Statistics

The Statistics standard library module contains basic statistics functionality.

`Statistics.std` — Function

```
std(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample standard deviation of collection `itr`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions, and `means` may contain means for each dimension of `itr`.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> **Note**
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.stdm` — Function

```
stdm(itr, mean; corrected::Bool=true)
```

Compute the sample standard deviation of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> **Note**
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.var` — Function

```
var(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample variance of collection `itr`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n` is the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> **Note**
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.varm` — Function

```
varm(itr, mean; dims, corrected::Bool=true)
```

Compute the sample variance of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sum((itr .- mean(itr)).^2) / (length(itr) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

Note

If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.cor` — Function

```
cor(x::AbstractVector)
```

Return the number one.

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix `X` along the dimension `dims`.

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors `x` and `y`.

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices `X` and `Y` along the dimension `dims`.

`Statistics.cov` — Function

```
cov(x::AbstractVector; corrected::Bool=true)
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

```
cov(X::AbstractMatrix; dims::Int=1, corrected::Bool=true)
```

Compute the covariance matrix of the matrix `X` along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims)`.

```
cov(x::AbstractVector, y::AbstractVector; corrected::Bool=true)
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$ where $*$ denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$.

```
cov(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims::Int=1, corrected::Bool=true)
```

Compute the covariance between the vectors or matrices `X` and `Y` along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims) = size(Y, dims)`.

`Statistics.mean!` — Function

```
mean!(r, v)
```

Compute the mean of `v` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> using Statistics

julia> v = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Array{Float64,1}:
 1.5
 3.5
```

```
julia> mean!([1. 1.], v)
1×2 Array{Float64,2}:
 2.0  3.0
```

`Statistics.mean` — Function

```
mean(itr)
```

Compute the mean of all elements in a collection.

Note

If `itr` contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the mean of non-missing values.

Examples

```
julia> using Statistics

julia> mean(1:20)
10.5

julia> mean([1, missing, 3])
missing

julia> mean(skipmissing([1, missing, 3]))
2.0
```

```
mean(f::Function, itr)
```

Apply the function `f` to each element of collection `itr` and take the mean.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
```

```
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

```
mean(f::Function, A::AbstractArray; dims)
```

Apply the function `f` to each element of array `A` and take the mean over dimensions `dims`.

> **Julia 1.3**
>
> This method requires at least Julia 1.3.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908

julia> mean(√, [1 2 3; 4 5 6], dims=2)
2×1 Array{Float64,2}:
 1.3820881233139908
 2.2285192400943226
```

```
mean(A::AbstractArray; dims)
```

Compute the mean of an array over the given dimensions.

> **Julia 1.1**
>
> `mean` for empty arrays requires at least Julia 1.1.

Examples

```
julia> using Statistics
```

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean(A, dims=1)
1×2 Array{Float64,2}:
 2.0  3.0

julia> mean(A, dims=2)
2×1 Array{Float64,2}:
 1.5
 3.5
```

`Statistics.median!` — Function

```
median!(v)
```

Like `median`, but may overwrite the input vector.

`Statistics.median` — Function

```
median(itr)
```

Compute the median of all elements in a collection. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

> **Note**
>
> If `itr` contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if `itr` contains both). Use the `skipmissing` function to omit `missing` entries and compute the median of non-missing values.

Examples

```
julia> using Statistics
```

```
julia> median([1, 2, 3])
2.0

julia> median([1, 2, 3, 4])
2.5

julia> median([1, 2, missing, 4])
missing

julia> median(skipmissing([1, 2, missing, 4]))
2.0
```

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

```
julia> using Statistics

julia> median([1 2; 3 4], dims=1)
1×2 Array{Float64,2}:
 2.0  3.0
```

`Statistics.middle` — Function

```
middle(x)
```

Compute the middle of a scalar value, which is equivalent to `x` itself, but of the type of `middle(x, x)` for consistency.

```
middle(x, y)
```

Compute the middle of two reals `x` and `y`, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

```
middle(range)
```

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

```
julia> using Statistics

julia> middle(1:10)
5.5
```

```
middle(a)
```

Compute the middle of an array `a`, which consists of finding its extrema and then computing their mean.

```
julia> using Statistics

julia> a = [1,2,3.6,10.9]
4-element Array{Float64,1}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

`Statistics.quantile!` — Function

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false, alpha::Real=
```

Compute the quantile(s) of a vector `v` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. If `p` is a vector, an optional output array `q` may also be specified. (If not provided, a new output array is created.) The keyword argument `sorted` indicates whether `v` can be assumed to be sorted; if `false` (the default), then the elements of `v` will be partially sorted in-place.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[k])`, for `k = 1:n` where `n = length(v)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5`
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

> **Note**
>
> An `ArgumentError` is thrown if `v` contains `NaN` or `missing` values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> x = [3, 2, 1];

julia> quantile!(x, 0.5)
2.0

julia> x
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> y = zeros(3);

julia> quantile!(y, x, [0.1, 0.5, 0.9]) === y
true

julia> y
3-element Array{Float64,1}:
 1.2000000000000002
 2.0
 2.8000000000000003
```

`Statistics.quantile` — Function

```
quantile(itr, p; sorted=false, alpha::Real=1.0, beta::Real=alpha)
```

Compute the quantile(s) of a collection `itr` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. The keyword argument `sorted` indicates whether `itr` can be assumed to be sorted.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where $x[j]$ is the j-th order statistic, and `γ` is a function of `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `g = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[k])`, for `k = 1:n` where `n = length(itr)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5`
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

> **Note**
>
> An `ArgumentError` is thrown if `v` contains `NaN` or `missing` values. Use the `skipmissing` function to omit `missing` entries and compute the quantiles of non-missing values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> quantile(0:20, 0.5)
10.0

julia> quantile(0:20, [0.1, 0.5, 0.9])
3-element Array{Float64,1}:
  2.0
 10.0
 18.000000000000004

julia> quantile(skipmissing([1, 10, missing]), 0.5)
5.5
```

« Sparse Arrays Unit Testing »

Powered by Documenter.jl and the Julia Programming Language.