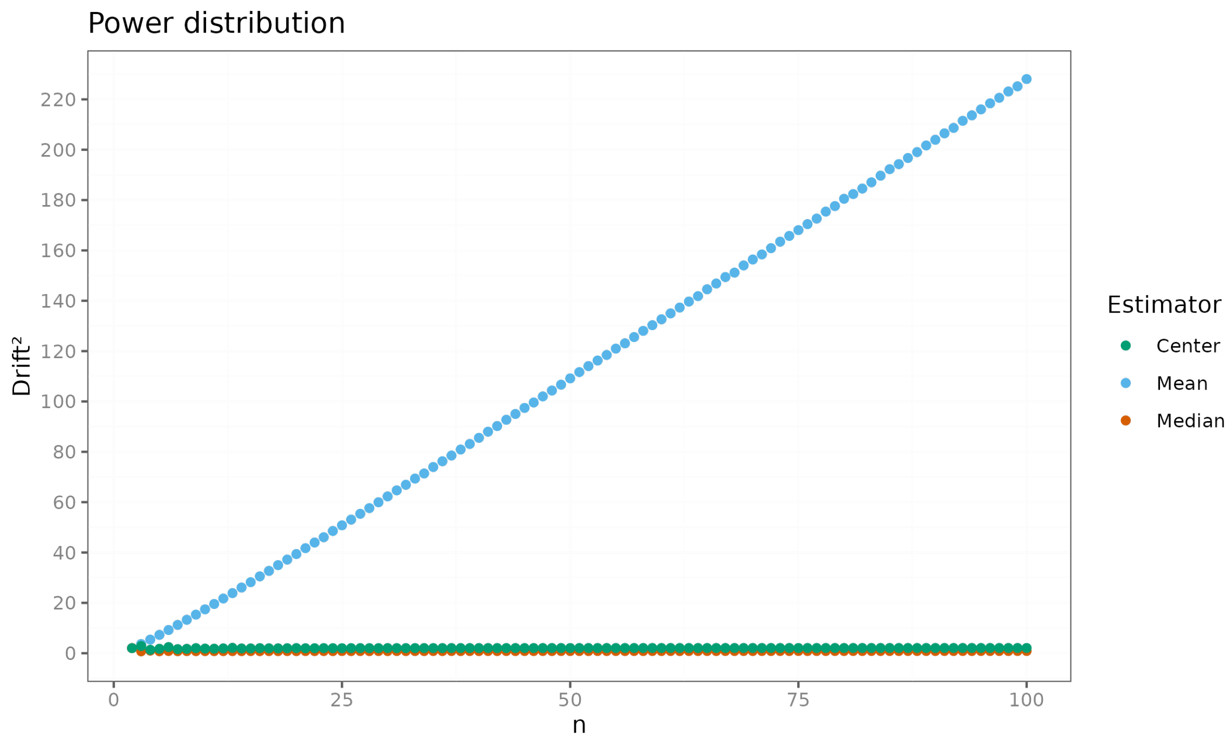
Rescaled to Center (sample size adjustment factors):

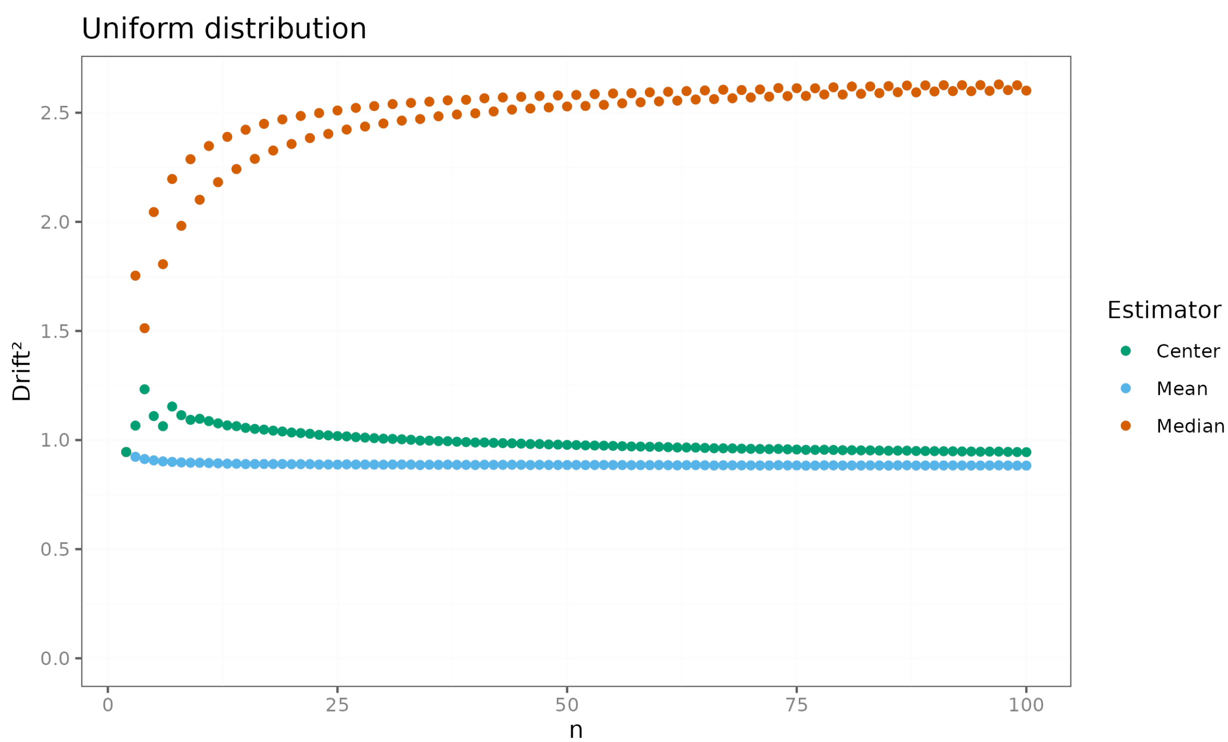
	Mean	Median	Center
Additive	0.96	1.50	1.0
Multiplic	2.32	0.82	1.0
Exp	1.11	1.11	1.0
Power	∞	0.43	1.0
Uniform	0.936	2.77	1.0




Multiplic distribution
Drift²
n
Estimator
Center
Mean
Median


Exp distribution
Drift²
n
Estimator
Center
Mean
Median



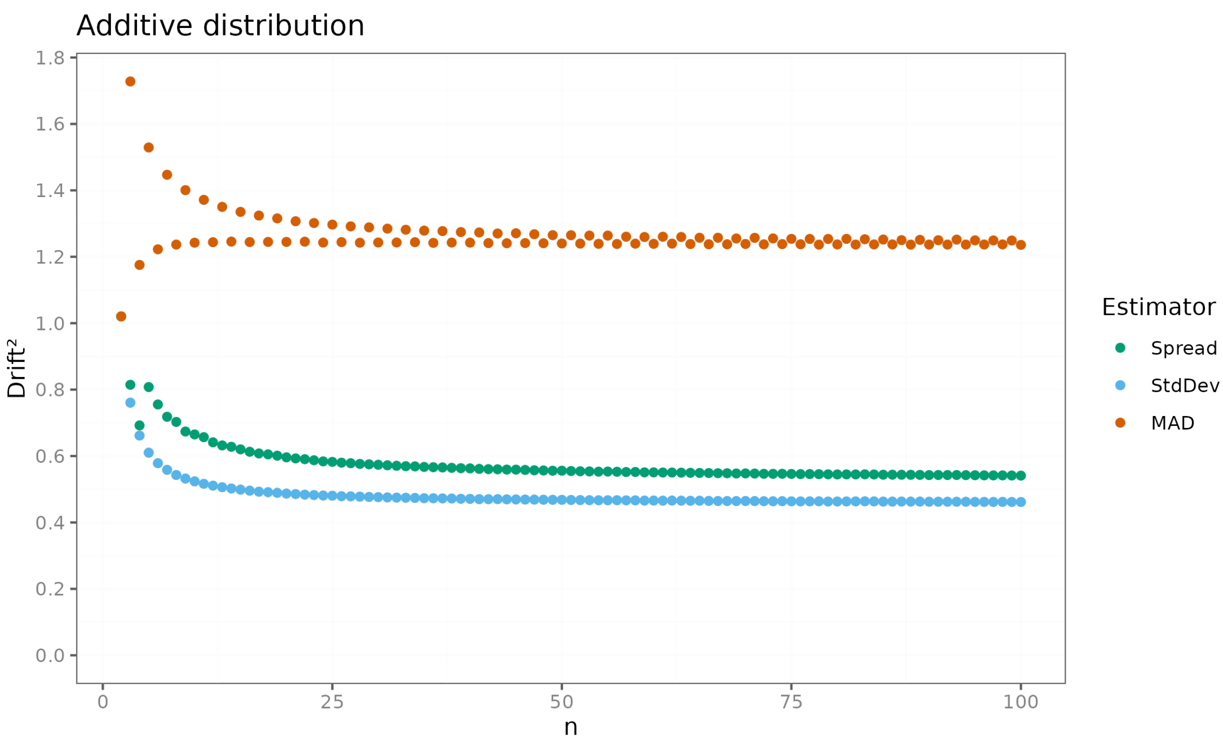


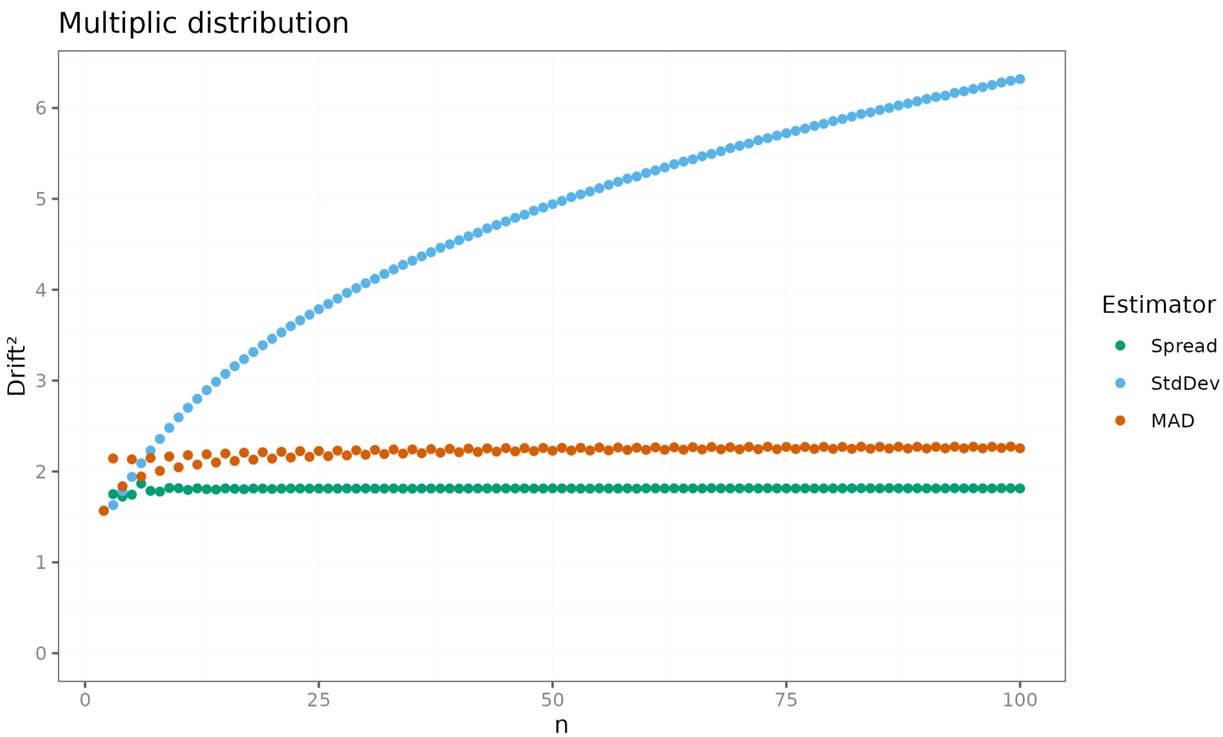
Asymptotic Dispersion estimator drift² (values are approximated):

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

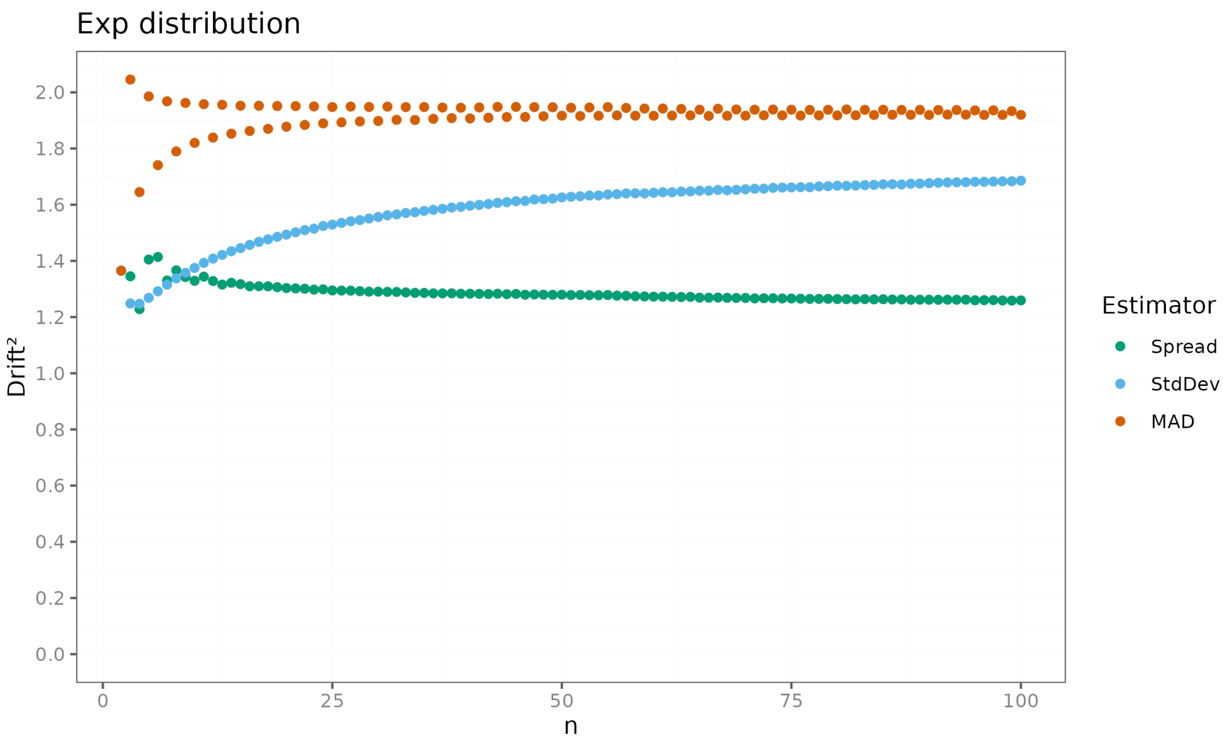
Rescaled to Spread (sample size adjustment factors):

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



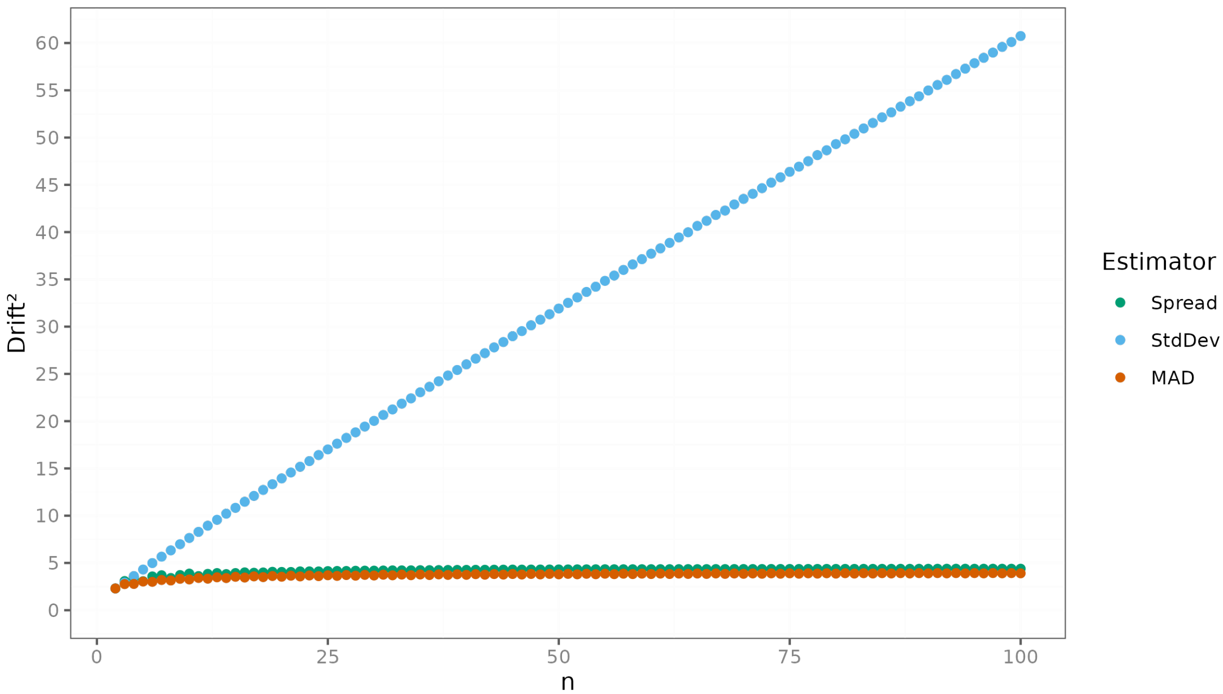


Multiplic distribution
Drift²
n
Estimator
Spread
StdDev
MAD

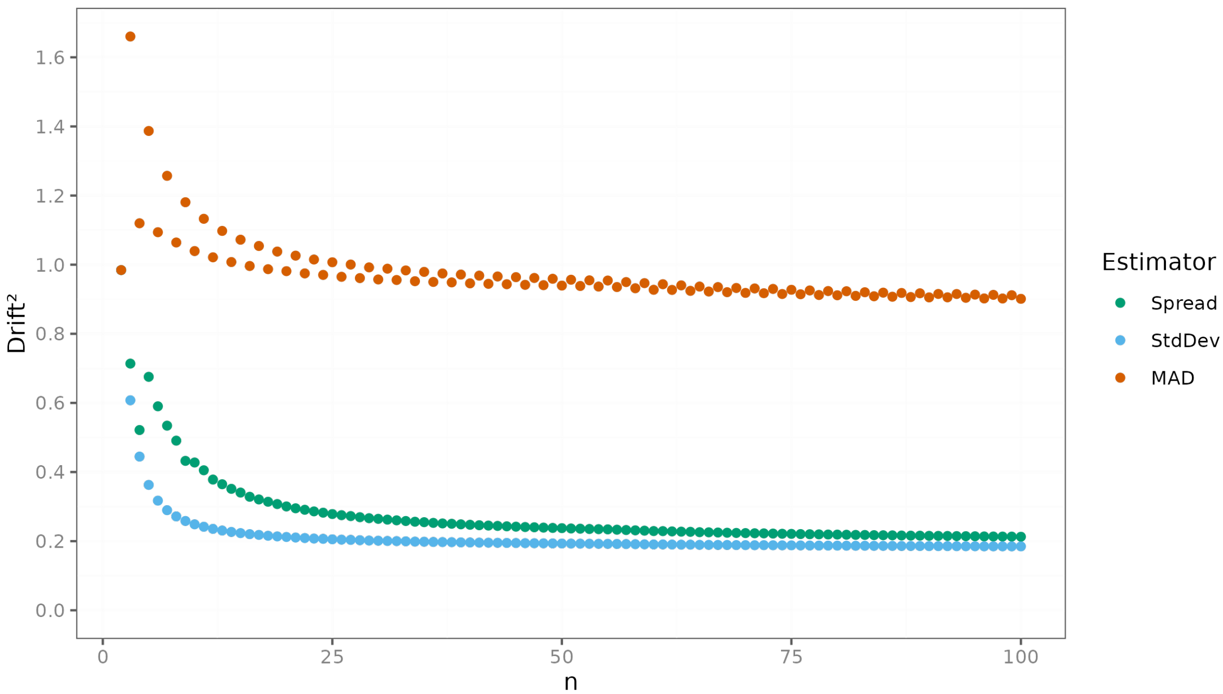


Exp distribution
Drift²
n
Estimator
Spread
StdDev
MAD

Power distribution



Uniform distribution



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 constants 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

	Location	Scale
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 *desired properties*:

- **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 works backward from mathematical convenience to practical application. Practitioners don't know the distribution in advance, so they lack clear guidance on which procedure to choose by default.

Most traditional statistics relies on Additivity ('Normality'). Procedures optimal for Additive ('Normal') distributions 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 lacks robustness 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 study 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 rate, 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 study 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.

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 Algorithm

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 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 [Mon84]). The algorithm exploits the inherent structure in pairwise sums rather than computing them explicitly. After sorting the input values $x_1 \le x_2 \le \cdots \le x_n$, consider the implicit upper triangular matrix M where $M_{i,j} = x_i + x_j$ for $i \le j$. This matrix has crucial properties: each row and column are sorted in non-decreasing order, enabling efficient median selection without materializing the quadratic structure.

Rather than sorting all pairwise sums, the algorithm uses a selection approach similar to quickselect. The process 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 contains 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 regardless of the number of pairwise sums. This matches the complexity of sorting a single array while avoiding the quadratic explosion of materializing all pairwise combinations.

```csharp
namespace Pragmastat.Algorithms;

internal static class FastCenterAlgorithm
{
    /// <summary>
    /// ACM Algorithm 616: fast computation of the Hodges-Lehmann location estimator
    /// </summary>
    /// <remarks>
    /// Computes the median of all pairwise averages (xi + xj)/2 efficiently.
    /// See: John F Monahan, "Algorithm 616: fast computation of the Hodges-Lehmann
    ↪ location estimator"
    /// (1984) DOI: 10.1145/1271.319414
    /// </remarks>
    /// <param name="values">A sorted sample of values</param>
```

```
    /// <param name="random">Random number generator</param>
    /// <param name="isSorted">If values are sorted</param>
    /// <returns>Exact center value (Hodges-Lehmann estimator)</returns>
    public static double Estimate(IReadOnlyList<double> values, Random? random = null,
    ↪  bool isSorted = false)
    {
        int n = values.Count;
        if (n == 1) return values[0];
        if (n == 2) return (values[0] + values[1]) / 2;
        random ??= new Random();
        if (!isSorted)
            values = values.OrderBy(x => x).ToList();

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

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

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

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

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

            for (int row = 1; row <= n; row++)
            {
                partitionCounts[row - 1] = 0;

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

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

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

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

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

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

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

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

        continue;
    }

    // === TARGET CHECK ===
    // Check if we've found the median rank(s)
    bool atTargetRank = countBelowPivot == medianRankLow || countBelowPivot ==
    ↪  medianRankHigh - 1;
    if (atTargetRank)
    {
        // Find the boundary values: largest < pivot and smallest >= pivot
        double largestBelowPivot = double.MinValue;
        double smallestAtOrAbovePivot = double.MaxValue;

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

```
                    long totalInRow = n - i + 1;

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

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

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

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

            // === PREPARE NEXT ITERATION ===
```

```
            previousCount = countBelowPivot;

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

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

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

                    cumulativeSize += rowSize;
                }

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

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

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

                    minRemainingSum = Min(minRemainingSum, minInRow);
```

```
                    maxRemainingSum = Max(maxRemainingSum, maxInRow);
                }

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

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

6.2 Fast Spread Algorithm

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:

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

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 apply to pairwise differences, 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 $y_j - y_i$. Consider the implicit upper triangular matrix D where $D_{i,j} = y_j - y_i$ for $i < j$. This matrix inherits crucial structural properties: for fixed row i, differences increase monotonically, while for fixed column j, differences decrease as i increases. The 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 consideration, initially spanning columns $i + 1$ through n. The algorithm 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, it counts how many differences fall below p using a single sweep, with the monotonic structure ensuring this counting requires only $O(n)$ operations. While counting, the algorithm maintains the largest difference below p and smallest difference at or above p — these boundary values become the exact answer when the target rank is reached.

The algorithm handles both odd and even cases naturally. For an odd number of differences, it returns the single middle element when the count exactly hits the median rank. For an even number of differences, it returns the average of the two middle elements, with boundary tracking during counting providing both values simultaneously. Unlike approximation methods, this algorithm returns the precise median of all pairwise differences, with randomness affecting 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, and when progress stalls due to tied values, it computes the range of remaining active differences and pivots to their midrange. This midrange strategy ensures convergence even with highly discrete data or datasets containing many identical values.

Several optimizations make the algorithm practical for production use. A global column pointer that never moves backward during counting exploits the matrix structure to avoid redundant comparisons. The algorithm captures exact boundary values 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, independent of the quadratic number of pairwise differences, the algorithm achieves $O(n \log n)$ time complexity with minimal memory overhead, making robust scale estimation practical for large datasets.

```
namespace Pragmastat.Algorithms;

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

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

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

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

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

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

        long prevCountBelow = -1;

        while (true)
        {
            // === PARTITION: count how many differences are < pivot; also track boundary
            ↪  neighbors ===
            long countBelow = 0;
```

```
double largestBelow = double.NegativeInfinity; // max difference < pivot
double smallestAtOrAbove = double.PositiveInfinity; // min difference >=
↪  pivot

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

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

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

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

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

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

// === STALL HANDLING (ties / no progress) ===
```

```
            if (countBelow == prevCountBelow)
            {
                // Compute min/max remaining difference in the ACTIVE set and pivot to
                ↪  their midrange.
                double minActive = double.PositiveInfinity;
                double maxActive = double.NegativeInfinity;
                long active = 0;

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

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

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

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

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

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

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

    prevCountBelow = countBelow;

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

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

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

        if (kLow < kHigh) return 0.5 * (minRem + maxRem);
        return (Abs((kLow - 1) - countBelow) <= Abs(countBelow - kLow)) ? minRem
        ↪  : maxRem;
    }
```

```
        else
        {
            long t = NextIndex(random, activeSize); // 0..activeSize-1
            long acc = 0;
            int row = 0;
            for (; row < n - 1; row++)
            {
                if (L[row] > R[row]) continue;
                long size = R[row] - L[row] + 1;
                if (t < acc + size) break;
                acc += size;
            }

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

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

    Array.Sort(a);
    return a;
}

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

    return a;
}

private static long NextIndex(Random rng, long limitExclusive)
{
    // Uniform 0..limitExclusive-1 even for large ranges.
```

```
        // Use rejection sampling for correctness.
        ulong uLimit = (ulong)limitExclusive;
        if (uLimit <= int.MaxValue)
        {
            return rng.Next((int)uLimit);
        }

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

7 Studies

This section analyzes the estimators' properties using mathematical proofs. Most proofs are adopted 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 $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$.

7.1.1 Asymptotic Spread Value

Consider two independent draws X and Y from the $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$ 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 $\underline{\text{Additive}}(0, \sqrt{2} \cdot \text{stdDev})$, and the problem reduces to finding the median of $|D|$. The location parameter mean disappears, as expected, because absolute differences are invariant under shifts.

Let $\tau = \sqrt{2} \cdot \text{stdDev}$, so that $D \sim \underline{\text{Additive}}(0, \tau)$. The random variable $|D|$ then follows the Half-Additive ('Folded Normal') distribution with scale τ. Its cumulative distribution function for $z \geq 0$ becomes

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

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

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

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 $T_n \sim \text{approx } \underline{\text{Additive}}(\mu, (a \cdot \text{stdDev})^2/n)$, 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 $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$, 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 Additive ('Normal') distribution, serving as the natural baseline for comparison. Mean is the optimal estimator under Additive ('Normal') distribution: no other estimators achieve lower Drift.

7.1.4 Asymptotic Median Drift

For the sample median $\text{Median}(\mathbf{x})$ applied to samples from $\underline{\text{Additive}}(\text{mean}, \text{stdDev})$, 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), we need to determine the asymptotic sampling distribution.

The center estimator computes all pairwise averages (including $i = j$) and takes their median. For the Additive ('Normal') distribution, the 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 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 $T_n \sim \text{approx}\,\text{Additive}(b \cdot \text{stdDev}, (a \cdot \text{stdDev})^2/n)$, 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 Additive(mean, stdDev), the sampling distribution of StdDev is approximately Additive ('Normal') for large n with mean stdDev and standard deviation $\text{stdDev}/\sqrt{2n}$.

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

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

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

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

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

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

Therefore:

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

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

7.1.8 Asymptotic MAD Drift

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

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

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

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

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

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

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

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

Therefore:

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

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

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:

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

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

7.1.10 Summary

Summary for average estimators:

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

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

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

Summary for dispersion estimators:

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

Estimator	$\mathrm{Drift}(E, X)$	$\mathrm{Drift}^2(E, X)$	$1/\mathrm{Drift}^2(E, X)$
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 StdDev precision, while offering greater robustness to outliers. The Spread requires about 1.16 times more data to match StdDev precision under purely Additive ('Normal') conditions while maintaining robustness.

8 Reference Implementations

8.1 Python

```
pip install pragmastat
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/python

Demo:

```python
from pragmastat import center, spread, rel_spread, shift, ratio, avg_spread, disparity


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
```

```python
    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


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

8.2 TypeScript

```
npm i pragmastat
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/ts

Demo:

```typescript
import { center, spread, relSpread, shift, ratio, avgSpread, disparity } 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
}

main();
```

8.3 R

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

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/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
```

```r
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
```

8.4 .NET

Install the NuGet package Pragmastat:

```
dotnet add package Pragmastat
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/dotnet

Demo:

```csharp
using static System.Console;
```

```
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
    }
}
```

8.5 Kotlin

A package is not available yet.

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/kotlin

Demo:

```kotlin
package dev.pragmastat.example

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
```

```
}
```

8.6 Rust

Add this to your `Cargo.toml`:

```toml
[dependencies]
pragmastat = "3.1.23"
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/rust

Demo:

```rust
use pragmastat::*;

fn main() {
    let x = vec![0.0, 2.0, 4.0, 6.0, 8.0];
    println!("{}", center(&x).unwrap()); // 4
    let x_plus_10: Vec<f64> = x.iter().map(|v| v + 10.0).collect();
    println!("{}", center(&x_plus_10).unwrap()); // 14
    let x_times_3: Vec<f64> = x.iter().map(|v| v * 3.0).collect();
    println!("{}", center(&x_times_3).unwrap()); // 12

    println!("{}", spread(&x).unwrap()); // 4
    println!("{}", spread(&x_plus_10).unwrap()); // 4
    let x_times_2: Vec<f64> = x.iter().map(|v| v * 2.0).collect();
    println!("{}", spread(&x_times_2).unwrap()); // 8

    println!("{}", rel_spread(&x).unwrap()); // 1
    let x_times_5: Vec<f64> = x.iter().map(|v| v * 5.0).collect();
    println!("{}", rel_spread(&x_times_5).unwrap()); // 1

    let y = vec![10.0, 12.0, 14.0, 16.0, 18.0];
    println!("{}", shift(&x, &y).unwrap()); // -10
    println!("{}", shift(&x, &x).unwrap()); // 0
    let x_plus_7: Vec<f64> = x.iter().map(|v| v + 7.0).collect();
    let y_plus_3: Vec<f64> = y.iter().map(|v| v + 3.0).collect();
    println!("{}", shift(&x_plus_7, &y_plus_3).unwrap()); // -6
    let y_times_2: Vec<f64> = y.iter().map(|v| v * 2.0).collect();
    println!("{}", shift(&x_times_2, &y_times_2).unwrap()); // -20
    println!("{}", shift(&y, &x).unwrap()); // 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];
    println!("{}", ratio(&x, &y).unwrap()); // 0.5
    println!("{}", ratio(&x, &x).unwrap()); // 1
    let x_times_2: Vec<f64> = x.iter().map(|v| v * 2.0).collect();
    let y_times_5: Vec<f64> = y.iter().map(|v| v * 5.0).collect();
    println!("{}", ratio(&x_times_2, &y_times_5).unwrap()); // 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];
```

```
    println!("{}", spread(&x).unwrap()); // 6
    println!("{}", spread(&y).unwrap()); // 4

    println!("{}", avg_spread(&x, &y).unwrap()); // 5
    println!("{}", avg_spread(&x, &x).unwrap()); // 6
    let x_times_2: Vec<f64> = x.iter().map(|v| v * 2.0).collect();
    let x_times_3: Vec<f64> = x.iter().map(|v| v * 3.0).collect();
    println!("{}", avg_spread(&x_times_2, &x_times_3).unwrap()); // 15
    println!("{}", avg_spread(&y, &x).unwrap()); // 5
    let y_times_2: Vec<f64> = y.iter().map(|v| v * 2.0).collect();
    println!("{}", avg_spread(&x_times_2, &y_times_2).unwrap()); // 10

    println!("{}", shift(&x, &y).unwrap()); // 2
    println!("{}", avg_spread(&x, &y).unwrap()); // 5

    println!("{}", disparity(&x, &y).unwrap()); // 0.4
    let x_plus_5: Vec<f64> = x.iter().map(|v| v + 5.0).collect();
    let y_plus_5: Vec<f64> = y.iter().map(|v| v + 5.0).collect();
    println!("{}", disparity(&x_plus_5, &y_plus_5).unwrap()); // 0.4
    println!("{}", disparity(&x_times_2, &y_times_2).unwrap()); // 0.4
    println!("{}", disparity(&y, &x).unwrap()); // -0.4
}
```

8.7 Go

To install the package from GitHub:

```
go get github.com/AndreyAkinshin/pragmastat/go/v3
```

Source code: https://github.com/AndreyAkinshin/pragmastat/tree/v3.1.23/go

Demo:

```
package main

import (
    "fmt"
    "log"

    pragmastat "github.com/AndreyAkinshin/pragmastat/go/v3"
)

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

func subtract(x []float64, val float64) []float64 {
    result := make([]float64, len(x))
```

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

func divide(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}
    fmt.Println(mustCenter(x))              // 4
    fmt.Println(mustCenter(add(x, 10)))     // 14
    fmt.Println(mustCenter(multiply(x, 3))) // 12

    fmt.Println(mustSpread(x))              // 4
    fmt.Println(mustSpread(add(x, 10)))     // 4
    fmt.Println(mustSpread(multiply(x, 2))) // 8

    fmt.Println(mustRelSpread(x))              // 1
    fmt.Println(mustRelSpread(multiply(x, 5))) // 1

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

    x = []float64{1, 2, 4, 8, 16}
    y = []float64{2, 4, 8, 16, 32}
    fmt.Println(mustRatio(x, y))                               // 0.5
    fmt.Println(mustRatio(x, x))                               // 1
    fmt.Println(mustRatio(multiply(x, 2), multiply(y, 5))) // 0.2

    x = []float64{0, 3, 6, 9, 12}
    y = []float64{0, 2, 4, 6, 8}
    fmt.Println(mustSpread(x)) // 6
    fmt.Println(mustSpread(y)) // 4

    fmt.Println(mustAvgSpread(x, y))                               // 5
```

```go
    fmt.Println(mustAvgSpread(x, x))                           // 6
    fmt.Println(mustAvgSpread(multiply(x, 2), multiply(x, 3))) // 15
    fmt.Println(mustAvgSpread(y, x))                           // 5
    fmt.Println(mustAvgSpread(multiply(x, 2), multiply(y, 2))) // 10

    fmt.Println(mustShift(x, y))     // 2
    fmt.Println(mustAvgSpread(x, y)) // 5

    fmt.Println(mustDisparity(x, y))                           // 0.4
    fmt.Println(mustDisparity(add(x, 5), add(y, 5)))           // 0.4
    fmt.Println(mustDisparity(multiply(x, 2), multiply(y, 2))) // 0.4
    fmt.Println(mustDisparity(y, x))                           // -0.4
}

func mustCenter(x []float64) float64 {
    result, err := pragmastat.Center(x)
    if err != nil {
        log.Fatal(err)
    }
    return result
}

func mustSpread(x []float64) float64 {
    result, err := pragmastat.Spread(x)
    if err != nil {
        log.Fatal(err)
    }
    return result
}

func mustRelSpread(x []float64) float64 {
    result, err := pragmastat.RelSpread(x)
    if err != nil {
        log.Fatal(err)
    }
    return result
}

func mustShift(x, y []float64) float64 {
    result, err := pragmastat.Shift(x, y)
    if err != nil {
        log.Fatal(err)
    }
    return result
}

func mustRatio(x, y []float64) float64 {
    result, err := pragmastat.Ratio(x, y)
    if err != nil {
        log.Fatal(err)
    }
    return result
}
```

```go
func mustAvgSpread(x, y []float64) float64 {
	result, err := pragmastat.AvgSpread(x, y)
	if err != nil {
		log.Fatal(err)
	}
	return result
}

func mustDisparity(x, y []float64) float64 {
	result, err := pragmastat.Disparity(x, y)
	if err != nil {
		log.Fatal(err)
	}
	return result
}
```

References

[HL63] J. L. Hodges and E. L. Lehmann. "Estimates of Location Based on Rank Tests." en. In: *The Annals of Mathematical Statistics* 34.2 (June 1963), pp. 598–611. ISSN: 0003-4851. DOI: 10.1214/aoms/1177704172. URL: http://projecteuclid.org/euclid.aoms/1177704172.

[Mon84] John F Monahan. "Algorithm 616: fast computation of the Hodges-Lehmann location estimator." en. In: *ACM Transactions on Mathematical Software* 10.3 (Aug. 1984), pp. 265–270. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1271.319414. URL: https://dl.acm.org/doi/10.1145/1271.319414.

[Sen63] Pranab Kumar Sen. "On the Estimation of Relative Potency in Dilution (-Direct) Assays by Distribution-Free Methods." In: *Biometrics* 19.4 (Dec. 1963), p. 532. ISSN: 0006341X. DOI: 10.2307/2527532. URL: https://www.jstor.org/stable/2527532.

[Sha76] Michael Ian Shamos. "Geometry and Statistics: Problems at the Interface." In: (1976).