JuliaPro v0.6.2.2 API Manual

March 1, 2018

Julia Computing Inc.
info@juliacomputing.com

Contents

1 Distributions 1

2 StatsBase 46

3 PyPlot 81

4 IndexedTables 108

5 Images 139

6 Knet 155

7 DataFrames 171

8 DataStructures 185

9 JDBC 197

10 NNlib 208

11 ImageCore 214

12 Reactive 223

13 JuliaDB 230

14 Combinatorics 237

15 HypothesisTests 243

16 DataArrays 249

17 GLM 257

18 Documenter 263

19 ColorTypes 268

20 Primes **273**

21 Roots **279**

22 ImageTransformations **284**

23 PyCall **288**

24 Gadfly **293**

25 IterTools **298**

26 Iterators **306**

27 Polynomials **314**

28 Colors **318**

29 FileIO **322**

30 Interact **326**

31 FFTW **330**

32 AxisArrays **334**

33 QuadGK **338**

34 BusinessDays **341**

35 NLSolversBase **344**

36 Dagger **347**

37 ShowItLikeYouBuildIt **350**

38 CoordinateTransformations **353**

39 Graphics **355**

40 MacroTools **358**

41 ImageMorphology **361**

42 Contour **363**

43 Compose **365**

44 CoupledFields **368**

45 AxisAlgorithms 370

46 Libz 372

47 NullableArrays 375

48 ImageMetadata 377

49 LegacyStrings 379

50 BufferedStreams 381

51 MbedTLS 383

52 DataValues 385

53 OnlineStats 387

54 NearestNeighbors 389

55 IJulia 390

56 WebSockets 392

57 AutoGrad 394

58 ComputationalResources 396

59 Clustering 398

60 JuliaWebAPI 400

61 DecisionTree 402

62 Blosc 404

63 Missings 406

64 Parameters 408

65 HDF5 410

66 HttpServer 412

67 MappedArrays 414

68 TextParse 415

69 LossFunctions 417

70 LearnBase **418**

71 Juno **420**

72 HttpParser **422**

73 ImageAxes **423**

74 Flux **424**

75 IntervalSets **425**

76 Media **426**

77 Rotations **427**

78 Mustache **428**

79 TiledIteration **429**

80 Distances **430**

81 HttpCommon **431**

82 StaticArrays **432**

83 SweepOperator **433**

84 PaddedViews **434**

85 SpecialFunctions **435**

86 NamedTuples **436**

87 Loess **437**

88 Nulls **438**

89 WoodburyMatrices **439**

90 Requests **440**

91 MemPool **441**

92 SimpleTraits **442**

93 BinDeps **443**

Chapter 1

Distributions

1.1 Base.LinAlg.scale!

`Base.LinAlg.scale!` — *Method*.

```
scale!{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix,::Abstract
```

Calculate the scale parameter, as defined for the location parameter above and store the result in .

source

1.2 Base.mean

`Base.mean` — *Method*.

```
mean(d::Union{UnivariateMixture, MultivariateMixture})
```

Compute the overall mean (expectation).

source

1.3 Base.mean

`Base.mean` — *Method*.

```
mean(d::MatrixDistribution)
```

Return the mean matrix of `d`.

source

1.4 Base.mean

`Base.mean` — *Method.*

```
mean(d::MultivariateDistribution)
```

Compute the mean vector of distribution `d`.
source

1.5 Base.mean

`Base.mean` — *Method.*

```
mean(d::UnivariateDistribution)
```

Compute the expectation.
source

1.6 Base.median

`Base.median` — *Method.*

```
median(d::UnivariateDistribution)
```

Return the median value of distribution `d`.
source

1.7 Base.median

`Base.median` — *Method.*

```
median(d::MvLogNormal)
```

Return the median vector of the lognormal distribution. which is strictly smaller than the mean.
source

1.8 Base.quantile

`Base.quantile` — *Method.*

```
quantile(d::UnivariateDistribution, q::Real)
```

Evaluate the inverse cumulative distribution function at `q`.
See also: `cquantile`, `invlogcdf`, and `invlogccdf`.
source

1.9 Base.std

`Base.std` — *Method*.

`std(d::UnivariateDistribution)`

Return the standard deviation of distribution `d`, i.e. `sqrt(var(d))`.
source

1.10 Base.var

`Base.var` — *Method*.

`var(d::UnivariateMixture)`

Compute the overall variance (only for $UnivariateMixture$).
source

1.11 Base.var

`Base.var` — *Method*.

`var(d::MultivariateDistribution)`

Compute the vector of element-wise variances for distribution `d`.
source

1.12 Base.var

`Base.var` — *Method*.

`var(d::UnivariateDistribution)`

Compute the variance. (A generic std is provided as `std(d) = sqrt(var(d))`)
source

1.13 Distributions.TruncatedNormal

`Distributions.TruncatedNormal` — *Method*.

`TruncatedNormal(mu, sigma, l, u)`

The *truncated normal distribution* is a particularly important one in the family of truncated distributions. We provide additional support for this type with `TruncatedNormal` which calls `Truncated(Normal(mu, sigma), l, u)`. Unlike the general case, truncated normal distributions support `mean`, `mode`, `modes`, `var`, `std`, and `entropy`.
source

1.14 Distributions.ccdf

`Distributions.ccdf` — *Method*.

```
ccdf(d::UnivariateDistribution, x::Real)
```

The complementary cumulative function evaluated at `x`, i.e. `1 - cdf(d, x)`.
source

1.15 Distributions.cdf

`Distributions.cdf` — *Method*.

```
cdf(d::UnivariateDistribution, x::Real)
```

Evaluate the cumulative probability at `x`.
See also `ccdf`, `logcdf`, and `logccdf`.
source

1.16 Distributions.cf

`Distributions.cf` — *Method*.

```
cf(d::UnivariateDistribution, t)
```

Evaluate the characteristic function of distribution `d`.
source

1.17 Distributions.components

`Distributions.components` — *Method*.

```
components(d::AbstractMixtureModel)
```

Get a list of components of the mixture model `d`.
source

1.18 Distributions.cquantile

`Distributions.cquantile` — *Method*.

```
cquantile(d::UnivariateDistribution, q::Real)
```

The complementary quantile value, i.e. `quantile(d, 1-q)`.
source

1.19 Distributions.failprob

`Distributions.failprob` — *Method*.

```
failprob(d::UnivariateDistribution)
```

Get the probability of failure.
source

1.20 Distributions.fit_mle

`Distributions.fit_mle` — *Method*.

```
fit_mle(D, x, w)
```

Fit a distribution of type `D` to a weighted data set `x`, with weights given by `w`.

Here, `w` should be an array with length `n`, where `n` is the number of samples contained in `x`.

source

1.21 Distributions.fit_mle

`Distributions.fit_mle` — *Method*.

```
fit_mle(D, x)
```

Fit a distribution of type `D` to a given data set `x`.

- For univariate distribution, x can be an array of arbitrary size.
- For multivariate distribution, x should be a matrix, where each column is a sample.

source

1.22 Distributions.insupport

`Distributions.insupport` — *Method*.

```
insupport(d::MultivariateMixture, x)
```

Evaluate whether `x` is within the support of mixture distribution `d`.
source

1.23 Distributions.insupport

`Distributions.insupport` — *Method.*

```
insupport(d::MultivariateDistribution, x::AbstractArray)
```

If x is a vector, it returns whether x is within the support of d. If x is a matrix, it returns whether every column in x is within the support of d.

source

1.24 Distributions.insupport

`Distributions.insupport` — *Method.*

```
insupport(d::UnivariateDistribution, x::Any)
```

When `x` is a scalar, it returns whether x is within the support of `d` (e.g., `insupport(d, x) = minimum(d) <= x <= maximum(d)`). When `x` is an array, it returns whether every element in x is within the support of `d`.

Generic fallback methods are provided, but it is often the case that `insupport` can be done more efficiently, and a specialized `insupport` is thus desirable. You should also override this function if the support is composed of multiple disjoint intervals.

source

1.25 Distributions.invcov

`Distributions.invcov` — *Method.*

```
invcov(d::AbstractMvNormal)
```

Return the inversed covariance matrix of d.

source

1.26 Distributions.invlogccdf

`Distributions.invlogccdf` — *Method.*

```
invlogcdf(d::UnivariateDistribution, lp::Real)
```

The inverse function of logcdf.

source

1.27 Distributions.invlogcdf

`Distributions.invlogcdf` — *Method*.

```
invlogcdf(d::UnivariateDistribution, lp::Real)
```

The inverse function of logcdf.
source

1.28 Distributions.isleptokurtic

`Distributions.isleptokurtic` — *Method*.

```
isleptokurtic(d)
```

Return whether `d` is leptokurtic (*i.e* `kurtosis(d) < 0`).
source

1.29 Distributions.ismesokurtic

`Distributions.ismesokurtic` — *Method*.

```
ismesokurtic(d)
```

Return whether `d` is mesokurtic (*i.e* `kurtosis(d) == 0`).
source

1.30 Distributions.isplatykurtic

`Distributions.isplatykurtic` — *Method*.

```
isplatykurtic(d)
```

Return whether `d` is platykurtic (*i.e* `kurtosis(d) > 0`).
source

1.31 Distributions.location!

`Distributions.location!` — *Method*.

```
location!{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix,::Abstr
```

Calculate the location vector (as above) and store the result in
source

1.32 Distributions.location

`Distributions.location` — *Method*.

```
location(d::UnivariateDistribution)
```

Get the location parameter.
source

1.33 Distributions.location

`Distributions.location` — *Method*.

```
location(d::MvLogNormal)
```

Return the location vector of the distribution (the mean of the underlying normal distribution).
source

1.34 Distributions.location

`Distributions.location` — *Method*.

```
location{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatri
```

Calculate the location vector (the mean of the underlying normal distribution).

- If `s == :meancov`, then m is taken as the mean, and S the covariance matrix of a lognormal distribution.
- If `s == :mean | :median | :mode`, then m is taken as the mean, median or mode of the lognormal respectively, and S is interpreted as the scale matrix (the covariance of the underlying normal distribution).

It is not possible to analytically calculate the location vector from e.g., median + covariance, or from mode + covariance.
source

1.35 Distributions.logccdf

`Distributions.logccdf` — *Method*.

```
logccdf(d::UnivariateDistribution, x::Real)
```

The logarithm of the complementary cumulative function values evaluated at x, i.e. `log(ccdf(x))`.
source

1.36 Distributions.logcdf

`Distributions.logcdf` — *Method*.

```
logcdf(d::UnivariateDistribution, x::Real)
```

The logarithm of the cumulative function value(s) evaluated at `x`, i.e. `log(cdf(x))`.
source

1.37 Distributions.logdetcov

`Distributions.logdetcov` — *Method*.

```
logdetcov(d::AbstractMvNormal)
```

Return the log-determinant value of the covariance matrix.
source

1.38 Distributions.logpdf

`Distributions.logpdf` — *Method*.

```
logpdf(d::Union{UnivariateMixture, MultivariateMixture}, x)
```

Evaluate the logarithm of the (mixed) probability density function over `x`. Here, `x` can be a single sample or an array of multiple samples.
source

1.39 Distributions.logpdf

`Distributions.logpdf` — *Method*.

```
logpdf(d::MultivariateDistribution, x::AbstractArray)
```

Return the logarithm of probability density evaluated at `x`.

- If `x` is a vector, it returns the result as a scalar.
- If `x` is a matrix with n columns, it returns a vector `r` of length n, where `r[i]` corresponds to `x[:,i]`.

`logpdf!(r, d, x)` will write the results to a pre-allocated array `r`.
source

1.40 Distributions.logpdf

`Distributions.logpdf` — *Method*.

```
logpdf(d::UnivariateDistribution, x::Real)
```

Evaluate the logarithm of probability density (mass) at `x`. Whereas there is a fallback implemented `logpdf(d, x) = log(pdf(d, x))`. Relying on this fallback is not recommended in general, as it is prone to overflow or underflow.

source

1.41 Distributions.logpdf

`Distributions.logpdf` — *Method*.

```
logpdf(d::MatrixDistribution, AbstractMatrix)
```

Compute the logarithm of the probability density at the input matrix `x`.

source

1.42 Distributions.mgf

`Distributions.mgf` — *Method*.

```
mgf(d::UnivariateDistribution, t)
```

Evaluate the moment generating function of distribution `d`.

source

1.43 Distributions.ncategories

`Distributions.ncategories` — *Method*.

```
ncategories(d::UnivariateDistribution)
```

Get the number of categories.

source

1.44 Distributions.nsamples

`Distributions.nsamples` — *Method*.

```
nsamples(s::Sampleable)
```

The number of samples contained in `A`. Multiple samples are often organized into an array, depending on the variate form.

source

1.45 Distributions.ntrials

`Distributions.ntrials` — *Method*.

```
ntrials(d::UnivariateDistribution)
```

Get the number of trials.
source

1.46 Distributions.pdf

`Distributions.pdf` — *Method*.

```
pdf(d::Union{UnivariateMixture, MultivariateMixture}, x)
```

Evaluate the (mixed) probability density function over `x`. Here, `x` can be a single sample or an array of multiple samples.
source

1.47 Distributions.pdf

`Distributions.pdf` — *Method*.

```
pdf(d::MultivariateDistribution, x::AbstractArray)
```

Return the probability density of distribution `d` evaluated at `x`.

- If `x` is a vector, it returns the result as a scalar.
- If `x` is a matrix with n columns, it returns a vector `r` of length n, where `r[i]` corresponds

to `x[:,i]` (i.e. treating each column as a sample).
`pdf!(r, d, x)` will write the results to a pre-allocated array `r`.
source

1.48 Distributions.pdf

`Distributions.pdf` — *Method*.

```
pdf(d::UnivariateDistribution, x::Real)
```

Evaluate the probability density (mass) at `x`.
See also: `logpdf`.
source

1.49 Distributions.pdf

`Distributions.pdf` — *Method*.

```
pdf(d::MatrixDistribution, x::AbstractArray)
```

Compute the probability density at the input matrix `x`.
source

1.50 Distributions.probs

`Distributions.probs` — *Method*.

```
probs(d::AbstractMixtureModel)
```

Get the vector of prior probabilities of all components of `d`.
source

1.51 Distributions.rate

`Distributions.rate` — *Method*.

```
rate(d::UnivariateDistribution)
```

Get the rate parameter.
source

1.52 Distributions.sampler

`Distributions.sampler` — *Method*.

```
sampler(d::Distribution) -> Sampleable
```

Samplers can often rely on pre-computed quantities (that are not parameters themselves) to improve efficiency. If such a sampler exists, it can be provide with this `sampler` method, which would be used for batch sampling. The general fallback is `sampler(d::Distribution) = d`.
source

1.53 Distributions.scale

`Distributions.scale` — *Method*.

```
scale(d::UnivariateDistribution)
```

Get the scale parameter.
source

1.54 Distributions.scale

`Distributions.scale` — *Method*.

```
scale(d::MvLogNormal)
```

Return the scale matrix of the distribution (the covariance matrix of the underlying normal distribution).

source

1.55 Distributions.scale

`Distributions.scale` — *Method*.

```
scale{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix)
```

Calculate the scale parameter, as defined for the location parameter above.

source

1.56 Distributions.shape

`Distributions.shape` — *Method*.

```
shape(d::UnivariateDistribution)
```

Get the shape parameter.

source

1.57 Distributions.sqmahal

`Distributions.sqmahal` — *Method*.

```
sqmahal(d, x)
```

Return the squared Mahalanobis distance from x to the center of d, w.r.t. the covariance. When x is a vector, it returns a scalar value. When x is a matrix, it returns a vector of length size(x,2).

`sqmahal!(r, d, x)` with write the results to a pre-allocated array `r`.

source

1.58 Distributions.succprob

`Distributions.succprob` — *Method*.

```
succprob(d::UnivariateDistribution)
```

Get the probability of success.

source

1.59 StatsBase.dof

`StatsBase.dof` — *Method*.

```
dof(d::UnivariateDistribution)
```

Get the degrees of freedom.
source

1.60 StatsBase.entropy

`StatsBase.entropy` — *Method*.

```
entropy(d::MultivariateDistribution, b::Real)
```

Compute the entropy value of distribution d, w.r.t. a given base.
source

1.61 StatsBase.entropy

`StatsBase.entropy` — *Method*.

```
entropy(d::MultivariateDistribution)
```

Compute the entropy value of distribution `d`.
source

1.62 StatsBase.entropy

`StatsBase.entropy` — *Method*.

```
entropy(d::UnivariateDistribution, b::Real)
```

Compute the entropy value of distribution `d`, w.r.t. a given base.
source

1.63 StatsBase.entropy

`StatsBase.entropy` — *Method*.

```
entropy(d::UnivariateDistribution)
```

Compute the entropy value of distribution `d`.
source

1.64 StatsBase.kurtosis

`StatsBase.kurtosis` — *Method*.

```
kurtosis(d::Distribution, correction::Bool)
```

Computes excess kurtosis by default. Proper kurtosis can be returned with correction=false
source

1.65 StatsBase.kurtosis

`StatsBase.kurtosis` — *Method*.

```
kurtosis(d::UnivariateDistribution)
```

Compute the excessive kurtosis.
source

1.66 StatsBase.loglikelihood

`StatsBase.loglikelihood` — *Method*.

```
loglikelihood(d::MultivariateDistribution, x::AbstractMatrix)
```

The log-likelihood of distribution `d` w.r.t. all columns contained in matrix `x`.
source

1.67 StatsBase.loglikelihood

`StatsBase.loglikelihood` — *Method*.

```
loglikelihood(d::UnivariateDistribution, X::AbstractArray)
```

The log-likelihood of distribution `d` w.r.t. all samples contained in array `x`.
source

1.68 StatsBase.mode

`StatsBase.mode` — *Method*.

```
mode(d::UnivariateDistribution)
```

Returns the first mode.
source

1.69 StatsBase.mode

`StatsBase.mode` — *Method*.

```
mode(d::MvLogNormal)
```

Return the mode vector of the lognormal distribution, which is strictly smaller than the mean and median.

source

1.70 StatsBase.modes

`StatsBase.modes` — *Method*.

```
modes(d::UnivariateDistribution)
```

Get all modes (if this makes sense).

source

1.71 StatsBase.params!

`StatsBase.params!` — *Method*.

```
params!{D<:AbstractMvLogNormal}(::Type{D},m::AbstractVector,S::AbstractMatrix,::Abstra
```

Calculate (scale,location) for a given mean and covariance, and store the results in and

source

1.72 StatsBase.params

`StatsBase.params` — *Method*.

```
params(d::UnivariateDistribution)
```

Return a tuple of parameters. Let `d` be a distribution of type `D`, then `D(params(d)...)` will construct exactly the same distribution as *d*.

source

1.73 StatsBase.params

`StatsBase.params` — *Method*.

```
params{D<:AbstractMvLogNormal}(::Type{D},m::AbstractVector,S::AbstractMatrix)
```

Return (scale,location) for a given mean and covariance

source

1.74 StatsBase.skewness

`StatsBase.skewness` — *Method.*

```
skewness(d::UnivariateDistribution)
```

Compute the skewness.

source

1.75 Distributions.AbstractMvNormal

`Distributions.AbstractMvNormal` — *Type.*

The Multivariate normal distribution is a multidimensional generalization of the *normal distribution*. The probability density function of a d-dimensional multivariate normal distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ is:

$$f(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

We realize that the mean vector and the covariance often have special forms in practice, which can be exploited to simplify the computation. For example, the mean vector is sometimes just a zero vector, while the covariance matrix can be a diagonal matrix or even in the form of $\sigma\mathbf{I}$. To take advantage of such special cases, we introduce a parametric type `MvNormal`, defined as below, which allows users to specify the special structure of the mean and covariance.

[] immutable MvNormal{Cov¡:AbstractPDMat,Mean¡:Union{Vector,ZeroVector}} ¡: AbstractMvNormal ::Mean ::Cov end

Here, the mean vector can be an instance of either `Vector` or `ZeroVector`, where the latter is simply an empty type indicating a vector filled with zeros. The covariance can be of any subtype of `AbstractPDMat`. Particularly, one can use `PDMat` for full covariance, `PDiagMat` for diagonal covariance, and `ScalMat` for the isotropic covariance – those in the form of $\sigma\mathbf{I}$. (See the Julia package PDMats for details).

We also define a set of alias for the types using different combinations of mean vectors and covariance:

[] const IsoNormal = MvNormal{ScalMat, Vector{Float64}} const DiagNormal = MvNormal{PDiagMat, Vector{Float64}} const FullNormal = MvNormal{PDMat, Vector{Float64}}

const ZeroMeanIsoNormal = MvNormal{ScalMat, ZeroVector{Float64}} const ZeroMeanDiagNormal = MvNormal{PDiagMat, ZeroVector{Float64}} const ZeroMeanFullNormal = MvNormal{PDMat, ZeroVector{Float64}}

source

1.76 Distributions.Arcsine

`Distributions.Arcsine` — *Type.*

`Arcsine(a,b)`

The *Arcsine distribution* has probability density function

$$f(x) = \frac{1}{\pi\sqrt{(x-a)(b-x)}}, \quad x \in [a, b]$$

[] Arcsine() Arcsine distribution with support [0, 1] Arcsine(b) Arcsine distribution with support [0, b] Arcsine(a, b) Arcsine distribution with support [a, b]

params(d) Get the parameters, i.e. (a, b) minimum(d) Get the lower bound, i.e. a maximum(d) Get the upper bound, i.e. b location(d) Get the left bound, i.e. a scale(d) Get the span of the support, i.e. b - a

External links

- Arcsine distribution on Wikipedia

source

1.77 Distributions.Bernoulli

`Distributions.Bernoulli` — *Type.*

`Bernoulli(p)`

A *Bernoulli distribution* is parameterized by a success rate `p`, which takes value 1 with probability `p` and 0 with probability `1-p`.

$$P(X = k) = \begin{cases} 1-p & \text{for } k = 0, \\ p & \text{for } k = 1. \end{cases}$$

[] Bernoulli() Bernoulli distribution with p = 0.5 Bernoulli(p) Bernoulli distribution with success rate p

params(d) Get the parameters, i.e. (p,) succprob(d) Get the success rate, i.e. p failprob(d) Get the failure rate, i.e. 1 - p

External links:

- Bernoulli distribution on Wikipedia

source

1.78 Distributions.Beta

`Distributions.Beta` — *Type.*

`Beta(,)`

The *Beta distribution* has probability density function

$$f(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \quad x \in [0,1]$$

The Beta distribution is related to the `Gamma` distribution via the property that if $X \sim \text{Gamma}(\alpha)$ and $Y \sim \text{Gamma}(\beta)$ independently, then $X/(X+Y) \sim Beta(\alpha,\beta)$.

[] Beta() equivalent to Beta(1, 1) Beta(a) equivalent to Beta(a, a) Beta(a, b) Beta distribution with shape parameters a and b

params(d) Get the parameters, i.e. (a, b)

External links

- Beta distribution on Wikipedia

source

1.79 Distributions.BetaBinomial

`Distributions.BetaBinomial` — *Type.*

`BetaBinomial(n,,)`

A *Beta-binomial distribution* is the compound distribution of the `Binomial` distribution where the probability of success `p` is distributed according to the `Beta`. It has three parameters: `n`, the number of trials and two shape parameters ,

$$P(X = k) = \binom{n}{k} B(k+\alpha, n-k+\beta)/B(\alpha,\beta), \quad \text{for } k = 0,1,2,\ldots,n.$$

[] BetaBinomial(n, a, b) BetaBinomial distribution with n trials and shape parameters a, b

params(d) Get the parameters, i.e. (n, a, b) ntrials(d) Get the number of trials, i.e. n

External links:

- Beta-binomial distribution on Wikipedia

source

1.80 Distributions.BetaPrime

`Distributions.BetaPrime` — *Type.*

`BetaPrime(,)`

The *Beta prime distribution* has probability density function

$$f(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1+x)^{-(\alpha+\beta)}, \quad x > 0$$

The Beta prime distribution is related to the `Beta` distribution via the relation ship that if $X \sim \text{Beta}(\alpha,\beta)$ then $\frac{X}{1-X} \sim \text{BetaPrime}(\alpha,\beta)$

[] BetaPrime() equivalent to BetaPrime(1, 1) BetaPrime(a) equivalent to BetaPrime(a, a) BetaPrime(a, b) Beta prime distribution with shape parameters a and b

params(d) Get the parameters, i.e. (a, b)

External links

- Beta prime distribution on Wikipedia

source

1.81 Distributions.Binomial

`Distributions.Binomial` — *Type.*

`Binomial(n,p)`

A *Binomial distribution* characterizes the number of successes in a sequence of independent trials. It has two parameters: `n`, the number of trials, and `p`, the probability of success in an individual trial, with the distribution:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{for } k = 0, 1, 2, \ldots, n.$$

[] Binomial() Binomial distribution with n = 1 and p = 0.5 Binomial(n) Binomial distribution for n trials with success rate p = 0.5 Binomial(n, p) Binomial distribution for n trials with success rate p

params(d) Get the parameters, i.e. (n, p) ntrials(d) Get the number of trials, i.e. n succprob(d) Get the success rate, i.e. p failprob(d) Get the failure rate, i.e. 1 - p

External links:

- Binomial distribution on Wikipedia

source

1.82 Distributions.Biweight

`Distributions.Biweight` — *Type.*

`Biweight(, )`

source

1.83 Distributions.Categorical

`Distributions.Categorical` — *Type.*

`Categorical(p)`

A *Categorical distribution* is parameterized by a probability vector `p` (of length `K`).

$$P(X = k) = p[k] \quad \text{for } k = 1, 2, \ldots, K.$$

[] Categorical(p) Categorical distribution with probability vector p params(d) Get the parameters, i.e. (p,) probs(d) Get the probability vector, i.e. p ncategories(d) Get the number of categories, i.e. K

Here, `p` must be a real vector, of which all components are nonnegative and sum to one. **Note:** The input vector `p` is directly used as a field of the constructed distribution, without being copied. External links:

- Categorical distribution on Wikipedia

source

1.84 Distributions.Cauchy

`Distributions.Cauchy` — *Type.*

`Cauchy(, )`

The *Cauchy distribution* with location and scale has probability density function

$$f(x; \mu, \sigma) = \frac{1}{\pi\sigma\left(1 + \left(\frac{x-\mu}{\sigma}\right)^2\right)}$$

[] Cauchy() Standard Cauchy distribution, i.e. Cauchy(0, 1) Cauchy(u) Cauchy distribution with location u and unit scale, i.e. Cauchy(u, 1) Cauchy(u, b) Cauchy distribution with location u and scale b

params(d) Get the parameters, i.e. (u, b) location(d) Get the location parameter, i.e. u scale(d) Get the scale parameter, i.e. b

External links

- Cauchy distribution on Wikipedia

source

1.85 Distributions.Chi

`Distributions.Chi` — *Type.*

`Chi()`

The *Chi distribution* degrees of freedom has probability density function

$$f(x;k) = \frac{1}{\Gamma(k/2)} 2^{1-k/2} x^{k-1} e^{-x^2/2}, \quad x > 0$$

It is the distribution of the square-root of a `Chisq` variate.

[] Chi(k) Chi distribution with k degrees of freedom

params(d) Get the parameters, i.e. (k,) dof(d) Get the degrees of freedom, i.e. k

External links

- Chi distribution on Wikipedia

source

1.86 Distributions.Chisq

`Distributions.Chisq` — *Type.*

`Chisq()`

The *Chi squared distribution* (typically written) with degrees of freedom has the probability density function

$$f(x;k) = \frac{x^{k/2-1} e^{-x/2}}{2^{k/2}\Gamma(k/2)}, \quad x > 0.$$

If is an integer, then it is the distribution of the sum of squares of independent standard `Normal` variates.

[] Chisq(k) Chi-squared distribution with k degrees of freedom

params(d) Get the parameters, i.e. (k,) dof(d) Get the degrees of freedom, i.e. k

External links

- Chi-squared distribution on Wikipedia

source

1.87 Distributions.Cosine

`Distributions.Cosine` — *Type.*

`Cosine(, )`

A raised Cosine distribution.
External link:

- Cosine distribution on wikipedia

source

1.88 Distributions.Dirichlet

`Distributions.Dirichlet` — *Type.*

`Dirichlet`

The Dirichlet distribution is often used the conjugate prior for Categorical or Multinomial distributions. The probability density function of a Dirichlet distribution with parameter $\alpha = (\alpha_1, \ldots, \alpha_k)$ is:

$$f(x;\alpha) = \frac{1}{B(\alpha)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}, \quad \text{with } B(\alpha) = \frac{\prod_{i=1}^{k} \Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{k} \alpha_i\right)}, \quad x_1 + \cdots + x_k = 1$$

[] Let alpha be a vector Dirichlet(alpha) Dirichlet distribution with parameter vector alpha

Let a be a positive scalar Dirichlet(k, a) Dirichlet distribution with parameter a * ones(k)

source

1.89 Distributions.DiscreteUniform

`Distributions.DiscreteUniform` — *Type.*

`DiscreteUniform(a,b)`

A *Discrete uniform distribution* is a uniform distribution over a consecutive sequence of integers between `a` and `b`, inclusive.

$$P(X = k) = 1/(b - a + 1) \quad \text{for } k = a, a + 1, \ldots, b.$$

[] DiscreteUniform(a, b) a uniform distribution over {a, a+1, ..., b}

params(d) Get the parameters, i.e. (a, b) span(d) Get the span of the support, i.e. (b - a + 1) probval(d) Get the probability value, i.e. 1 / (b - a + 1) minimum(d) Return a maximum(d) Return b

External links

- Discrete uniform distribution on Wikipedia

source

1.90 Distributions.Epanechnikov

`Distributions.Epanechnikov` — *Type.*

`Epanechnikov(, )`

source

1.91 Distributions.Erlang

`Distributions.Erlang` — *Type.*

`Erlang(,)`

The *Erlang distribution* is a special case of a `Gamma` distribution with integer shape parameter.

[] Erlang() Erlang distribution with unit shape and unit scale, i.e. Erlang(1, 1) Erlang(a) Erlang distribution with shape parameter a and unit scale, i.e. Erlang(a, 1) Erlang(a, s) Erlang distribution with shape parameter a and scale b

External links

- Erlang distribution on Wikipedia

source

1.92 Distributions.Exponential

`Distributions.Exponential` — *Type.*

`Exponential()`

The *Exponential distribution* with scale parameter has probability density function

$$f(x;\theta) = \frac{1}{\theta} e^{-\frac{x}{\theta}}, \quad x > 0$$

[] Exponential() Exponential distribution with unit scale, i.e. Exponential(1) Exponential(b) Exponential distribution with scale b

params(d) Get the parameters, i.e. (b,) scale(d) Get the scale parameter, i.e. b rate(d) Get the rate parameter, i.e. 1 / b

External links

- Exponential distribution on Wikipedia

source

1.93 Distributions.FDist

`Distributions.FDist` — *Type.*

`FDist(1, 2)`

The *F distribution* has probability density function

$$f(x; \nu_1, \nu_2) = \frac{1}{xB(\nu_1/2, \nu_2/2)} \sqrt{\frac{(\nu_1 x)^{\nu_1} \cdot \nu_2^{\nu_2}}{(\nu_1 x + \nu_2)^{\nu_1+\nu_2}}}, \quad x > 0$$

It is related to the `Chisq` distribution via the property that if $X_1 \sim \text{Chisq}(\nu_1)$ and $X_2 \sim \text{Chisq}(\nu_2)$, then $(X_1/\nu_1)/(X_2/\nu_2) \sim \text{FDist}(\nu_1, \nu_2)$.

[] FDist(1, 2) F-Distribution with parameters 1 and 2

params(d) Get the parameters, i.e. (1, 2)

External links

- F distribution on Wikipedia

source

1.94 Distributions.Frechet

`Distributions.Frechet` — *Type.*

`Frechet(,)`

The *Frchet distribution* with shape and scale has probability density function

$$f(x; \alpha, \theta) = \frac{\alpha}{\theta} \left(\frac{x}{\theta}\right)^{-\alpha-1} e^{-(x/\theta)^{-\alpha}}, \quad x > 0$$

[] Frechet() Frchet distribution with unit shape and unit scale, i.e. Frechet(1, 1) Frechet(a) Frchet distribution with shape a and unit scale, i.e. Frechet(a, 1) Frechet(a, b) Frchet distribution with shape a and scale b

params(d) Get the parameters, i.e. (a, b) shape(d) Get the shape parameter, i.e. a scale(d) Get the scale parameter, i.e. b

External links

- Frchet_distribution on Wikipedia

source

1.95 Distributions.Gamma

`Distributions.Gamma` — *Type.*

`Gamma(,)`

The *Gamma distribution* with shape parameter and scale has probability density function

$$f(x;\alpha,\theta)=\frac{x^{\alpha-1}e^{-x/\theta}}{\Gamma(\alpha)\theta^{\alpha}},\quad x>0$$

[] Gamma() Gamma distribution with unit shape and unit scale, i.e. Gamma(1, 1) Gamma() Gamma distribution with shape and unit scale, i.e. Gamma(, 1) Gamma(,) Gamma distribution with shape and scale

params(d) Get the parameters, i.e. (,) shape(d) Get the shape parameter, i.e. scale(d) Get the scale parameter, i.e.

External links

- Gamma distribution on Wikipedia

source

1.96 Distributions.GeneralizedExtremeValue

`Distributions.GeneralizedExtremeValue` — *Type.*

`GeneralizedExtremeValue(, , )`

The *Generalized extreme value distribution* with shape parameter , scale and location has probability density function

$$f(x;\xi,\sigma,\mu)=\begin{cases}\frac{1}{\sigma}\left[1+\left(\frac{x-\mu}{\sigma}\right)\xi\right]^{-1/\xi-1}\exp\left\{-\left[1+\left(\frac{x-\mu}{\sigma}\right)\xi\right]^{-1/\xi}\right\} & \text{for } \xi\neq 0\\ \frac{1}{\sigma}\exp\left\{-\frac{x-\mu}{\sigma}\right\}\exp\left\{-\exp\left[-\frac{x-\mu}{\sigma}\right]\right\} & \text{for } \xi=0\end{cases}$$

for

$$x\in\begin{cases}\left[\mu-\frac{\sigma}{\xi},+\infty\right) & \text{for } \xi>0\\ (-\infty,+\infty) & \text{for } \xi=0\\ \left(-\infty,\mu-\frac{\sigma}{\xi}\right] & \text{for } \xi<0\end{cases}$$

[] GeneralizedExtremeValue(m, s, k) Generalized Pareto distribution with shape k, scale s and location m.

params(d) Get the parameters, i.e. (m, s, k) location(d) Get the location parameter, i.e. m scale(d) Get the scale parameter, i.e. s shape(d) Get the shape parameter, i.e. k (sometimes called c)

External links

- Generalized extreme value distribution on Wikipedia

source

1.97 Distributions.GeneralizedPareto

`Distributions.GeneralizedPareto` — *Type*.

`GeneralizedPareto(, , )`

The *Generalized Pareto distribution* with shape parameter , scale and location has probability density function

$$f(x;\mu,\sigma,\xi) = \begin{cases} \frac{1}{\sigma}(1+\xi\frac{x-\mu}{\sigma})^{-\frac{1}{\xi}-1} & \text{for } \xi \neq 0 \\ \frac{1}{\sigma}e^{-\frac{(x-\mu)}{\sigma}} & \text{for } \xi = 0 \end{cases}, \quad x \in \begin{cases} [\mu,\infty] & \text{for } \xi \geq 0 \\ [\mu,\mu-\sigma/\xi] & \text{for } \xi < 0 \end{cases}$$

[] GeneralizedPareto() Generalized Pareto distribution with unit shape and unit scale, i.e. GeneralizedPareto(0, 1, 1) GeneralizedPareto(k, s) Generalized Pareto distribution with shape k and scale s, i.e. GeneralizedPareto(0, k, s) GeneralizedPareto(m, k, s) Generalized Pareto distribution with shape k, scale s and location m.

params(d) Get the parameters, i.e. (m, s, k) location(d) Get the location parameter, i.e. m scale(d) Get the scale parameter, i.e. s shape(d) Get the shape parameter, i.e. k

External links

- Generalized Pareto distribution on Wikipedia

source

1.98 Distributions.Geometric

`Distributions.Geometric` — *Type*.

`Geometric(p)`

A *Geometric distribution* characterizes the number of failures before the first success in a sequence of independent Bernoulli trials with success rate `p`.

$$P(X = k) = p(1-p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

[] Geometric() Geometric distribution with success rate 0.5 Geometric(p) Geometric distribution with success rate p

params(d) Get the parameters, i.e. (p,) succprob(d) Get the success rate, i.e. p failprob(d) Get the failure rate, i.e. 1 - p

External links

- Geometric distribution on Wikipedia

source

1.99 Distributions.Gumbel

`Distributions.Gumbel` — *Type.*

`Gumbel(, )`

The *Gumbel distribution* with location and scale has probability density function

$$f(x; \mu, \theta) = \frac{1}{\theta} e^{-(z+e^z)}, \quad \text{with } z = \frac{x-\mu}{\theta}$$

[] Gumbel() Gumbel distribution with zero location and unit scale, i.e. Gumbel(0, 1) Gumbel(u) Gumbel distribution with location u and unit scale, i.e. Gumbel(u, 1) Gumbel(u, b) Gumbel distribution with location u and scale b

params(d) Get the parameters, i.e. (u, b) location(d) Get the location parameter, i.e. u scale(d) Get the scale parameter, i.e. b

External links

- Gumbel distribution on Wikipedia

source

1.100 Distributions.Hypergeometric

`Distributions.Hypergeometric` — *Type.*

`Hypergeometric(s, f, n)`

A *Hypergeometric distribution* describes the number of successes in `n` draws without replacement from a finite population containing `s` successes and `f` failures.

$$P(X = k) = \frac{\binom{s}{k}\binom{f}{n-k}}{\binom{s+f}{n}}, \quad \text{for } k = \max(0, n-f), \ldots, \min(n, s).$$

[] Hypergeometric(s, f, n) Hypergeometric distribution for a population with s successes and f failures, and a sequence of n trials.

params(d) Get the parameters, i.e. (s, f, n)

External links

- Hypergeometric distribution on Wikipedia

source

1.101 Distributions.InverseGamma

`Distributions.InverseGamma` — *Type.*

`InverseGamma(, )`

The *inverse gamma distribution* with shape parameter and scale has probability density function

$$f(x; \alpha, \theta) = \frac{\theta^\alpha x^{-(\alpha+1)}}{\Gamma(\alpha)} e^{-\frac{\theta}{x}}, \quad x > 0$$

It is related to the `Gamma` distribution: if $X \sim \text{Gamma}(\alpha, \beta)$, then ‘1 / X $\sim$ InverseGamma$(\alpha, \beta\{-1\})$“.

[] InverseGamma() Inverse Gamma distribution with unit shape and unit scale, i.e. InverseGamma(1, 1) InverseGamma(a) Inverse Gamma distribution with shape a and unit scale, i.e. InverseGamma(a, 1) InverseGamma(a, b) Inverse Gamma distribution with shape a and scale b

params(d) Get the parameters, i.e. (a, b) shape(d) Get the shape parameter, i.e. a scale(d) Get the scale parameter, i.e. b

External links

- Inverse gamma distribution on Wikipedia

source

1.102 Distributions.InverseGaussian

`Distributions.InverseGaussian` — *Type.*

`InverseGaussian(,)`

The *inverse Gaussian distribution* with mean and shape has probability density function

$$f(x; \mu, \lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left(\frac{-\lambda(x-\mu)^2}{2\mu^2 x}\right), \quad x > 0$$

[] InverseGaussian() Inverse Gaussian distribution with unit mean and unit shape, i.e. InverseGaussian(1, 1) InverseGaussian(mu), Inverse Gaussian distribution with mean mu and unit shape, i.e. InverseGaussian(u, 1) InverseGaussian(mu, lambda) Inverse Gaussian distribution with mean mu and shape lambda

params(d) Get the parameters, i.e. (mu, lambda) mean(d) Get the mean parameter, i.e. mu shape(d) Get the shape parameter, i.e. lambda

External links

- Inverse Gaussian distribution on Wikipedia

source

1.103 Distributions.InverseWishart

`Distributions.InverseWishart` — *Type.*

`InverseWishart(nu, P)`

The [Inverse Wishart distribution](http://en.wikipedia.org/wiki/Inverse-Wishart_distribution is usually used a the conjugate prior for the covariance matrix of a multivariate normal distribution, which is characterized by a degree of freedom , and a base matrix .

source

1.104 Distributions.KSDist

`Distributions.KSDist` — *Type.*

`KSDist(n)`

Distribution of the (two-sided) Kolmogorov-Smirnoff statistic

$$D_n = \sup_x |\hat{F}_n(x) - F(x)| \sqrt(n)$$

D_n converges a.s. to the Kolmogorov distribution.

source

1.105 Distributions.KSOneSided

`Distributions.KSOneSided` — *Type.*

`KSOneSided(n)`

Distribution of the one-sided Kolmogorov-Smirnov test statistic:

$$D_n^+ = \sup_x (\hat{F}_n(x) - F(x))$$

source

1.106 Distributions.Kolmogorov

`Distributions.Kolmogorov` — *Type.*

`Kolmogorov()`

Kolmogorov distribution defined as

$$\sup_{t \in [0,1]} |B(t)|$$

where $B(t)$ is a Brownian bridge used in the Kolmogorov–Smirnov test for large n.

source

1.107 Distributions.Laplace

`Distributions.Laplace` — *Type.*

`Laplace(,)`

The *Laplace distribution* with location and scale has probability density function

$$f(x; \mu, \beta) = \frac{1}{2\beta} \exp\left(-\frac{|x-\mu|}{\beta}\right)$$

[] Laplace() Laplace distribution with zero location and unit scale, i.e. Laplace(0, 1) Laplace(u) Laplace distribution with location u and unit scale, i.e. Laplace(u, 1) Laplace(u, b) Laplace distribution with location u ans scale b

params(d) Get the parameters, i.e. (u, b) location(d) Get the location parameter, i.e. u scale(d) Get the scale parameter, i.e. b

External links

- Laplace distribution on Wikipedia

source

1.108 Distributions.Levy

`Distributions.Levy` — *Type.*

`Levy(, )`

The *Lvy distribution* with location and scale has probability density function

$$f(x; \mu, \sigma) = \sqrt{\frac{\sigma}{2\pi(x-\mu)^3}} \exp\left(-\frac{\sigma}{2(x-\mu)}\right), \quad x > \mu$$

[] Levy() Levy distribution with zero location and unit scale, i.e. Levy(0, 1) Levy(u) Levy distribution with location u and unit scale, i.e. Levy(u, 1) Levy(u, c) Levy distribution with location u ans scale c

params(d) Get the parameters, i.e. (u, c) location(d) Get the location parameter, i.e. u

External links

- Lvy distribution on Wikipedia

source

1.109 Distributions.LocationScale

`Distributions.LocationScale` — *Type.*

`LocationScale(,,)`

A location-scale transformed distribution with location parameter , scale parameter , and given distribution .

$$f(x) = \frac{1}{}\left(\frac{x - }{}\right)$$

[] LocationScale(,,) location-scale transformed distribution params(d) Get the parameters, i.e. (, , and the base distribution) location(d) Get the location parameter scale(d) Get the scale parameter

External links Location-Scale family on Wikipedia

source

1.110 Distributions.LogNormal

`Distributions.LogNormal` — *Type.*

`LogNormal(,)`

The *log normal distribution* is the distribution of the exponential of a `Normal` variate: if $X \sim \text{Normal}(\mu, \sigma)$ then $\exp(X) \sim \text{LogNormal}(\mu, \sigma)$. The probability density function is

$$f(x; \mu, \sigma) = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\log(x) - \mu)^2}{2\sigma^2}\right), \quad x > 0$$

[] LogNormal() Log-normal distribution with zero log-mean and unit scale LogNormal(mu) Log-normal distribution with log-mean mu and unit scale LogNormal(mu, sig) Log-normal distribution with log-mean mu and scale sig

params(d) Get the parameters, i.e. (mu, sig) meanlogx(d) Get the mean of log(X), i.e. mu varlogx(d) Get the variance of log(X), i.e. $\text{sig}^2 stdlogx(d) Getthestandarddeviationoflog($

External links

- Log normal distribution on Wikipedia

source

1.111 Distributions.Logistic

`Distributions.Logistic` — *Type*.

`Logistic(,)`

The *Logistic distribution* with location and scale has probability density function

$$f(x; \mu, \theta) = \frac{1}{4\theta} \operatorname{sech}^2 \left(\frac{x - \mu}{2\theta} \right)$$

[] Logistic() Logistic distribution with zero location and unit scale, i.e. Logistic(0, 1) Logistic(u) Logistic distribution with location u and unit scale, i.e. Logistic(u, 1) Logistic(u, b) Logistic distribution with location u ans scale b

params(d) Get the parameters, i.e. (u, b) location(d) Get the location parameter, i.e. u scale(d) Get the scale parameter, i.e. b

External links

- Logistic distribution on Wikipedia

source

1.112 Distributions.MixtureModel

`Distributions.MixtureModel` — *Method*.

`MixtureModel(components, [prior])`

Construct a mixture model with a vector of `components` and a `prior` probability vector. If no `prior` is provided then all components will have the same prior probabilities.

source

1.113 Distributions.MixtureModel

`Distributions.MixtureModel` — *Method*.

`MixtureModel(C, params, [prior])`

Construct a mixture model with component type C, a vector of parameters for constructing the components given by *params*, and a prior probability vector. If no `prior` is provided then all components will have the same prior probabilities.

source

1.114 Distributions.Multinomial

`Distributions.Multinomial` — *Type.*

The Multinomial distribution generalizes the *binomial distribution.* Consider n independent draws from a Categorical distribution over a finite set of size k, and let $X = (X_1, ..., X_k)$ where X_i represents the number of times the element i occurs, then the distribution of X is a multinomial distribution. Each sample of a multinomial distribution is a k-dimensional integer vector that sums to n.

The probability mass function is given by

$$f(x; n, p) = \frac{n!}{x_1! \cdots x_k!} \prod_{i=1}^{k} p_i^{x_i}, \quad x_1 + \cdots + x_k = n$$

[] Multinomial(n, p) Multinomial distribution for n trials with probability vector p Multinomial(n, k) Multinomial distribution for n trials with equal probabilities over 1:k

source

1.115 Distributions.MvLogNormal

`Distributions.MvLogNormal` — *Type.*

```
MvLogNormal(d::MvNormal)
```

The Multivariate lognormal distribution is a multidimensional generalization of the *lognormal distribution.*

If $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ has a multivariate normal distribution then $\boldsymbol{Y} = \exp(\boldsymbol{X})$ has a multivariate lognormal distribution.

Mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ of the underlying normal distribution are known as the *location* and *scale* parameters of the corresponding lognormal distribution.

source

1.116 Distributions.MvNormal

`Distributions.MvNormal` — *Type.*

```
MvNormal
```

Generally, users don't have to worry about these internal details. We provide a common constructor `MvNormal`, which will construct a distribution of appropriate type depending on the input arguments.

```
MvNormal(sig)
```

Construct a multivariate normal distribution with zero mean and covariance represented by `sig`.

`MvNormal(mu, sig)`

Construct a multivariate normal distribution with mean `mu` and covariance represented by `sig`.

`MvNormal(d, sig)`

Construct a multivariate normal distribution of dimension `d`, with zero mean, and an isotropic covariance as `abs2(sig) * eye(d)`.

Arguments

- `mu::Vector{T<:Real}`: The mean vector.
- `d::Real`: dimension of distribution.
- `sig`: The covariance, which can in of either of the following forms (with `T<:Real`):
  1. subtype of `AbstractPDMat`
  2. symmetric matrix of type `Matrix{T}`
  3. vector of type `Vector{T}`: indicating a diagonal covariance as `diagm(abs2(sig))`.
  4. real-valued number: indicating an isotropic covariance as `abs2(sig) * eye(d)`.

Note: The constructor will choose an appropriate covariance form internally, so that special structure of the covariance can be exploited.

source

1.117 Distributions.MvNormalCanon

`Distributions.MvNormalCanon` — *Type.*

`MvNormalCanon`

Multivariate normal distribution is an exponential family distribution, with two *canonical parameters*: the *potential vector* $\mathbf{h}$ and the *precision matrix* $\mathbf{J}$. The relation between these parameters and the conventional representation (*i.e.* the one using mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$) is:

$$\mathbf{h} = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}, \quad \text{and} \quad \mathbf{J} = \boldsymbol{\Sigma}^{-1}$$

The canonical parameterization is widely used in Bayesian analysis. We provide a type `MvNormalCanon`, which is also a subtype of `AbstractMvNormal` to represent a multivariate normal distribution using canonical parameters. Particularly, `MvNormalCanon` is defined as:

[] immutable MvNormalCanon{P¡:AbstractPDMat,V¡:Union{Vector,ZeroVector}} ¡: AbstractMvNormal ::V the mean vector h::V potential vector, i.e. inv() * J::P precision matrix, i.e. inv() end

We also define aliases for common specializations of this parametric type:

[] const FullNormalCanon = MvNormalCanon{PDMat, Vector{Float64}} const DiagNormalCanon = MvNormalCanon{PDiagMat, Vector{Float64}} const IsoNormalCanon = MvNormalCanon{ScalMat, Vector{Float64}}

const ZeroMeanFullNormalCanon = MvNormalCanon{PDMat, ZeroVector{Float64}} const ZeroMeanDiagNormalCanon = MvNormalCanon{PDiagMat, ZeroVector{Float64}} const ZeroMeanIsoNormalCanon = MvNormalCanon{ScalMat, ZeroVector{Float64}}

A multivariate distribution with canonical parameterization can be constructed using a common constructor `MvNormalCanon` as:

`MvNormalCanon(h, J)`

Construct a multivariate normal distribution with potential vector `h` and precision matrix represented by `J`.

`MvNormalCanon(J)`

Construct a multivariate normal distribution with zero mean (thus zero potential vector) and precision matrix represented by `J`.

`MvNormalCanon(d, J)`

Construct a multivariate normal distribution of dimension `d`, with zero mean and a precision matrix as `J * eye(d)`.

Arguments

- `d::Int`: dimension of distribution
- `h::Vector{T<:Real}`: the potential vector, of type `Vector{T}` with `T<:Real`.
- `J`: the representation of the precision matrix, which can be in either of the following forms (`T<:Real`):
  1. an instance of a subtype of `AbstractPDMat`
  2. a square matrix of type `Matrix{T}`
  3. a vector of type `Vector{T}`: indicating a diagonal precision matrix as `diagm(J)`.
  4. a real number: indicating an isotropic precision matrix as `J * eye(d)`.

Note: `MvNormalCanon` share the same set of methods as `MvNormal`.

source

1.118 Distributions.NegativeBinomial

`Distributions.NegativeBinomial` — *Type.*

`NegativeBinomial(r,p)`

A *Negative binomial distribution* describes the number of failures before the `r`th success in a sequence of independent Bernoulli trials. It is parameterized by `r`, the number of successes, and `p`, the probability of success in an individual trial.

$$P(X = k) = \binom{k+r-1}{k} p^r (1-p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

The distribution remains well-defined for any positive `r`, in which case

$$P(X = k) = \frac{\Gamma(k+r)}{k!\Gamma(r)} p^r (1-p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

[] NegativeBinomial() Negative binomial distribution with r = 1 and p = 0.5 NegativeBinomial(r, p) Negative binomial distribution with r successes and success rate p

params(d) Get the parameters, i.e. (r, p) succprob(d) Get the success rate, i.e. p failprob(d) Get the failure rate, i.e. 1 - p

External links:

- Negative binomial distribution on Wikipedia

source

1.119 Distributions.NoncentralBeta

`Distributions.NoncentralBeta` — *Type.*

`NoncentralBeta(, , )`

source

1.120 Distributions.NoncentralChisq

`Distributions.NoncentralChisq` — *Type.*

`NoncentralChisq(, )`

The *noncentral chi-squared distribution* with degrees of freedom and non-centrality parameter has the probability density function

$$f(x; \nu, \lambda) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{\nu/4-1/2} I_{\nu/2-1}(\sqrt{\lambda x}), \quad x > 0$$

It is the distribution of the sum of squares of independent Normal variates with individual means μ_i and

$$\lambda = \sum_{i=1}^{\nu} \mu_i^2$$

[] NoncentralChisq(,) Noncentral chi-squared distribution with degrees of freedom and noncentrality parameter

params(d) Get the parameters, i.e. (,)

External links

- Noncentral chi-squared distribution on Wikipedia

source

1.121 Distributions.NoncentralF

`Distributions.NoncentralF` — *Type.*

```
NoncentralF(1, 2, )
```

source

1.122 Distributions.NoncentralT

`Distributions.NoncentralT` — *Type.*

```
NoncentralT(, )
```

source

1.123 Distributions.Normal

`Distributions.Normal` — *Type.*

```
Normal(,)
```

The *Normal distribution* with mean and standard deviation has probability density function

$$f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

[] Normal() standard Normal distribution with zero mean and unit variance Normal(mu) Normal distribution with mean mu and unit variance Normal(mu, sig) Normal distribution with mean mu and variance sig^2

params(d) Get the parameters, i.e. (mu, sig) mean(d) Get the mean, i.e. mu std(d) Get the standard deviation, i.e. sig

External links

- Normal distribution on Wikipedia

source

1.124 Distributions.NormalCanon

`Distributions.NormalCanon` — *Type.*

`NormalCanon(, )`

Canonical Form of Normal distribution
source

1.125 Distributions.NormalInverseGaussian

`Distributions.NormalInverseGaussian` — *Type.*

`NormalInverseGaussian(,,,)`

The *Normal-inverse Gaussian distribution* with location , tail heaviness , asymmetry parameter and scale has probability density function

$$f(x;\mu,\alpha,\beta,\delta) = \frac{\alpha\delta K_1\left(\alpha\sqrt{\delta^2+(x-\mu)^2}\right)}{\pi\sqrt{\delta^2+(x-\mu)^2}}\, e^{\delta\gamma+\beta(x-\mu)}$$

where K_j denotes a modified Bessel function of the third kind.
External links

- Normal-inverse Gaussian distribution on Wikipedia

source

1.126 Distributions.Pareto

`Distributions.Pareto` — *Type.*

`Pareto(,)`

The *Pareto distribution* with shape and scale has probability density function

$$f(x;\alpha,\theta) = \frac{\alpha\theta^\alpha}{x^{\alpha+1}}, \quad x \geq \theta$$

[] Pareto() Pareto distribution with unit shape and unit scale, i.e. Pareto(1, 1) Pareto(a) Pareto distribution with shape a and unit scale, i.e. Pareto(a, 1) Pareto(a, b) Pareto distribution with shape a and scale b

params(d) Get the parameters, i.e. (a, b) shape(d) Get the shape parameter, i.e. a scale(d) Get the scale parameter, i.e. b
External links

- Pareto distribution on Wikipedia

source

1.127 Distributions.Poisson

`Distributions.Poisson` — *Type.*

`Poisson()`

A *Poisson distribution* descibes the number of independent events occurring within a unit time interval, given the average rate of occurrence .

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}, \quad \text{for } k = 0, 1, 2, \ldots.$$

[] Poisson() Poisson distribution with rate parameter 1 Poisson(lambda) Poisson distribution with rate parameter lambda

params(d) Get the parameters, i.e. (,) mean(d) Get the mean arrival rate, i.e.

External links:

- Poisson distribution on Wikipedia

source

1.128 Distributions.PoissonBinomial

`Distributions.PoissonBinomial` — *Type.*

`PoissonBinomial(p)`

A *Poisson-binomial distribution* describes the number of successes in a sequence of independent trials, wherein each trial has a different success rate. It is parameterized by a vector `p` (of length K), where K is the total number of trials and `p[i]` corresponds to the probability of success of the `i`th trial.

$$P(X = k) = \sum_{A \in F_k} \prod_{i \in A} p[i] \prod_{j \in A^c} (1 - p[j]), \quad \text{for } k = 0, 1, 2, \ldots, K,$$

where F_k is the set of all subsets of k integers that can be selected from $\{1, 2, 3, ..., K\}$.

[] PoissonBinomial(p) Poisson Binomial distribution with success rate vector p

params(d) Get the parameters, i.e. (p,) succprob(d) Get the vector of success rates, i.e. p failprob(d) Get the vector of failure rates, i.e. 1-p

External links:

- Poisson-binomial distribution on Wikipedia

source

1.129 Distributions.Rayleigh

`Distributions.Rayleigh` — *Type.*

`Rayleigh()`

The *Rayleigh distribution* with scale has probability density function

$$f(x;\sigma) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}, \quad x > 0$$

It is related to the `Normal` distribution via the property that if $X, Y \sim \text{Normal}(0, \sigma)$, independently, then $\sqrt{X^2 + Y^2} \sim \text{Rayleigh}(\sigma)$.

[] Rayleigh() Rayleigh distribution with unit scale, i.e. Rayleigh(1) Rayleigh(s) Rayleigh distribution with scale s

params(d) Get the parameters, i.e. (s,) scale(d) Get the scale parameter, i.e. s

External links

- Rayleigh distribution on Wikipedia

source

1.130 Distributions.Semicircle

`Distributions.Semicircle` — *Type.*

`Semicircle(r)`

The Wigner semicircle distribution with radius parameter `r` has probability density function

$$f(x;r) = \frac{2}{\pi r^2}\sqrt{r^2 - x^2}, \quad x \in [-r, r].$$

[] Semicircle(r) Wigner semicircle distribution with radius r

params(d) Get the radius parameter, i.e. (r,)

External links

- Wigner semicircle distribution on Wikipedia

source

1.131 Distributions.Skellam

`Distributions.Skellam` — *Type.*

`Skellam(1, 2)`

A *Skellam distribution* describes the difference between two independent `Poisson` variables, respectively with rate `1` and `2`.

$$P(X = k) = e^{-(\mu_1+\mu_2)} \left(\frac{\mu_1}{\mu_2}\right)^{k/2} I_k(2\sqrt{\mu_1\mu_2}) \quad \text{for integer } k$$

where I_k is the modified Bessel function of the first kind.

[] Skellam(mu1, mu2) Skellam distribution for the difference between two Poisson variables, respectively with expected values mu1 and mu2.

params(d) Get the parameters, i.e. (mu1, mu2)

External links:

- Skellam distribution on Wikipedia

source

1.132 Distributions.SymTriangularDist

`Distributions.SymTriangularDist` — *Type.*

`SymTriangularDist(,)`

The *Symmetric triangular distribution* with location and scale has probability density function

$$f(x; \mu, \sigma) = \frac{1}{\sigma}\left(1 - \left|\frac{x-\mu}{\sigma}\right|\right), \quad \mu - \sigma \le x \le \mu + \sigma$$

[] SymTriangularDist() Symmetric triangular distribution with zero location and unit scale SymTriangularDist(u) Symmetric triangular distribution with location u and unit scale SymTriangularDist(u, s) Symmetric triangular distribution with location u and scale s

params(d) Get the parameters, i.e. (u, s) location(d) Get the location parameter, i.e. u scale(d) Get the scale parameter, i.e. s

source

1.133 Distributions.TDist

`Distributions.TDist` — *Type.*

`TDist()`

The *Students T distribution* with degrees of freedom has probability density function

$$f(x;d) = \frac{1}{\sqrt{d}B(1/2,d/2)}\left(1+\frac{x^2}{d}\right)^{-\frac{d+1}{2}}$$

[] TDist(d) t-distribution with d degrees of freedom

params(d) Get the parameters, i.e. (d,) dof(d) Get the degrees of freedom, i.e. d

External links

Student's T distribution on Wikipedia

source

1.134 Distributions.TriangularDist

`Distributions.TriangularDist` — *Type.*

`TriangularDist(a,b,c)`

The *triangular distribution* with lower limit `a`, upper limit `b` and mode `c` has probability density function

$$f(x;a,b,c) = \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \le x \le c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \le b, \\ 0 & \text{for } b < x, \end{cases}$$

[] TriangularDist(a, b) Triangular distribution with lower limit a, upper limit b, and mode (a+b)/2 TriangularDist(a, b, c) Triangular distribution with lower limit a, upper limit b, and mode c

params(d) Get the parameters, i.e. (a, b, c) minimum(d) Get the lower bound, i.e. a maximum(d) Get the upper bound, i.e. b mode(d) Get the mode, i.e. c

External links

- Triangular distribution on Wikipedia

source

1.135 Distributions.Triweight

`Distributions.Triweight` — *Type.*

`Triweight(, )`

source

1.136 Distributions.Truncated

`Distributions.Truncated` — *Type.*

`Truncated(d, l, u):`

Construct a truncated distribution.
Arguments

- `d::UnivariateDistribution`: The original distribution.
- `l::Real`: The lower bound of the truncation, which can be a finite value or `-Inf`.
- `u::Real`: The upper bound of the truncation, which can be a finite value of `Inf`.

source

1.137 Distributions.Uniform

`Distributions.Uniform` — *Type.*

`Uniform(a,b)`

The *continuous uniform distribution* over an interval $[a, b]$ has probability density function

$$f(x; a, b) = \frac{1}{b-a}, \quad a \leq x \leq b$$

[] Uniform() Uniform distribution over [0, 1] Uniform(a, b) Uniform distribution over [a, b]

params(d) Get the parameters, i.e. (a, b) minimum(d) Get the lower bound, i.e. a maximum(d) Get the upper bound, i.e. b location(d) Get the location parameter, i.e. a scale(d) Get the scale parameter, i.e. b - a

External links

- Uniform distribution (continuous) on Wikipedia

source

1.138 Distributions.VonMises

`Distributions.VonMises` — *Type.*

`VonMises(, )`

The *von Mises distribution* with mean and concentration has probability density function

$$f(x;\mu,\kappa) = \frac{1}{2\pi I_0(\kappa)} \exp\left(\kappa \cos(x-\mu)\right)$$

[] VonMises() von Mises distribution with zero mean and unit concentration VonMises() von Mises distribution with zero mean and concentration VonMises(,) von Mises distribution with mean and concentration

External links

- von Mises distribution on Wikipedia

source

1.139 Distributions.Weibull

`Distributions.Weibull` — *Type.*

`Weibull(,)`

The *Weibull distribution* with shape and scale has probability density function

$$f(x;\alpha,\theta) = \frac{\alpha}{\theta}\left(\frac{x}{\theta}\right)^{\alpha-1} e^{-(x/\theta)^{\alpha}}, \quad x \geq 0$$

[] Weibull() Weibull distribution with unit shape and unit scale, i.e. Weibull(1, 1) Weibull(a) Weibull distribution with shape a and unit scale, i.e. Weibull(a, 1) Weibull(a, b) Weibull distribution with shape a and scale b

params(d) Get the parameters, i.e. (a, b) shape(d) Get the shape parameter, i.e. a scale(d) Get the scale parameter, i.e. b

External links

- Weibull distribution on Wikipedia

source

1.140 Distributions.Wishart

`Distributions.Wishart` — *Type.*

`Wishart(nu, S)`

The Wishart distribution is a multidimensional generalization of the Chi-square distribution, which is characterized by a degree of freedom , and a base matrix S.

source

Chapter 2

StatsBase

2.1 Base.stdm

`Base.stdm` — *Method.*

```
stdm(v, w::AbstractWeights, m, [dim]; corrected=false)
```

Compute the standard deviation of a real-valued array `x` with a known mean `m`, optionally over a dimension `dim`. Observations in `x` are weighted using weight vector `w`. The uncorrected (when `corrected=false`) sample standard deviation is defined as:

$$\sqrt{\frac{1}{\sum w}\sum_{i=1}^{n} w_i \left(x_i - m\right)^2}$$

where n is the length of the input. The unbiased estimate (when `corrected=true`) of the population standard deviation is computed by replacing $\frac{1}{\sum w}$ with a factor dependent on the type of weights used:

- `AnalyticWeights`: $\frac{1}{\sum w - \sum w^2 / \sum w}$
- `FrequencyWeights`: $\frac{1}{\sum w - 1}$
- `ProbabilityWeights`: $\frac{n}{(n-1)\sum w}$ where n equals `count(!iszero, w)`
- `Weights`: `ArgumentError` (bias correction not supported)

source

2.2 Base.varm

`Base.varm` — *Method.*

```
varm(x, w::AbstractWeights, m, [dim]; corrected=false)
```

Compute the variance of a real-valued array `x` with a known mean `m`, optionally over a dimension `dim`. Observations in `x` are weighted using weight vector `w`. The uncorrected (when `corrected=false`) sample variance is defined as:

$$\frac{1}{\sum w} \sum_{i=1}^{n} w_i \left(x_i - m\right)^2$$

where n is the length of the input. The unbiased estimate (when `corrected=true`) of the population variance is computed by replacing $\frac{1}{\sum w}$ with a factor dependent on the type of weights used:

- `AnalyticWeights`: $\frac{1}{\sum w - \sum w^2 / \sum w}$
- `FrequencyWeights`: $\frac{1}{\sum w - 1}$
- `ProbabilityWeights`: $\frac{n}{(n-1)\sum w}$ where n equals `count(!iszero, w)`
- `Weights`: `ArgumentError` (bias correction not supported)

source

2.3 StatsBase.L1dist

`StatsBase.L1dist` — *Method.*

```
L1dist(a, b)
```

Compute the L1 distance between two arrays: $\sum_{i=1}^{n} |a_i - b_i|$. Efficient equivalent of `sum(abs, a - b)`.

source

2.4 StatsBase.L2dist

`StatsBase.L2dist` — *Method.*

```
L2dist(a, b)
```

Compute the L2 distance between two arrays: $\sqrt{\sum_{i=1}^{n} |a_i - b_i|^2}$. Efficient equivalent of `sqrt(sumabs2(a - b))`.

source

2.5 StatsBase.Linfdist

`StatsBase.Linfdist` — *Method.*

```
Linfdist(a, b)
```

Compute the L distance, also called the Chebyshev distance, between two arrays: $\max_{i \in 1:n} |a_i - b_i|$. Efficient equivalent of `maxabs(a - b)`.

source

2.6 StatsBase.addcounts!

`StatsBase.addcounts!` — *Method.*

```
addcounts!(r, x, levels::UnitRange{<:Int}, [wv::AbstractWeights])
```

Add the number of occurrences in `x` of each value in `levels` to an existing array `r`. If a weighting vector `wv` is specified, the sum of weights is used rather than the raw counts.

source

2.7 StatsBase.addcounts!

`StatsBase.addcounts!` — *Method.*

```
addcounts!(dict, x[, wv]; alg = :auto)
```

Add counts based on `x` to a count map. New entries will be added if new values come up. If a weighting vector `wv` is specified, the sum of the weights is used rather than the raw counts.

`alg` can be one of:

- `:auto` (default): if `StatsBase.radixsort_safe(eltype(x)) == true` then use `:radixsort`, otherwise use `:dict`.
- `:radixsort`: if `radixsort_safe(eltype(x)) == true` then use the radix sort algorithm to sort the input vector which will generally lead to shorter running time. However the radix sort algorithm creates a copy of the input vector and hence uses more RAM. Choose `:dict` if the amount of available RAM is a limitation.
- `:dict`: use `Dict`-based method which is generally slower but uses less RAM and is safe for any data type.

source

2.8 StatsBase.adjr2

`StatsBase.adjr2` — *Method.*

```
adjr2(obj::StatisticalModel, variant::Symbol)
adjr(obj::StatisticalModel, variant::Symbol)
```

Adjusted coefficient of determination (adjusted R-squared).

For linear models, the adjusted R is defined as $1-(1-(1-R^2)(n-1)/(n-p))$, with R^2 the coefficient of determination, n the number of observations, and p the number of coefficients (including the intercept). This definition is generally known as the Wherry Formula I.

For other models, one of the several pseudo R definitions must be chosen via `variant`. The only currently supported variant is `:MacFadden`, defined as $1 - (\log L - k)/\log L0$. In this formula, L is the likelihood of the model, $L0$ that of the null model (the model including only the intercept). These two quantities are taken to be minus half `deviance` of the corresponding models. k is the number of consumed degrees of freedom of the model (as returned by `dof`).

source

2.9 StatsBase.aic

`StatsBase.aic` — *Method.*

```
aic(obj::StatisticalModel)
```

Akaike's Information Criterion, defined as $-2\log L+2k$, with L the likelihood of the model, and `k` its number of consumed degrees of freedom (as returned by `dof`).

source

2.10 StatsBase.aicc

`StatsBase.aicc` — *Method.*

```
aicc(obj::StatisticalModel)
```

Corrected Akaike's Information Criterion for small sample sizes (Hurvich and Tsai 1989), defined as $-2\log L+2k+2k(k-1)/(n-k-1)$, with L the likelihood of the model, k its number of consumed degrees of freedom (as returned by `dof`), and n the number of observations (as returned by `nobs`).

source

2.11 StatsBase.autocor!

`StatsBase.autocor!` — *Method*.

```
autocor!(r, x, lags; demean=true)
```

Compute the autocorrelation function (ACF) of a vector or matrix `x` at `lags` and store the result in `r`. `demean` denotes whether the mean of `x` should be subtracted from `x` before computing the ACF.

If `x` is a vector, `r` must be a vector of the same length as `x`. If `x` is a matrix, `r` must be a matrix of size `(length(lags), size(x,2))`, and where each column in the result will correspond to a column in `x`.

The output is normalized by the variance of `x`, i.e. so that the lag 0 autocorrelation is 1. See `autocov!` for the unnormalized form.

source

2.12 StatsBase.autocor

`StatsBase.autocor` — *Method*.

```
autocor(x, [lags]; demean=true)
```

Compute the autocorrelation function (ACF) of a vector or matrix `x`, optionally specifying the `lags`. `demean` denotes whether the mean of `x` should be subtracted from `x` before computing the ACF.

If `x` is a vector, return a vector of the same length as `x`. If `x` is a matrix, return a matrix of size `(length(lags), size(x,2))`, where each column in the result corresponds to a column in `x`.

When left unspecified, the lags used are the integers from 0 to `min(size(x,1)-1, 10*log10(size(x,1)))`.

The output is normalized by the variance of `x`, i.e. so that the lag 0 autocorrelation is 1. See `autocov` for the unnormalized form.

source

2.13 StatsBase.autocov!

`StatsBase.autocov!` — *Method*.

```
autocov!(r, x, lags; demean=true)
```

Compute the autocovariance of a vector or matrix `x` at `lags` and store the result in `r`. `demean` denotes whether the mean of `x` should be subtracted from `x` before computing the autocovariance.

If `x` is a vector, `r` must be a vector of the same length as `x`. If `x` is a matrix, `r` must be a matrix of size `(length(lags), size(x,2))`, and where each column in the result will correspond to a column in `x`.

The output is not normalized. See `autocor!` for a method with normalization.

source

2.14 StatsBase.autocov

`StatsBase.autocov` — *Method.*

```
autocov(x, [lags]; demean=true)
```

Compute the autocovariance of a vector or matrix `x`, optionally specifying the `lags` at which to compute the autocovariance. `demean` denotes whether the mean of `x` should be subtracted from `x` before computing the autocovariance.

If `x` is a vector, return a vector of the same length as `x`. If `x` is a matrix, return a matrix of size `(length(lags), size(x,2))`, where each column in the result corresponds to a column in `x`.

When left unspecified, the lags used are the integers from 0 to `min(size(x,1)-1, 10*log10(size(x,1)))`.

The output is not normalized. See `autocor` for a function with normalization.

source

2.15 StatsBase.aweights

`StatsBase.aweights` — *Method.*

```
aweights(vs)
```

Construct an `AnalyticWeights` vector from array `vs`. See the documentation for `AnalyticWeights` for more details.

source

2.16 StatsBase.bic

`StatsBase.bic` — *Method.*

```
bic(obj::StatisticalModel)
```

Bayesian Information Criterion, defined as $-2 \log L + k \log n$, with L the likelihood of the model, k its number of consumed degrees of freedom (as returned by `dof`), and n the number of observations (as returned by `nobs`).

source

2.17 StatsBase.coef

`StatsBase.coef` — *Method*.

```
coef(obj::StatisticalModel)
```

Return the coefficients of the model.
source

2.18 StatsBase.coefnames

`StatsBase.coefnames` — *Method*.

```
coefnames(obj::StatisticalModel)
```

Return the names of the coefficients.
source

2.19 StatsBase.coeftable

`StatsBase.coeftable` — *Method*.

```
coeftable(obj::StatisticalModel)
```

Return a table of class `CoefTable` with coefficients and related statistics.
source

2.20 StatsBase.competerank

`StatsBase.competerank` — *Method*.

```
competerank(x; lt = isless, rev::Bool = false)
```

Return the standard competition ranking ("1224" ranking) of an array. The `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. Items that compare equal are given the same rank, then a gap is left in the rankings the size of the number of tied items - 1. Missing values are assigned rank `missing`.
source

2.21 StatsBase.confint

`StatsBase.confint` — *Method*.

```
confint(obj::StatisticalModel)
```

Compute confidence intervals for coefficients.
source

2.22 StatsBase.cor2cov

`StatsBase.cor2cov` — *Method*.

```
cor2cov(C, s)
```

Compute the covariance matrix from the correlation matrix `C` and a vector of standard deviations `s`. Use `StatsBase.cor2cov!` for an in-place version.

source

2.23 StatsBase.corkendall

`StatsBase.corkendall` — *Method*.

```
corkendall(x, y=x)
```

Compute Kendall's rank correlation coefficient, . `x` and `y` must both be either matrices or vectors.

source

2.24 StatsBase.corspearman

`StatsBase.corspearman` — *Method*.

```
corspearman(x, y=x)
```

Compute Spearman's rank correlation coefficient. If `x` and `y` are vectors, the output is a float, otherwise it's a matrix corresponding to the pairwise correlations of the columns of `x` and `y`.

source

2.25 StatsBase.counteq

`StatsBase.counteq` — *Method*.

```
counteq(a, b)
```

Count the number of indices at which the elements of the arrays `a` and `b` are equal.

source

2.26 StatsBase.countmap

`StatsBase.countmap` — *Method.*

```
countmap(x; alg = :auto)
```

Return a dictionary mapping each unique value in `x` to its number of occurrences.

- `:auto` (default): if `StatsBase.radixsort_safe(eltype(x)) == true` then use `:radixsort`, otherwise use `:dict`.
- `:radixsort`: if `radixsort_safe(eltype(x)) == true` then use the radix sort algorithm to sort the input vector which will generally lead to shorter running time. However the radix sort algorithm creates a copy of the input vector and hence uses more RAM. Choose `:dict` if the amount of available RAM is a limitation.
- `:dict`: use `Dict`-based method which is generally slower but uses less RAM and is safe for any data type.

source

2.27 StatsBase.countne

`StatsBase.countne` — *Method.*

```
countne(a, b)
```

Count the number of indices at which the elements of the arrays `a` and `b` are not equal.

source

2.28 StatsBase.counts

`StatsBase.counts` — *Function.*

```
counts(x, [wv::AbstractWeights])
counts(x, levels::UnitRange{<:Integer}, [wv::AbstractWeights])
counts(x, k::Integer, [wv::AbstractWeights])
```

Count the number of times each value in `x` occurs. If `levels` is provided, only values falling in that range will be considered (the others will be ignored without raising an error or a warning). If an integer `k` is provided, only values in the range `1:k` will be considered.

If a weighting vector `wv` is specified, the sum of the weights is used rather than the raw counts.

The output is a vector of length `length(levels)`.

source

2.29 StatsBase.cov2cor

`StatsBase.cov2cor` — *Method.*

```
cov2cor(C, s)
```

Compute the correlation matrix from the covariance matrix `C` and a vector of standard deviations `s`. Use `Base.cov2cor!` for an in-place version.

source

2.30 StatsBase.crosscor!

`StatsBase.crosscor!` — *Method.*

```
crosscor!(r, x, y, lags; demean=true)
```

Compute the cross correlation between real-valued vectors or matrices `x` and `y` at `lags` and store the result in `r`. `demean` specifies whether the respective means of `x` and `y` should be subtracted from them before computing their cross correlation.

If both `x` and `y` are vectors, `r` must be a vector of the same length as `lags`. If either `x` is a matrix and `y` is a vector, `r` must be a matrix of size `(length(lags), size(x, 2))`; if `x` is a vector and `y` is a matrix, `r` must be a matrix of size `(length(lags), size(y, 2))`. If both `x` and `y` are matrices, `r` must be a three-dimensional array of size `(length(lags), size(x, 2), size(y, 2))`.

The output is normalized by `sqrt(var(x)*var(y))`. See `crosscov!` for the unnormalized form.

source

2.31 StatsBase.crosscor

`StatsBase.crosscor` — *Method.*

```
crosscor(x, y, [lags]; demean=true)
```

Compute the cross correlation between real-valued vectors or matrices `x` and `y`, optionally specifying the `lags`. `demean` specifies whether the respective means of `x` and `y` should be subtracted from them before computing their cross correlation.

If both `x` and `y` are vectors, return a vector of the same length as `lags`. Otherwise, compute cross covariances between each pairs of columns in `x` and `y`.

When left unspecified, the lags used are the integers from `-min(size(x,1)-1, 10*log10(size(x,1)))` to `min(size(x,1), 10*log10(size(x,1)))`.

The output is normalized by `sqrt(var(x)*var(y))`. See `crosscov` for the unnormalized form.

source

2.32 StatsBase.crosscov!

`StatsBase.crosscov!` — *Method.*

```
crosscov!(r, x, y, lags; demean=true)
```

Compute the cross covariance function (CCF) between real-valued vectors or matrices `x` and `y` at `lags` and store the result in `r`. `demean` specifies whether the respective means of `x` and `y` should be subtracted from them before computing their CCF.

If both `x` and `y` are vectors, `r` must be a vector of the same length as `lags`. If either `x` is a matrix and `y` is a vector, `r` must be a matrix of size `(length(lags), size(x, 2))`; if `x` is a vector and `y` is a matrix, `r` must be a matrix of size `(length(lags), size(y, 2))`. If both `x` and `y` are matrices, `r` must be a three-dimensional array of size `(length(lags), size(x, 2), size(y, 2))`.

The output is not normalized. See `crosscor!` for a function with normalization.

source

2.33 StatsBase.crosscov

`StatsBase.crosscov` — *Method.*

```
crosscov(x, y, [lags]; demean=true)
```

Compute the cross covariance function (CCF) between real-valued vectors or matrices `x` and `y`, optionally specifying the `lags`. `demean` specifies whether the respective means of `x` and `y` should be subtracted from them before computing their CCF.

If both `x` and `y` are vectors, return a vector of the same length as `lags`. Otherwise, compute cross covariances between each pairs of columns in `x` and `y`.

When left unspecified, the lags used are the integers from `-min(size(x,1)-1, 10*log10(size(x,1)))` to `min(size(x,1), 10*log10(size(x,1)))`.

The output is not normalized. See `crosscor` for a function with normalization.

source

2.34 StatsBase.crossentropy

`StatsBase.crossentropy` — *Method.*

```
crossentropy(p, q, [b])
```

Compute the cross entropy between `p` and `q`, optionally specifying a real number `b` such that the result is scaled by `1/log(b)`.

source

2.35 StatsBase.denserank

`StatsBase.denserank` — *Method.*

```
denserank(x)
```

Return the dense ranking ("1223" ranking) of an array. The `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. Items that compare equal receive the same ranking, and the next subsequent rank is assigned with no gap. Missing values are assigned rank `missing`.

source

2.36 StatsBase.describe

`StatsBase.describe` — *Method.*

```
describe(a)
```

Pretty-print the summary statistics provided by `summarystats`: the mean, minimum, 25th percentile, median, 75th percentile, and maximum.

source

2.37 StatsBase.deviance

`StatsBase.deviance` — *Method.*

```
deviance(obj::StatisticalModel)
```

Return the deviance of the model relative to a reference, which is usually when applicable the saturated model. It is equal, *up to a constant*, to $-2 \log L$, with L the likelihood of the model.

source

2.38 StatsBase.dof

`StatsBase.dof` — *Method.*

```
dof(obj::StatisticalModel)
```

Return the number of degrees of freedom consumed in the model, including when applicable the intercept and the distribution's dispersion parameter.

source

2.39 StatsBase.dof_residual

`StatsBase.dof_residual` — *Method*.

```
dof_residual(obj::RegressionModel)
```

Return the residual degrees of freedom of the model.
source

2.40 StatsBase.ecdf

`StatsBase.ecdf` — *Method*.

```
ecdf(X)
```

Return an empirical cumulative distribution function (ECDF) based on a vector of samples given in `X`.

Note: this is a higher-level function that returns a function, which can then be applied to evaluate CDF values on other samples.
source

2.41 StatsBase.entropy

`StatsBase.entropy` — *Method*.

```
entropy(p, [b])
```

Compute the entropy of an array `p`, optionally specifying a real number `b` such that the entropy is scaled by `1/log(b)`.
source

2.42 StatsBase.findat

`StatsBase.findat` — *Method*.

```
findat(a, b)
```

For each element in `b`, find its first index in `a`. If the value does not occur in `a`, the corresponding index is 0.
source

2.43 StatsBase.fit!

`StatsBase.fit!` — *Method*.

Fit a statistical model in-place.
source

2.44 StatsBase.fit

`StatsBase.fit` — *Method*.

Fit a statistical model.

source

2.45 StatsBase.fit

`StatsBase.fit` — *Method*.

```
fit(Histogram, data[, weight][, edges]; closed=:right, nbins)
```

Fit a histogram to `data`.

Arguments

- `data`: either a vector (for a 1-dimensional histogram), or a tuple of vectors of equal length (for an n-dimensional histogram).
- `weight`: an optional `AbstractWeights` (of the same length as the data vectors), denoting the weight each observation contributes to the bin. If no weight vector is supplied, each observation has weight 1.
- `edges`: a vector (typically an `AbstractRange` object), or tuple of vectors, that gives the edges of the bins along each dimension. If no edges are provided, these are determined from the data.

Keyword arguments

- `closed=:right`: if `:left`, the bin intervals are left-closed [a,b); if `:right` (the default), intervals are right-closed (a,b].
- `nbins`: if no `edges` argument is supplied, the approximate number of bins to use along each dimension (can be either a single integer, or a tuple of integers).

Examples

[] Univariate h = fit(Histogram, rand(100)) h = fit(Histogram, rand(100), 0:0.1:1.0) h = fit(Histogram, rand(100), nbins=10) h = fit(Histogram, rand(100), weights(rand(100)), 0:0.1:1.0) h = fit(Histogram, [20], 0:20:100) h = fit(Histogram, [20], 0:20:100, closed=:left)

Multivariate h = fit(Histogram, (rand(100),rand(100))) h = fit(Histogram, (rand(100),rand(100)),nbins=10)

source

2.46 StatsBase.fitted

`StatsBase.fitted` — *Method.*

`fitted(obj::RegressionModel)`

Return the fitted values of the model.
source

2.47 StatsBase.fweights

`StatsBase.fweights` — *Method.*

`fweights(vs)`

Construct a `FrequencyWeights` vector from a given array. See the documentation for `FrequencyWeights` for more details.
source

2.48 StatsBase.genmean

`StatsBase.genmean` — *Method.*

`genmean(a, p)`

Return the generalized/power mean with exponent `p` of a real-valued array, i.e. $\left(\frac{1}{n}\sum_{i=1}^{n} a_i^p\right)^{\frac{1}{p}}$, where `n = length(a)`. It is taken to be the geometric mean when `p == 0`.
source

2.49 StatsBase.geomean

`StatsBase.geomean` — *Method.*

`geomean(a)`

Return the geometric mean of a real-valued array.
source

2.50 StatsBase.gkldiv

`StatsBase.gkldiv` — *Method.*

`gkldiv(a, b)`

Compute the generalized Kullback-Leibler divergence between two arrays: $\sum_{i=1}^{n}(a_i \log(a_i/b_i) - a_i + b_i)$. Efficient equivalent of `sum(a*log(a/b)-a+b)`.
source

2.51 StatsBase.harmmean

`StatsBase.harmmean` — *Method*.

`harmmean(a)`

Return the harmonic mean of a real-valued array.
source

2.52 StatsBase.indexmap

`StatsBase.indexmap` — *Method*.

`indexmap(a)`

Construct a dictionary that maps each unique value in `a` to the index of its first occurrence in `a`.
source

2.53 StatsBase.indicatormat

`StatsBase.indicatormat` — *Method*.

`indicatormat(x, c=sort(unique(x)); sparse=false)`

Construct a boolean matrix `I` of size `(length(c), length(x))`. Let `ci` be the index of `x[i]` in `c`. Then `I[ci, i] = true` and all other elements are `false`.
source

2.54 StatsBase.indicatormat

`StatsBase.indicatormat` — *Method*.

`indicatormat(x, k::Integer; sparse=false)`

Construct a boolean matrix `I` of size `(k, length(x))` such that `I[x[i], i] = true` and all other elements are set to `false`. If `sparse` is `true`, the output will be a sparse matrix, otherwise it will be dense (default).

Examples

```
julia> using StatsBase

julia> indicatormat([1 2 2], 2)
23 Array{Bool,2}:
  true  false  false
 false   true   true
```

source

2.55 StatsBase.inverse_rle

`StatsBase.inverse_rle` — *Method*.

```
inverse_rle(vals, lens)
```

Reconstruct a vector from its run-length encoding (see `rle`). `vals` is a vector of the values and `lens` is a vector of the corresponding run lengths.

source

2.56 StatsBase.iqr

`StatsBase.iqr` — *Method*.

```
iqr(v)
```

Compute the interquartile range (IQR) of an array, i.e. the 75th percentile minus the 25th percentile.

source

2.57 StatsBase.kldivergence

`StatsBase.kldivergence` — *Method*.

```
kldivergence(p, q, [b])
```

Compute the Kullback-Leibler divergence of `q` from `p`, optionally specifying a real number `b` such that the divergence is scaled by `1/log(b)`.

source

2.58 StatsBase.kurtosis

`StatsBase.kurtosis` — *Method*.

```
kurtosis(v, [wv::AbstractWeights], m=mean(v))
```

Compute the excess kurtosis of a real-valued array `v`, optionally specifying a weighting vector `wv` and a center `m`.

source

2.59 StatsBase.levelsmap

`StatsBase.levelsmap` — *Method*.

```
levelsmap(a)
```

Construct a dictionary that maps each of the `n` unique values in `a` to a number between 1 and `n`.

source

2.60 StatsBase.loglikelihood

`StatsBase.loglikelihood` — *Method*.

```
loglikelihood(obj::StatisticalModel)
```

Return the log-likelihood of the model.
source

2.61 StatsBase.mad!

`StatsBase.mad!` — *Method*.

```
StatsBase.mad!(v; center=median!(v), normalize=true)
```

Compute the median absolute deviation (MAD) of `v` around `center` (by default, around the median), overwriting `v` in the process.

If `normalize` is set to `true`, the MAD is multiplied by `1 / quantile(Normal(), 3/4)  1.4826`, in order to obtain a consistent estimator of the standard deviation under the assumption that the data is normally distributed.
source

2.62 StatsBase.mad

`StatsBase.mad` — *Method*.

```
mad(v; center=median(v), normalize=true)
```

Compute the median absolute deviation (MAD) of `v` around `center` (by default, around the median).

If `normalize` is set to `true`, the MAD is multiplied by `1 / quantile(Normal(), 3/4)  1.4826`, in order to obtain a consistent estimator of the standard deviation under the assumption that the data is normally distributed.
source

2.63 StatsBase.maxad

`StatsBase.maxad` — *Method*.

```
maxad(a, b)
```

Return the maximum absolute deviation between two arrays: `maxabs(a - b)`.
source

2.64 StatsBase.mean_and_cov

`StatsBase.mean_and_cov` — *Function*.

```
mean_and_cov(x, [wv::AbstractWeights]; vardim=1, corrected=false) -> (mean, cov)
```

Return the mean and covariance matrix as a tuple. A weighting vector `wv` can be specified. `vardim` that designates whether the variables are columns in the matrix (`1`) or rows (`2`). Finally, bias correction is applied to the covariance calculation if `corrected=true`. See `cov` documentation for more details.

source

2.65 StatsBase.mean_and_std

`StatsBase.mean_and_std` — *Method*.

```
mean_and_std(x, [w::AbstractWeights], [dim]; corrected=false) -> (mean, std)
```

Return the mean and standard deviation of a real-valued array `x`, optionally over a dimension `dim`, as a tuple. A weighting vector `w` can be specified to weight the estimates. Finally, bias correction is applied to the standard deviation calculation if `corrected=true`. See `std` documentation for more details.

source

2.66 StatsBase.mean_and_var

`StatsBase.mean_and_var` — *Method*.

```
mean_and_var(x, [w::AbstractWeights], [dim]; corrected=false) -> (mean, var)
```

Return the mean and variance of a real-valued array `x`, optionally over a dimension `dim`, as a tuple. Observations in `x` can be weighted using weight vector `w`. Finally, bias correction is be applied to the variance calculation if `corrected=true`. See `var` documentation for more details.

source

2.67 StatsBase.meanad

`StatsBase.meanad` — *Method*.

```
meanad(a, b)
```

Return the mean absolute deviation between two arrays: `mean(abs(a - b))`.

source

2.68 StatsBase.mode

`StatsBase.mode` — *Method*.

```
mode(a, [r])
```

Return the mode (most common number) of an array, optionally over a specified range `r`. If several modes exist, the first one (in order of appearance) is returned.

source

2.69 StatsBase.model_response

`StatsBase.model_response` — *Method*.

```
model_response(obj::RegressionModel)
```

Return the model response (a.k.a. the dependent variable).

source

2.70 StatsBase.modelmatrix

`StatsBase.modelmatrix` — *Method*.

```
modelmatrix(obj::RegressionModel)
```

Return the model matrix (a.k.a. the design matrix).

source

2.71 StatsBase.modes

`StatsBase.modes` — *Method*.

```
modes(a, [r])::Vector
```

Return all modes (most common numbers) of an array, optionally over a specified range `r`.

source

2.72 StatsBase.moment

`StatsBase.moment` — *Method*.

```
moment(v, k, [wv::AbstractWeights], m=mean(v))
```

Return the `k`th order central moment of a real-valued array `v`, optionally specifying a weighting vector `wv` and a center `m`.

source

2.73 StatsBase.msd

`StatsBase.msd` — *Method.*

```
msd(a, b)
```

Return the mean squared deviation between two arrays: `mean(abs2(a - b))`.

source

2.74 StatsBase.nobs

`StatsBase.nobs` — *Method.*

```
nobs(obj::StatisticalModel)
```

Return the number of independent observations on which the model was fitted. Be careful when using this information, as the definition of an independent observation may vary depending on the model, on the format used to pass the data, on the sampling plan (if specified), etc.

source

2.75 StatsBase.nquantile

`StatsBase.nquantile` — *Method.*

```
nquantile(v, n)
```

Return the n-quantiles of a real-valued array, i.e. the values which partition `v` into `n` subsets of nearly equal size.

Equivalent to `quantile(v, [0:n]/n)`. For example, `nquantiles(x, 5)` returns a vector of quantiles, respectively at `[0.0, 0.2, 0.4, 0.6, 0.8, 1.0]`.

source

2.76 StatsBase.nulldeviance

`StatsBase.nulldeviance` — *Method.*

```
nulldeviance(obj::StatisticalModel)
```

Return the deviance of the null model, that is the one including only the intercept.

source

2.77 StatsBase.nullloglikelihood

`StatsBase.nullloglikelihood` — *Method*.

```
loglikelihood(obj::StatisticalModel)
```

Return the log-likelihood of the null model corresponding to model `obj`. This is usually the model containing only the intercept.

source

2.78 StatsBase.ordinalrank

`StatsBase.ordinalrank` — *Method*.

```
ordinalrank(x; lt = isless, rev::Bool = false)
```

Return the ordinal ranking ("1234" ranking) of an array. The `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. All items in `x` are given distinct, successive ranks based on their position in `sort(x; lt = lt, rev = rev)`. Missing values are assigned rank `missing`.

source

2.79 StatsBase.pacf!

`StatsBase.pacf!` — *Method*.

```
pacf!(r, X, lags; method=:regression)
```

Compute the partial autocorrelation function (PACF) of a matrix `X` at `lags` and store the result in `r`. `method` designates the estimation method. Recognized values are `:regression`, which computes the partial autocorrelations via successive regression models, and `:yulewalker`, which computes the partial autocorrelations using the Yule-Walker equations.

`r` must be a matrix of size `(length(lags), size(x, 2))`.

source

2.80 StatsBase.pacf

`StatsBase.pacf` — *Method*.

```
pacf(X, lags; method=:regression)
```

Compute the partial autocorrelation function (PACF) of a real-valued vector or matrix `X` at `lags`. `method` designates the estimation method. Recognized values are `:regression`, which computes the partial autocorrelations via successive regression models, and `:yulewalker`, which computes the partial autocorrelations using the Yule-Walker equations.

If `x` is a vector, return a vector of the same length as `lags`. If `x` is a matrix, return a matrix of size `(length(lags), size(x, 2))`, where each column in the result corresponds to a column in `x`.

source

2.81 StatsBase.percentile

`StatsBase.percentile` — *Method.*

```
percentile(v, p)
```

Return the `p`th percentile of a real-valued array `v`, i.e. `quantile(x, p / 100)`.

source

2.82 StatsBase.predict

`StatsBase.predict` — *Function.*

```
predict(obj::RegressionModel, [newX])
```

Form the predicted response of model `obj`. An object with new covariate values `newX` can be supplied, which should have the same type and structure as that used to fit `obj`; e.g. for a GLM it would generally be a `DataFrame` with the same variable names as the original predictors.

source

2.83 StatsBase.predict!

`StatsBase.predict!` — *Function.*

```
predict!
```

In-place version of `predict`.

source

2.84 StatsBase.proportionmap

`StatsBase.proportionmap` — *Method.*

```
proportionmap(x)
```

Return a dictionary mapping each unique value in `x` to its proportion in `x`.
source

2.85 StatsBase.proportions

`StatsBase.proportions` — *Method.*

```
proportions(x, k::Integer, [wv::AbstractWeights])
```

Return the proportion of integers in 1 to `k` that occur in `x`.
source

2.86 StatsBase.proportions

`StatsBase.proportions` — *Method.*

```
proportions(x, levels=span(x), [wv::AbstractWeights])
```

Return the proportion of values in the range `levels` that occur in `x`. Equivalent to `counts(x, levels) / length(x)`. If a weighting vector `wv` is specified, the sum of the weights is used rather than the raw counts.
source

2.87 StatsBase.psnr

`StatsBase.psnr` — *Method.*

```
psnr(a, b, maxv)
```

Compute the peak signal-to-noise ratio between two arrays `a` and `b`. `maxv` is the maximum possible value either array can take. The PSNR is computed as `10 * log10(maxv^2 / msd(a, b))`.
source

2.88 StatsBase.pweights

`StatsBase.pweights` — *Method.*

```
pweights(vs)
```

Construct a `ProbabilityWeights` vector from a given array. See the documentation for `ProbabilityWeights` for more details.
source

2.89 StatsBase.r2

`StatsBase.r2` — *Method.*

```
r2(obj::StatisticalModel, variant::Symbol)
r(obj::StatisticalModel, variant::Symbol)
```

Coefficient of determination (R-squared).

For a linear model, the R is defined as ESS/TSS, with ESS the explained sum of squares and TSS the total sum of squares, and `variant` can be omitted.

For other models, one of several pseudo R definitions must be chosen via `variant`. Supported variants are:

- `:MacFadden` (a.k.a. likelihood ratio index), defined as $1 - \log L / \log L0$.
- `:CoxSnell`, defined as $1 - (L0/L)^{2/n}$
- `:Nagelkerke`, defined as $(1 - (L0/L)^{2/n})/(1 - L0^{2/n})$, with n the number

of observations (as returned by `nobs`).

In the above formulas, L is the likelihood of the model, $L0$ that of the null model (the model including only the intercept). These two quantities are taken to be minus half `deviance` of the corresponding models.

source

2.90 StatsBase.renyientropy

`StatsBase.renyientropy` — *Method.*

```
renyientropy(p, )
```

Compute the Rnyi (generalized) entropy of order of an array `p`.

source

2.91 StatsBase.residuals

`StatsBase.residuals` — *Method.*

```
residuals(obj::RegressionModel)
```

Return the residuals of the model.

source

2.92 StatsBase.rle

`StatsBase.rle` — *Method*.

```
rle(v) -> (vals, lens)
```

Return the run-length encoding of a vector as a tuple. The first element of the tuple is a vector of values of the input and the second is the number of consecutive occurrences of each element.

Examples

```
julia> using StatsBase

julia> rle([1,1,1,2,2,3,3,3,3,2,2,2])
([1, 2, 3, 2], [3, 2, 4, 3])
```

source

2.93 StatsBase.rmsd

`StatsBase.rmsd` — *Method*.

```
rmsd(a, b; normalize=false)
```

Return the root mean squared deviation between two optionally normalized arrays. The root mean squared deviation is computed as `sqrt(msd(a, b))`.

source

2.94 StatsBase.sample!

`StatsBase.sample!` — *Method*.

```
sample!([rng], a, [wv::AbstractWeights], x; replace=true, ordered=false)
```

Draw a random sample of `length(x)` elements from an array `a` and store the result in `x`. A polyalgorithm is used for sampling. Sampling probabilities are proportional to the weights given in `wv`, if provided. `replace` dictates whether sampling is performed with replacement and `order` dictates whether an ordered sample, also called a sequential sample, should be taken.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.95 StatsBase.sample

`StatsBase.sample` — *Method*.

```
sample([rng], a, [wv::AbstractWeights])
```

Select a single random element of `a`. Sampling probabilities are proportional to the weights given in `wv`, if provided.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.96 StatsBase.sample

`StatsBase.sample` — *Method*.

```
sample([rng], wv::AbstractWeights)
```

Select a single random integer in `1:length(wv)` with probabilities proportional to the weights given in `wv`.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.97 StatsBase.sample

`StatsBase.sample` — *Method*.

```
sample([rng], a, [wv::AbstractWeights], n::Integer; replace=true, ordered=false)
```

Select a random, optionally weighted sample of size `n` from an array `a` using a polyalgorithm. Sampling probabilities are proportional to the weights given in `wv`, if provided. `replace` dictates whether sampling is performed with replacement and `order` dictates whether an ordered sample, also called a sequential sample, should be taken.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.98 StatsBase.sample

`StatsBase.sample` — *Method*.

```
sample([rng], a, [wv::AbstractWeights], dims::Dims; replace=true, ordered=false)
```

Select a random, optionally weighted sample from an array `a` specifying the dimensions `dims` of the output array. Sampling probabilities are proportional to the weights given in `wv`, if provided. `replace` dictates whether sampling is performed with replacement and `order` dictates whether an ordered sample, also called a sequential sample, should be taken.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.99 StatsBase.samplepair

`StatsBase.samplepair` — *Method*.

```
samplepair([rng], a)
```

Draw a pair of distinct elements from the array `a` without replacement.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.100 StatsBase.samplepair

`StatsBase.samplepair` — *Method*.

```
samplepair([rng], n)
```

Draw a pair of distinct integers between 1 and `n` without replacement.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.101 StatsBase.scattermat

`StatsBase.scattermat` — *Function*.

```
scattermat(X, [wv::AbstractWeights]; mean=nothing, vardim=1)
```

Compute the scatter matrix, which is an unnormalized covariance matrix. A weighting vector `wv` can be specified to weight the estimate.

Arguments

- `mean=nothing`: a known mean value. `nothing` indicates that the mean is unknown, and the function will compute the mean. Specifying `mean=0` indicates that the data are centered and hence there's no need to subtract the mean.

- `vardim=1`: the dimension along which the variables are organized. When `vardim = 1`, the variables are considered columns with observations in rows; when `vardim = 2`, variables are in rows with observations in columns.

source

2.102 StatsBase.sem

`StatsBase.sem` — *Method.*

```
sem(a)
```

Return the standard error of the mean of `a`, i.e. `sqrt(var(a) / length(a))`.
source

2.103 StatsBase.skewness

`StatsBase.skewness` — *Method.*

```
skewness(v, [wv::AbstractWeights], m=mean(v))
```

Compute the standardized skewness of a real-valued array `v`, optionally specifying a weighting vector `wv` and a center `m`.
source

2.104 StatsBase.span

`StatsBase.span` — *Method.*

```
span(x)
```

Return the span of an integer array, i.e. the range `minimum(x):maximum(x)`. The minimum and maximum of `x` are computed in one-pass using `extrema`.
source

2.105 StatsBase.sqL2dist

`StatsBase.sqL2dist` — *Method.*

```
sqL2dist(a, b)
```

Compute the squared L2 distance between two arrays: $\sum_{i=1}^{n} |a_i - b_i|^2$. Efficient equivalent of `sumabs2(a - b)`.
source

2.106 StatsBase.stderr

`StatsBase.stderr` — *Method*.

```
stderr(obj::StatisticalModel)
```

Return the standard errors for the coefficients of the model.
source

2.107 StatsBase.summarystats

`StatsBase.summarystats` — *Method*.

```
summarystats(a)
```

Compute summary statistics for a real-valued array `a`. Returns a `SummaryStats` object containing the mean, minimum, 25th percentile, median, 75th percentile, and maxmimum.
source

2.108 StatsBase.tiedrank

`StatsBase.tiedrank` — *Method*.

```
tiedrank(x)
```

Return the tied ranking, also called fractional or "1 2.5 2.5 4" ranking, of an array. The `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. Items that compare equal receive the mean of the rankings they would have been assigned under ordinal ranking. Missing values are assigned rank `missing`.
source

2.109 StatsBase.trim!

`StatsBase.trim!` — *Method*.

```
trim!(x; prop=0.0, count=0)
```

A variant of `trim` that modifies `x` in place.
source

2.110 StatsBase.trim

`StatsBase.trim` — *Method.*

```
trim(x; prop=0.0, count=0)
```

Return a copy of `x` with either `count` or proportion `prop` of the highest and lowest elements removed. To compute the trimmed mean of `x` use `mean(trim(x))`; to compute the variance use `trimvar(x)` (see `trimvar`).

Example

[] julia¿ trim([1,2,3,4,5], prop=0.2) 3-element Array{Int64,1}: 2 3 4

source

2.111 StatsBase.trimvar

`StatsBase.trimvar` — *Method.*

```
trimvar(x; prop=0.0, count=0)
```

Compute the variance of the trimmed mean of `x`. This function uses the Winsorized variance, as described in Wilcox (2010).

source

2.112 StatsBase.variation

`StatsBase.variation` — *Method.*

```
variation(x, m=mean(x))
```

Return the coefficient of variation of an array `x`, optionally specifying a precomputed mean `m`. The coefficient of variation is the ratio of the standard deviation to the mean.

source

2.113 StatsBase.vcov

`StatsBase.vcov` — *Method.*

```
vcov(obj::StatisticalModel)
```

Return the variance-covariance matrix for the coefficients of the model.

source

2.114 StatsBase.weights

`StatsBase.weights` — *Method*.

```
weights(vs)
```

Construct a `Weights` vector from array `vs`. See the documentation for `Weights` for more details.

source

2.115 StatsBase.winsor!

`StatsBase.winsor!` — *Method*.

```
winsor!(x; prop=0.0, count=0)
```

A variant of `winsor` that modifies vector `x` in place.

source

2.116 StatsBase.winsor

`StatsBase.winsor` — *Method*.

```
winsor(x; prop=0.0, count=0)
```

Return a copy of `x` with either `count` or proportion `prop` of the lowest elements of `x` replaced with the next-lowest, and an equal number of the highest elements replaced with the previous-highest. To compute the Winsorized mean of `x` use `mean(winsor(x))`.

Example

[] julia¿ winsor([1,2,3,4,5], prop=0.2) 5-element Array{Int64,1}: 2 2 3 4 4

source

2.117 StatsBase.wmean

`StatsBase.wmean` — *Method*.

```
wmean(v, w::AbstractVector)
```

Compute the weighted mean of an array `v` with weights `w`.

source

2.118 StatsBase.wmedian

`StatsBase.wmedian` — *Method*.

```
wmedian(v, w)
```

Compute the weighted median of an array `v` with weights `w`, given as either a vector or an `AbstractWeights` vector.

source

2.119 StatsBase.wquantile

`StatsBase.wquantile` — *Method*.

```
wquantile(v, w, p)
```

Compute the `p`th quantile(s) of `v` with weights `w`, given as either a vector or an `AbstractWeights` vector.

source

2.120 StatsBase.wsample!

`StatsBase.wsample!` — *Method*.

```
wsample!([rng], a, w, x; replace=true, ordered=false)
```

Select a weighted sample from an array `a` and store the result in `x`. Sampling probabilities are proportional to the weights given in `w`. `replace` dictates whether sampling is performed with replacement and `order` dictates whether an ordered sample, also called a sequential sample, should be taken.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.121 StatsBase.wsample

`StatsBase.wsample` — *Method*.

```
wsample([rng], [a], w)
```

Select a weighted random sample of size 1 from `a` with probabilities proportional to the weights given in `w`. If `a` is not present, select a random weight from `w`.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.122 StatsBase.wsample

`StatsBase.wsample` — *Method*.

```
wsample([rng], [a], w, n::Integer; replace=true, ordered=false)
```

Select a weighted random sample of size `n` from `a` with probabilities proportional to the weights given in `w` if `a` is present, otherwise select a random sample of size `n` of the weights given in `w`. `replace` dictates whether sampling is performed with replacement and `order` dictates whether an ordered sample, also called a sequential sample, should be taken.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.123 StatsBase.wsample

`StatsBase.wsample` — *Method*.

```
wsample([rng], [a], w, dims::Dims; replace=true, ordered=false)
```

Select a weighted random sample from `a` with probabilities proportional to the weights given in `w` if `a` is present, otherwise select a random sample of size `n` of the weights given in `w`. The dimensions of the output are given by `dims`.

Optionally specify a random number generator `rng` as the first argument (defaults to `Base.GLOBAL_RNG`).

source

2.124 StatsBase.wsum!

`StatsBase.wsum!` — *Method*.

```
wsum!(R, A, w, dim; init=true)
```

Compute the weighted sum of `A` with weights `w` over the dimension `dim` and store the result in `R`. If `init=false`, the sum is added to `R` rather than starting from zero.

source

2.125 StatsBase.wsum

`StatsBase.wsum` — *Method*.

```
wsum(v, w::AbstractVector, [dim])
```

Compute the weighted sum of an array `v` with weights `w`, optionally over the dimension `dim`.

source

2.126 StatsBase.zscore!

`StatsBase.zscore!` — *Method.*

```
zscore!([Z], X, , )
```

Compute the z-scores of an array `X` with mean and standard deviation . z-scores are the signed number of standard deviations above the mean that an observation lies, i.e. $(x -)/$.

If a destination array `Z` is provided, the scores are stored in `Z` and it must have the same shape as `X`. Otherwise `X` is overwritten.

source

2.127 StatsBase.zscore

`StatsBase.zscore` — *Method.*

```
zscore(X, [, ])
```

Compute the z-scores of `X`, optionally specifying a precomputed mean and standard deviation . z-scores are the signed number of standard deviations above the mean that an observation lies, i.e. $(x -)/$.

 and should be both scalars or both arrays. The computation is broadcasting. In particular, when and are arrays, they should have the same size, and `size(, i) == 1 || size(, i) == size(X, i)` for each dimension.

source

Chapter 3

PyPlot

3.1 Base.step

`Base.step` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), (“step”,))

source

3.2 PyPlot.Axes3D

`PyPlot.Axes3D` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), (“Axes3D”,))

source

3.3 PyPlot.acorr

`PyPlot.acorr` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), (“acorr”,))

source

3.4 PyPlot.annotate

`PyPlot.annotate` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), (“annotate”,))

source

3.5 PyPlot.arrow

`PyPlot.arrow` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("arrow",))

source

3.6 PyPlot.autoscale

`PyPlot.autoscale` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("autoscale",))

source

3.7 PyPlot.autumn

`PyPlot.autumn` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("autumn",))

source

3.8 PyPlot.axes

`PyPlot.axes` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axes",))

source

3.9 PyPlot.axhline

`PyPlot.axhline` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axhline",))

source

3.10 PyPlot.axhspan

`PyPlot.axhspan` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axhspan",))

source

3.11 PyPlot.axis

`PyPlot.axis` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axis",))

source

3.12 PyPlot.axvline

`PyPlot.axvline` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axvline",))

source

3.13 PyPlot.axvspan

`PyPlot.axvspan` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("axvspan",))

source

3.14 PyPlot.bar

`PyPlot.bar` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("bar",))

source

3.15 PyPlot.bar3D

`PyPlot.bar3D` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "bar3d"))

source

3.16 PyPlot.barbs

`PyPlot.barbs` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("barbs",))

source

3.17 PyPlot.barh

`PyPlot.barh` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("barh",))

source

3.18 PyPlot.bone

`PyPlot.bone` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("bone",))

source

3.19 PyPlot.box

`PyPlot.box` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("box",))

source

3.20 PyPlot.boxplot

`PyPlot.boxplot` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("boxplot",))

source

3.21 PyPlot.broken_barh

`PyPlot.broken_barh` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("broken_barh",))

source

3.22 PyPlot.cla

`PyPlot.cla` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("cla",))

source

3.23 PyPlot.clabel

`PyPlot.clabel` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("clabel",))

source

3.24 PyPlot.clf

`PyPlot.clf` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("clf",))

source

3.25 PyPlot.clim

`PyPlot.clim` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("clim",))

source

3.26 PyPlot.cohere

`PyPlot.cohere` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("cohere",))

source

3.27 PyPlot.colorbar

`PyPlot.colorbar` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("colorbar",))

source

3.28 PyPlot.colors

`PyPlot.colors` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("colors",))

source

3.29 PyPlot.contour

`PyPlot.contour` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("contour",))

source

3.30 PyPlot.contour3D

`PyPlot.contour3D` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "contour3D"))

source

3.31 PyPlot.contourf

`PyPlot.contourf` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("contourf",))

source

3.32 PyPlot.contourf3D

`PyPlot.contourf3D` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "contourf3D"))

source

3.33 PyPlot.cool

`PyPlot.cool` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("cool",))

source

3.34 PyPlot.copper

`PyPlot.copper` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("copper",))

source

3.35 PyPlot.csd

`PyPlot.csd` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("csd",))

source

3.36 PyPlot.delaxes

`PyPlot.delaxes` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("delaxes",))

source

3.37 PyPlot.disconnect

`PyPlot.disconnect` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("disconnect",))

source

3.38 PyPlot.draw

`PyPlot.draw` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("draw",))

source

3.39 PyPlot.errorbar

`PyPlot.errorbar` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("errorbar",))

source

3.40 PyPlot.eventplot

`PyPlot.eventplot` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("eventplot",))

source

3.41 PyPlot.figaspect

`PyPlot.figaspect` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("figaspect",))

source

3.42 PyPlot.figimage

`PyPlot.figimage` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("figimage",))

source

3.43 PyPlot.figlegend

`PyPlot.figlegend` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("figlegend",))

source

3.44 PyPlot.figtext

`PyPlot.figtext` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("figtext",))

source

3.45 PyPlot.figure

`PyPlot.figure` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fc52f0), ())

source

3.46 PyPlot.fill_between

`PyPlot.fill_between` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("fill_between",))

source

3.47 PyPlot.fill_betweenx

`PyPlot.fill_betweenx` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("fill_betweenx",))

source

3.48 PyPlot.findobj

`PyPlot.findobj` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("findobj",))

source

3.49 PyPlot.flag

`PyPlot.flag` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("flag",))

source

3.50 PyPlot.gca

`PyPlot.gca` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("gca",))

source

3.51 PyPlot.gcf

`PyPlot.gcf` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fc5400), ())

source

3.52 PyPlot.gci

`PyPlot.gci` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("gci",))

source

3.53 PyPlot.get_cmap

`PyPlot.get_cmap` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002586f2f0), ())

source

3.54 PyPlot.get_current_fig_manager

`PyPlot.get_current_fig_manager` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("get_current_fig_manager",))

source

3.55 PyPlot.get_figlabels

`PyPlot.get_figlabels` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("get_figlabels",))

source

3.56 PyPlot.get_fignums

`PyPlot.get_fignums` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("get_fignums",))

source

3.57 PyPlot.get_plot_commands

`PyPlot.get_plot_commands` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("get_plot_commands",))

source

3.58 PyPlot.ginput

`PyPlot.ginput` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ginput",))

source

3.59 PyPlot.gray

`PyPlot.gray` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("gray",))

source

3.60 PyPlot.grid

`PyPlot.grid` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("grid",))

source

3.61 PyPlot.hexbin

`PyPlot.hexbin` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hexbin",))

source

3.62 PyPlot.hist2D

`PyPlot.hist2D` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hist2d",))

source

3.63 PyPlot.hlines

`PyPlot.hlines` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hlines",))

source

3.64 PyPlot.hold

`PyPlot.hold` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hold",))

source

3.65 PyPlot.hot

`PyPlot.hot` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hot",))

source

3.66 PyPlot.hsv

`PyPlot.hsv` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("hsv",))

source

3.67 PyPlot.imread

`PyPlot.imread` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("imread",))

source

3.68 PyPlot.imsave

`PyPlot.imsave` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("imsave",))

source

3.69 PyPlot.imshow

`PyPlot.imshow` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("imshow",))

source

3.70 PyPlot.ioff

`PyPlot.ioff` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ioff",))

source

3.71 PyPlot.ion

`PyPlot.ion` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ion",))

source

3.72 PyPlot.ishold

`PyPlot.ishold` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ishold",))

source

3.73 PyPlot.jet

`PyPlot.jet` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("jet",))

source

3.74 PyPlot.legend

`PyPlot.legend` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("legend",))

source

3.75 PyPlot.locator_params

`PyPlot.locator_params` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("locator_params",))

source

3.76 PyPlot.loglog

`PyPlot.loglog` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("loglog",))

source

3.77 PyPlot.margins

`PyPlot.margins` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("margins",))

source

3.78 PyPlot.matshow

`PyPlot.matshow` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("matshow",))

source

3.79 PyPlot.mesh

`PyPlot.mesh` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot_wireframe"))

source

3.80 PyPlot.minorticks_off

`PyPlot.minorticks_off` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("minorticks_off",))

source

3.81 PyPlot.minorticks_on

`PyPlot.minorticks_on` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("minorticks_on",))

source

3.82 PyPlot.over

`PyPlot.over` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("over",))

source

3.83 PyPlot.pause

`PyPlot.pause` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("pause",))

source

3.84 PyPlot.pcolor

`PyPlot.pcolor` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("pcolor",))

source

3.85 PyPlot.pcolormesh

`PyPlot.pcolormesh` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("pcolormesh",))

source

3.86 PyPlot.pie

`PyPlot.pie` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("pie",))

source

3.87 PyPlot.pink

`PyPlot.pink` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("pink",))

source

3.88 PyPlot.plot

`PyPlot.plot` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("plot",))

source

3.89 PyPlot.plot3D

`PyPlot.plot3D` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot3D"))

source

3.90 PyPlot.plot_date

`PyPlot.plot_date` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("plot_date",))

source

3.91 PyPlot.plot_surface

`PyPlot.plot_surface` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot_surface"))

source

3.92 PyPlot.plot_trisurf

`PyPlot.plot_trisurf` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot_trisurf"))

source

3.93 PyPlot.plot_wireframe

`PyPlot.plot_wireframe` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot_wireframe"))

source

3.94 PyPlot.plotfile

`PyPlot.plotfile` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("plotfile",))

source

3.95 PyPlot.polar

`PyPlot.polar` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("polar",))

source

3.96 PyPlot.prism

`PyPlot.prism` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("prism",))

source

3.97 PyPlot.psd

`PyPlot.psd` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("psd",))

source

3.98 PyPlot.quiver

`PyPlot.quiver` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("quiver",))

source

3.99 PyPlot.quiverkey

`PyPlot.quiverkey` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("quiverkey",))

source

3.100 PyPlot.rc

`PyPlot.rc` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("rc",))

source

3.101 PyPlot.rc_context

`PyPlot.rc_context` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("rc_context",))

source

3.102 PyPlot.rcdefaults

`PyPlot.rcdefaults` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("rcdefaults",))

source

3.103 PyPlot.register_cmap

`PyPlot.register_cmap` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002586f268), ())

source

3.104 PyPlot.rgrids

`PyPlot.rgrids` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("rgrids",))

source

3.105 PyPlot.savefig

`PyPlot.savefig` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("savefig",))

source

3.106 PyPlot.sca

`PyPlot.sca` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("sca",))

source

3.107 PyPlot.scatter

`PyPlot.scatter` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("scatter",))

source

3.108 PyPlot.scatter3D

`PyPlot.scatter3D` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "scatter3D"))

source

3.109 PyPlot.sci

`PyPlot.sci` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("sci",))

source

3.110 PyPlot.semilogx

`PyPlot.semilogx` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("semilogx",))

source

3.111 PyPlot.semilogy

`PyPlot.semilogy` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("semilogy",))

source

3.112 PyPlot.set_cmap

`PyPlot.set_cmap` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("set_cmap",))

source

3.113 PyPlot.setp

`PyPlot.setp` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("setp",))

source

3.114 PyPlot.specgram

`PyPlot.specgram` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("specgram",))

source

3.115 PyPlot.spectral

`PyPlot.spectral` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("spectral",))

source

3.116 PyPlot.spring

`PyPlot.spring` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("spring",))

source

3.117 PyPlot.spy

`PyPlot.spy` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("spy",))

source

3.118 PyPlot.stackplot

`PyPlot.stackplot` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("stackplot",))

source

3.119 PyPlot.stem

`PyPlot.stem` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("stem",))

source

3.120 PyPlot.streamplot

`PyPlot.streamplot` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("streamplot",))

source

3.121 PyPlot.subplot

`PyPlot.subplot` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("subplot",))

source

3.122 PyPlot.subplot2grid

`PyPlot.subplot2grid` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("subplot2grid",))

source

3.123 PyPlot.subplot_tool

`PyPlot.subplot_tool` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("subplot_tool",))

source

3.124 PyPlot.subplots

`PyPlot.subplots` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("subplots",))

source

3.125 PyPlot.subplots_adjust

`PyPlot.subplots_adjust` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("subplots_adjust",))

source

3.126 PyPlot.summer

`PyPlot.summer` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("summer",))

source

3.127 PyPlot.suptitle

`PyPlot.suptitle` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("suptitle",))

source

3.128 PyPlot.surf

`PyPlot.surf` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "plot_surface"))

source

3.129 PyPlot.table

`PyPlot.table` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("table",))

source

3.130 PyPlot.text

`PyPlot.text` — *Method.*

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("text",))

source

3.131 PyPlot.text2D

`PyPlot.text2D` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "text2D"))

source

3.132 PyPlot.text3D

`PyPlot.text3D` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "text3D"))

source

3.133 PyPlot.thetagrids

`PyPlot.thetagrids` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("thetagrids",))

source

3.134 PyPlot.tick_params

`PyPlot.tick_params` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("tick_params",))

source

3.135 PyPlot.ticklabel_format

`PyPlot.ticklabel_format` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ticklabel_format",))

source

3.136 PyPlot.tight_layout

`PyPlot.tight_layout` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("tight_layout",))

source

3.137 PyPlot.title

`PyPlot.title` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("title",))

source

3.138 PyPlot.tricontour

`PyPlot.tricontour` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("tricontour",))

source

3.139 PyPlot.tricontourf

`PyPlot.tricontourf` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("tricontourf",))

source

3.140 PyPlot.tripcolor

`PyPlot.tripcolor` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("tripcolor",))

source

3.141 PyPlot.triplot

`PyPlot.triplot` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("triplot",))

source

3.142 PyPlot.twinx

`PyPlot.twinx` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("twinx",))

source

3.143 PyPlot.twiny

`PyPlot.twiny` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("twiny",))

source

3.144 PyPlot.vlines

`PyPlot.vlines` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("vlines",))

source

3.145 PyPlot.waitforbuttonpress

`PyPlot.waitforbuttonpress` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("waitforbuttonpress",))

source

3.146 PyPlot.winter

`PyPlot.winter` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("winter",))

source

3.147 PyPlot.xkcd

`PyPlot.xkcd` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("xkcd",))

source

3.148 PyPlot.xlabel

`PyPlot.xlabel` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("xlabel",))

source

3.149 PyPlot.xlim

`PyPlot.xlim` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("xlim",))

source

3.150 PyPlot.xscale

`PyPlot.xscale` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("xscale",))

source

3.151 PyPlot.xticks

`PyPlot.xticks` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("xticks",))

source

3.152 PyPlot.ylabel

`PyPlot.ylabel` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ylabel",))

source

3.153 PyPlot.ylim

`PyPlot.ylim` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("ylim",))

source

3.154 PyPlot.yscale

`PyPlot.yscale` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("yscale",))

source

3.155 PyPlot.yticks

`PyPlot.yticks` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x000000002573e818), ("yticks",))

source

3.156 PyPlot.zlabel

`PyPlot.zlabel` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "set_zlabel"))

source

3.157 PyPlot.zlim

`PyPlot.zlim` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "set_zlim"))

source

3.158 PyPlot.zscale

`PyPlot.zscale` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "set_zscale"))

source

3.159 PyPlot.zticks

`PyPlot.zticks` — *Method*.

PyPlot.LazyHelp(PyCall.PyObject(Ptr{PyCall.PyObject_struct} @0x0000000031fb9228), ("Axes3D", "set_zticks"))

source

Chapter 4

IndexedTables

4.1 Base.Sort.select

`Base.Sort.select` — *Method.*

`select(t::Table, which::Selection)`

Select all or a subset of columns, or a single column from the table.

`Selection` is a type union of many types that can select from a table. It can be:

1. `Integer` – returns the column at this position.
2. `Symbol` – returns the column with this name.
3. `Pair{Selection => Function}` – selects and maps a function over the selection, returns the result.
4. `AbstractArray` – returns the array itself. This must be the same length as the table.
5. `Tuple` of `Selection` – returns a table containing a column for every selector in the tuple. The tuple may also contain the type `Pair{Symbol, Selection}`, which the selection a name. The most useful form of this when introducing a new column.

Examples:

Selection with `Integer` – returns the column at this position.

```
julia> tbl = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  2  3
0.05  1  4
```

```
julia> select(tbl, 2)
2-element Array{Int64,1}:
 2
 1
```

Selection with `Symbol` – returns the column with this name.

```
julia> select(tbl, :t)
2-element Array{Float64,1}:
 0.01
 0.05
```

Selection with `Pair{Selection => Function}` – selects some columns and maps a function over it, then returns the mapped column.

```
julia> select(tbl, :t=>t->1/t)
2-element Array{Float64,1}:
 100.0
  20.0
```

Selection with `AbstractArray` – returns the array itself.

```
julia> select(tbl, [3,4])
2-element Array{Int64,1}:
 3
 4
```

Selection with `Tuple`– returns a table containing a column for every selector in the tuple.

```
julia> select(tbl, (2,1))
Table with 2 rows, 2 columns:
x  t

2  0.01
1  0.05

julia> vx = select(tbl, (:x, :t)=>p->p.x/p.t)
2-element Array{Float64,1}:
 200.0
  20.0

julia> select(tbl, (:x,:t=>-))
Table with 2 rows, 2 columns:
x  t

1  -0.05
2  -0.01
```

Note that since `tbl` was initialized with `t` as the primary key column, selections that retain the key column will retain its status as a key. The same applies when multiple key columns are selected.

Selection with a custom array in the tuple will cause the name of the columns to be removed and replaced with integers.

```
julia> select(tbl, (:x, :t, [3,4]))
Table with 2 rows, 3 columns:
1  2     3

2  0.01  3
1  0.05  4
```

This is because the third column's name is unknown. In general if a column's name cannot be determined, then selection returns an iterable of tuples rather than named tuples. In other words, it strips column names.

To specify a new name to a custom column, you can use `Symbol => Selection` selector.

```
julia> select(tbl, (:x,:t,:z=>[3,4]))
Table with 2 rows, 3 columns:
x  t     z

2  0.01  3
1  0.05  4

julia> select(tbl, (:x, :t, :minust=>:t=>-))
Table with 2 rows, 3 columns:
x  t     minust

2  0.01  -0.01
1  0.05  -0.05

julia> select(tbl, (:x, :t, :vx=>(:x,:t)=>p->p.x/p.t))
Table with 2 rows, 3 columns:
x  t     vx

2  0.01  200.0
1  0.05  20.0
```

source

4.2 DataValues.dropna

`DataValues.dropna` — *Function.*

`dropna(t[, select])`

Drop rows which contain NA values.

```
julia> t = table([0.1, 0.5, NA,0.7], [2,NA,4,5], [NA,6,NA,7],
                 names=[:t,:x,:y])
Table with 4 rows, 3 columns:
t    x    y

0.1  2    #NA
0.5  #NA  6
#NA  4    #NA
0.7  5    7

julia> dropna(t)
Table with 1 rows, 3 columns:
t    x  y

0.7  5  7
```

Optionally `select` can be speicified to limit columns to look for NAs in.

```
julia> dropna(t, :y)
Table with 2 rows, 3 columns:
t    x    y

0.5  #NA  6
0.7  5    7

julia> t1 = dropna(t, (:t, :x))
Table with 2 rows, 3 columns:
t    x  y

0.1  2  #NA
0.7  5  7
```

Any columns whose NA rows have been dropped will be converted to non-na array type. In our last example, columns `t` and `x` got converted from `Array{DataValue{Int}}` to `Array{Int}`. Similarly if the vectors are of type `DataValueArray{T}` (default for `loadtable`) they will be converted to `Array{T}`.

[] julia¿ typeof(column(dropna(t,:x), :x)) Array{Int64,1}

source

4.3 IndexedTables.aggregate!

`IndexedTables.aggregate!` — *Method.*

`aggregate!(f::Function, arr::NDSparse)`

Combine adjacent rows with equal indices using the given 2-argument reduction function, in place.

source

4.4 IndexedTables.asofjoin

`IndexedTables.asofjoin` — *Method.*

`asofjoin(left::NDSparse, right::NDSparse)`

asofjoin is most useful on two time-series. It joins rows from `left` with the "most recent" value from `right`.

```
julia> x = ndsparse((["ko","ko", "xrx","xrx"],
                     Date.(["2017-11-11", "2017-11-12",
                            "2017-11-11", "2017-11-12"])), [1,2,3,4]);

julia> y = ndsparse((["ko","ko", "xrx","xrx"],
                     Date.(["2017-11-12", "2017-11-13",
                            "2017-11-10", "2017-11-13"])), [5,6,7,8])

julia> asofjoin(x,y)
2-d NDSparse with 4 values (Int64):
1      2

"ko"   2017-11-11  1
"ko"   2017-11-12  5
"xrx"  2017-11-11  7
"xrx"  2017-11-12  7
```

source

4.5 IndexedTables.colnames

`IndexedTables.colnames` — *Function.*

```
colnames(itr)
```

Returns the names of the "columns" in `itr`.

Examples:

```
julia> colnames([1,2,3])
1-element Array{Int64,1}:
 1

julia> colnames(Columns([1,2,3], [3,4,5]))
2-element Array{Int64,1}:
 1
 2

julia> colnames(table([1,2,3], [3,4,5]))
2-element Array{Int64,1}:
```

```
 1
 2

julia> colnames(Columns(x=[1,2,3], y=[3,4,5]))
2-element Array{Symbol,1}:
 :x
 :y

julia> colnames(table([1,2,3], [3,4,5], names=[:x,:y]))
2-element Array{Symbol,1}:
 :x
 :y

julia> colnames(ndsparse(Columns(x=[1,2,3]), Columns(y=[3,4,5])))
2-element Array{Symbol,1}:
 :x
 :y

julia> colnames(ndsparse(Columns(x=[1,2,3]), [3,4,5]))
2-element Array{Any,1}:
 :x
 1
 2

julia> colnames(ndsparse(Columns(x=[1,2,3]), [3,4,5]))
2-element Array{Any,1}:
 :x
 2

julia> colnames(ndsparse(Columns([1,2,3], [4,5,6]), Columns(x=[6,7,8])))
3-element Array{Any,1}:
 1
 2
 :x

julia> colnames(ndsparse(Columns(x=[1,2,3]), Columns([3,4,5],[6,7,8])))
3-element Array{Any,1}:
 :x
 2
 3
```

source

4.6 IndexedTables.columns

`IndexedTables.columns` — *Function.*

`columns(itr[, select::Selection])`

Select one or more columns from an iterable of rows as a tuple of vectors.

`select` specifies which columns to select. See `Selection convention` for possible values. If unspecified, returns all columns.

`itr` can be `NDSparse`, `Columns` and `AbstractVector`, and their distributed counterparts.

Examples

```
julia> t = table([1,2],[3,4], names=[:x,:y])
Table with 2 rows, 2 columns:
x  y

1  3
2  4

julia> columns(t)
(x = [1, 2], y = [3, 4])

julia> columns(t, :x)
2-element Array{Int64,1}:
 1
 2

julia> columns(t, (:x,))
(x = [1, 2])

julia> columns(t, (:y,:x=>-))
(y = [3, 4], x = [-1, -2])
```

source

4.7 IndexedTables.columns

`IndexedTables.columns` — *Method.*

`columns(itr, which)`

Returns a vector or a tuple of vectors from the iterator.

source

4.8 IndexedTables.convertdim

`IndexedTables.convertdim` — *Method.*

`convertdim(x::NDSparse, d::DimName, xlate; agg::Function, vecagg::Function, name)`

Apply function or dictionary `xlate` to each index in the specified dimension. If the mapping is many-to-one, `agg` or `vecagg` is used to aggregate the results.

If `agg` is passed, it is used as a 2-argument reduction function over the data. If `vecagg` is passed, it is used as a vector-to-scalar function to aggregate the data. `name` optionally specifies a new name for the translated dimension.

source

4.9 IndexedTables.dimlabels

`IndexedTables.dimlabels` — *Method*.

`dimlabels(t::NDSparse)`

Returns an array of integers or symbols giving the labels for the dimensions of `t`. `ndims(t) == length(dimlabels(t))`.

source

4.10 IndexedTables.flatten

`IndexedTables.flatten` — *Method*.

`flatten(t::Table, col)`

Flatten `col` column which may contain a vector of vectors while repeating the other fields.

Examples:

```
julia> x = table([1,2], [[3,4], [5,6]], names=[:x, :y])
Table with 2 rows, 2 columns:
x  y

1  [3, 4]
2  [5, 6]

julia> flatten(x, 2)
Table with 4 rows, 2 columns:
x  y

1  3
1  4
2  5
2  6

julia> x = table([1,2], [table([3,4],[5,6], names=[:a,:b]),
                         table([7,8], [9,10], names=[:a,:b])], names=[:x, :y]);

julia> flatten(x, :y)
Table with 4 rows, 3 columns:
x  a  b

1  3  5
```

```
1  4  6
2  7  9
2  8  10
```

source

4.11 IndexedTables.flush!

`IndexedTables.flush!` — *Method.*

`flush!(arr::NDSparse)`

Commit queued assignment operations, by sorting and merging the internal temporary buffer.

source

4.12 IndexedTables.groupby

`IndexedTables.groupby` — *Function.*

`groupby(f, t[, by::Selection]; select::Selection, flatten)`

Group rows by `by`, and apply `f` to each group. `f` can be a function or a tuple of functions. The result of `f` on each group is put in a table keyed by unique `by` values. `flatten` will flatten the result and can be used when `f` returns a vector instead of a single scalar value.

Examples

```
julia> t=table([1,1,1,2,2,2], [1,1,2,2,1,1], [1,2,3,4,5,6],
               names=[:x,:y,:z]);

julia> groupby(mean, t, :x, select=:z)
Table with 2 rows, 2 columns:
x  mean

1  2.0
2  5.0

julia> groupby(identity, t, (:x, :y), select=:z)
Table with 4 rows, 3 columns:
x  y  identity

1  1  [1, 2]
1  2  [3]
2  1  [5, 6]
2  2  [4]

julia> groupby(mean, t, (:x, :y), select=:z)
Table with 4 rows, 3 columns:
```

```
x  y  mean

1  1  1.5
1  2  3.0
2  1  5.5
2  2  4.0
```

multiple aggregates can be computed by passing a tuple of functions:

```
julia> groupby((mean, std, var), t, :y, select=:z)
Table with 2 rows, 4 columns:
y  mean  std       var

1  3.5   2.38048   5.66667
2  3.5   0.707107  0.5

julia> groupby(@NT(q25=z->quantile(z, 0.25), q50=median,
                   q75=z->quantile(z, 0.75)), t, :y, select=:z)
Table with 2 rows, 4 columns:
y  q25   q50  q75

1  1.75  3.5  5.25
2  3.25  3.5  3.75
```

Finally, it's possible to select different inputs for different functions by using a named tuple of `slector => function` pairs:

```
julia> groupby(@NT(xmean=:z=>mean, ystd=(:y=>-)=>std), t, :x)
Table with 2 rows, 3 columns:
x  xmean  ystd

1  2.0    0.57735
2  5.0    0.57735
```

By default, the result of groupby when `f` returns a vector or iterator of values will not be expanded. Pass the `flatten` option as `true` to flatten the grouped column:

```
julia> t = table([1,1,2,2], [3,4,5,6], names=[:x,:y])

julia> groupby((:normy => x->Iterators.repeated(mean(x), length(x)),),
               t, :x, select=:y, flatten=true)
Table with 4 rows, 2 columns:
x  normy

1  3.5
1  3.5
2  5.5
2  5.5
```

The keyword option `usekey = true` allows to use information from the indexing column. `f` will need to accept two arguments, the first being the key (as a `Tuple` or `NamedTuple`) the second the data (as `Columns`).

```
julia> t = table([1,1,2,2], [3,4,5,6], names=[:x,:y])

julia> groupby((:x_plus_mean_y => (key, d) -> key.x + mean(d),),
                                t, :x, select=:y, usekey = true)
Table with 2 rows, 2 columns:
x  x_plus_mean_y

1  4.5
2  7.5
```

source

4.13 IndexedTables.groupjoin

`IndexedTables.groupjoin` — *Method.*

`groupjoin([f, ] left, right; how, <options>)`

Join `left` and `right` creating groups of values with matching keys.

Inner join

Inner join is the default join (when `how` is unspecified). It looks up keys from `left` in `right` and only joins them when there is a match. This generates the "intersection" of keys from `left` and `right`.

One-to-many and many-to-many matches

If the same key appears multiple times in either table (say, `m` and `n` times respectively), each row with a key from `left` is matched with each row from `right` with that key. The resulting group has `mn` output elements.

```
julia> l = table([1,1,1,2], [1,2,2,1], [1,2,3,4],
                 names=[:a,:b,:c], pkey=(:a, :b))
Table with 4 rows, 3 columns:
a  b  c

1  1  1
1  2  2
1  2  3
2  1  4

julia> r = table([0,1,1,2], [1,2,2,1], [1,2,3,4],
                 names=[:a,:b,:d], pkey=(:a, :b))
Table with 4 rows, 3 columns:
a  b  d

0  1  1
```

```
1  2  2
1  2  3
2  1  4

julia> groupjoin(l,r)
Table with 2 rows, 3 columns:
a  b  groups

1  2  NamedTuples._NT_c_d{Int64,Int64}[(c = 2, d = 2), (c = 2, d = 3), (c = 3, d = 2), (c = 3, d
2  1  NamedTuples._NT_c_d{Int64,Int64}[(c = 4, d = 4)]
```

Left join

Left join looks up rows from `right` where keys match that in `left`. If there are no such rows in `right`, an NA value is used for every selected field from right.

```
julia> groupjoin(l,r, how=:left)
Table with 3 rows, 3 columns:
a  b  groups

1  1  NamedTuples._NT_c_d{Int64,Int64}[]
1  2  NamedTuples._NT_c_d{Int64,Int64}[(c = 2, d = 2), (c = 2, d = 3), (c = 3, d = 2), (c = 3, d
2  1  NamedTuples._NT_c_d{Int64,Int64}[(c = 4, d = 4)]
```

Outer join

Outer (aka Union) join looks up rows from `right` where keys match that in `left`, and also rows from `left` where keys match those in `left`, if there are no matches on either side, a tuple of NA values is used. The output is guarranteed to contain

```
julia> groupjoin(l,r, how=:outer)
Table with 4 rows, 3 columns:
a  b  groups

0  1  NamedTuples._NT_c_d{Int64,Int64}[]
1  1  NamedTuples._NT_c_d{Int64,Int64}[]
1  2  NamedTuples._NT_c_d{Int64,Int64}[(c = 2, d = 2), (c = 2, d = 3), (c = 3, d = 2), (c = 3, d
2  1  NamedTuples._NT_c_d{Int64,Int64}[(c = 4, d = 4)]
```

Options

- `how::Symbol` – join method to use. Described above.
- `lkey::Selection` – fields from `left` to match on
- `rkey::Selection` – fields from `right` to match on

- `lselect::Selection` – fields from `left` to use as input to use as output columns, or input to `f` if it is specified. By default, this is all fields not selected in `lkey`.
- `rselect::Selection` – fields from `left` to use as input to use as output columns, or input to `f` if it is specified. By default, this is all fields not selected in `rkey`.

```
julia> groupjoin(l,r, lkey=:a, rkey=:a, lselect=:c, rselect=:d, how=:outer)
Table with 3 rows, 2 columns:
a  groups

0  NamedTuples._NT_c_d{Int64,Int64}[]
1  NamedTuples._NT_c_d{Int64,Int64}[(c = 1, d = 2), (c = 1, d = 3), (c = 2, d = 2), (c
2  NamedTuples._NT_c_d{Int64,Int64}[(c = 4, d = 4)]
```

source

4.14 IndexedTables.groupreduce

`IndexedTables.groupreduce` — *Function.*

`groupreduce(f, t[, by::Selection]; select::Selection)`

Group rows by `by`, and apply `f` to reduce each group. `f` can be a function, OnlineStat or a struct of these as described in `reduce`. Recommended: see documentation for `reduce` first. The result of reducing each group is put in a table keyed by unique `by` values, the names of the output columns are the same as the names of the fields of the reduced tuples.

Examples

```
julia> t=table([1,1,1,2,2,2], [1,1,2,2,1,1], [1,2,3,4,5,6],
               names=[:x,:y,:z]);

julia> groupreduce(+, t, :x, select=:z)
Table with 2 rows, 2 columns:
x  +

1  6
2  15

julia> groupreduce(+, t, (:x, :y), select=:z)
Table with 4 rows, 3 columns:
x  y  +

1  1  3
1  2  3
2  1  11
```

```
2  2  4

julia> groupreduce((+, min, max), t, (:x, :y), select=:z)
Table with 4 rows, 5 columns:
x  y  +   min  max

1  1  3   1    2
1  2  3   3    3
2  1  11  5    6
2  2  4   4    4
```

If `f` is a single function or a tuple of functions, the output columns will be named the same as the functions themselves. To change the name, pass a named tuple:

```
julia> groupreduce(@NT(zsum=+, zmin=min, zmax=max), t, (:x, :y), select=:z)
Table with 4 rows, 5 columns:
x  y  zsum  zmin  zmax

1  1  3     1     2
1  2  3     3     3
2  1  11    5     6
2  2  4     4     4
```

Finally, it's possible to select different inputs for different reducers by using a named tuple of `slector => function` pairs:

```
julia> groupreduce(@NT(xsum=:x=>+, negysum=(:y=>-)=>+), t, :x)
Table with 2 rows, 3 columns:
x  xsum  negysum

1  3     -4
2  6     -4
```

source

4.15 IndexedTables.insertcol

`IndexedTables.insertcol` — *Method*.

`insertcol(t, position::Integer, name, x)`

Insert a column `x` named `name` at `position`. Returns a new table.

```
julia> t = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y
```

```
0.01  2  3
0.05  1  4

julia> insertcol(t, 2, :w, [0,1])
Table with 2 rows, 4 columns:
t     w  x  y

0.01  0  2  3
0.05  1  1  4
```

source

4.16 IndexedTables.insertcolafter

`IndexedTables.insertcolafter` — *Method.*

`insertcolafter(t, after, name, col)`

Insert a column `col` named `name` after `after`. Returns a new table.

```
julia> t = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  2  3
0.05  1  4

julia> insertcolafter(t, :t, :w, [0,1])
Table with 2 rows, 4 columns:
t     w  x  y

0.01  0  2  3
0.05  1  1  4
```

source

4.17 IndexedTables.insertcolbefore

`IndexedTables.insertcolbefore` — *Method.*

`insertcolbefore(t, before, name, col)`

Insert a column `col` named `name` before `before`. Returns a new table.

```
julia> t = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  2  3
```

```
0.05  1  4

julia> insertcolbefore(t, :x, :w, [0,1])
Table with 2 rows, 4 columns:
t     w  x  y

0.01  0  2  3
0.05  1  1  4
```

source

4.18 IndexedTables.ncols

`IndexedTables.ncols` — *Function.*

`ncols(itr)`

Returns the number of columns in `itr`.

```
julia> ncols([1,2,3])
1

julia> d = ncols(rows(([1,2,3],[4,5,6])))
2

julia> ncols(table(([1,2,3],[4,5,6])))
2

julia> ncols(table(@NT(x=[1,2,3],y=[4,5,6])))
2

julia> ncols(ndsparse(d, [7,8,9]))
3
```

source

4.19 IndexedTables.ndsparse

`IndexedTables.ndsparse` — *Function.*

`ndsparse(indices, data; agg, presorted, copy, chunks)`

Construct an NDSparse array with the given indices and data. Each vector in `indices` represents the index values for one dimension. On construction, the indices and data are sorted in lexicographic order of the indices.

Arguments:

- `agg::Function`: If `indices` contains duplicate entries, the corresponding data items are reduced using this 2-argument function.

- `presorted::Bool`: If true, the indices are assumed to already be sorted and no sorting is done.
- `copy::Bool`: If true, the storage for the new array will not be shared with the passed indices and data. If false (the default), the passed arrays will be copied only if necessary for sorting. The only way to guarantee sharing of data is to pass `presorted=true`.
- `chunks::Integer`: distribute the table into `chunks` (Integer) chunks (a safe bet is nworkers()). Not distributed by default. See Distributed docs.

Examples:
1-dimensional NDSparse can be constructed with a single array as index.

```
julia> x = ndsparse(["a","b"],[3,4])
1-d NDSparse with 2 values (Int64):
1

"a"  3
"b"  4

julia> keytype(x), eltype(x)
(Tuple{String}, Int64)
```

A dimension will be named if constructed with a named tuple of columns as index.

```
julia> x = ndsparse(@NT(date=Date.(2014:2017)), [4:7;])
1-d NDSparse with 4 values (Int64):
date

2014-01-01  4
2015-01-01  5
2016-01-01  6
2017-01-01  7

julia> x[Date("2015-01-01")]
5

julia> keytype(x), eltype(x)
(Tuple{Date}, Int64)
```

Multi-dimensional `NDSparse` can be constructed by passing a tuple of index columns:

```
julia> x = ndsparse((["a","b"],[3,4]), [5,6])
2-d NDSparse with 2 values (Int64):
1    2
```

```
"a"  3  5
"b"  4  6

julia> keytype(x), eltype(x)
(Tuple{String,Int64}, Int64)

julia> x["a", 3]
5
```

The data itself can also contain tuples (these are stored in columnar format, just like in `table`.)

```
julia> x = ndsparse((["a","b"],[3,4]), ([5,6], [7.,8.]))
2-d NDSparse with 2 values (2-tuples):
1    2  3  4

"a"  3  5  7.0
"b"  4  6  8.0

julia> x = ndsparse(@NT(x=["a","a","b"],y=[3,4,4]),
                    @NT(p=[5,6,7], q=[8.,9.,10.]))
2-d NDSparse with 3 values (2 field named tuples):
x    y  p  q

"a"  3  5  8.0
"a"  4  6  9.0
"b"  4  7  10.0

julia> keytype(x), eltype(x)
(Tuple{String,Int64}, NamedTuples._NT_p_q{Int64,Float64})

julia> x["a", :]
2-d NDSparse with 2 values (2 field named tuples):
x    y  p  q

"a"  3  5  8.0
"a"  4  6  9.0
```

Passing a `chunks` option to `ndsparse`, or constructing with a distributed array will cause the result to be distributed. Use `distribute` function to distribute an array.

```
julia> x = ndsparse(@NT(date=Date.(2014:2017)), [4:7.;], chunks=2)
1-d Distributed NDSparse with 4 values (Float64) in 2 chunks:
date
```

```
2014-01-01  4.0
2015-01-01  5.0
2016-01-01  6.0
2017-01-01  7.0

julia> x = ndsparse(@NT(date=Date.(2014:2017)), distribute([4:7.0;], 2))
1-d Distributed NDSparse with 4 values (Float64) in 2 chunks:
date

2014-01-01  4.0
2015-01-01  5.0
2016-01-01  6.0
2017-01-01  7.0
```

Distribution is done to match the first distributed column from left to right. Specify `chunks` to override this.

source

4.20 IndexedTables.pairs

`IndexedTables.pairs` — *Method*.

`pairs(arr::NDSparse, indices...)`

Similar to `where`, but returns an iterator giving `index=>value` pairs. `index` will be a tuple.

source

4.21 IndexedTables.pkeynames

`IndexedTables.pkeynames` — *Method*.

`pkeynames(t::Table)`

Names of the primary key columns in `t`.

Example

```
julia> t = table([1,2], [3,4]);

julia> pkeynames(t)
()

julia> t = table([1,2], [3,4], pkey=1);

julia> pkeynames(t)
(1,)

julia> t = table([2,1],[1,3],[4,5], names=[:x,:y,:z], pkey=(1,2));
```

```
julia> pkeys(t)
2-element IndexedTables.Columns{NamedTuples._NT_x_y{Int64,Int64},NamedTuples._NT_x_y{Array{Int64,
 (x = 1, y = 3)
 (x = 2, y = 1)
```

source

4.22 IndexedTables.pkeynames

`IndexedTables.pkeynames` — *Method*.

```
pkeynames(t::NDSparse)
```

Names of the primary key columns in `t`.
Example

```
julia> x = ndsparse([1,2],[3,4])
1-d NDSparse with 2 values (Int64):
1

1  3
2  4

julia> pkeynames(x)
(1,)
```

source

4.23 IndexedTables.pkeys

`IndexedTables.pkeys` — *Method*.

```
pkeys(itr::Table)
```

Primary keys of the table. If Table doesn't have any designated primary key columns (constructed without `pkey` argument) then a default key of tuples `(1,):(n,)` is generated.
Example

```
julia> a = table(["a","b"], [3,4]) # no pkey
Table with 2 rows, 2 columns:
1    2
```

```
"a"  3
"b"  4

julia> pkeys(a)
2-element Columns{Tuple{Int64}}:
 (1,)
 (2,)

julia> a = table(["a","b"], [3,4], pkey=1)
Table with 2 rows, 2 columns:
1    2

"a"  3
"b"  4

julia> pkeys(a)
2-element Columns{Tuple{String}}:
 ("a",)
 ("b",)
```

source

4.24 IndexedTables.popcol

`IndexedTables.popcol` — *Method.*

`popcol(t, col)`

Remove the column `col` from the table. Returns a new table.

```
julia> t = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  2  3
0.05  1  4

julia> popcol(t, :x)
Table with 2 rows, 2 columns:
t     y

0.01  3
0.05  4
```

source

4.25 IndexedTables.pushcol

`IndexedTables.pushcol` — *Method.*

`pushcol(t, name, x)`

Push a column `x` to the end of the table. `name` is the name for the new column. Returns a new table.

Example:

```
julia> t = table([0.01, 0.05], [2,1], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  2  3
0.05  1  4

julia> pushcol(t, :z, [1//2, 3//4])
Table with 2 rows, 4 columns:
t     x  y  z

0.01  2  3  1//2
0.05  1  4  3//4
```

source

4.26 IndexedTables.reducedim_vec

`IndexedTables.reducedim_vec` — *Method.*

`reducedim_vec(f::Function, arr::NDSparse, dims)`

Like `reducedim`, except uses a function mapping a vector of values to a scalar instead of a 2-argument scalar function.

source

4.27 IndexedTables.reindex

`IndexedTables.reindex` — *Function.*

`reindex(t::Table, by[, select])`

Reindex `t` by columns selected in `by`. Keeps columns selected by `select` as non-indexed columns. By default all columns not mentioned in `by` are kept.

Use `selectkeys` to reindex and NDSparse object.

```
julia> t = table([2,1],[1,3],[4,5], names=[:x,:y,:z], pkey=(1,2))

julia> reindex(t, (:y, :z))
Table with 2 rows, 3 columns:
y  z  x
```

```
1  4  2
3  5  1

julia> pkeynames(t)
(:y, :z)

julia> reindex(t, (:w=>[4,5], :z))
Table with 2 rows, 4 columns:
w  z  x  y

4  5  1  3
5  4  2  1

julia> pkeynames(t)
(:w, :z)
```

source

4.28 IndexedTables.renamecol

`IndexedTables.renamecol` — *Method.*

`renamecol(t, col, newname)`

Set `newname` as the new name for column `col` in `t`. Returns a new table.

```
julia> t = table([0.01, 0.05], [2,1], names=[:t, :x])
Table with 2 rows, 2 columns:
t     x

0.01  2
0.05  1

julia> renamecol(t, :t, :time)
Table with 2 rows, 2 columns:
time  x

0.01  2
0.05  1
```

source

4.29 IndexedTables.rows

`IndexedTables.rows` — *Function.*

`rows(itr[, select::Selection])`

Select one or more fields from an iterable of rows as a vector of their values.

`select` specifies which fields to select. See `Selection convention` for possible values. If unspecified, returns all columns.

`itr` can be `NDSparse`, `Columns` and `AbstractVector`, and their distributed counterparts.

Examples

```
julia> t = table([1,2],[3,4], names=[:x,:y])
Table with 2 rows, 2 columns:
x  y

1  3
2  4

julia> rows(t)
2-element IndexedTables.Columns{NamedTuples._NT_x_y{Int64,Int64},NamedTuples._NT_x_y{Array{Int64,
 (x = 1, y = 3)
 (x = 2, y = 4)

julia> rows(t, :x)
2-element Array{Int64,1}:
 1
 2

julia> rows(t, (:x,))
2-element IndexedTables.Columns{NamedTuples._NT_x{Int64},NamedTuples._NT_x{Array{Int64,1}}}:
 (x = 1)
 (x = 2)

julia> rows(t, (:y,:x=>-))
2-element IndexedTables.Columns{NamedTuples._NT_y_x{Int64,Int64},NamedTuples._NT_y_x{Array{Int64,
 (y = 3, x = -1)
 (y = 4, x = -2)
```

Note that vectors of tuples returned are `Columns` object and have columnar internal storage.

source

4.30 IndexedTables.setcol

`IndexedTables.setcol` — *Method.*

`setcol(t::Table, col::Union{Symbol, Int}, x::Selection)`

Sets a `x` as the column identified by `col`. Returns a new table.

`setcol(t::Table, map::Pair...)`

Set many columns at a time.

Examples:

```
julia> t = table([1,2], [3,4], names=[:x, :y])
Table with 2 rows, 2 columns:
x  y

1  3
2  4

julia> setcol(t, 2, [5,6])
Table with 2 rows, 2 columns:
x  y

1  5
2  6
```

x can be any selection that transforms existing columns.

```
julia> setcol(t, :x, :x => x->1/x)
Table with 2 rows, 2 columns:
x    y

1.0  5
0.5  6
```

`setcol` will result in a re-sorted copy if a primary key column is replaced.

```
julia> t = table([0.01, 0.05], [1,2], [3,4], names=[:t, :x, :y], pkey=:t)
Table with 2 rows, 3 columns:
t     x  y

0.01  1  3
0.05  2  4

julia> t2 = setcol(t, :t, [0.1,0.05])
Table with 2 rows, 3 columns:
t     x  y

0.05  2  4
0.1   1  3

julia> t == t2
false
```

source

4.31 IndexedTables.stack

`IndexedTables.stack` — *Method.*

stack(t, by = pkeynames(t); select = excludecols(t, by), variable = :variable, value = :value)

Reshape a table from the wide to the long format. Columns in `by` are kept as indexing columns. Columns in `select` are stacked. In addition to the id columns, two additional columns labeled `variable` and `value` are added, containg the column identifier and the stacked columns.

Examples

```
julia> t = table(1:4, [1, 4, 9, 16], [1, 8, 27, 64], names = [:x, :xsquare, :xcube], pkey = :x);

julia> stack(t)
Table with 8 rows, 3 columns:
x  variable  value

1  :xsquare  1
1  :xcube    1
2  :xsquare  4
2  :xcube    8
3  :xsquare  9
3  :xcube    27
4  :xsquare  16
4  :xcube    64
```

source

4.32 IndexedTables.summarize

`IndexedTables.summarize` — *Function.*

summarize(f, t, by = pkeynames(t); select = excludecols(t, by))

Apply summary functions column-wise to a table. Return a `NamedTuple` in the non-grouped case and a table in the grouped case.

Examples

```
julia> t = table([1, 2, 3], [1, 1, 1], names = [:x, :y]);

julia> summarize((mean, std), t)
(x_mean = 2.0, y_mean = 1.0, x_std = 1.0, y_std = 0.0)

julia> s = table(["a","a","b","b"], [1,3,5,7], [2,2,2,2], names = [:x, :y, :z], pkey = :x);

julia> summarize(mean, s)
Table with 2 rows, 3 columns:
x    y    z

"a"  2.0  2.0
"b"  6.0  2.0
```

Use a `NamedTuple` to have different names for the summary functions:

```
julia> summarize(@NT(m = mean, s = std), t)
(x_m = 2.0, y_m = 1.0, x_s = 1.0, y_s = 0.0)
```

Use `select` to only summarize some columns:

```
julia> summarize(@NT(m = mean, s = std), t, select = :x)
(m = 2.0, s = 1.0)
```

source

4.33 IndexedTables.table

`IndexedTables.table` — *Function*.

`table(cols::AbstractVector...; names, <options>)`

Create a table with columns given by `cols`.

```
julia> a = table([1,2,3], [4,5,6])
Table with 3 rows, 2 columns:
1  2

1  4
2  5
3  6
```

`names` specify names for columns. If specified, the table will be an iterator of named tuples.

```
julia> b = table([1,2,3], [4,5,6], names=[:x, :y])
Table with 3 rows, 2 columns:
x  y

1  4
2  5
3  6
```

`table(cols::Union{Tuple, NamedTuple}; <options>)`

Convert a struct of columns to a table of structs.

```
julia> table(([1,2,3], [4,5,6])) == a
true

julia> table(@NT(x=[1,2,3], y=[4,5,6])) == b
true
```

`table(cols::Columns; <options>)`

Construct a table from a vector of tuples. See `rows`.

```
julia> table(Columns([1,2,3], [4,5,6])) == a
true

julia> table(Columns(x=[1,2,3], y=[4,5,6])) == b
true
```

`table(t::Union{Table, NDSparse}; <options>)`

Copy a Table or NDSparse to create a new table. The same primary keys as the input are used.

```
julia> b == table(b)
true
```

`table(iter; <options>)`

Construct a table from an iterable table.

Options:

- `pkey`: select columns to act as the primary key. By default, no columns are used as primary key.
- `presorted`: is the data pre-sorted by primary key columns? If so, skip sorting. `false` by default. Irrelevant if `chunks` is specified.
- `copy`: creates a copy of the input vectors if `true`. `true` by default. Irrelavant if `chunks` is specified.
- `chunks`: distribute the table into `chunks` (Integer) chunks (a safe bet is nworkers()). Table is not distributed by default. See Distributed docs.

Examples:

Specifying `pkey` will cause the table to be sorted by the columns named in pkey:

```
julia> b = table([2,3,1], [4,5,6], names=[:x, :y], pkey=:x)
Table with 3 rows, 2 columns:
x  y

1  6
2  4
3  5

julia> b = table([2,1,2,1],[2,3,1,3],[4,5,6,7],
                 names=[:x, :y, :z], pkey=(:x,:y))
Table with 4 rows, 3 columns:
x  y  z

1  3  5
1  3  7
2  1  6
2  2  4
```

Note that the keys do not have to be unique.
`chunks` option creates a distributed table.
`chunks` can be:

1. An integer – number of chunks to create
2. An vector of `k` integers – number of elements in each of the `k` chunks.
3. The distribution of another array. i.e. `vec.subdomains` where `vec` is a distributed array.

```
julia> t = table([2,3,1,4], [4,5,6,7],
                 names=[:x, :y], pkey=:x, chunks=2)
Distributed Table with 4 rows in 2 chunks:
x  y

1  6
2  4
3  5
4  7
```

A distributed table will be constructed if one of the arrays passed into `table` constructor is a distributed array. A distributed Array can be constructed using `distribute`:

```
julia> x = distribute([1,2,3,4], 2);

julia> t = table(x, [5,6,7,8], names=[:x,:y])
Distributed Table with 4 rows in 2 chunks:
x  y

1  5
2  6
3  7
4  8

julia> table(columns(t)..., [9,10,11,12],
             names=[:x,:y,:z])
Distributed Table with 4 rows in 2 chunks:
x  y  z

1  5  9
2  6  10
3  7  11
4  8  12
```

Distribution is done to match the first distributed column from left to right. Specify `chunks` to override this.

source

4.34 IndexedTables.unstack

`IndexedTables.unstack` — *Method*.

`unstack(t, by = pkeynames(t); variable = :variable, value = :value)`

Reshape a table from the long to the wide format. Columns in `by` are kept as indexing columns. Keyword arguments `variable` and `value` denote which column contains the column identifier and which the corresponding values.

Examples

```
julia> t = table(1:4, [1, 4, 9, 16], [1, 8, 27, 64], names = [:x, :xsquare, :xcube], pkey = :x);

julia> long = stack(t)
Table with 8 rows, 3 columns:
x  variable  value

1  :xsquare  1
1  :xcube    1
2  :xsquare  4
2  :xcube    8
3  :xsquare  9
3  :xcube    27
4  :xsquare  16
4  :xcube    64

julia> unstack(long)
Table with 4 rows, 3 columns:
x  xsquare  xcube

1  1        1
2  4        8
3  9        27
4  16       64
```

source

4.35 IndexedTables.update!

`IndexedTables.update!` — *Method*.

`update!(f::Function, arr::NDSparse, indices...)`

Replace data values `x` with `f(x)` at each location that matches the given indices.

source

4.36 IndexedTables.where

`IndexedTables.where` — *Method*.

`where(arr::NDSparse, indices...)`

Returns an iterator over data items where the given indices match. Accepts the same index arguments as `getindex`.

source

Chapter 5

Images

5.1 ColorTypes.blue

`ColorTypes.blue` — *Function.*

`b = blue(img)` extracts the blue channel from an RGB image `img`

source

5.2 ColorTypes.green

`ColorTypes.green` — *Function.*

`g = green(img)` extracts the green channel from an RGB image `img`

source

5.3 ColorTypes.red

`ColorTypes.red` — *Function.*

`r = red(img)` extracts the red channel from an RGB image `img`

source

5.4 Images.adjust_gamma

`Images.adjust_gamma` — *Method.*

```
gamma_corrected_img = adjust_gamma(img, gamma)
```

Returns a gamma corrected image.

The `adjust_gamma` function can handle a variety of input types. The returned image depends on the input type. If the input is an `Image` then the resulting image is of the same type and has the same properties.

For coloured images, the input is converted to YIQ type and the Y channel is gamma corrected. This is the combined with the I and Q channels and the resulting image converted to the same type as the input.

source

5.5 Images.bilinear_interpolation

`Images.bilinear_interpolation` — *Method.*

```
P = bilinear_interpolation(img, r, c)
```

Bilinear Interpolation is used to interpolate functions of two variables on a rectilinear 2D grid.

The interpolation is done in one direction first and then the values obtained are used to do the interpolation in the second direction.

source

5.6 Images.blob_LoG

`Images.blob_LoG` — *Method.*

```
blob_LoG(img, scales, [edges], [shape]) -> Vector{BlobLoG}
```

Find "blobs" in an N-D image using the negative Lapacian of Gaussians with the specifed vector or tuple of values. The algorithm searches for places where the filtered image (for a particular) is at a peak compared to all spatially- and -adjacent voxels, where is `scales[i] * shape` for some i. By default, `shape` is an ntuple of 1s.

The optional `edges` argument controls whether peaks on the edges are included. `edges` can be `true` or `false`, or a N+1-tuple in which the first entry controls whether edge- values are eligible to serve as peaks, and the remaining N entries control each of the N dimensions of `img`.

Citation:

Lindeberg T (1998), "Feature Detection with Automatic Scale Selection", International Journal of Computer Vision, 30(2), 79–116.

See also: `BlobLoG`.

source

5.7 Images.boxdiff

`Images.boxdiff` — *Method.*

```
sum = boxdiff(integral_image, ytop:ybot, xtop:xbot)
sum = boxdiff(integral_image, CartesianIndex(tl_y, tl_x), CartesianIndex(br_y, br_x))
sum = boxdiff(integral_image, tl_y, tl_x, br_y, br_x)
```

An integral image is a data structure which helps in efficient calculation of sum of pixels in a rectangular subset of an image. It stores at each pixel the sum of all pixels above it and to its left. The sum of a window in an image can be directly calculated using four array references of the integral image, irrespective of the size of the window, given the `yrange` and `xrange` of the window. Given an integral image -

```
A - - - - - - B -
- * * * * * * * -
- * * * * * * * -
- * * * * * * * -
- * * * * * * * -
- * * * * * * * -
C * * * * * * D -
- - - - - - - - -
```

The sum of pixels in the area denoted by * is given by S = D + A - B - C.
source

5.8 Images.canny

`Images.canny` — *Method.*

```
canny_edges = canny(img, (upper, lower), sigma=1.4)
```

Performs Canny Edge Detection on the input image.
Parameters :
(upper, lower) : Bounds for hysteresis thresholding sigma : Specifies the standard deviation of the gaussian filter
Example
[] imgedg = canny(img, (Percentile(80), Percentile(20)))
source

5.9 Images.clahe

`Images.clahe` — *Method.*

```
hist_equalised_img = clahe(img, nbins, xblocks = 8, yblocks = 8, clip = 3)
```

Performs Contrast Limited Adaptive Histogram Equalisation (CLAHE) on the input image. It differs from ordinary histogram equalization in the respect that the adaptive method computes several histograms, each corresponding to a distinct section of the image, and uses them to redistribute the lightness values of the image. It is therefore suitable for improving the local contrast and enhancing the definitions of edges in each region of an image.

In the straightforward form, CLAHE is done by calculation a histogram of a window around each pixel and using the transformation function of the equalised histogram to rescale the pixel. Since this is computationally expensive, we use interpolation which gives a significant rise in efficiency without compromising the result. The image is divided into a grid and equalised histograms are calculated for each block. Then, each pixel is interpolated using the closest histograms.

The `xblocks` and `yblocks` specify the number of blocks to divide the input image into in each direction. `nbins` specifies the granularity of histogram calculation of each local region. `clip` specifies the value at which the histogram is clipped. The excess in the histogram bins with value exceeding `clip` is redistributed among the other bins.

source

5.10 Images.cliphist

`Images.cliphist` — *Method*.

```
clipped_hist = cliphist(hist, clip)
```

Clips the histogram above a certain value `clip`. The excess left in the bins exceeding `clip` is redistributed among the remaining bins.

source

5.11 Images.complement

`Images.complement` — *Method*.

```
y = complement(x)
```

Take the complement `1-x` of `x`. If `x` is a color with an alpha channel, the alpha channel is left untouched.

source

5.12 Images.component_boxes

`Images.component_boxes` — *Method*.

`component_boxes(labeled_array)` -> an array of bounding boxes for each label, including the background label 0

source

5.13 Images.component_centroids

`Images.component_centroids` — *Method*.

`component_centroids(labeled_array)` -> an array of centroids for each label, including the background label 0

source

5.14 Images.component_indices

`Images.component_indices` — *Method*.

`component_indices(labeled_array)` -> an array of pixels for each label, including the background label 0

source

5.15 Images.component_lengths

`Images.component_lengths` — *Method*.

`component_lengths(labeled_array)` -> an array of areas (2D), volumes (3D), etc. for each label, including the background label 0

source

5.16 Images.component_subscripts

`Images.component_subscripts` — *Method*.

`component_subscripts(labeled_array)` -> an array of pixels for each label, including the background label 0

source

5.17 Images.convexhull

`Images.convexhull` — *Method*.

```
chull = convexhull(img)
```

Computes the convex hull of a binary image and returns the vertices of convex hull as a CartesianIndex array.

source

5.18 Images.corner2subpixel

`Images.corner2subpixel` — *Method*.

```
corners = corner2subpixel(responses::AbstractMatrix,corner_indicator::AbstractMatrix{Bool})
        -> Vector{HomogeneousPoint{Float64,3}}
```

Refines integer corner coordinates to sub-pixel precision.

The function takes as input a matrix representing corner responses and a boolean indicator matrix denoting the integer coordinates of a corner in the image. The output is a vector of type `HomogeneousPoint` storing the sub-pixel coordinates of the corners.

The algorithm computes a correction factor which is added to the original integer coordinates. In particular, a univariate quadratic polynomial is fit separately to the x-coordinates and y-coordinates of a corner and its immediate east/west, and north/south neighbours. The fit is achieved using a local coordinate system for each corner, where the origin of the coordinate system is a given corner, and its immediate neighbours are assigned coordinates of minus one and plus one.

The corner and its two neighbours form a system of three equations. For example, let $x_1 = -1$, $x_2 = 0$ and $x_3 = 1$ denote the local x coordinates of the west, center and east pixels and let the vector $\mathbf{b} = [r_1, r_2, r_3]$ denote the corresponding corner response values. With

$$\mathbf{A} = \begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix},$$

the coefficients of the quadratic polynomial can be found by solving the system of equations $\mathbf{b} = \mathbf{A}\mathbf{x}$. The result is given by $x = \mathbf{A}^{-1}\mathbf{b}$.

The vertex of the quadratic polynomial yields a sub-pixel estimate of the true corner position. For example, for a univariate quadratic polynomial px^2+qx+r, the x-coordinate of the vertex is $\frac{-q}{2p}$. Hence, the refined sub-pixel coordinate is equal to: $c + \frac{-q}{2p}$, where c is the integer coordinate.

!!! note Corners on the boundary of the image are not refined to sub-pixel precision.

source

5.19 Images.entropy

`Images.entropy` — *Method.*

```
entropy(log, img)
entropy(img; [kind=:shannon])
```

Compute the entropy of a grayscale image defined as `-sum(p.*log(p))`. The base of the logarithm (a.k.a. entropy unit) is one of the following:

- `:shannon` (log base 2, default), or use log = log2
- `:nat` (log base e), or use log = log
- `:hartley` (log base 10), or use log = log10

source

5.20 Images.fastcorners

`Images.fastcorners` — *Method.*

```
fastcorners(img, n, threshold) -> corners
```

Performs FAST Corner Detection. `n` is the number of contiguous pixels which need to be greater (lesser) than intensity + threshold (intensity - threshold) for a pixel to be marked as a corner. The default value for n is 12.

source

5.21 Images.findlocalmaxima

`Images.findlocalmaxima` — *Function.*

`findlocalmaxima(img, [region, edges]) -> Vector{CartesianIndex}`

Returns the coordinates of elements whose value is larger than all of their immediate neighbors. `region` is a list of dimensions to consider. `edges` is a boolean specifying whether to include the first and last elements of each dimension, or a tuple-of-Bool specifying edge behavior for each dimension separately.

source

5.22 Images.findlocalminima

`Images.findlocalminima` — *Function.*

Like `findlocalmaxima`, but returns the coordinates of the smallest elements.

source

5.23 Images.gaussian_pyramid

`Images.gaussian_pyramid` — *Method.*

```
pyramid = gaussian_pyramid(img, n_scales, downsample, sigma)
```

Returns a gaussian pyramid of scales `n_scales`, each downsampled by a factor `downsample` and `sigma` for the gaussian kernel.

source

5.24 Images.harris

`Images.harris` — *Method.*

```
harris_response = harris(img; [k], [border], [weights])
```

Performs Harris corner detection. The covariances can be taken using either a mean weighted filter or a gamma kernel.

source

5.25 Images.hausdorff_distance

`Images.hausdorff_distance` — *Method*.

`hausdorff_distance(imgA, imgB)` is the modified Hausdorff distance between binary images (or point sets).

References

Dubuisson, M-P; Jain, A. K., 1994. A Modified Hausdorff Distance for Object-Matching.

source

5.26 Images.histeq

`Images.histeq` — *Method*.

```
hist_equalised_img = histeq(img, nbins)
hist_equalised_img = histeq(img, nbins, minval, maxval)
```

Returns a histogram equalised image with a granularity of approximately `nbins` number of bins.

The `histeq` function can handle a variety of input types. The returned image depends on the input type. If the input is an `Image` then the resulting image is of the same type and has the same properties.

For coloured images, the input is converted to YIQ type and the Y channel is equalised. This is the combined with the I and Q channels and the resulting image converted to the same type as the input.

If minval and maxval are specified then intensities are equalized to the range (minval, maxval). The default values are 0 and 1.

source

5.27 Images.histmatch

`Images.histmatch` — *Function*.

```
hist_matched_img = histmatch(img, oimg, nbins)
```

Returns a grayscale histogram matched image with a granularity of `nbins` number of bins. `img` is the image to be matched and `oimg` is the image having the desired histogram to be matched to.

source

5.28 Images.imROF

`Images.imROF` — *Method*.

```
imgr = imROF(img, , iterations)
```

Perform Rudin-Osher-Fatemi (ROF) filtering, more commonly known as Total Variation (TV) denoising or TV regularization. is the regularization coefficient for the derivative, and `iterations` is the number of relaxation iterations taken. 2d only.

See https://en.wikipedia.org/wiki/Total_variation_denoising and Chambolle, A. (2004). "An algorithm for total variation minimization and applications". Journal of Mathematical Imaging and Vision. 20: 89–97

source

5.29 Images.imadjustintensity

`Images.imadjustintensity` — *Method*.

`imadjustintensity(img [, (minval,maxval)]) -> Image`

Map intensities over the interval `(minval,maxval)` to the interval `[0,1]`. This is equivalent to `map(ScaleMinMax(eltype(img), minval, maxval), img)`. (minval,maxval) defaults to `extrema(img)`.

source

5.30 Images.imcorner

`Images.imcorner` — *Method*.

```
corners = imcorner(img; [method])
corners = imcorner(img, threshold, percentile; [method])
```

Performs corner detection using one of the following methods -

```
1. harris
2. shi_tomasi
3. kitchen_rosenfeld
```

The parameters of the individual methods are described in their documentation. The maxima values of the resultant responses are taken as corners. If a threshold is specified, the values of the responses are thresholded to give the corner pixels. The threshold is assumed to be a percentile value unless `percentile` is set to false.

source

5.31 Images.imcorner_subpixel

`Images.imcorner_subpixel` — *Method*.

```
corners = imcorner_subpixel(img; [method])
         -> Vector{HomogeneousPoint{Float64,3}}
corners = imcorner_subpixel(img, threshold, percentile; [method])
         -> Vector{HomogeneousPoint{Float64,3}}
```

Same as imcorner, but estimates corners to sub-pixel precision.

Sub-pixel precision is achieved by interpolating the corner response values using the 4-connected neighbourhood of a maximum response value. See corner2subpixel for more details of the interpolation scheme.

source

5.32 Images.imedge

Images.imedge — *Function*.

```
grad_y, grad_x, mag, orient = imedge(img, kernelfun=KernelFactors.ando3, border="repli
```

Edge-detection filtering. kernelfun is a valid kernel function for imgradients, defaulting to KernelFactors.ando3. border is any of the boundary conditions specified in padarray.

Returns a tuple (grad_y, grad_x, mag, orient), which are the horizontal gradient, vertical gradient, and the magnitude and orientation of the strongest edge, respectively.

source

5.33 Images.imhist

Images.imhist — *Method*.

```
edges, count = imhist(img, nbins)
edges, count = imhist(img, nbins, minval, maxval)
```

Generates a histogram for the image over nbins spread between (minval, maxval]. If minval and maxval are not given, then the minimum and maximum values present in the image are taken.

edges is a vector that specifies how the range is divided; count[i+1] is the number of values x that satisfy edges[i] <= x < edges[i+1]. count[1] is the number satisfying x < edges[1], and count[end] is the number satisfying x >= edges[end]. Consequently, length(count) == length(edges)+1.

source

5.34 Images.imstretch

Images.imstretch — *Method*.

imgs = imstretch(img, m, slope) enhances or reduces (for slope > 1 or < 1, respectively) the contrast near saturation (0 and 1). This is essentially a symmetric gamma-correction. For a pixel of brightness p, the new intensity is 1/(1+(m/(p+eps))^slope).

This assumes the input img has intensities between 0 and 1.

source

5.35 Images.integral_image

`Images.integral_image` — *Method.*

```
integral_img = integral_image(img)
```

Returns the integral image of an image. The integral image is calculated by assigning to each pixel the sum of all pixels above it and to its left, i.e. the rectangle from (1, 1) to the pixel. An integral image is a data structure which helps in efficient calculation of sum of pixels in a rectangular subset of an image. See `boxdiff` for more information.

source

5.36 Images.kitchen_rosenfeld

`Images.kitchen_rosenfeld` — *Method.*

```
kitchen_rosenfeld_response = kitchen_rosenfeld(img; [border])
```

Performs Kitchen Rosenfeld corner detection. The covariances can be taken using either a mean weighted filter or a gamma kernel.

source

5.37 Images.label_components

`Images.label_components` — *Function.*

```
label = label_components(tf, [connectivity])
label = label_components(tf, [region])
```

Find the connected components in a binary array `tf`. There are two forms that `connectivity` can take:

- It can be a boolean array of the same dimensionality as `tf`, of size 1 or 3

along each dimension. Each entry in the array determines whether a given neighbor is used for connectivity analyses. For example, `connectivity = trues(3,3)` would use 8-connectivity and test all pixels that touch the current one, even the corners.

- You can provide a list indicating which dimensions are used to

determine connectivity. For example, `region = [1,3]` would not test neighbors along dimension 2 for connectivity. This corresponds to just the nearest neighbors, i.e., 4-connectivity in 2d and 6-connectivity in 3d.

The default is `region = 1:ndims(A)`.

The output `label` is an integer array, where 0 is used for background pixels, and each connected region gets a different integer index.

source

5.38 Images.magnitude

`Images.magnitude` — *Method.*

```
m = magnitude(grad_x, grad_y)
```

Calculates the magnitude of the gradient images given by `grad_x` and `grad_y`. Equivalent to $sqrt(grad_x.^2 + grad_y.^2)$.

Returns a magnitude image the same size as `grad_x` and `grad_y`.

source

5.39 Images.magnitude_phase

`Images.magnitude_phase` — *Method.*

```
magnitude_phase(grad_x, grad_y) -> m, p
```

Convenience function for calculating the magnitude and phase of the gradient images given in `grad_x` and `grad_y`. Returns a tuple containing the magnitude and phase images. See `magnitude` and `phase` for details.

source

5.40 Images.maxabsfinite

`Images.maxabsfinite` — *Method.*

`m = maxabsfinite(A)` calculates the maximum absolute value in `A`, ignoring any values that are not finite (Inf or NaN).

source

5.41 Images.maxfinite

`Images.maxfinite` — *Method.*

`m = maxfinite(A)` calculates the maximum value in `A`, ignoring any values that are not finite (Inf or NaN).

source

5.42 Images.meanfinite

`Images.meanfinite` — *Method.*

`M = meanfinite(img, region)` calculates the mean value along the dimensions listed in `region`, ignoring any non-finite values.

source

5.43 Images.minfinite

`Images.minfinite` — *Method*.

`m = minfinite(A)` calculates the minimum value in `A`, ignoring any values that are not finite (Inf or NaN).

source

5.44 Images.ncc

`Images.ncc` — *Method*.

`C = ncc(A, B)` computes the normalized cross-correlation of `A` and `B`.

source

5.45 Images.orientation

`Images.orientation` — *Method*.

```
orientation(grad_x, grad_y) -> orient
```

Calculate the orientation angle of the strongest edge from gradient images given by `grad_x` and `grad_y`. Equivalent to `atan2(grad_x, grad_y)`. When both `grad_x` and `grad_y` are effectively zero, the corresponding angle is set to zero.

source

5.46 Images.otsu_threshold

`Images.otsu_threshold` — *Method*.

```
thres = otsu_threshold(img)
thres = otsu_threshold(img, bins)
```

Computes threshold for grayscale image using Otsu's method.

Parameters:

- img = Grayscale input image
- bins = Number of bins used to compute the histogram. Needed for floating-point images.

source

5.47 Images.phase

`Images.phase` — *Method.*

```
phase(grad_x, grad_y) -> p
```

Calculate the rotation angle of the gradient given by `grad_x` and `grad_y`. Equivalent to `atan2(-grad_y, grad_x)`, except that when both `grad_x` and `grad_y` are effectively zero, the corresponding angle is set to zero.

source

5.48 Images.sad

`Images.sad` — *Method.*

`s = sad(A, B)` computes the sum-of-absolute differences over arrays/images A and B

source

5.49 Images.sadn

`Images.sadn` — *Method.*

`s = sadn(A, B)` computes the sum-of-absolute differences over arrays/images A and B, normalized by array size

source

5.50 Images.shepp_logan

`Images.shepp_logan` — *Method.*

```
phantom = shepp_logan(N,[M]; highContrast=true)
```

output the NxM Shepp-Logan phantom, which is a standard test image usually used for comparing image reconstruction algorithms in the field of computed tomography (CT) and magnetic resonance imaging (MRI). If the argument M is omitted, the phantom is of size NxN. When setting the keyword argument `highConstrast` to false, the CT version of the phantom is created. Otherwise, the high contrast MRI version is calculated.

source

5.51 Images.shi_tomasi

`Images.shi_tomasi` — *Method.*

```
shi_tomasi_response = shi_tomasi(img; [border], [weights])
```

Performs Shi Tomasi corner detection. The covariances can be taken using either a mean weighted filter or a gamma kernel.

source

5.52 Images.ssd

`Images.ssd` — *Method*.

`s = ssd(A, B)` computes the sum-of-squared differences over arrays/images A and B

source

5.53 Images.ssdn

`Images.ssdn` — *Method*.

`s = ssdn(A, B)` computes the sum-of-squared differences over arrays/images A and B, normalized by array size

source

5.54 Images.thin_edges

`Images.thin_edges` — *Method*.

```
thinned = thin_edges(img, gradientangle, [border])
thinned, subpix = thin_edges_subpix(img, gradientangle, [border])
thinned, subpix = thin_edges_nonmaxsup(img, gradientangle, [border]; [radius::Float64=1.35], [the
thinned, subpix = thin_edges_nonmaxsup_subpix(img, gradientangle, [border]; [radius::Float64=1.35
```

Edge thinning for 2D edge images. Currently the only algorithm available is non-maximal suppression, which takes an edge image and its gradient angle, and checks each edge point for local maximality in the direction of the gradient. The returned image is non-zero only at maximal edge locations.

`border` is any of the boundary conditions specified in `padarray`.

In addition to the maximal edge image, the `_subpix` versions of these functions also return an estimate of the subpixel location of each local maxima, as a 2D array or image of `Graphics.Point` objects. Additionally, each local maxima is adjusted to the estimated value at the subpixel location.

Currently, the `_nonmaxsup` functions are identical to the first two function calls, except that they also accept additional keyword arguments. `radius` indicates the step size to use when searching in the direction of the gradient; values between 1.2 and 1.5 are suggested (default 1.35). `theta` indicates the step size to use when discretizing angles in the `gradientangle` image, in radians (default: 1 degree in radians = pi/180).

Example:

```
g = rgb2gray(rgb_image)
gx, gy = imgradients(g)
mag, grad_angle = magnitude_phase(gx,gy)
mag[mag .< 0.5] = 0.0  # Threshold magnitude image
thinned, subpix =  thin_edges_subpix(mag, grad_angle)
```

source

Chapter 6

Knet

6.1 Knet.accuracy

`Knet.accuracy` — *Method*.

```
accuracy(model, data, predict; average=true)
```

Compute `accuracy(predict(model,x), y)` for `(x,y)` in `data` and return the ratio (if average=true) or the count (if average=false) of correct answers.

source

6.2 Knet.accuracy

`Knet.accuracy` — *Method*.

```
accuracy(scores, answers, d=1; average=true)
```

Given an unnormalized `scores` matrix and an `Integer` array of correct `answers`, return the ratio of instances where the correct answer has the maximum score. `d=1` means instances are in columns, `d=2` means instances are in rows. Use `average=false` to return the number of correct answers instead of the ratio.

source

6.3 Knet.batchnorm

`Knet.batchnorm` — *Function*.

`batchnorm(x[, moments, params]; kwargs...)` performs batch normalization to `x` with optional scaling factor and bias stored in `params`.

2d, 4d and 5d inputs are supported. Mean and variance are computed over dimensions (2,), (1,2,4) and (1,2,3,5) for 2d, 4d and 5d arrays, respectively.

`moments` stores running mean and variance to be used in testing. It is optional in the training mode, but mandatory in the test mode. Training and test modes are controlled by the `training` keyword argument.

`params` stores the optional affine parameters gamma and beta. `bnparams` function can be used to initialize `params`.

Example

```
# Inilization, C is an integer
moments = bnmoments()
params = bnparams(C)
...
# size(x) -> (H, W, C, N)
y = batchnorm(x, moments, params)
# size(y) -> (H, W, C, N)
```

Keywords

`eps=1e-5`: The epsilon parameter added to the variance to avoid division by 0.

`training`: When `training` is true, the mean and variance of `x` are used and `moments` argument is modified if it is provided. When `training` is false, mean and variance stored in the `moments` argument are used. Default value is `true` when at least one of `x` and `params` is `AutoGrad.Rec`, `false` otherwise.

source

6.4 Knet.bilinear

`Knet.bilinear` — *Method*.

Bilinear interpolation filter weights; used for initializing deconvolution layers.

Adapted from https://github.com/shelhamer/fcn.berkeleyvision.org/blob/master/surgery.py#L33

Arguments:

`T` : Data Type

`fw`: Width upscale factor

`fh`: Height upscale factor

`IN`: Number of input filters

`ON`: Number of output filters

Example usage:

w = bilinear(Float32,2,2,128,128)

source

6.5 Knet.bnmoments

`Knet.bnmoments` — *Method*.

`bnmoments(;momentum=0.1, mean=nothing, var=nothing, meaninit=zeros, varinit=ones)` can be used directly load moments from data. `meaninit` and `varinit` are called if `mean` and `var` are nothing. Type and size of the `mean` and

`var` are determined automatically from the inputs in the `batchnorm` calls. A `BNMoments` object is returned.

BNMoments

A high-level data structure used to store running mean and running variance of batch normalization with the following fields:

`momentum::AbstractFloat`: A real number between 0 and 1 to be used as the scale of last mean and variance. The existing running mean or variance is multiplied by (1-momentum).

`mean`: The running mean.

`var`: The running variance.

`meaninit`: The function used for initialize the running mean. Should either be `nothing` or of the form `(eltype, dims...)->data`. `zeros` is a good option.

`varinit`: The function used for initialize the running variance. Should either be `nothing` or `(eltype, dims...)->data`. `ones` is a good option.

source

6.6 Knet.bnparams

`Knet.bnparams` — *Method*.

`bnparams(etype, channels)` creates a single 1d array that contains both scale and bias of batchnorm, where the first half is scale and the second half is bias.

`bnparams(channels)` calls `bnparams` with `etype=Float64`, following Julia convention

source

6.7 Knet.conv4

`Knet.conv4` — *Method*.

```
conv4(w, x; kwargs...)
```

Execute convolutions or cross-correlations using filters specified with `w` over tensor `x`.

Currently KnetArray{Float32/64,4/5} and Array{Float32/64,4} are supported as `w` and `x`. If `w` has dimensions `(W1,W2,...,I,O)` and `x` has dimensions `(X1,X2,...,I,N)`, the result `y` will have dimensions `(Y1,Y2,...,O,N)` where

```
Yi=1+floor((Xi+2*padding[i]-Wi)/stride[i])
```

Here `I` is the number of input channels, `O` is the number of output channels, `N` is the number of instances, and `Wi,Xi,Yi` are spatial dimensions. `padding` and `stride` are keyword arguments that can be specified as a single number (in which case they apply to all dimensions), or an array/tuple with entries for each spatial dimension.

Keywords

- `padding=0`: the number of extra zeros implicitly concatenated at the start and at the end of each dimension.
- `stride=1`: the number of elements to slide to reach the next filtering window.
- `upscale=1`: upscale factor for each dimension.
- `mode=0`: 0 for convolution and 1 for cross-correlation.
- `alpha=1`: can be used to scale the result.
- `handle`: handle to a previously created cuDNN context. Defaults to a Knet allocated handle.

source

6.8 Knet.deconv4

`Knet.deconv4` — *Method*.

```
y = deconv4(w, x; kwargs...)
```

Simulate 4-D deconvolution by using *transposed convolution* operation. Its forward pass is equivalent to backward pass of a convolution (gradients with respect to input tensor). Likewise, its backward pass (gradients with respect to input tensor) is equivalent to forward pass of a convolution. Since it swaps forward and backward passes of convolution operation, padding and stride options belong to output tensor. See this report for further explanation.

Currently KnetArray{Float32/64,4} and Array{Float32/64,4} are supported as `w` and `x`. If `w` has dimensions `(W1,W2,...,O,I)` and `x` has dimensions `(X1,X2,...,I,N)`, the result `y` will have dimensions `(Y1,Y2,...,O,N)` where

Yi = Wi+stride[i]*(Xi-1)*-2padding[i]

Here I is the number of input channels, O is the number of output channels, N is the number of instances, and Wi,Xi,Yi are spatial dimensions. padding and stride are keyword arguments that can be specified as a single number (in which case they apply to all dimensions), or an array/tuple with entries for each spatial dimension.

Keywords

- `padding=0`: the number of extra zeros implicitly concatenated at the start and at the end of each dimension.
- `stride=1`: the number of elements to slide to reach the next filtering window.
- `mode=0`: 0 for convolution and 1 for cross-correlation.
- `alpha=1`: can be used to scale the result.

- `handle`: handle to a previously created cuDNN context. Defaults to a Knet allocated handle.

source

6.9 Knet.dropout

`Knet.dropout` — *Method*.

```
dropout(x, p)
```

Given an array `x` and probability `0<=p<=1`, just return `x` if testing, return an array `y` in which each element is 0 with probability `p` or `x[i]/(1-p)` with probability `1-p` if training. Training mode is detected automatically based on the type of `x`, which is `AutoGrad.Rec` during gradient calculation. Use the keyword argument `training::Bool` to change the default mode and `seed::Number` to set the random number seed for reproducible results. See (Srivastava et al. 2014) for a reference.

source

6.10 Knet.gaussian

`Knet.gaussian` — *Method*.

```
gaussian(a...; mean=0.0, std=0.01)
```

Return a Gaussian array with a given mean and standard deviation. The `a` arguments are passed to `randn`.

source

6.11 Knet.goldensection

`Knet.goldensection` — *Method*.

```
goldensection(f,n;kwargs) => (fmin,xmin)
```

Find the minimum of `f` using concurrent golden section search in `n` dimensions. See `Knet.goldensection_demo()` for an example.

`f` is a function from a `Vector{Float64}` of length `n` to a `Number`. It can return `NaN` for out of range inputs. Goldensection will always start with a zero vector as the initial input to `f`, and the initial step size will be 1 in each dimension. The user should define `f` to scale and shift this input range into a vector meaningful for their application. For positive inputs like learning rate or hidden size, you can use a transformation such as `x0*exp(x)` where `x` is a value `goldensection` passes to `f` and `x0` is your initial guess for this value. This will effectively start the search at `x0`, then move with multiplicative steps.

I designed this algorithm combining ideas from Golden Section Search and Hill Climbing Search. It essentially runs golden section search concurrently in each dimension, picking the next step based on estimated gain.

Keyword arguments

- `dxmin=0.1`: smallest step size.
- `accel=`: acceleration rate. Golden ratio `=1.618...` is best.
- `verbose=false`: use `true` to print individual steps.
- `history=[]`: cache of `[(x,f(x)),...]` function evaluations.

source

6.12 Knet.gpu

`Knet.gpu` — *Function*.

`gpu()` returns the id of the active GPU device or -1 if none are active.

`gpu(true)` resets all GPU devices and activates the one with the most available memory.

`gpu(false)` resets and deactivates all GPU devices.

`gpu(d::Int)` activates the GPU device `d` if `0 <= d < gpuCount()`, otherwise deactivates devices.

`gpu(true/false)` resets all devices. If there are any allocated KnetArrays their pointers will be left dangling. Thus `gpu(true/false)` should only be used during startup. If you want to suspend GPU use temporarily, use `gpu(-1)`.

`gpu(d::Int)` does not reset the devices. You can select a previous device and find allocated memory preserved. However trying to operate on arrays of an inactive device will result in error.

source

6.13 Knet.hyperband

`Knet.hyperband` — *Function*.

```
hyperband(getconfig, getloss, maxresource=27, reduction=3)
```

Hyperparameter optimization using the hyperband algorithm from (Lisha et al. 2016). You can try a simple MNIST example using `Knet.hyperband_demo()`.

Arguments

- `getconfig()` returns random configurations with a user defined type and distribution.
- `getloss(c,n)` returns loss for configuration `c` and number of resources (e.g. epochs) `n`.

- `maxresource` is the maximum number of resources any one configuration should be given.
- `reduction` is an algorithm parameter (see paper), 3 is a good value.

source

6.14 Knet.invx

`Knet.invx` — *Function.*

```
invx(x) = (1./x)
```

source

6.15 Knet.knetgc

`Knet.knetgc` — *Function.*

```
knetgc(dev=gpu())
```

cudaFree all pointers allocated on device `dev` that were previously allocated and garbage collected. Normally Knet holds on to all garbage collected pointers for reuse. Try this if you run out of GPU memory.

source

6.16 Knet.logp

`Knet.logp` — *Method.*

```
logp(x,[dims])
```

Treat entries in `x` as as unnormalized log probabilities and return normalized log probabilities.

`dims` is an optional argument, if not specified the normalization is over the whole `x`, otherwise the normalization is performed over the given dimensions. In particular, if `x` is a matrix, `dims=1` normalizes columns of `x` and `dims=2` normalizes rows of `x`.

source

6.17 Knet.logsumexp

`Knet.logsumexp` — *Method.*

```
logsumexp(x,[dims])
```

Compute `log(sum(exp(x),dims))` in a numerically stable manner.

`dims` is an optional argument, if not specified the summation is over the whole `x`, otherwise the summation is performed over the given dimensions. In particular if `x` is a matrix, `dims=1` sums columns of `x` and `dims=2` sums rows of `x`.

source

6.18 Knet.mat

`Knet.mat` — *Method.*

```
mat(x)
```

Reshape x into a two-dimensional matrix.

This is typically used when turning the output of a 4-D convolution result into a 2-D input for a fully connected layer. For 1-D inputs returns `reshape(x, (length(x),1))`. For inputs with more than two dimensions of size `(X1,X2,...,XD)`, returns

```
reshape(x, (X1*X2*...*X[D-1],XD))
```

source

6.19 Knet.minibatch

`Knet.minibatch` — *Method.*

```
minibatch(x, y, batchsize; shuffle, partial, xtype, ytype)
```

Return an iterable of minibatches [(xi,yi)...] given data tensors x, y and batchsize. The last dimension of x and y should match and give the number of instances. Keyword arguments:

- `shuffle=false`: Shuffle the instances before minibatching.
- `partial=false`: If true include the last partial minibatch < batchsize.
- `xtype=typeof(x)`: Convert xi in minibatches to this type.
- `ytype=typeof(y)`: Convert yi in minibatches to this type.

source

6.20 Knet.minibatch

`Knet.minibatch` — *Method*.

```
minibatch(x, batchsize; shuffle, partial, xtype, ytype)
```

Return an iterable of minibatches [x1,x2,...] given data tensor x and batchsize. The last dimension of x gives the number of instances. Keyword arguments:

- `shuffle=false`: Shuffle the instances before minibatching.
- `partial=false`: If true include the last partial minibatch < batchsize.
- `xtype=typeof(x)`: Convert xi in minibatches to this type.

source

6.21 Knet.nll

`Knet.nll` — *Method*.

```
nll(model, data, predict; average=true)
```

Compute `nll(predict(model,x), y)` for `(x,y)` in `data` and return the per-instance average (if average=true) or total (if average=false) negative log likelihood.
source

6.22 Knet.nll

`Knet.nll` — *Method*.

```
nll(scores, answers, d=1; average=true)
```

Given an unnormalized `scores` matrix and an `Integer` array of correct `answers`, return the per-instance negative log likelihood. `d=1` means instances are in columns, `d=2` means instances are in rows. Use `average=false` to return the sum instead of per-instance average.
source

6.23 Knet.optimizers

`Knet.optimizers` — *Method*.

```
optimizers(model, otype; options...)
```

Given parameters of a `model`, initialize and return corresponding optimization parameters for a given optimization type `otype` and optimization options `options`. This is useful because each numeric array in model needs its own distinct optimization parameter. `optimizers` makes the creation of optimization parameters that parallel model parameters easy when all of them use the same type and options.

source

6.24 Knet.pool

`Knet.pool` — *Method.*

```
pool(x; kwargs...)
```

Compute pooling of input values (i.e., the maximum or average of several adjacent values) to produce an output with smaller height and/or width.

Currently 4 or 5 dimensional KnetArrays with `Float32` or `Float64` entries are supported. If `x` has dimensions `(X1,X2,...,I,N)`, the result `y` will have dimensions `(Y1,Y2,...,I,N)` where

```
Yi=1+floor((Xi+2*padding[i]-window[i])/stride[i])
```

Here `I` is the number of input channels, `N` is the number of instances, and `Xi,Yi` are spatial dimensions. `window`, `padding` and `stride` are keyword arguments that can be specified as a single number (in which case they apply to all dimensions), or an array/tuple with entries for each spatial dimension.

Keywords:

- `window=2`: the pooling window size for each dimension.
- `padding=0`: the number of extra zeros implicitly concatenated at the start and at the end of each dimension.
- `stride=window`: the number of elements to slide to reach the next pooling window.
- `mode=0`: 0 for max, 1 for average including padded values, 2 for average excluding padded values.
- `maxpoolingNanOpt=0`: Nan numbers are not propagated if 0, they are propagated if 1.
- `alpha=1`: can be used to scale the result.
- `handle`: Handle to a previously created cuDNN context. Defaults to a Knet allocated handle.

source

6.25 Knet.relu

`Knet.relu` — *Function.*

```
relu(x) = max(0,x)
```

source

6.26 Knet.rnnforw

`Knet.rnnforw` — *Method.*

```
rnnforw(r, w, x[, hx, cx]; batchSizes, hy, cy)
```

Returns a tuple (y,hyout,cyout,rs) given rnn `r`, weights `w`, input `x` and optionally the initial hidden and cell states `hx` and `cx` (`cx` is only used in LSTMs). `r` and `w` should come from a previous call to `rnninit`. Both `hx` and `cx` are optional, they are treated as zero arrays if not provided. The output `y` contains the hidden states of the final layer for each time step, `hyout` and `cyout` give the final hidden and cell states for all layers, `rs` is a buffer the RNN needs for its gradient calculation.

The boolean keyword arguments `hy` and `cy` control whether `hyout` and `cyout` will be output. By default `hy = (hx!=nothing)` and `cy = (cx!=nothing && r.mode==2)`, i.e. a hidden state will be output if one is provided as input and for cell state we also require an LSTM. If `hy`/`cy` is `false`, `hyout`/`cyout` will be `nothing`. `batchSizes` can be an integer array that specifies non-uniform batch sizes as explained below. By default `batchSizes=nothing` and the same batch size, `size(x,2)`, is used for all time steps.

The input and output dimensions are:

- `x`: (X,[B,T])
- `y`: (H/2H,[B,T])
- `hx`,`cx`,`hyout`,`cyout`: (H,B,L/2L)
- `batchSizes`: `nothing` or `Vector{Int}(T)`

where X is inputSize, H is hiddenSize, B is batchSize, T is seqLength, L is numLayers. `x` can be 1, 2, or 3 dimensional. If `batchSizes==nothing`, a 1-D `x` represents a single instance, a 2-D `x` represents a single minibatch, and a 3-D `x` represents a sequence of identically sized minibatches. If `batchSizes` is an array of (non-increasing) integers, it gives us the batch size for each time step in the sequence, in which case `sum(batchSizes)` should equal `div(length(x),size(x,1))`. `y` has the same dimensionality as `x`, differing only in its first dimension, which is H if the RNN is unidirectional, 2H if bidirectional. Hidden vectors `hx`, `cx`, `hyout`, `cyout` all have size (H,B1,L) for unidirectional RNNs, and (H,B1,2L) for bidirectional RNNs where B1 is the size of the first minibatch.

source

6.27 Knet.rnninit

`Knet.rnninit` — *Method.*

```
rnninit(inputSize, hiddenSize; opts...)
```

Return an `(r,w)` pair where `r` is a RNN struct and `w` is a single weight array that includes all matrices and biases for the RNN. Keyword arguments:

- `rnnType=:lstm` Type of RNN: One of :relu, :tanh, :lstm, :gru.
- `numLayers=1`: Number of RNN layers.
- `bidirectional=false`: Create a bidirectional RNN if `true`.
- `dropout=0.0`: Dropout probability. Ignored if `numLayers==1`.
- `skipInput=false`: Do not multiply the input with a matrix if `true`.
- `dataType=Float32`: Data type to use for weights.
- `algo=0`: Algorithm to use, see CUDNN docs for details.
- `seed=0`: Random number seed. Uses `time()` if 0.
- `winit=xavier`: Weight initialization method for matrices.
- `binit=zeros`: Weight initialization method for bias vectors.
- 'usegpu=(gpu()>=0): GPU used by default if one exists.

RNNs compute the output h[t] for a given iteration from the recurrent input h[t-1] and the previous layer input x[t] given matrices W, R and biases bW, bR from the following equations:

`:relu` and `:tanh`: Single gate RNN with activation function f:

```
h[t] = f(W * x[t] .+ R * h[t-1] .+ bW .+ bR)
```

`:gru`: Gated recurrent unit:

```
i[t] = sigm(Wi * x[t] .+ Ri * h[t-1] .+ bWi .+ bRi) # input gate
r[t] = sigm(Wr * x[t] .+ Rr * h[t-1] .+ bWr .+ bRr) # reset gate
n[t] = tanh(Wn * x[t] .+ r[t] .* (Rn * h[t-1] .+ bRn) .+ bWn) # new gate
h[t] = (1 - i[t]) .* n[t] .+ i[t] .* h[t-1]
```

`:lstm`: Long short term memory unit with no peephole connections:

```
i[t] = sigm(Wi * x[t] .+ Ri * h[t-1] .+ bWi .+ bRi) # input gate
f[t] = sigm(Wf * x[t] .+ Rf * h[t-1] .+ bWf .+ bRf) # forget gate
o[t] = sigm(Wo * x[t] .+ Ro * h[t-1] .+ bWo .+ bRo) # output gate
n[t] = tanh(Wn * x[t] .+ Rn * h[t-1] .+ bWn .+ bRn) # new gate
c[t] = f[t] .* c[t-1] .+ i[t] .* n[t]               # cell output
h[t] = o[t] .* tanh(c[t])
```

source

6.28 Knet.rnnparam

`Knet.rnnparam` — *Method*.

```
rnnparam{T}(r::RNN, w::KnetArray{T}, layer, id, param)
```

Return a single weight matrix or bias vector as a slice of w.
Valid `layer` values:

- For unidirectional RNNs 1:numLayers
- For bidirectional RNNs 1:2*numLayers, forw and back layers alternate.

Valid `id` values:

- For RELU and TANH RNNs, input = 1, hidden = 2.
- For GRU reset = 1,4; update = 2,5; newmem = 3,6; 1:3 for input, 4:6 for hidden
- For LSTM inputgate = 1,5; forget = 2,6; newmem = 3,7; output = 4,8; 1:4 for input, 5:8 for hidden

Valid `param` values:

- Return the weight matrix (transposed!) if `param==1`.
- Return the bias vector if `param==2`.

The effect of skipInput: Let I=1 for RELU/TANH, 1:3 for GRU, 1:4 for LSTM

- For skipInput=false (default), rnnparam(r,w,1,I,1) is a (inputSize,hiddenSize) matrix.
- For skipInput=true, rnnparam(r,w,1,I,1) is `nothing`.
- For bidirectional, the same applies to rnnparam(r,w,2,I,1): the first back layer.

source

6.29 Knet.rnnparams

`Knet.rnnparams` — *Method*.

```
rnnparams(r::RNN, w)
```

Split w into individual parameters and return them as an array.
The order of params returned (subject to change):

- All weight matrices come before all bias vectors.
- Matrices and biases are sorted lexically based on (layer,id).
- See @doc rnnparam for valid layer and id values.
- Input multiplying matrices are `nothing` if r.inputMode = 1.

source

6.30 Knet.setseed

`Knet.setseed` — *Method*.

```
setseed(n::Integer)
```

Run srand(n) on both cpu and gpu.
source

6.31 Knet.sigm

`Knet.sigm` — *Function*.

```
sigm(x) = (1./(1+exp(-x)))
```

source

6.32 Knet.unpool

`Knet.unpool` — *Method*.

Unpooling; `reverse` of pooling.

```
x == pool(unpool(x;o...); o...)
```

source

6.33 Knet.update!

`Knet.update!` — *Function*.

```
update!(weights, gradients, params)
update!(weights, gradients; lr=0.001, gclip=0)
```

Update the `weights` using their `gradients` and the optimization algorithm parameters specified by `params`. The 2-arg version defaults to the Sgd algorithm with learning rate `lr` and gradient clip `gclip`. `gclip==0` indicates no clipping. The `weights` and possibly `gradients` and `params` are modified in-place.

`weights` can be an individual numeric array or a collection of arrays represented by an iterator or dictionary. In the individual case, `gradients` should be a similar numeric array of `size(weights)` and `params` should be a single object. In the collection case, each individual weight array should have a corresponding params object. This way different weight arrays can have their own optimization state, different learning rates, or even different optimization algorithms running in parallel. In the iterator case, `gradients` and `params` should be iterators of the same length as `weights` with corresponding elements. In the dictionary case, `gradients` and `params` should be dictionaries with the same keys as `weights`.

Individual optimization parameters can be one of the following types. The keyword arguments for each type's constructor and their default values are listed as well.

- `Sgd(;lr=0.001, gclip=0)`
- `Momentum(;lr=0.001, gclip=0, gamma=0.9)`
- `Nesterov(;lr=0.001, gclip=0, gamma=0.9)`
- `Rmsprop(;lr=0.001, gclip=0, rho=0.9, eps=1e-6)`
- `Adagrad(;lr=0.1, gclip=0, eps=1e-6)`
- `Adadelta(;lr=0.01, gclip=0, rho=0.9, eps=1e-6)`
- `Adam(;lr=0.001, gclip=0, beta1=0.9, beta2=0.999, eps=1e-8)`

Example:

```
w = rand(d)                 # an individual weight array
g = lossgradient(w)         # gradient g has the same shape as w
update!(w, g)               # update w in-place with Sgd()
update!(w, g; lr=0.1)       # update w in-place with Sgd(lr=0.1)
update!(w, g, Sgd(lr=0.1))  # update w in-place with Sgd(lr=0.1)

w = (rand(d1), rand(d2))    # a tuple of weight arrays
g = lossgradient2(w)        # g will also be a tuple
p = (Adam(), Sgd())         # p has params for each w[i]
update!(w, g, p)            # update each w[i] in-place with g[i],p[i]

w = Any[rand(d1), rand(d2)] # any iterator can be used
g = lossgradient3(w)        # g will be similar to w
p = Any[Adam(), Sgd()]      # p should be an iterator of same length
update!(w, g, p)            # update each w[i] in-place with g[i],p[i]

w = Dict(:a => rand(d1), :b => rand(d2)) # dictionaries can be used
g = lossgradient4(w)
p = Dict(:a => Adam(), :b => Sgd())
update!(w, g, p)
```

source

6.34 Knet.xavier

`Knet.xavier` — *Method*.

```
xavier(a...)
```

Xavier initialization. The `a` arguments are passed to `rand`. See (Glorot and Bengio 2010) for a description. Caffe implements this slightly differently. Lasagne calls it `GlorotUniform`.

source

Chapter 7

DataFrames

7.1 DataFrames.aggregate

`DataFrames.aggregate` — *Method*.

Split-apply-combine that applies a set of functions over columns of an AbstractDataFrame or GroupedDataFrame

[] aggregate(d::AbstractDataFrame, cols, fs) aggregate(gd::GroupedDataFrame, fs)

Arguments

- `d` : an AbstractDataFrame
- `gd` : a GroupedDataFrame
- `cols` : a column indicator (Symbol, Int, Vector{Symbol}, etc.)
- `fs` : a function or vector of functions to be applied to vectors within groups; expects each argument to be a column vector

Each `fs` should return a value or vector. All returns must be the same length.

Returns

- `::DataFrame`

Examples

[] df = DataFrame(a = repeat([1, 2, 3, 4], outer=[2]), b = repeat([2, 1], outer=[4]), c = randn(8)) aggregate(df, :a, sum) aggregate(df, :a, [sum, mean]) aggregate(groupby(df, :a), [sum, mean]) df —¿ groupby(:a) —¿ [sum, mean] equivalent

source

7.2 DataFrames.by

`DataFrames.by` — *Method.*

Split-apply-combine in one step; apply `f` to each grouping in `d` based on columns `col`

[] by(d::AbstractDataFrame, cols, f::Function) by(f::Function, d::AbstractDataFrame, cols)

Arguments

- `d` : an AbstractDataFrame
- `cols` : a column indicator (Symbol, Int, Vector{Symbol}, etc.)
- `f` : a function to be applied to groups; expects each argument to be an AbstractDataFrame

`f` can return a value, a vector, or a DataFrame. For a value or vector, these are merged into a column along with the `cols` keys. For a DataFrame, `cols` are combined along columns with the resulting DataFrame. Returning a DataFrame is the clearest because it allows column labeling.

A method is defined with `f` as the first argument, so do-block notation can be used.

`by(d, cols, f)` is equivalent to `combine(map(f, groupby(d, cols)))`.

Returns

- `::DataFrame`

Examples

[] df = DataFrame(a = repeat([1, 2, 3, 4], outer=[2]), b = repeat([2, 1], outer=[4]), c = randn(8)) by(df, :a, d -¿ sum(d[:c])) by(df, :a, d -¿ 2 * d[:c]) by(df, :a, d -¿ DataFrame(c$_s$$um$ = $sum(d[: c]), c_m ean$ = $mean(d[: c]))) by(df, : a, d- > DataFrame(c = d[: c], c_m ean = mean(d[: c]))) by(df, [: a, : b]) dodDataFrame(m = m$

source

7.3 DataFrames.coefnames

`DataFrames.coefnames` — *Method.*

`coefnames(mf::ModelFrame)`

Returns a vector of coefficient names constructed from the Terms member and the types of the evaluation columns.

source

7.4 DataFrames.colwise

`DataFrames.colwise` — *Method*.

Apply a function to each column in an AbstractDataFrame or GroupedDataFrame

[] colwise(f::Function, d) colwise(d)

Arguments

- `f` : a function or vector of functions
- `d` : an AbstractDataFrame of GroupedDataFrame

If `d` is not provided, a curried version of groupby is given.

Returns

- various, depending on the call

Examples

[] df = DataFrame(a = repeat([1, 2, 3, 4], outer=[2]), b = repeat([2, 1], outer=[4]), c = randn(8)) colwise(sum, df) colwise(sum, groupby(df, :a))

source

7.5 DataFrames.combine

`DataFrames.combine` — *Method*.

Combine a GroupApplied object (rudimentary)

[] combine(ga::GroupApplied)

Arguments

- `ga` : a GroupApplied

Returns

- `::DataFrame`

Examples

[] df = DataFrame(a = repeat([1, 2, 3, 4], outer=[2]), b = repeat([2, 1], outer=[4]), c = randn(8)) combine(map(d -¿ mean(d[:c]), gd))

source

7.6 DataFrames.completecases!

`DataFrames.completecases!` — *Method*.

Delete rows with NA's.

[] completecases!(df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame

Result

- `::AbstractDataFrame` : the updated version

See also `completecases`.

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) df[[1,4,5], :x] = NA df[[9,10], :y] = NA completecases!(df)

source

7.7 DataFrames.completecases

`DataFrames.completecases` — *Method.*

Indexes of complete cases (rows without NA's)

[] completecases(df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame

Result

- `::Vector{Bool}` : indexes of complete cases

See also `completecases!`.

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) df[[1,4,5], :x] = NA df[[9,10], :y] = NA completecases(df)

source

7.8 DataFrames.eltypes

`DataFrames.eltypes` — *Method.*

Column elemental types

[] eltypes(df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame

Result

- `::Vector{Type}` : the elemental type of each column

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) eltypes(df)

source

7.9 DataFrames.groupby

`DataFrames.groupby` — *Method.*

A view of an AbstractDataFrame split into row groups

[] groupby(d::AbstractDataFrame, cols) groupby(cols)

Arguments

- `d` : an AbstractDataFrame
- `cols` : an

If `d` is not provided, a curried version of groupby is given.

Returns

- `::GroupedDataFrame` : a grouped view into `d`

Details

An iterator over a `GroupedDataFrame` returns a `SubDataFrame` view for each grouping into `d`. A `GroupedDataFrame` also supports indexing by groups and `map`.

See the following for additional split-apply-combine operations:

- `by` : split-apply-combine using functions
- `aggregate` : split-apply-combine; applies functions in the form of a cross product
- `combine` : combine (obviously)
- `colwise` : apply a function to each column in an AbstractDataFrame or GroupedDataFrame

Piping methods |> are also provided.

See the DataFramesMeta package for more operations on GroupedDataFrames.

Examples

[] df = DataFrame(a = repeat([1, 2, 3, 4], outer=[2]), b = repeat([2, 1], outer=[4]), c = randn(8)) gd = groupby(df, :a) gd[1] last(gd) vcat([g[:b] for g in gd]...) for g in gd println(g) end map(d -¿ mean(d[:c]), gd) returns a GroupApplied object combine(map(d -¿ mean(d[:c]), gd)) df —¿ groupby(:a) —¿ [sum, length] df —¿ groupby([:a, :b]) —¿ [sum, length]

source

7.10 DataFrames.head

`DataFrames.head` — *Function.*

Show the first or last part of an AbstractDataFrame

[] head(df::AbstractDataFrame, r::Int = 6) tail(df::AbstractDataFrame, r::Int = 6)

Arguments

- `df` : the AbstractDataFrame
- `r` : the number of rows to show

Result

- `::AbstractDataFrame` : the first or last part of `df`

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) head(df) tail(df)

source

7.11 DataFrames.melt

`DataFrames.melt` — *Method.*

Stacks a DataFrame; convert from a wide to long format; see `stack`.

source

7.12 DataFrames.meltdf

`DataFrames.meltdf` — *Method.*

A stacked view of a DataFrame (long format); see `stackdf`

source

7.13 DataFrames.names!

`DataFrames.names!` — *Method.*

Set column names

[] names!(df::AbstractDataFrame, vals)

Arguments

- `df` : the AbstractDataFrame
- `vals` : column names, normally a Vector{Symbol} the same length as the number of columns in `df`
- `allow_duplicates` : if `false` (the default), an error will be raised if duplicate names are found; if `true`, duplicate names will be suffixed with `_i` (`i` starting at 1 for the first duplicate).

Result

- `::AbstractDataFrame` : the updated result

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) names!(df, [:a, :b, :c]) names!(df, [:a, :b, :a]) throws ArgumentError names!(df, [:a, :b, :a], allow$_d uplicates = true) renamessecond : ato : a_1$

source

7.14 DataFrames.nonunique

`DataFrames.nonunique` — *Method*.

Indexes of complete cases (rows without NA's)
[] nonunique(df::AbstractDataFrame) nonunique(df::AbstractDataFrame, cols)

Arguments

- `df` : the AbstractDataFrame
- `cols` : a column indicator (Symbol, Int, Vector{Symbol}, etc.) specifying the column(s) to compare

Result

- `::Vector{Bool}` : indicates whether the row is a duplicate of some prior row

See also `unique` and `unique!`.
Examples
[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) df = vcat(df, df) nonunique(df) nonunique(df, 1)
source

7.15 DataFrames.readtable

`DataFrames.readtable` — *Method*.

Read data from a tabular-file format (CSV, TSV, . . .)
[] readtable(filename, [keyword options])
Arguments

- `filename::AbstractString` : the filename to be read

Keyword Arguments

- `header::Bool` – Use the information from the file's header line to determine column names. Defaults to `true`.
- `separator::Char` – Assume that fields are split by the `separator` character. If not specified, it will be guessed from the filename: `.csv` defaults to `,`, `.tsv` defaults to `	`, `.wsv` defaults to ` `.
- `quotemark::Vector{Char}` – Assume that fields contained inside of two `quotemark` characters are quoted, which disables processing of separators and linebreaks. Set to `Char[]` to disable this feature and slightly improve performance. Defaults to `["]`.
- `decimal::Char` – Assume that the decimal place in numbers is written using the `decimal` character. Defaults to `.`.

- `nastrings::Vector{String}` – Translate any of the strings into this vector into an `NA`. Defaults to `["", "NA"]`.
- `truestrings::Vector{String}` – Translate any of the strings into this vector into a Boolean `true`. Defaults to `["T", "t", "TRUE", "true"]`.
- `falsestrings::Vector{String}` – Translate any of the strings into this vector into a Boolean `false`. Defaults to `["F", "f", "FALSE", "false"]`.
- `makefactors::Bool` – Convert string columns into `PooledDataVector`'s for use as factors. Defaults to `false`.
- `nrows::Int` – Read only `nrows` from the file. Defaults to `-1`, which indicates that the entire file should be read.
- `names::Vector{Symbol}` – Use the values in this array as the names for all columns instead of or in lieu of the names in the file's header. Defaults to `[]`, which indicates that the header should be used if present or that numeric names should be invented if there is no header.
- `eltypes::Vector` – Specify the types of all columns. Defaults to `[]`.
- `allowcomments::Bool` – Ignore all text inside comments. Defaults to `false`.
- `commentmark::Char` – Specify the character that starts comments. Defaults to `#`.
- `ignorepadding::Bool` – Ignore all whitespace on left and right sides of a field. Defaults to `true`.
- `skipstart::Int` – Specify the number of initial rows to skip. Defaults to `0`.
- `skiprows::Vector{Int}` – Specify the indices of lines in the input to ignore. Defaults to `[]`.
- `skipblanks::Bool` – Skip any blank lines in input. Defaults to `true`.
- `encoding::Symbol` – Specify the file's encoding as either `:utf8` or `:latin1`. Defaults to `:utf8`.
- `normalizenames::Bool` – Ensure that column names are valid Julia identifiers. For instance this renames a column named `"a b"` to `"a_b"` which can then be accessed with `:a_b` instead of `Symbol("a b")`. Defaults to `true`.

Result

- `::DataFrame`

Examples

[] df = readtable("data.csv") df = readtable("data.tsv") df = readtable("data.wsv") df = readtable("data.txt", separator = ' ') df = readtable("data.txt", header = false)

source

7.16 DataFrames.rename

`DataFrames.rename` — *Function.*

Rename columns

[] rename!(df::AbstractDataFrame, from::Symbol, to::Symbol) rename!(df::AbstractDataFrame, d::Associative) rename!(f::Function, df::AbstractDataFrame) rename(df::AbstractDataFrame, from::Symbol, to::Symbol) rename(f::Function, df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame
- `d` : an Associative type that maps the original name to a new name
- `f` : a function that has the old column name (a symbol) as input and new column name (a symbol) as output

Result

- `::AbstractDataFrame` : the updated result

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) rename(x -¿ Symbol(uppercase(string(x))), df) rename(df, Dict(:i=¿:A, :x=¿:X)) rename(df, :y, :Y) rename!(df, Dict(:i=¿:A, :x=¿:X))

source

7.17 DataFrames.rename!

`DataFrames.rename!` — *Function.*

Rename columns

[] rename!(df::AbstractDataFrame, from::Symbol, to::Symbol) rename!(df::AbstractDataFrame, d::Associative) rename!(f::Function, df::AbstractDataFrame) rename(df::AbstractDataFrame, from::Symbol, to::Symbol) rename(f::Function, df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame
- `d` : an Associative type that maps the original name to a new name
- `f` : a function that has the old column name (a symbol) as input and new column name (a symbol) as output

Result

- `::AbstractDataFrame` : the updated result

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) rename(x -¿ Symbol(uppercase(string(x))), df) rename(df, Dict(:i=¿:A, :x=¿:X)) rename(df, :y, :Y) rename!(df, Dict(:i=¿:A, :x=¿:X))

source

7.18 DataFrames.stack

`DataFrames.stack` — *Method.*

Stacks a DataFrame; convert from a wide to long format

[] stack(df::AbstractDataFrame, measure$_v$$ars$, $id_v ars$)$stack(df :: AbstractDataFrame, measure_v ars)$

Arguments

- `df` : the AbstractDataFrame to be stacked
- `measure_vars` : the columns to be stacked (the measurement variables), a normal column indexing type, like a Symbol, Vector{Symbol}, Int, etc.; for `melt`, defaults to all variables that are not `id_vars`
- `id_vars` : the identifier columns that are repeated during stacking, a normal column indexing type; for `stack` defaults to all variables that are not `measure_vars`

If neither `measure_vars` or `id_vars` are given, `measure_vars` defaults to all floating point columns.

Result

- `::DataFrame` : the long-format dataframe with column `:value` holding the values of the stacked columns (`measure_vars`), with column `:variable` a Vector of Symbols with the `measure_vars` name, and with columns for each of the `id_vars`.

See also `stackdf` and `meltdf` for stacking methods that return a view into the original DataFrame. See `unstack` for converting from long to wide format.

Examples

[] d1 = DataFrame(a = repeat([1:3;], inner = [4]), b = repeat([1:4;], inner = [3]), c = randn(12), d = randn(12), e = map(string, 'a':'l'))

d1s = stack(d1, [:c, :d]) d1s2 = stack(d1, [:c, :d], [:a]) d1m = melt(d1, [:a, :b, :e])

source

7.19 DataFrames.stackdf

`DataFrames.stackdf` — *Method.*

A stacked view of a DataFrame (long format)

Like `stack` and `melt`, but a view is returned rather than data copies.

[] stackdf(df::AbstractDataFrame, measure$_v ars$, $id_v ars$)$stackdf(df :: AbstractDataFrame, measure_v ars)meltdf($

Arguments

- `df` : the wide AbstractDataFrame
- `measure_vars` : the columns to be stacked (the measurement variables), a normal column indexing type, like a Symbol, Vector{Symbol}, Int, etc.; for `melt`, defaults to all variables that are not `id_vars`
- `id_vars` : the identifier columns that are repeated during stacking, a normal column indexing type; for `stack` defaults to all variables that are not `measure_vars`

Result

- `::DataFrame` : the long-format dataframe with column `:value` holding the values of the stacked columns (`measure_vars`), with column `:variable` a Vector of Symbols with the `measure_vars` name, and with columns for each of the `id_vars`.

The result is a view because the columns are special AbstractVectors that return indexed views into the original DataFrame.

Examples

[] d1 = DataFrame(a = repeat([1:3;], inner = [4]), b = repeat([1:4;], inner = [3]), c = randn(12), d = randn(12), e = map(string, 'a':'l'))

d1s = stackdf(d1, [:c, :d]) d1s2 = stackdf(d1, [:c, :d], [:a]) d1m = meltdf(d1, [:a, :b, :e])

source

7.20 DataFrames.tail

`DataFrames.tail` — *Function.*

Show the first or last part of an AbstractDataFrame

[] head(df::AbstractDataFrame, r::Int = 6) tail(df::AbstractDataFrame, r::Int = 6)

Arguments

- `df` : the AbstractDataFrame
- `r` : the number of rows to show

Result

- `::AbstractDataFrame` : the first or last part of `df`

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) head(df) tail(df)

source

7.21 DataFrames.unique!

`DataFrames.unique!` — *Function.*

Delete duplicate rows

[] unique(df::AbstractDataFrame) unique(df::AbstractDataFrame, cols) unique!(df::AbstractDataFran unique!(df::AbstractDataFrame, cols)

Arguments

- `df` : the AbstractDataFrame

- `cols` : column indicator (Symbol, Int, Vector{Symbol}, etc.)

specifying the column(s) to compare.

Result

- `::AbstractDataFrame` : the updated version of `df` with unique rows.

When `cols` is specified, the return DataFrame contains complete rows, retaining in each case the first instance for which `df[cols]` is unique.

See also `nonunique`.

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) df = vcat(df, df) unique(df) doesn't modify df unique(df, 1) unique!(df) modifies df

source

7.22 DataFrames.unstack

`DataFrames.unstack` — *Method.*

Unstacks a DataFrame; convert from a long to wide format

[] unstack(df::AbstractDataFrame, rowkey, colkey, value) unstack(df::AbstractDataFrame, colkey, value) unstack(df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame to be unstacked

- `rowkey` : the column with a unique key for each row, if not given, find a key by grouping on anything not a `colkey` or `value`

- `colkey` : the column holding the column names in wide format, defaults to `:variable`
- `value` : the value column, defaults to `:value`

Result

- `::DataFrame` : the wide-format dataframe

Examples

[] wide = DataFrame(id = 1:12, a = repeat([1:3;], inner = [4]), b = repeat([1:4;], inner = [3]), c = randn(12), d = randn(12))

long = stack(wide) wide0 = unstack(long) wide1 = unstack(long, :variable, :value) wide2 = unstack(long, :id, :variable, :value)

Note that there are some differences between the widened results above.

source

7.23 DataFrames.writetable

`DataFrames.writetable` — *Method*.

Write data to a tabular-file format (CSV, TSV, . . .)

[] writetable(filename, df, [keyword options])

Arguments

- `filename::AbstractString` : the filename to be created
- `df::AbstractDataFrame` : the AbstractDataFrame to be written

Keyword Arguments

- `separator::Char` – The separator character that you would like to use. Defaults to the output of `getseparator(filename)`, which uses commas for files that end in `.csv`, tabs for files that end in `.tsv` and a single space for files that end in `.wsv`.
- `quotemark::Char` – The character used to delimit string fields. Defaults to `"`.
- `header::Bool` – Should the file contain a header that specifies the column names from `df`. Defaults to `true`.
- `nastring::AbstractString` – What to write in place of missing data. Defaults to `"NA"`.

Result

- `::DataFrame`

Examples

[] df = DataFrame(A = 1:10) writetable("output.csv", df) writetable("output.dat", df, separator = ',', header = false) writetable("output.dat", df, quotemark = '', separator = ',') writetable("output.dat", df, header = false)

source

7.24 StatsBase.describe

`StatsBase.describe` — *Method*.

Summarize the columns of an AbstractDataFrame

[] describe(df::AbstractDataFrame) describe(io, df::AbstractDataFrame)

Arguments

- `df` : the AbstractDataFrame
- `io` : optional output descriptor

Result

- nothing

Details

If the column's base type derives from Number, compute the minimum, first quantile, median, mean, third quantile, and maximum. NA's are filtered and reported separately.

For boolean columns, report trues, falses, and NAs.

For other types, show column characteristics and number of NAs.

Examples

[] df = DataFrame(i = 1:10, x = rand(10), y = rand(["a", "b", "c"], 10)) describe(df)

source

7.25 StatsBase.model_response

`StatsBase.model_response` — *Method*.

```
StatsBase.model_response(mf::ModelFrame)
```

Extract the response column, if present. `DataVector` or `PooledDataVector` columns are converted to `Array`s

source

Chapter 8

DataStructures

8.1 Base.pop!

`Base.pop!` — *Method.*

```
pop!(sc, k)
```

Deletes the item with key `k` in SortedDict or SortedSet `sc` and returns the value that was associated with `k` in the case of SortedDict or `k` itself in the case of SortedSet. A `KeyError` results if `k` is not in `sc`. Time: O(c log n)

source

8.2 Base.pop!

`Base.pop!` — *Method.*

```
pop!(sc, k)
```

Deletes the item with key `k` in SortedDict or SortedSet `sc` and returns the value that was associated with `k` in the case of SortedDict or `k` itself in the case of SortedSet. A `KeyError` results if `k` is not in `sc`. Time: O(c log n)

source

8.3 Base.pop!

`Base.pop!` — *Method.*

```
pop!(ss)
```

Deletes the item with first key in SortedSet `ss` and returns the key. A `BoundsError` results if `ss` is empty. Time: O(c log n)

source

8.4 Base.push!

`Base.push!` — *Method.*

```
push!(sc, k)
```

Argument `sc` is a SortedSet and `k` is a key. This inserts the key into the container. If the key is already present, this overwrites the old value. (This is not necessarily a no-op; see below for remarks about the customizing the sort order.) The return value is `sc`. Time: O(c log n)

source

8.5 Base.push!

`Base.push!` — *Method.*

```
push!(sc, k=>v)
```

Argument `sc` is a SortedDict or SortedMultiDict and `k=>v` is a key-value pair. This inserts the key-value pair into the container. If the key is already present, this overwrites the old value. The return value is `sc`. Time: O(c log n)

source

8.6 Base.push!

`Base.push!` — *Method.*

```
push!(sc, k=>v)
```

Argument `sc` is a SortedDict or SortedMultiDict and `k=>v` is a key-value pair. This inserts the key-value pair into the container. If the key is already present, this overwrites the old value. The return value is `sc`. Time: O(c log n)

source

8.7 DataStructures.back

`DataStructures.back` — *Method.*

```
back(q::Deque)
```

Returns the last element of the deque `q`.

source

8.8 DataStructures.compare

`DataStructures.compare` — *Method*.

```
compare(m::SAContainer, s::IntSemiToken, t::IntSemiToken)
```

Determines the relative positions of the data items indexed by `(m,s)` and `(m,t)` in the sorted order. The return value is `-1` if `(m,s)` precedes `(m,t)`, `0` if they are equal, and `1` if `(m,s)` succeeds `(m,t)`. `s` and `t` are semitokens for the same container `m`.

source

8.9 DataStructures.counter

`DataStructures.counter` — *Method*.

```
counter(seq)
```

Returns an `Accumulator` object containing the elements from `seq`.

source

8.10 DataStructures.dec!

`DataStructures.dec!` — *Method*.

```
dec!(ct, x, [v=1])
```

Decrements the count for `x` by `v` (defaulting to one)

source

8.11 DataStructures.deque

`DataStructures.deque` — *Method*.

```
deque(T)
```

Create a deque of type `T`.

source

8.12 DataStructures.dequeue!

`DataStructures.dequeue!` — *Method*.

```
dequeue!(pq)
```

Remove and return the lowest priority key from a priority queue.

```
julia> a = PriorityQueue(["a","b","c"],[2,3,1],Base.Order.Forward)
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 3 entries:
  "c" => 1
  "b" => 3
  "a" => 2

julia> dequeue!(a)
"c"

julia> a
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 2 entries:
  "b" => 3
  "a" => 2
```

source

8.13 DataStructures.dequeue!

`DataStructures.dequeue!` — *Method.*

`dequeue!(s::Queue)`

Removes an element from the front of the queue `s` and returns it.
source

8.14 DataStructures.dequeue_pair!

`DataStructures.dequeue_pair!` — *Method.*

`dequeue_pair!(pq)`

Remove and return a the lowest priority key and value from a priority queue as a pair.

```
julia> a = PriorityQueue(["a","b","c"],[2,3,1],Base.Order.Forward)
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 3 entries:
  "c" => 1
  "b" => 3
  "a" => 2

julia> dequeue_pair!(a)
"c" => 1

julia> a
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 2 entries:
  "b" => 3
  "a" => 2
```

source

8.15 DataStructures.enqueue!

`DataStructures.enqueue!` — *Method.*

enqueue!(pq, k, v)

Insert the a key `k` into a priority queue `pq` with priority `v`.

source

8.16 DataStructures.enqueue!

`DataStructures.enqueue!` — *Method.*

```
enqueue!(s::Queue, x)
```

Inserts the value `x` to the end of the queue `s`.

source

8.17 DataStructures.enqueue!

`DataStructures.enqueue!` — *Method.*

```
enqueue!(pq, k=>v)
```

Insert the a key `k` into a priority queue `pq` with priority `v`.

```
julia> a = PriorityQueue(PriorityQueue("a"=>1, "b"=>2, "c"=>3))
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1

julia> enqueue!(a, "d"=>4)
PriorityQueue{String,Int64,Base.Order.ForwardOrdering} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

source

8.18 DataStructures.find_root

`DataStructures.find_root` — *Method.*

```
find_root{T}(s::DisjointSets{T}, x::T)
```

Finds the root element of the subset in `s` which has the element `x` as a member.

source

8.19 DataStructures.front

`DataStructures.front` — *Method.*

```
front(q::Deque)
```

Returns the first element of the deque `q`.
source

8.20 DataStructures.heapify

`DataStructures.heapify` — *Function.*

```
heapify(v, ord::Ordering=Forward)
```

Returns a new vector in binary heap order, optionally using the given ordering.

```
julia> a = [1,3,4,5,2];

julia> heapify(a)
5-element Array{Int64,1}:
 1
 2
 4
 5
 3

julia> heapify(a, Base.Order.Reverse)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2
```

source

8.21 DataStructures.heapify!

`DataStructures.heapify!` — *Function.*

```
heapify!(v, ord::Ordering=Forward)
```

In-place `heapify`.
source

8.22 DataStructures.heappop!

`DataStructures.heappop!` — *Function.*

```
heappop!(v, [ord])
```

Given a binary heap-ordered array, remove and return the lowest ordered element. For efficiency, this function does not check that the array is indeed heap-ordered.

source

8.23 DataStructures.heappush!

`DataStructures.heappush!` — *Function.*

```
heappush!(v, x, [ord])
```

Given a binary heap-ordered array, push a new element `x`, preserving the heap property. For efficiency, this function does not check that the array is indeed heap-ordered.

source

8.24 DataStructures.in_same_set

`DataStructures.in_same_set` — *Method.*

```
in_same_set(s::IntDisjointSets, x::Integer, y::Integer)
```

Returns `true` if `x` and `y` belong to the same subset in `s` and `false` otherwise.

source

8.25 DataStructures.inc!

`DataStructures.inc!` — *Method.*

```
inc!(ct, x, [v=1])
```

Increments the count for `x` by `v` (defaulting to one)

source

8.26 DataStructures.isheap

`DataStructures.isheap` — *Function*.

```
isheap(v, ord::Ordering=Forward)
```

Return `true` if an array is heap-ordered according to the given order.

```
julia> a = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> isheap(a,Base.Order.Forward)
true

julia> isheap(a,Base.Order.Reverse)
false
```

source

8.27 DataStructures.nlargest

`DataStructures.nlargest` — *Method*.

Returns the `n` largest elements of `arr`.
Equivalent to `sort(arr, lt = >)[1:min(n, end)]`
source

8.28 DataStructures.nsmallest

`DataStructures.nsmallest` — *Method*.

Returns the `n` smallest elements of `arr`.
Equivalent to `sort(arr, lt = <)[1:min(n, end)]`
source

8.29 DataStructures.orderobject

`DataStructures.orderobject` — *Method*.

```
orderobject(sc)
```

Returns the order object used to construct the container. Time: O(1)
source

8.30 DataStructures.orderobject

`DataStructures.orderobject` — *Method*.

`orderobject(sc)`

Returns the order object used to construct the container. Time: O(1)
source

8.31 DataStructures.orderobject

`DataStructures.orderobject` — *Method*.

`orderobject(sc)`

Returns the order object used to construct the container. Time: O(1)
source

8.32 DataStructures.ordtype

`DataStructures.ordtype` — *Method*.

`ordtype(sc)`

Returns the order type for SortedDict, SortedMultiDict and SortedSet. This function may also be applied to the type itself. Time: O(1)
source

8.33 DataStructures.ordtype

`DataStructures.ordtype` — *Method*.

`ordtype(sc)`

Returns the order type for SortedDict, SortedMultiDict and SortedSet. This function may also be applied to the type itself. Time: O(1)
source

8.34 DataStructures.ordtype

`DataStructures.ordtype` — *Method*.

`ordtype(sc)`

Returns the order type for SortedDict, SortedMultiDict and SortedSet. This function may also be applied to the type itself. Time: O(1)
source

8.35 DataStructures.packcopy

`DataStructures.packcopy` — *Method.*

```
packcopy(sc)
```

This returns a copy of `sc` in which the data is packed. When deletions take place, the previously allocated memory is not returned. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.36 DataStructures.packcopy

`DataStructures.packcopy` — *Method.*

```
packcopy(sc)
```

This returns a copy of `sc` in which the data is packed. When deletions take place, the previously allocated memory is not returned. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.37 DataStructures.packcopy

`DataStructures.packcopy` — *Method.*

```
packcopy(sc)
```

This returns a copy of `sc` in which the data is packed. When deletions take place, the previously allocated memory is not returned. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.38 DataStructures.packdeepcopy

`DataStructures.packdeepcopy` — *Method.*

```
packdeepcopy(sc)
```

This returns a packed copy of `sc` in which the keys and values are deepcopied. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.39 DataStructures.packdeepcopy

`DataStructures.packdeepcopy` — *Method*.

```
packdeepcopy(sc)
```

This returns a packed copy of `sc` in which the keys and values are deepcopied. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.40 DataStructures.packdeepcopy

`DataStructures.packdeepcopy` — *Method*.

```
packdeepcopy(sc)
```

This returns a packed copy of `sc` in which the keys and values are deepcopied. This function can be used to reclaim memory after many deletions. Time: O(cn log n)

source

8.41 DataStructures.peek

`DataStructures.peek` — *Method*.

```
peek(pq)
```

Return the lowest priority key from a priority queue without removing that key from the queue.

source

8.42 DataStructures.reset!

`DataStructures.reset!` — *Method*.

```
reset!(ct::Accumulator, x)
```

Resets the count of `x` to zero. Returns its former count.

source

8.43 DataStructures.top

`DataStructures.top` — *Method*.

```
top(h::BinaryHeap)
```

Returns the element at the top of the heap `h`.

source

8.44 DataStructures.top_with_handle

`DataStructures.top_with_handle` — *Method*.

```
top_with_handle(h::MutableBinaryHeap)
```

Returns the element at the top of the heap `h` and its handle.
source

8.45 DataStructures.update!

`DataStructures.update!` — *Method*.

```
update!{T}(h::MutableBinaryHeap{T}, i::Int, v::T)
```

Replace the element at index `i` in heap `h` with `v`. This is equivalent to `h[i]=v`.
source

Chapter 9

JDBC

9.1 Base.close

`Base.close` — *Method*.

Closes the JDBCConnection `conn`. Throws a `JDBCError` if connection is null.

Returns `nothing`.

source

9.2 Base.close

`Base.close` — *Method*.

Close the JDBCCursor `csr`. Throws a `JDBCError` if cursor is not initialized.

Returns `nothing`.

source

9.3 Base.connect

`Base.connect` — *Method*.

Open a JDBC Connection to the specified `host`. The username and password can be optionally passed as a Dictionary `props` of the form `Dict("user" => "username", "passwd" => "password")`. The JDBC connector location can be optionally passed as `connectorpath`, if it is not added to the java class path.

Returns a `JDBCConnection` instance.

source

9.4 Base.isopen

`Base.isopen` — *Method*.

Returns a boolean indicating whether connection `conn` is open.

source

9.5 DBAPI.DBAPIBase.connection

`DBAPI.DBAPIBase.connection` — *Method*.

Return the corresponding connection for a given cursor.

source

9.6 DBAPI.DBAPIBase.cursor

`DBAPI.DBAPIBase.cursor` — *Method*.

Create a new database cursor.

Returns a `JDBCCursor` instance.

source

9.7 DBAPI.DBAPIBase.execute!

`DBAPI.DBAPIBase.execute!` — *Method*.

Run a query on a database.

The results of the query are not returned by this function but are accessible through the cursor.

`parameters` can be any iterable of positional parameters, or of some T<:Associative for keyword/named parameters.

Throws a `JDBCError` if query caused an error, cursor is not initialized or connection is null.

Returns `nothing`.

source

9.8 DBAPI.DBAPIBase.rows

`DBAPI.DBAPIBase.rows` — *Method*.

Create a row iterator.

This method returns an instance of an iterator type which returns one row on each iteration. Each row returnes a Tuple{...}.

Throws a `JDBCError` if `execute!` was not called on the cursor or connection is null.

Returns a `JDBCRowIterator` instance.

source

9.9 JDBC.commit

`JDBC.commit` — *Method*.

Commit any pending transaction to the database. Throws a `JDBCError` if connection is null.

Returns `nothing`.

source

9.10 JDBC.commit

`JDBC.commit` — *Method*.

```
commit(connection::JConnection)
```

Commits the transaction

Args

- connection: The connection object

Returns

None

source

9.11 JDBC.createStatement

`JDBC.createStatement` — *Method*.

```
createStatement(connection::JConnection)
```

Initializes a Statement

Args

- connection: The connection object

Returns

The JStatement object

source

9.12 JDBC.execute

`JDBC.execute` — *Method*.

```
execute(stmt::JStatement, query::AbstractString)
```

Executes the auery based on JStatement or any of its sub-types

Args

- stmt: The JStatement object or any of its sub-types
- query: The query to be executed

Returns

A boolean indicating whether the execution was successful or not

source

9.13 JDBC.execute

`JDBC.execute` — *Method.*

```
execute(stmt::@compat(Union{JPreparedStatement, JCallableStatement}))
```

Executes the auery based on the Prepared Statement or Callable Statement
Args

- stmt: The Prepared Statement or the Callable Statement object

Returns
A boolean indicating whether the execution was successful or not
source

9.14 JDBC.executeQuery

`JDBC.executeQuery` — *Method.*

```
executeQuery(stmt::JStatement, query::AbstractString)
```

Executes the auery and returns the results as a JResultSet object.
Args

- stmt: The Statement object
- query: The query to be executed

Returns
The result set as a JResultSet object
source

9.15 JDBC.executeQuery

`JDBC.executeQuery` — *Method.*

```
executeQuery(stmt::@compat(Union{JPreparedStatement, JCallableStatement}))
```

Executes the auery based on a JPreparedStatement object or a JCallableStatement object
Args

- stmt: The JPreparedStatement object or JCallableStatement object

Returns
The result set as a JResultSet object
source

9.16 JDBC.executeUpdate

`JDBC.executeUpdate` — *Method*.

```
executeUpdate(stmt::JStatement, query::AbstractString)
```

Executes the update auery and returns the status of the execution of the query

Args

- stmt: The Statement object
- query: The query to be executed

Returns
An integer representing the status of the execution
source

9.17 JDBC.executeUpdate

`JDBC.executeUpdate` — *Method*.

```
executeUpdate(stmt::@compat(Union{JPreparedStatement, JCallableStatement}))
```

Executes the update auery based on a JPreparedStatement object or a JCallableStatement object

Args

- stmt: The JPreparedStatement object or JCallableStatement object

Returns
An integer indicating the status of the execution of the query
source

9.18 JDBC.getColumnCount

`JDBC.getColumnCount` — *Method*.

```
getColumnCount(rsmd::JResultSetMetaData)
```

Returns the number of columns based on the JResultSetMetaData object

Args

- rsmd: The JResultSetMetaData object

Returns
The number of columns.
source

9.19 JDBC.getColumnName

`JDBC.getColumnName` — *Method*.

```
getColumnName(rsmd::JResultSetMetaData, col::Integer)
```

Returns the column's name based on the JResultSetMetaData object and the column number

Args

- rsmd: The JResultSetMetaData object
- col: The column number

Returns
The column name
source

9.20 JDBC.getColumnType

`JDBC.getColumnType` — *Method*.

```
getColumnType(rsmd::JResultSetMetaData, col::Integer)
```

Returns the column's data type based on the JResultSetMetaData object and the column number

Args

- rsmd: The JResultSetMetaData object
- col: The column number

Returns
The column type as an integer
source

9.21 JDBC.getDate

`JDBC.getDate` — *Method*.

```
getDate(rs::@compat(Union{JResultSet, JCallableStatement}), fld::AbstractString)
```

Returns the Date object based on the result set or a callable statement. The value is extracted based on the column name.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column name

Returns
The Date object.
source

9.22 JDBC.getDate

`JDBC.getDate` — *Method.*

```
getDate(rs::@compat(Union{JResultSet, JCallableStatement}), fld::Integer)
```

Returns the Date object based on the result set or a callable statement. The value is extracted based on the column number.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column number

Returns
The Date object.
source

9.23 JDBC.getMetaData

`JDBC.getMetaData` — *Method.*

```
getMetaData(rs::JResultSet)
```

Returns information about the types and properties of the columns in the ResultSet object

Args

- stmt: The JResultSet object

Returns
The JResultSetMetaData object.
source

9.24 JDBC.getResultSet

`JDBC.getResultSet` — *Method.*

```
getResultSet(stmt::JStatement)
```

Returns the result set based on the previous execution of the query based on a JStatement

Args

- stmt: The JStatement object

Returns
The JResultSet object.
source

9.25 JDBC.getTableMetaData

`JDBC.getTableMetaData` — *Method*.

Get the metadata (column name and type) for each column of the table in the result set `rs`.

Returns an array of (column name, column type) tuples.

source

9.26 JDBC.getTime

`JDBC.getTime` — *Method*.

```
getTime(rs::@compat(Union{JResultSet, JCallableStatement}), fld::AbstractString)
```

Returns the Time object based on the result set or a callable statement. The value is extracted based on the column name.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column name

Returns

The Time object.

source

9.27 JDBC.getTime

`JDBC.getTime` — *Method*.

```
getTime(rs::@compat(Union{JResultSet, JCallableStatement}), fld::Integer)
```

Returns the Time object based on the result set or a callable statement. The value is extracted based on the column number.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column number

Returns

The Time object.

source

9.28 JDBC.getTimestamp

`JDBC.getTimestamp` — *Method*.

```
getTimestamp(rs::@compat(Union{JResultSet, JCallableStatement}), fld::AbstractString)
```

Returns the Timestamp object based on the result set or a callable statement. The value is extracted based on the column name.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column name

Returns
The Timestamp object.
source

9.29 JDBC.getTimestamp

`JDBC.getTimestamp` — *Method*.

```
getTimestamp(rs::@compat(Union{JResultSet, JCallableStatement}), fld::Integer)
```

Returns the Timestamp object based on the result set or a callable statement. The value is extracted based on the column number.

Args

- stmt: The JResultSet or JCallableStatement object
- fld: The column number

Returns
The Timestamp object.
source

9.30 JDBC.prepareCall

`JDBC.prepareCall` — *Method*.

```
prepareCall(connection::JConnection, query::AbstractString)
```

Prepares the Callable Statement for the given query

Args

- connection: The connection object
- query: The query string

Returns
The JCallableStatement object
source

9.31 JDBC.prepareStatement

`JDBC.prepareStatement` — *Method*.

```
prepareStatement(connection::JConnection, query::AbstractString)
```

Prepares the Statement for the given query
Args

- connection: The connection object
- query: The query string

Returns
The JPreparedStatement object
source

9.32 JDBC.rollback

`JDBC.rollback` — *Method*.
Roll back to the start of any pending transaction. Throws a `JDBCError` if connection is null.
Returns `nothing`.
source

9.33 JDBC.rollback

`JDBC.rollback` — *Method*.

```
rollback(connection::JConnection)
```

Rolls back the transactions.
Args

- connection: The connection object

Returns
None
source

9.34 JDBC.setAutoCommit

`JDBC.setAutoCommit` — *Method*.

```
setAutoCommit(connection::JConnection, x::Bool)
```

Set the Auto Commit flag to either true or false. If set to false, commit has to be called explicitly

Args

- connection: The connection object

Returns
None
source

9.35 JDBC.setFetchSize

`JDBC.setFetchSize` — *Method*.

```
setFetchSize(stmt::@compat(Union{JStatement, JPreparedStatement, JCallableStatement }), x::Intege
```

Sets the fetch size in a JStatement or a JPreparedStatement object or a JCallableStatement object. The number of records that are returned in subsequent query executions are determined by what is set here.

Args

- stmt: The JPreparedStatement object or JCallableStatement object
- x: The number of records to be returned

Returns
None
source

Chapter 10

NNlib

10.1 NNlib.elu

`NNlib.elu` — *Function.*

```
elu(x,  = 1) =
  x > 0 ? x :  * (exp(x) - 1)
```

Exponential Linear Unit activation function. See Fast and Accurate Deep Network Learning by Exponential Linear Units. You can also specify the coefficient explicitly, e.g. `elu(x, 1)`.

```
3
```

```
-1
```

```
  -3                    0                    3
```

source

10.2 NNlib.leakyrelu

`NNlib.leakyrelu` — *Function.*

```
leakyrelu(x) = max(0.01x, x)
```

Leaky Rectified Linear Unit activation function. You can also specify the coefficient explicitly, e.g. `leakyrelu(x, 0.01)`.

```
 3
```

```
-1
```

```
   -3                    0                    3
```

source

10.3 NNlib.logsoftmax

`NNlib.logsoftmax` — *Function.*

```
logsoftmax(xs) = log.(exp.(xs) ./ sum(exp.(xs)))
```

logsoftmax computes the log of softmax(xs) and it is more numerically stable than softmax function in computing the cross entropy loss.

source

10.4 NNlib.log

`NNlib.log` — *Method.*

```
log(x)
```

Return `log((x))` which is computed in a numerically stable way.

```
julia> log(0.)
-0.6931471805599453
julia> log.([-100, -10, 100.])
3-element Array{Float64,1}:
 -100.0
  -10.0
   -0.0
```

source

10.5 NNlib.relu

`NNlib.relu` — *Method.*

```
relu(x) = max(0, x)
```

Rectified Linear Unit activation function.

```
3
```

```
0
  -3                        0                        3
```

source

10.6 NNlib.selu

`NNlib.selu` — *Method.*

```
selu(x) =  * (x  0 ? x :  * (exp(x) - 1))

  1.0507
  1.6733
```

Scaled exponential linear units. See Self-Normalizing Neural Networks.
4 -2 -3 0 3
source

10.7 NNlib.softmax

`NNlib.softmax` — *Function*.

```
softmax(xs) = exp.(xs) ./ sum(exp.(xs))
```

Softmax takes log-probabilities (any real vector) and returns a probability distribution that sums to 1.

If given a matrix it will treat it as a batch of vectors, with each column independent.

```
julia> softmax([1,2,3.])
3-element Array{Float64,1}:
  0.0900306
  0.244728
  0.665241
```

source

10.8 NNlib.softplus

`NNlib.softplus` — *Method*.

```
softplus(x) = log(exp(x) + 1)
```

See Deep Sparse Rectifier Neural Networks.

```
4
```

```
0
  -3                    0                    3
```

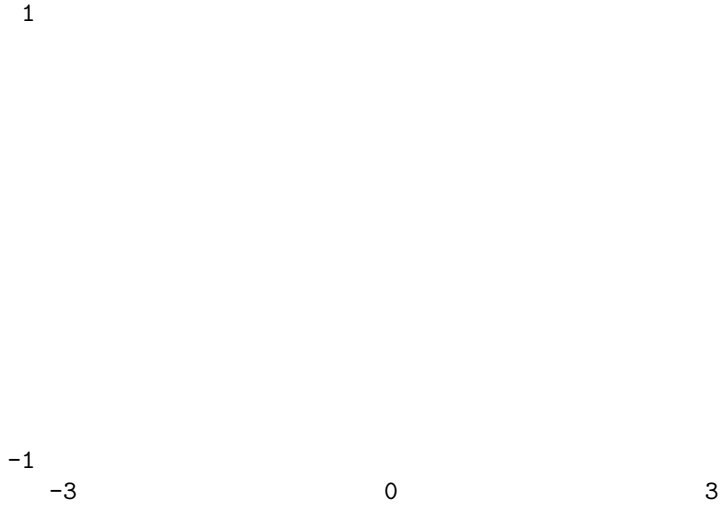
source

10.9 NNlib.softsign

`NNlib.softsign` — *Method.*

```
softsign(x) = x / (1 + |x|)
```

See Quadratic Polynomials Learn Better Image Features.



source

10.10 NNlib.swish

`NNlib.swish` — *Method.*

```
swish(x) = x * (x)
```

Self-gated actvation function. See Swish: a Self-Gated Activation Function.

source

10.11 NNlib.

`NNlib.` — *Method.*

```
(x) = 1 / (1 + exp(-x))
```

Classic sigmoid activation function.

source

Chapter 11

ImageCore

11.1 ImageCore.assert_timedim_last

`ImageCore.assert_timedim_last` — *Method*.

```
assert_timedim_last(img)
```

Throw an error if the image has a time dimension that is not the last dimension.

source

11.2 ImageCore.channelview

`ImageCore.channelview` — *Method*.

```
channelview(A)
```

returns a view of `A`, splitting out (if necessary) the color channels of `A` into a new first dimension. This is almost identical to `ChannelView(A)`, except that if `A` is a `ColorView`, it will simply return the parent of `A`, or will use `reinterpret` when appropriate. Consequently, the output may not be a `ChannelView` array.

Of relevance for types like RGB and BGR, the channels of the returned array will be in constructor-argument order, not memory order (see `reinterpret` if you want to use memory order).

source

11.3 ImageCore.clamp01

`ImageCore.clamp01` — *Method*.

```
clamp01(x) -> y
```

Produce a value `y` that lies between 0 and 1, and equal to `x` when `x` is already in this range. Equivalent to `clamp(x, 0, 1)` for numeric values. For colors, this function is applied to each color channel separately.

See also: `clamp01nan`.

source

11.4 ImageCore.clamp01nan

`ImageCore.clamp01nan` — *Method*.

```
clamp01nan(x) -> y
```

Similar to `clamp01`, except that any `NaN` values are changed to 0.

See also: `clamp01`.

source

11.5 ImageCore.colorsigned

`ImageCore.colorsigned` — *Method*.

```
colorsigned()
colorsigned(colorneg, colorpos) -> f
colorsigned(colorneg, colorcenter, colorpos) -> f
```

Define a function that maps negative values (in the range [-1,0]) to the linear colormap between `colorneg` and `colorcenter`, and positive values (in the range [0,1]) to the linear colormap between `colorcenter` and `colorpos`.

The default colors are:

- `colorcenter`: white
- `colorneg`: green1
- `colorpos`: magenta

See also: `scalesigned`.

source

11.6 ImageCore.colorview

`ImageCore.colorview` — *Method*.

```
colorview(C, gray1, gray2, ...) -> imgC
```

Combine numeric/grayscale images `gray1`, `gray2`, etc., into the separate color channels of an array `imgC` with element type `C<:Colorant`.

As a convenience, the constant `zeroarray` fills in an array of matched size with all zeros.

Example

[] imgC = colorview(RGB, r, zeroarray, b)

creates an image with `r` in the red chanel, `b` in the blue channel, and nothing in the green channel.

See also: `StackedView`.

source

11.7 ImageCore.colorview

`ImageCore.colorview` — *Method*.

```
colorview(C, A)
```

returns a view of the numeric array `A`, interpreting successive elements of `A` as if they were channels of Colorant `C`. This is almost identical to `ColorView{C}(A)`, except that if `A` is a `ChannelView`, it will simply return the parent of `A`, or use `reinterpret` when appropriate. Consequently, the output may not be a `ColorView` array.

Of relevance for types like RGB and BGR, the elements of `A` are interpreted in constructor-argument order, not memory order (see `reinterpret` if you want to use memory order).

Example

[] A = rand(3, 10, 10) img = colorview(RGB, A)

source

11.8 ImageCore.coords_spatial

`ImageCore.coords_spatial` — *Method*.

coords_spatial(img)

Return a tuple listing the spatial dimensions of `img`.

Note that a better strategy may be to use ImagesAxes and take slices along the time axis.

source

11.9 ImageCore.float32

`ImageCore.float32` — *Function*.

```
float32.(img)
```

converts the raw storage type of `img` to `Float32`, without changing the color space.

source

11.10 ImageCore.float64

`ImageCore.float64` — *Function*.

```
float64.(img)
```

converts the raw storage type of `img` to `Float64`, without changing the color space.

source

11.11 ImageCore.indices_spatial

`ImageCore.indices_spatial` — *Method*.

```
indices_spatial(img)
```

Return a tuple with the indices of the spatial dimensions of the image. Defaults to the same as `indices`, but using ImagesAxes you can mark some axes as being non-spatial.

source

11.12 ImageCore.n0f16

`ImageCore.n0f16` — *Function*.

```
n0f16.(img)
```

converts the raw storage type of `img` to `N0f16`, without changing the color space.

source

11.13 ImageCore.n0f8

`ImageCore.n0f8` — *Function*.

```
n0f8.(img)
```

converts the raw storage type of `img` to `N0f8`, without changing the color space.

source

11.14 ImageCore.n2f14

`ImageCore.n2f14` — *Function*.

`n2f14.(img)`

converts the raw storage type of `img` to `N2f14`, without changing the color space.

source

11.15 ImageCore.n4f12

`ImageCore.n4f12` — *Function*.

`n4f12.(img)`

converts the raw storage type of `img` to `N4f12`, without changing the color space.

source

11.16 ImageCore.n6f10

`ImageCore.n6f10` — *Function*.

`n6f10.(img)`

converts the raw storage type of `img` to `N6f10`, without changing the color space.

source

11.17 ImageCore.nimages

`ImageCore.nimages` — *Method*.

`nimages(img)`

Return the number of time-points in the image array. Defaults to

1. Use ImagesAxes if you want to use an explicit time dimension.

source

11.18 ImageCore.normedview

`ImageCore.normedview` — *Method*.

```
normedview([T], img::AbstractArray{Unsigned})
```

returns a "view" of `img` where the values are interpreted in terms of `Normed` number types. For example, if `img` is an `Array{UInt8}`, the view will act like an `Array{N0f8}`. Supply `T` if the element type of `img` is `UInt16`, to specify whether you want a `N6f10`, `N4f12`, `N2f14`, or `N0f16` result.

source

11.19 ImageCore.permuteddimsview

`ImageCore.permuteddimsview` — *Method*.

```
permuteddimsview(A, perm)
```

returns a "view" of `A` with its dimensions permuted as specified by `perm`. This is like `permutedims`, except that it produces a view rather than a copy of `A`; consequently, any manipulations you make to the output will be mirrored in `A`. Compared to the copy, the view is much faster to create, but generally slower to use.

source

11.20 ImageCore.pixelspacing

`ImageCore.pixelspacing` — *Method*.

```
pixelspacing(img) -> (sx, sy, ...)
```

Return a tuple representing the separation between adjacent pixels along each axis of the image. Defaults to (1,1,...). Use ImagesAxes for images with anisotropic spacing or to encode the spacing using physical units.

source

11.21 ImageCore.rawview

`ImageCore.rawview` — *Method*.

```
rawview(img::AbstractArray{FixedPoint})
```

returns a "view" of `img` where the values are interpreted in terms of their raw underlying storage. For example, if `img` is an `Array{N0f8}`, the view will act like an `Array{UInt8}`.

source

11.22 ImageCore.scaleminmax

`ImageCore.scaleminmax` — *Method*.

```
scaleminmax(min, max) -> f
scaleminmax(T, min, max) -> f
```

Return a function `f` which maps values less than or equal to `min` to 0, values greater than or equal to `max` to 1, and uses a linear scale in between. `min` and `max` should be real values.

Optionally specify the return type `T`. If `T` is a colorant (e.g., RGB), then scaling is applied to each color channel.

Examples

Example 1

[] julia¿ f = scaleminmax(-10, 10) (::9) (generic function with 1 method)
julia¿ f(10) 1.0
julia¿ f(-10) 0.0
julia¿ f(5) 0.75

Example 2

[] julia¿ c = RGB(255.0,128.0,0.0) RGB{Float64}(255.0,128.0,0.0)
julia¿ f = scaleminmax(RGB, 0, 255) (::13) (generic function with 1 method)
julia¿ f(c) RGB{Float64}(1.0,0.5019607843137255,0.0)

See also: `takemap`.

source

11.23 ImageCore.scalesigned

`ImageCore.scalesigned` — *Method*.

```
scalesigned(maxabs) -> f
```

Return a function `f` which scales values in the range `[-maxabs, maxabs]` (clamping values that lie outside this range) to the range `[-1, 1]`.

See also: `colorsigned`.

source

11.24 ImageCore.scalesigned

`ImageCore.scalesigned` — *Method*.

```
scalesigned(min, center, max) -> f
```

Return a function `f` which scales values in the range `[min, center]` to `[-1,0]` and `[center,max]` to `[0,1]`. Values smaller than `min`/`max` get clamped to `min`/`max`, respectively.

See also: `colorsigned`.

source

11.25 ImageCore.sdims

`ImageCore.sdims` — *Method.*

```
sdims(img)
```

Return the number of spatial dimensions in the image. Defaults to the same as `ndims`, but with ImagesAxes you can specify that some axes correspond to other quantities (e.g., time) and thus not included by `sdims`.

source

11.26 ImageCore.size_spatial

`ImageCore.size_spatial` — *Method.*

```
size_spatial(img)
```

Return a tuple listing the sizes of the spatial dimensions of the image. Defaults to the same as `size`, but using ImagesAxes you can mark some axes as being non-spatial.

source

11.27 ImageCore.spacedirections

`ImageCore.spacedirections` — *Method.*

```
spacedirections(img) -> (axis1, axis2, ...)
```

Return a tuple-of-tuples, each `axis[i]` representing the displacement vector between adjacent pixels along spatial axis `i` of the image array, relative to some external coordinate system ("physical coordinates").

By default this is computed from `pixelspacing`, but you can set this manually using ImagesMeta.

source

11.28 ImageCore.takemap

`ImageCore.takemap` — *Function.*

```
takemap(f, A) -> fnew
takemap(f, T, A) -> fnew
```

Given a value-mapping function `f` and an array `A`, return a "concrete" mapping function `fnew`. When applied to elements of `A`, `fnew` should return valid values for storage or display, for example in the range from 0 to 1 (for grayscale)

or valid colorants. `fnew` may be adapted to the actual values present in `A`, and may not produce valid values for any inputs not in `A`.

Optionally one can specify the output type `T` that `fnew` should produce.

Example:

[] julia¿ A = [0, 1, 1000];

julia¿ f = takemap(scaleminmax, A) (::7) (generic function with 1 method)

julia¿ f.(A) 3-element Array{Float64,1}: 0.0 0.001 1.0

source

Chapter 12

Reactive

12.1 Base.filter

`Base.filter` — *Method*.

```
filter(f, default, signal)
```

remove updates from the `signal` where `f` returns `false`. The filter will hold the value default until f(value(signal)) returns true, when it will be updated to value(signal).

source

12.2 Base.map

`Base.map` — *Method*.

```
map(f, s::Signal...) -> signal
```

Transform signal `s` by applying `f` to each element. For multiple signal arguments, apply `f` elementwise.

source

12.3 Base.merge

`Base.merge` — *Method*.

```
merge(inputs...)
```

Merge many signals into one. Returns a signal which updates when any of the inputs update. If many signals update at the same time, the value of the *youngest* (most recently created) input signal is taken.

source

12.4 Base.push!

`Base.push!` — *Function.*

`push!(signal, value, onerror=Reactive.print_error)`

Queue an update to a signal. The update will be propagated when all currently queued updates are done processing.

The third (optional) argument, `onerror`, is a callback triggered when the update ends in an error. The callback receives 4 arguments, `onerror(sig, val, node, capex)`, where `sig` and `val` are the Signal and value that `push!` was called with, respectively, `node` is the Signal whose action triggered the error, and `capex` is a `CapturedException` with the fields `ex` which is the original exception object, and `processed_bt` which is the backtrace of the exception.

The default error callback will print the error and backtrace to STDERR.

source

12.5 Reactive.async_map

`Reactive.async_map` — *Method.*

```
tasks, results = async_map(f, init, input...;typ=typeof(init), onerror=Reactive.print_
```

Spawn a new task to run a function when input signal updates. Returns a signal of tasks and a `results` signal which updates asynchronously with the results. `init` will be used as the default value of `results`. `onerror` is the callback to be called when an error occurs, by default it is set to a callback which prints the error to STDERR. It's the same as the `onerror` argument to `push!` but is run in the spawned task.

source

12.6 Reactive.bind!

`Reactive.bind!` — *Function.*

```
‘bind!(dest, src, twoway=true; initial=true)‘
```

for every update to `src` also update `dest` with the same value and, if `twoway` is true, vice-versa. If `initial` is false, `dest` will only be updated to `src`'s value when `src` next updates, otherwise (if `initial` is true) both `dest` and `src` will take `src`'s value immediately.

source

12.7 Reactive.bound_dests

`Reactive.bound_dests` — *Method.*

`bound_dests(src::Signal)` returns a vector of all signals that will update when `src` updates, that were bound using `bind!(dest, src)`

source

12.8 Reactive.bound_srcs

`Reactive.bound_srcs` — *Method*.

`bound_srcs(dest::Signal)` returns a vector of all signals that will cause an update to `dest` when they update, that were bound using `bind!(dest, src)`

source

12.9 Reactive.debounce

`Reactive.debounce` — *Method*.

```
debounce(dt, input, f=(acc,x)->x, init=value(input), reinit=x->x;
            typ=typeof(init), name=auto_name!(string("debounce ",dt,"s"), input))
```

Creates a signal that will delay updating until `dt` seconds have passed since the last time `input` has updated. By default, the debounce signal holds the last update of the `input` signal since the debounce signal last updated.

This behavior can be changed by the `f`, `init` and `reinit` arguments. The `init` and `f` functions are similar to `init` and `f` in `foldp`. `reinit` is called after the debounce sends an update, to reinitialize the initial value for accumulation, it gets one argument, the previous accumulated value.

For example `y = debounce(0.2, x, push!, Int[], _->Int[])` will accumulate a vector of updates to the integer signal `x` and push it after `x` is inactive (doesn't update) for 0.2 seconds.

source

12.10 Reactive.delay

`Reactive.delay` — *Method*.

```
delay(input, default=value(input))
```

Schedule an update to happen after the current update propagates throughout the signal graph.

Returns the delayed signal.

source

12.11 Reactive.droprepeats

`Reactive.droprepeats` — *Method*.

```
droprepeats(input)
```

Drop updates to `input` whenever the new value is the same as the previous value of the signal.

source

12.12 Reactive.every

`Reactive.every` — *Method*.

```
every(dt)
```

A signal that updates every `dt` seconds to the current timestamp. Consider using `fpswhen` or `fps` if you want specify the timing signal by frequency, rather than delay.

source

12.13 Reactive.filterwhen

`Reactive.filterwhen` — *Method*.

```
filterwhen(switch::Signal{Bool}, default, input)
```

Keep updates to `input` only when `switch` is true.

If switch is false initially, the specified default value is used.

source

12.14 Reactive.flatten

`Reactive.flatten` — *Method*.

```
flatten(input::Signal{Signal}; typ=Any)
```

Flatten a signal of signals into a signal which holds the value of the current signal. The `typ` keyword argument specifies the type of the flattened signal. It is `Any` by default.

source

12.15 Reactive.foldp

`Reactive.foldp` — *Method*.

```
foldp(f, init, inputs...)
```

Fold over past values.

Accumulate a value as the input signals change. `init` is the initial value of the accumulator. `f` should take `1 + length(inputs)` arguments: the first is the current accumulated value and the rest are the current input signal values. `f` will be called when one or more of the `inputs` updates. It should return the next accumulated value.

source

12.16 Reactive.fps

`Reactive.fps` — *Method*.

```
fps(rate)
```

Same as `fpswhen(Input(true), rate)`

source

12.17 Reactive.fpswhen

`Reactive.fpswhen` — *Method*.

```
fpswhen(switch, rate)
```

returns a signal which when `switch` signal is true, updates `rate` times every second. If `rate` is not possible to attain because of slowness in computing dependent signal values, the signal will self adjust to provide the best possible rate.

source

12.18 Reactive.preserve

`Reactive.preserve` — *Method*.

```
preserve(signal::Signal)
```

prevents `signal` from being garbage collected as long as any of its parents are around. Useful for when you want to do some side effects in a signal. e.g. `preserve(map(println, x))` - this will continue to print updates to x, until x goes out of scope. `foreach` is a shorthand for `map` with `preserve`.

source

12.19 Reactive.previous

`Reactive.previous` — *Method*.

```
previous(input, default=value(input))
```

Create a signal which holds the previous value of `input`. You can optionally specify a different initial value.

source

12.20 Reactive.remote_map

`Reactive.remote_map` — *Method*.

```
remoterefs, results = remote_map(procid, f, init, input...;typ=typeof(init), onerror=Re
```

Spawn a new task on process `procid` to run a function when input signal updates. Returns a signal of remote refs and a `results` signal which updates asynchronously with the results. `init` will be used as the default value of `results`. `onerror` is the callback to be called when an error occurs, by default it is set to a callback which prints the error to STDERR. It's the same as the `onerror` argument to `push!` but is run in the spawned task.

source

12.21 Reactive.rename!

`Reactive.rename!` — *Method*.

`rename!(s::Signal, name::String)`

Change a Signal's name

source

12.22 Reactive.sampleon

`Reactive.sampleon` — *Method*.

```
sampleon(a, b)
```

Sample the value of `b` whenever `a` updates.

source

12.23 Reactive.throttle

`Reactive.throttle` — *Method*.

```
throttle(dt, input, f=(acc,x)->x, init=value(input), reinit=x->x;
            typ=typeof(init), name=auto_name!(string("throttle ",dt,"s"), input), leadi
```

Throttle a signal to update at most once every dt seconds. By default, the throttled signal holds the last update of the `input` signal during each `dt` second time window.

This behavior can be changed by the `f`, `init` and `reinit` arguments. The `init` and `f` functions are similar to `init` and `f` in `foldp`. `reinit` is called when a new throttle time window opens to reinitialize the initial value for accumulation, it gets one argument, the previous accumulated value.

For example `y = throttle(0.2, x, push!, Int[], _->Int[])` will create vectors of updates to the integer signal `x` which occur within 0.2 second time windows.

If `leading` is `true`, the first update from `input` will be sent immediately by the throttle signal. If it is false, the first update will happen `dt` seconds after `input`'s first update

New in v0.4.1: `throttle`'s behaviour from previous versions is now available with the `debounce` signal type.

source

12.24 Reactive.unbind!

`Reactive.unbind!` — *Function.*

```
‘unbind!(dest, src, twoway=true)‘
```

remove a link set up using `bind!`

source

12.25 Reactive.unpreserve

`Reactive.unpreserve` — *Method.*

```
unpreserve(signal::Signal)
```

allow `signal` to be garbage collected. See also `preserve`.

source

Chapter 13

JuliaDB

13.1 Dagger.compute

`Dagger.compute` — *Method*.

```
compute(t::DNDSparse; allowoverlap, closed)
```

Computes any delayed-evaluations in the `DNDSparse`. The computed data is left on the worker processes. Subsequent operations on the results will reuse the chunks.

If `allowoverlap` is false then the computed data is re-sorted if required to have no chunks with overlapping index ranges if necessary.

If `closed` is true then the computed data is re-sorted if required to have no chunks with overlapping OR continuous boundaries.

See also `collect`.

!!! warning `compute(t)` requires at least as much memory as the size of the result of the computing `t`. You usually don't need to do this for the whole dataset. If the result is expected to be big, try `compute(save(t, "output_dir"))` instead. See `save` for more.

source

13.2 Dagger.distribute

`Dagger.distribute` — *Function*.

```
distribute(itable::NDSparse, nchunks::Int=nworkers())
```

Distributes an NDSparse object into a DNDSparse of `nchunks` chunks of approximately equal size.

Returns a `DNDSparse`.

source

13.3 Dagger.distribute

`Dagger.distribute` — *Method*.

```
distribute(t::Table, chunks)
```

Distribute a table in `chunks` pieces. Equivalent to `table(t, chunks=chunks)`.
source

13.4 Dagger.distribute

`Dagger.distribute` — *Method*.

```
distribute(itable::NDSparse, rowgroups::AbstractArray)
```

Distributes an NDSparse object into a DNDSparse by splitting it up into chunks of `rowgroups` elements. `rowgroups` is a vector specifying the number of rows in the chunks.

Returns a `DNDSparse`.
source

13.5 Dagger.load

`Dagger.load` — *Method*.

```
load(dir::AbstractString; tomemory)
```

Load a saved `DNDSparse` from `dir` directory. Data can be saved using the `save` function.
source

13.6 Dagger.save

`Dagger.save` — *Method*.

```
save(t::Union{DNDSparse, DNDSparse}, outputdir::AbstractString)
```

Saves a distributed dataset to disk. Saved data can be loaded with `load`.
source

13.7 IndexedTables.convertdim

`IndexedTables.convertdim` — *Method*.

```
convertdim(x::DNDSparse, d::DimName, xlate; agg::Function, name)
```

Apply function or dictionary `xlate` to each index in the specified dimension. If the mapping is many-to-one, `agg` is used to aggregate the results. `name` optionally specifies a name for the new dimension. `xlate` must be a monotonically increasing function.

See also `reducedim` and `aggregate`

source

13.8 IndexedTables.leftjoin

`IndexedTables.leftjoin` — *Method*.

```
leftjoin(left::DNDSparse, right::DNDSparse, [op::Function])
```

Keeps only rows with indices in `left`. If rows of the same index are present in `right`, then they are combined using `op`. `op` by default picks the value from `right`.

source

13.9 IndexedTables.naturaljoin

`IndexedTables.naturaljoin` — *Method*.

```
naturaljoin(op, left::DNDSparse, right::DNDSparse, ascolumns=false)
```

Returns a new `DNDSparse` containing only rows where the indices are present both in `left` AND `right` tables. The data columns are concatenated. The data of the matching rows from `left` and `right` are combined using `op`. If `op` returns a tuple or NamedTuple, and `ascolumns` is set to true, the output table will contain the tuple elements as separate data columns instead as a single column of resultant tuples.

source

13.10 IndexedTables.naturaljoin

`IndexedTables.naturaljoin` — *Method*.

```
naturaljoin(left::DNDSparse, right::DNDSparse, [op])
```

Returns a new `DNDSparse` containing only rows where the indices are present both in `left` AND `right` tables. The data columns are concatenated.

source

13.11 IndexedTables.reducedim_vec

`IndexedTables.reducedim_vec` — *Method*.

```
reducedim_vec(f::Function, t::DNDSparse, dims)
```

Like `reducedim`, except uses a function mapping a vector of values to a scalar instead of a 2-argument scalar function.

See also `reducedim` and `aggregate_vec`.

source

13.12 JuliaDB.loadndsparse

`JuliaDB.loadndsparse` — *Method*.

`loadndsparse(files::Union{AbstractVector,String}; <options>)`

Load an NDSparse from CSV files.

`files` is either a vector of file paths, or a directory name.

Options:

- `indexcols::Vector` – columns to use as indexed columns. (by default a `1:n` implicit index is used.)
- `datacols::Vector` – non-indexed columns. (defaults to all columns but indexed columns). Specify this to only load a subset of columns. In place of the name of a column, you can specify a tuple of names – this will treat any column with one of those names as the same column, but use the first name in the tuple. This is useful when the same column changes name between CSV files. (e.g. `vendor_id` and `VendorId`)

All other options are identical to those in `loadtable`

source

13.13 JuliaDB.loadtable

`JuliaDB.loadtable` — *Method*.

`loadtable(files::Union{AbstractVector,String}; <options>)`

Load a table from CSV files.

`files` is either a vector of file paths, or a directory name.

Options:

- `output::AbstractString` – directory name to write the table to. By default data is loaded directly to memory. Specifying this option will allow you to load data larger than the available memory.
- `indexcols::Vector` – columns to use as primary key columns. (defaults to [])

- `datacols::Vector` – non-indexed columns. (defaults to all columns but indexed columns). Specify this to only load a subset of columns. In place of the name of a column, you can specify a tuple of names – this will treat any column with one of those names as the same column, but use the first name in the tuple. This is useful when the same column changes name between CSV files. (e.g. `vendor_id` and `VendorId`)
- `distributed::Bool` – should the output dataset be loaded as a distributed table? If true, this will use all available worker processes to load the data. (defaults to true if workers are available, false if not)
- `chunks::Int` – number of chunks to create when loading distributed. (defaults to number of workers)
- `delim::Char` – the delimiter character. (defaults to `,`). Use `spacedelim=true` to split by spaces.
- `spacedelim::Bool`: parse space-delimited files. `delim` has no effect if true.
- `quotechar::Char` – quote character. (defaults to `"`)
- `escapechar::Char` – escape character. (defaults to `"`)
- `filenamecol::Union{Symbol, Pair}` – create a column containing the file names from where each row came from. This argument gives a name to the column. By default, `basename(name)` of the name is kept, and ".csv" suffix will be stripped. To provide a custom function to apply on the names, use a `name => Function` pair. By default, no file name column will be created.
- `header_exists::Bool` – does header exist in the files? (defaults to true)
- `colnames::Vector{String}` – specify column names for the files, use this with (`header_exists=false`, otherwise first row is discarded). By default column names are assumed to be present in the file.
- `samecols` – a vector of tuples of strings where each tuple contains alternative names for the same column. For example, if some files have the name "vendor_id" and others have the name "VendorID", pass `samecols=[("VendorID", "vendor_id")]`.
- `colparsers` – either a vector or dictionary of data types or an `AbstractToken` object from TextParse package. By default, these are inferred automatically. See `type_detect_rows` option below.
- `type_detect_rows`: number of rows to use to infer the initial `colparsers` defaults to 20.
- `nastrings::Vector{String}` – strings that are to be considered NA. (defaults to `TextParse.NA_STRINGS`)

- `skiplines_begin::Char` – skip some lines in the beginning of each file. (doesn't skip by default)
- `usecache::Bool`: (vestigial)

source

13.14 JuliaDB.partitionplot

`JuliaDB.partitionplot` — *Function.*

```
partitionplot(table, y;    stat=Extrema(), nparts=100, by=nothing, dropmissing=false)
partitionplot(table, x, y; stat=Extrema(), nparts=100, by=nothing, dropmissing=false)
```

Plot a summary of variable `y` against `x` (`1:length(y)` if not specified). Using `nparts` approximately-equal sections along the x-axis, the data in `y` over each section is summarized by `stat`.

source

13.15 JuliaDB.rechunk

`JuliaDB.rechunk` — *Function.*

`rechunk(t::Union{DNDSparse, DNDSparse}[, by[, select]]; <options>)`

Reindex and sort a distributed dataset by keys selected by `by`.

Optionally `select` specifies which non-indexed fields are kept. By default this is all fields not mentioned in `by` for Table and the value columns for NDSparse.

Options:

- `chunks` – how to distribute the data. This can be:
  1. An integer – number of chunks to create
  2. An vector of `k` integers – number of elements in each of the `k` chunks. `sum(k)` must be same as `length(t)`
  3. The distribution of another array. i.e. `vec.subdomains` where `vec` is a distributed array.
- `merge::Function` – a function which merges two sub-table or sub-ndsparse into one NDSparse. They may have overlaps in their indices.
- `splitters::AbstractVector` – specify keys to split by. To create `n` chunks you would need to pass `n-1` splitters and also the `chunks=n` option.
- `chunks_sorted::Bool` – are the chunks sorted locally? If true, this skips sorting or re-indexing them.

- `affinities::Vector{<:Integer}` – which processes (Int pid) should each output chunk be created on. If unspecified all workers are used.
- `closed::Bool` – if true, the same key will not be present in multiple chunks (although sorted). `true` by default.
- `nsamples::Integer` – number of keys to randomly sample from each chunk to estimate splitters in the sorting process. (See samplesort). Defaults to 2000.
- `batchsize::Integer` – how many chunks at a time from the input should be loaded into memory at any given time. This will essentially sort in batches of `batchsize` chunks.

source

13.16 JuliaDB.tracktime

`JuliaDB.tracktime` — *Method*.

`tracktime(f)`

Track the time spent on different processes in different categories in running `f`.

source

Chapter 14

Combinatorics

14.1 Base.factorial

`Base.factorial` — *Method*.

computes n!/k!

source

14.2 Combinatorics.bellnum

`Combinatorics.bellnum` — *Method*.

Returns the n-th Bell number

source

14.3 Combinatorics.catalannum

`Combinatorics.catalannum` — *Method*.

Returns the n-th Catalan number

source

14.4 Combinatorics.character

`Combinatorics.character` — *Method*.

Computes character () of the partition in the th irrep of the symmetric group S_n

Implements the Murnaghan-Nakayama algorithm as described in: Dan Bernstein, "The computational complexity of rules for the character table of Sn", Journal of Symbolic Computation, vol. 37 iss. 6 (2004), pp 727-748. doi:10.1016/j.jsc.2003.11.001

source

14.5 Combinatorics.combinations

`Combinatorics.combinations` — *Method*.

Generate all combinations of `n` elements from an indexable object. Because the number of combinations can be very large, this function returns an iterator object. Use `collect(combinations(array,n))` to get an array of all combinations.

source

14.6 Combinatorics.combinations

`Combinatorics.combinations` — *Method*.

generate combinations of all orders, chaining of order iterators is eager, but sequence at each order is lazy

source

14.7 Combinatorics.derangement

`Combinatorics.derangement` — *Method*.

The number of permutations of n with no fixed points (subfactorial)

source

14.8 Combinatorics.integer_partitions

`Combinatorics.integer_partitions` — *Method*.

Lists the partitions of the number n, the order is consistent with GAP

source

14.9 Combinatorics.isrimhook

`Combinatorics.isrimhook` — *Method*.

Checks if skew diagram is a rim hook

source

14.10 Combinatorics.isrimhook

`Combinatorics.isrimhook` — *Method*.

Takes two elements of a partition sequence, with a to the left of b

source

14.11 Combinatorics.lassallenum

`Combinatorics.lassallenum` — *Method*.

Computes Lassalle's sequence OEIS entry A180874

source

14.12 Combinatorics.leglength

`Combinatorics.leglength` — *Method*.

Strictly speaking, defined for rim hook only, but here we define it for all skew diagrams

source

14.13 Combinatorics.levicivita

`Combinatorics.levicivita` — *Method*.

Levi-Civita symbol of a permutation.

Returns 1 is the permutation is even, -1 if it is odd and 0 otherwise.

The parity is computed by using the fact that a permutation is odd if and only if the number of even-length cycles is odd.

source

14.14 Combinatorics.multiexponents

`Combinatorics.multiexponents` — *Method*.

```
multiexponents(m, n)
```

Returns the exponents in the multinomial expansion (x + x + ... + x).

For example, the expansion (x + x + x) = x + xx + xx + ... has the exponents:

```
julia> collect(multiexponents(3, 2))

6-element Array{Any,1}:
 [2, 0, 0]
 [1, 1, 0]
 [1, 0, 1]
 [0, 2, 0]
 [0, 1, 1]
 [0, 0, 2]
```

source

14.15 Combinatorics.multinomial

`Combinatorics.multinomial` — *Method*.

Multinomial coefficient where n = sum(k)

source

14.16 Combinatorics.multiset_combinations

`Combinatorics.multiset_combinations` — *Method*.

generate all combinations of size t from an array a with possibly duplicated elements.

source

14.17 Combinatorics.multiset_permutations

`Combinatorics.multiset_permutations` — *Method*.

generate all permutations of size t from an array a with possibly duplicated elements.

source

14.18 Combinatorics.nthperm!

`Combinatorics.nthperm!` — *Method*.

In-place version of nthperm.

source

14.19 Combinatorics.nthperm

`Combinatorics.nthperm` — *Method*.

Compute the kth lexicographic permutation of the vector a.

source

14.20 Combinatorics.nthperm

`Combinatorics.nthperm` — *Method*.

Return the `k` that generated permutation `p`. Note that `nthperm(nthperm([1:n], k)) == k` for `1 <= k <= factorial(n)`.

source

14.21 Combinatorics.parity

`Combinatorics.parity` — *Method*.

Computes the parity of a permutation using the levicivita function, so you can ask iseven(parity(p)). If p is not a permutation throws an error.

source

14.22 Combinatorics.partitions

`Combinatorics.partitions` — *Method*.

Generate all set partitions of the elements of an array into exactly m subsets, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array,m))` to get an array of all partitions. The number of partitions into m subsets is equal to the Stirling number of the second kind and can be efficiently computed using `length(partitions(array,m))`.

source

14.23 Combinatorics.partitions

`Combinatorics.partitions` — *Method*.

Generate all set partitions of the elements of an array, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(array))`.

source

14.24 Combinatorics.partitions

`Combinatorics.partitions` — *Method*.

Generate all arrays of `m` integers that sum to `n`. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n,m))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(n,m))`.

source

14.25 Combinatorics.partitions

`Combinatorics.partitions` — *Method*.

Generate all integer arrays that sum to `n`. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(n))`.

source

14.26 Combinatorics.partitionsequence

`Combinatorics.partitionsequence` — *Method*.

Computes essential part of the partition sequence of lambda

source

14.27 Combinatorics.permutations

`Combinatorics.permutations` — *Method*.

Generate all size t permutations of an indexable object.

source

14.28 Combinatorics.permutations

`Combinatorics.permutations` — *Method*.

Generate all permutations of an indexable object. Because the number of permutations can be very large, this function returns an iterator object. Use `collect(permutations(array))` to get an array of all permutations.

source

14.29 Combinatorics.prevprod

`Combinatorics.prevprod` — *Method*.

Previous integer not greater than `n` that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

For a list of integers i1, i2, i3, find the largest i1^n1 * i2^n2 * i3^n3 <= x for integer n1, n2, n3

source

14.30 Combinatorics.with_replacement_combinations

`Combinatorics.with_replacement_combinations` — *Method*.

generate all combinations with replacement of size t from an array a.

source

Chapter 15

HypothesisTests

15.1 HypothesisTests.ChisqTest

`HypothesisTests.ChisqTest` — *Method.*

```
ChisqTest(x[, y][, theta0 = ones(length(x))/length(x)])
```

Perform a `PowerDivergenceTest` with = 1, i.e. in the form of Pearson's chi-squared statistic.

If `y` is not given and `x` is a matrix with one row or column, or `x` is a vector, then a goodness-of-fit test is performed (`x` is treated as a one-dimensional contingency table). In this case, the hypothesis tested is whether the population probabilities equal those in `theta0`, or are all equal if `theta0` is not given.

If `x` is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table. Otherwise, `x` and `y` must be vectors of the same length. The contingency table is calculated using `counts` function from the `StatsBase` package. Then the power divergence test is conducted under the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals.

Note that the entries of `x` (and `y` if provided) must be non-negative integers.

Implements: `pvalue`, `confint`

source

15.2 HypothesisTests.MannWhitneyUTest

`HypothesisTests.MannWhitneyUTest` — *Method.*

```
MannWhitneyUTest(x::AbstractVector{<:Real}, y::AbstractVector{<:Real})
```

Perform a Mann-Whitney U test of the null hypothesis that the probability that an observation drawn from the same population as `x` is greater than an observation drawn from the same population as `y` is equal to the probability that

an observation drawn from the same population as `y` is greater than an observation drawn from the same population as `x` against the alternative hypothesis that these probabilities are not equal.

The Mann-Whitney U test is sometimes known as the Wilcoxon rank-sum test.

When there are no tied ranks and 50 samples, or tied ranks and 10 samples, `MannWhitneyUTest` performs an exact Mann-Whitney U test. In all other cases, `MannWhitneyUTest` performs an approximate Mann-Whitney U test. Behavior may be further controlled by using `ExactMannWhitneyUTest` or `ApproximateMannWhitneyUTest` directly.

Implements: `pvalue`

source

15.3 HypothesisTests.MultinomialLRT

`HypothesisTests.MultinomialLRT` — *Method.*

```
MultinomialLRT(x[, y][, theta0 = ones(length(x))/length(x)])
```

Perform a `PowerDivergenceTest` with $\lambda = 0$, i.e. in the form of the likelihood ratio test statistic.

If `y` is not given and `x` is a matrix with one row or column, or `x` is a vector, then a goodness-of-fit test is performed (`x` is treated as a one-dimensional contingency table). In this case, the hypothesis tested is whether the population probabilities equal those in `theta0`, or are all equal if `theta0` is not given.

If `x` is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table. Otherwise, `x` and `y` must be vectors of the same length. The contingency table is calculated using `counts` function from the `StatsBase` package. Then the power divergence test is conducted under the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals.

Note that the entries of `x` (and `y` if provided) must be non-negative integers.

Implements: `pvalue`, `confint`

source

15.4 HypothesisTests.SignedRankTest

`HypothesisTests.SignedRankTest` — *Method.*

```
SignedRankTest(x::AbstractVector{<:Real})
SignedRankTest(x::AbstractVector{<:Real}, y::AbstractVector{T<:Real})
```

Perform a Wilcoxon signed rank test of the null hypothesis that the distribution of `x` (or the difference `x - y` if `y` is provided) has zero median against the alternative hypothesis that the median is non-zero.

When there are no tied ranks and 50 samples, or tied ranks and 15 samples, `SignedRankTest` performs an exact signed rank test. In all other cases, `SignedRankTest` performs an approximate signed rank test. Behavior may be further controlled by using `ExactSignedRankTest` or `ApproximateSignedRankTest` directly.

Implements: `pvalue`, `confint`

source

15.5 HypothesisTests.pvalue

`HypothesisTests.pvalue` — *Function.*

```
pvalue(test::HypothesisTest; tail = :both)
```

Compute the p-value for a given significance test.

If `tail` is `:both` (default), then the p-value for the two-sided test is returned. If `tail` is `:left` or `:right`, then a one-sided test is performed.

source

15.6 HypothesisTests.pvalue

`HypothesisTests.pvalue` — *Method.*

```
pvalue(x::FisherExactTest; tail = :both, method = :central)
```

Compute the p-value for a given Fisher exact test.

The one-sided p-values are based on Fisher's non-central hypergeometric distribution $f(i)$ with odds ratio :

$$p^{(\text{left})} = \sum_{ia} f(i)$$

$$p^{(\text{right})} = \sum_{ia} f(i)$$

For `tail = :both`, possible values for `method` are:

- `:central` (default): Central interval, i.e. the p-value is two times the minimum of the one-sided p-values.
- `:minlike`: Minimum likelihood interval, i.e. the p-value is computed by summing all tables with the same marginals that are equally or less probable:

$$p = \sum_{f(i) f(a)} f(i)$$

References

- Gibbons, J.D., Pratt, J.W., P-values: Interpretation and Methodology, American Statistican, 29(1):20-25, 1975.
- Fay, M.P., Supplementary material to "Confidence intervals that match Fisher's exact or Blaker's exact tests". Biostatistics, Volume 11, Issue 2, 1 April 2010, Pages 373–374, link

source

15.7 HypothesisTests.testname

`HypothesisTests.testname` — *Method*.

```
testname(::HypothesisTest)
```

Returns the string value, e.g. "Binomial test" or "Sign Test".
source

15.8 StatsBase.confint

`StatsBase.confint` — *Function*.

```
confint(test::HypothesisTest, alpha = 0.05; tail = :both)
```

Compute a confidence interval C with coverage 1-`alpha`.

If `tail` is `:both` (default), then a two-sided confidence interval is returned. If `tail` is `:left` or `:right`, then a one-sided confidence interval is returned.

!!! note Most of the implemented confidence intervals are *strongly consistent*, that is, the confidence interval with coverage 1-`alpha` does not contain the test statistic under h_0 if and only if the corresponding test rejects the null hypothesis $h_0 : = {}_0$:

```
$$
    C (x, 1  ) = \{ : p_ (x) > \},
$$

where $p_$ is the [‘pvalue‘](HypothesisTests.md#HypothesisTests.pvalue) of the corresp
```

source

15.9 StatsBase.confint

`StatsBase.confint` — *Function*.

```
confint(test::BinomialTest, alpha = 0.05; tail = :both, method = :clopper_pearson)
```

Compute a confidence interval with coverage 1-`alpha` for a binomial proportion using one of the following methods. Possible values for `method` are:

- `:clopper_pearson` (default): Clopper-Pearson interval is based on the binomial distribution. The empirical coverage is never less than the nominal coverage of 1-`alpha`; it is usually too conservative.
- `:wald`: Wald (or normal approximation) interval relies on the standard approximation of the actual binomial distribution by a normal distribution. Coverage can be erratically poor for success probabilities close to zero or one.
- `:wilson`: Wilson score interval relies on a normal approximation. In contrast to `:wald`, the standard deviation is not approximated by an empirical estimate, resulting in good empirical coverages even for small numbers of draws and extreme success probabilities.
- `:jeffrey`: Jeffreys interval is a Bayesian credible interval obtained by using a non-informative Jeffreys prior. The interval is very similar to the Wilson interval.
- `:agresti_coull`: Agresti-Coull interval is a simplified version of the Wilson interval; both are centered around the same value. The Agresti Coull interval has higher or equal coverage.
- `:arcsine`: Confidence interval computed using the arcsine transformation to make $var(p)$ independent of the probability p.

References

- Brown, L.D., Cai, T.T., and DasGupta, A. Interval estimation for a binomial proportion. Statistical Science, 16(2):101–117, 2001.

External links

- Binomial confidence interval on Wikipedia

source

15.10 StatsBase.confint

`StatsBase.confint` — *Function.*

```
confint(x::FisherExactTest, alpha::Float64=0.05; tail=:both, method=:central)
```

Compute a confidence interval with coverage 1 - `alpha`. One-sided intervals are based on Fisher's non-central hypergeometric distribution. For `tail = :both`, the only `method` implemented yet is the central interval (`:central`).

!!! note Since the p-value is not necessarily unimodal, the corresponding confidence region might not be an interval.

References

- Gibbons, J.D, Pratt, J.W. P-values: Interpretation and Methodology, American Statistican, 29(1):20-25, 1975.
- Fay, M.P., Supplementary material to "Confidence intervals that match Fisher's exact or Blaker's exact tests". Biostatistics, Volume 11, Issue 2, 1 April 2010, Pages 373–374, link

source

15.11 StatsBase.confint

`StatsBase.confint` — *Function.*

```
confint(test::PowerDivergenceTest, alpha = 0.05; tail = :both, method = :sison_glaz)
```

Compute a confidence interval with coverage 1-`alpha` for multinomial proportions using one of the following methods. Possible values for `method` are:

- `:sison_glaz` (default): Sison-Glaz intervals
- `:bootstrap`: Bootstrap intervals
- `:quesenberry_hurst`: Quesenberry-Hurst intervals
- `:gold`: Gold intervals (asymptotic simultaneous intervals)

References

- Agresti, Alan. Categorical Data Analysis, 3rd Edition. Wiley, 2013.
- Sison, C.P and Glaz, J. Simultaneous confidence intervals and sample size determination for multinomial proportions. Journal of the American Statistical Association, 90:366-369, 1995.
- Quesensberry, C.P. and Hurst, D.C. Large Sample Simultaneous Confidence Intervals for Multinational Proportions. Technometrics, 6:191-195, 1964.
- Gold, R. Z. Tests Auxiliary to ^{2} Tests in a Markov Chain. Annals of Mathematical Statistics, 30:56-74, 1963.

source

Chapter 16

DataArrays

16.1 DataArrays.PooledDataVecs

`DataArrays.PooledDataVecs` — *Method*.

```
PooledDataVecs(v1, v2) -> (pda1, pda2)
```

Return a tuple of `PooledDataArrays` created from the data in `v1` and `v2`, respectively, but sharing a common value pool.

source

16.2 DataArrays.compact

`DataArrays.compact` — *Method*.

```
compact(d::PooledDataArray)
```

Return a `PooledDataArray` with the smallest possible reference type for the data in `d`.

!!! note If the reference type is already the smallest possible for the data, the input array is returned, i.e. the function *aliases* the input.

Examples

```
julia> p = @pdata(repeat(["A", "B"], outer=4))
8-element DataArrays.PooledDataArray{String,UInt32,1}:
 "A"
 "B"
 "A"
 "B"
 "A"
 "B"
 "A"
```

```
 "B"

julia> compact(p) # second type parameter compacts to UInt8 (only need 2 unique values)
8-element DataArrays.PooledDataArray{String,UInt8,1}:
 "A"
 "B"
 "A"
 "B"
 "A"
 "B"
 "A"
 "B"
```

source

16.3 DataArrays.cut

`DataArrays.cut` — *Method.*

```
cut(x::AbstractVector, breaks::Vector) -> PooledDataArray
cut(x::AbstractVector, ngroups::Integer) -> PooledDataArray
```

Divide the range of `x` into intervals based on the cut points specified in `breaks`, or into `ngroups` intervals of approximately equal length.

Examples

```
julia> cut([1, 2, 3, 4], [1, 3])
4-element DataArrays.PooledDataArray{String,UInt32,1}:
 "[1,3]"
 "[1,3]"
 "[1,3]"
 "(3,4]"
```

source

16.4 DataArrays.data

`DataArrays.data` — *Method.*

```
data(a::AbstractArray) -> DataArray
```

Convert `a` to a `DataArray`.

Examples

```
julia> data([1, 2, 3])
3-element DataArrays.DataArray{Int64,1}:
 1
 2
 3

julia> data(@data [1, 2, NA])
3-element DataArrays.DataArray{Int64,1}:
 1
 2
  NA
```

source

16.5 DataArrays.dropna

`DataArrays.dropna` — *Method*.

```
dropna(v::AbstractVector) -> AbstractVector
```

Return a copy of `v` with all `NA` elements removed.

Examples

```
julia> dropna(@data [NA, 1, NA, 2])
2-element Array{Int64,1}:
 1
 2

julia> dropna([4, 5, 6])
3-element Array{Int64,1}:
 4
 5
 6
```

source

16.6 DataArrays.getpoolidx

`DataArrays.getpoolidx` — *Method*.

```
getpoolidx(pda::PooledDataArray, val)
```

Return the index of `val` in the value pool for `pda`. If `val` is not already in the value pool, `pda` is modified to include it in the pool.

source

16.7 DataArrays.gl

`DataArrays.gl` — *Method.*

```
gl(n::Integer, k::Integer, l::Integer = n*k) -> PooledDataArray
```

Generate a `PooledDataArray` with `n` levels and `k` replications, optionally specifying an output length `l`. If specified, `l` must be a multiple of `n*k`.

Examples

```
julia> gl(2, 1)
2-element DataArrays.PooledDataArray{Int64,UInt8,1}:
 1
 2

julia> gl(2, 1, 4)
4-element DataArrays.PooledDataArray{Int64,UInt8,1}:
 1
 2
 1
 2
```

source

16.8 DataArrays.isna

`DataArrays.isna` — *Method.*

```
isna(x) -> Bool
```

Determine whether `x` is missing, i.e. `NA`.

Examples

```
julia> isna(1)
false

julia> isna(NA)
true
```

source

16.9 DataArrays.isna

`DataArrays.isna` — *Method.*

```
isna(a::AbstractArray, i) -> Bool
```

Determine whether the element of `a` at index `i` is missing, i.e. `NA`.

Examples

```
julia> X = @data [1, 2, NA];

julia> isna(X, 2)
false

julia> isna(X, 3)
true
```

source

16.10 DataArrays.levels

`DataArrays.levels` — *Method*.

```
levels(da::DataArray) -> DataVector
```

Return a vector of the unique values in `da`, excluding any `NA`s.

```
levels(a::AbstractArray) -> Vector
```

Equivalent to `unique(a)`.

Examples

```
julia> levels(@data [1, 2, NA])
2-element DataArrays.DataArray{Int64,1}:
 1
 2
```

source

16.11 DataArrays.padna

`DataArrays.padna` — *Method*.

```
padna(dv::AbstractDataVector, front::Integer, back::Integer) -> DataVector
```

Pad `dv` with `NA` values. `front` is an integer number of `NA`s to add at the beginning of the array and `back` is the number of `NA`s to add at the end.

Examples

```
julia> padna(@data([1, 2, 3]), 1, 2)
6-element DataArrays.DataArray{Int64,1}:
  NA
 1
```

```
 2
 3
  NA
  NA
```

source

16.12 DataArrays.reorder

`DataArrays.reorder` — *Method.*

```
reorder(x::PooledDataArray) -> PooledDataArray
```

Return a `PooledDataArray` containing the same data as `x` but with the value pool sorted.

source

16.13 DataArrays.replace!

`DataArrays.replace!` — *Method.*

```
replace!(x::PooledDataArray, from, to)
```

Replace all occurrences of `from` in `x` with `to`, modifying `x` in place.

source

16.14 DataArrays.setlevels!

`DataArrays.setlevels!` — *Method.*

```
setlevels!(x::PooledDataArray, newpool::Union{AbstractVector, Dict})
```

Set the value pool for the `PooledDataArray` `x` to `newpool`, modifying `x` in place. The values can be replaced using a mapping specified in a `Dict` or with an array, since the order of the levels is used to identify values. The pool can be enlarged to contain values not present in the data, but it cannot be reduced to exclude present values.

Examples

```
julia> p = @pdata repeat(["A", "B"], inner=3)
6-element DataArrays.PooledDataArray{String,UInt32,1}:
 "A"
 "A"
 "A"
 "B"
 "B"
```

```
 "B"

julia> setlevels!(p, Dict("A"=>"C"));

julia> p # has been modified
6-element DataArrays.PooledDataArray{String,UInt32,1}:
 "C"
 "C"
 "C"
 "B"
 "B"
 "B"
```

source

16.15 DataArrays.setlevels

`DataArrays.setlevels` — *Method.*

```
setlevels(x::PooledDataArray, newpool::Union{AbstractVector, Dict})
```

Create a new `PooledDataArray` based on `x` but with the new value pool specified by `newpool`. The values can be replaced using a mapping specified in a `Dict` or with an array, since the order of the levels is used to identify values. The pool can be enlarged to contain values not present in the data, but it cannot be reduced to exclude present values.

Examples

```
julia> p = @pdata repeat(["A", "B"], inner=3)
6-element DataArrays.PooledDataArray{String,UInt32,1}:
 "A"
 "A"
 "A"
 "B"
 "B"
 "B"

julia> p2 = setlevels(p, ["C", "D"]) # could also be Dict("A"=>"C", "B"=>"D")
6-element DataArrays.PooledDataArray{String,UInt32,1}:
 "C"
 "C"
 "C"
 "D"
 "D"
 "D"
```

```
julia> p3 = setlevels(p2, ["C", "D", "E"])
6-element DataArrays.PooledDataArray{String,UInt32,1}:
 "C"
 "C"
 "C"
 "D"
 "D"
 "D"

julia> p3.pool # the pool can contain values not in the array
3-element Array{String,1}:
 "C"
 "D"
 "E"
```

source

Chapter 17

GLM

17.1 GLM.canonicallink

`GLM.canonicallink` — *Function.*

```
canonicallink(D::Distribution)
```

Return the canonical link for distribution `D`, which must be in the exponential family.

Examples

```
julia> canonicallink(Bernoulli())
GLM.LogitLink()
```

source

17.2 GLM.devresid

`GLM.devresid` — *Method.*

```
devresid(D, y, )
```

Return the squared deviance residual of from `y` for distribution `D`

The deviance of a GLM can be evaluated as the sum of the squared deviance residuals. This is the principal use for these values. The actual deviance residual, say for plotting, is the signed square root of this value

[] sign(y -) * sqrt(devresid(D, y,))

Examples

```
julia> showcompact(devresid(Normal(), 0, 0.25))     # abs2(y - )
0.0625
julia> showcompact(devresid(Bernoulli(), 1, 0.75))  # -2log() when y == 1
0.575364
```

```
julia> showcompact(devresid(Bernoulli(), 0, 0.25))  # -2log1p(-) = -2log(1-) when y ==
0.575364
```

source

17.3 GLM.ftest

`GLM.ftest` — *Method.*

```
ftest(mod::LinearModel...; atol=0::Real)
```

For each sequential pair of linear predictors in `mod`, perform an F-test to determine if the first one fits significantly better than the next.

A table is returned containing residual degrees of freedom (DOF), degrees of freedom, difference in DOF from the preceding model, sum of squared residuals (SSR), difference in SSR from the preceding model, R, difference in R from the preceding model, and F-statistic and p-value for the comparison between the two models.

!!! note This function can be used to perform an ANOVA by testing the relative fit of two models to the data

Optional keyword argument `atol` controls the numerical tolerance when testing whether the models are nested.

Examples

Suppose we want to compare the effects of two or more treatments on some result. Because this is an ANOVA, our null hypothesis is that `Result ~ 1` fits the data as well as `Result ~ 1 + Treatment`.

```
julia> dat = DataFrame(Treatment=[1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2.],
                       Result=[1.1, 1.2, 1, 2.2, 1.9, 2, .9, 1, 1, 2.2, 2, 2],
                       Other=[1, 1, 2, 1, 2, 1, 3, 1, 1, 2, 2, 1]);

julia> mod = lm(@formula(Result ~ 1 + Treatment), dat);

julia> nullmod = lm(@formula(Result ~ 1), dat);

julia> bigmod = lm(@formula(Result ~ 1 + Treatment + Other), dat);

julia> ft = ftest(mod.model, nullmod.model)
        Res. DOF DOF DOF    SSR    SSR     R    R       F* p(>F)
Model 1       10   3      0.1283         0.9603
Model 2       11   2   -1 3.2292 -3.1008 0.0000 0.9603 241.6234 <1e-7


julia> ftest(bigmod.model, mod.model, nullmod.model)
        Res. DOF DOF DOF    SSR    SSR     R    R       F*  p(>F)
Model 1        9   4      0.1038         0.9678
```

```
Model 2       10   3   -1 0.1283 -0.0245 0.9603 0.0076   2.1236 0.1790
Model 3       11   2   -1 3.2292 -3.1008 0.0000 0.9603 241.6234  <1e-7
```

source

17.4 GLM.glmvar

`GLM.glmvar` — *Function.*

```
glmvar(D::Distribution, )
```

Return the value of the variance function for `D` at

The variance of `D` at is the product of the dispersion parameter, , which does not depend on and the value of `glmvar`. In other words `glmvar` returns the factor of the variance that depends on .

Examples

```
julia>  = inv(6):inv(3):1; showcompact(collect())
[0.166667, 0.5, 0.833333]
julia> showcompact(glmvar.(Normal(), ))    # constant for Normal()
[1.0, 1.0, 1.0]
julia> showcompact(glmvar.(Bernoulli(), )) #  * (1 - ) for Bernoulli()
[0.138889, 0.25, 0.138889]
julia> showcompact(glmvar.(Poisson(), ))   #  for Poisson()
[0.166667, 0.5, 0.833333]
```

source

17.5 GLM.inverselink

`GLM.inverselink` — *Function.*

```
inverselink(L::Link, )
```

Return a 3-tuple of the inverse link, the derivative of the inverse link, and when appropriate, the variance function `*(1 - )`.

The variance function is returned as NaN unless the range of is (0, 1)

Examples

```
julia> showcompact(inverselink(LogitLink(), 0.0))
(0.5, 0.25, 0.25)
julia> showcompact(inverselink(CloglogLink(), 0.0))
(0.632121, 0.367879, 0.232544)
julia> showcompact(inverselink(LogLink(), 2.0))
(7.38906, 7.38906, NaN)
```

source

17.6 GLM.linkfun

`GLM.linkfun` — *Function.*

`linkfun(L::Link, )`

Return , the value of the linear predictor for link `L` at mean .
Examples

```
julia>  = inv(10):inv(5):1
0.1:0.2:0.9

julia> show(linkfun.(LogitLink(), ))
[-2.19722, -0.847298, 0.0, 0.847298, 2.19722]
```

source

17.7 GLM.linkinv

`GLM.linkinv` — *Function.*

`linkinv(L::Link, )`

Return , the mean value, for link `L` at linear predictor value .
Examples

```
julia>  = inv(10):inv(5):1; showcompact(collect())
[0.1, 0.3, 0.5, 0.7, 0.9]
julia>  = logit.(); showcompact()
[-2.19722, -0.847298, 0.0, 0.847298, 2.19722]
julia> showcompact(linkinv.(LogitLink(), ))
[0.1, 0.3, 0.5, 0.7, 0.9]
```

source

17.8 GLM.mueta

`GLM.mueta` — *Function.*

`mueta(L::Link, )`

Return the derivative of `linkinv`, `d/d`, for link `L` at linear predictor value .
Examples

```
julia> showcompact(mueta(LogitLink(), 0.0))
0.25
julia> showcompact(mueta(CloglogLink(), 0.0))
0.367879
julia> showcompact(mueta(LogLink(), 2.0))
7.38906
```

source

17.9 GLM.mustart

`GLM.mustart` — *Function*.

```
mustart(D::Distribution, y, wt)
```

Return a starting value for .

For some distributions it is appropriate to set = y to initialize the IRLS algorithm but for others, notably the Bernoulli, the values of y are not allowed as values of and must be modified.

Examples

```
julia> showcompact(mustart(Bernoulli(), 0.0, 1))
0.25
julia> showcompact(mustart(Bernoulli(), 1.0, 1))
0.75
julia> showcompact(mustart(Binomial(), 0.0, 10))
0.0454545
julia> showcompact(mustart(Normal(), 0.0, 1))
0.0
```

source

17.10 GLM.update!

`GLM.update!` — *Method*.

```
update!{T<:FPVector}(r::GlmResp{T}, linPr::T)
```

Update the mean, working weights and working residuals, in `r` given a value of the linear predictor, `linPr`.

source

17.11 GLM.wrkresp

`GLM.wrkresp` — *Method*.

```
wrkresp(r::GlmResp)
```

The working response, `r.eta + r.wrkresid - r.offset`.

source

17.12 StatsBase.deviance

`StatsBase.deviance` — *Method*.

```
deviance(obj::LinearModel)
```

For linear models, the deviance is equal to the residual sum of squares (RSS).

source

17.13 StatsBase.nobs

`StatsBase.nobs` — *Method.*

```
nobs(obj::LinearModel)
nobs(obj::GLM)
```

For linear and generalized linear models, returns the number of rows, or, when prior weights are specified, the sum of weights.

source

17.14 StatsBase.nulldeviance

`StatsBase.nulldeviance` — *Method.*

```
nulldeviance(obj::LinearModel)
```

For linear models, the deviance of the null model is equal to the total sum of squares (TSS).

source

17.15 StatsBase.predict

`StatsBase.predict` — *Function.*

```
predict(mm::LinearModel, newx::AbstractMatrix, interval_type::Symbol, level::Real = 0.
```

Specifying `interval_type` will return a 3-column matrix with the prediction and the lower and upper confidence bounds for a given `level` (0.95 equates alpha = 0.05). Valid values of `interval_type` are `:confint` delimiting the uncertainty of the predicted relationship, and `:predint` delimiting estimated bounds for new data points.

source

17.16 StatsBase.predict

`StatsBase.predict` — *Method.*

```
predict(mm::AbstractGLM, newX::AbstractMatrix; offset::FPVector=Vector{eltype(newX)}(0
```

Form the predicted response of model `mm` from covariate values `newX` and, optionally, an offset.

source

Chapter 18

Documenter

18.1 Documenter.deploydocs

`Documenter.deploydocs` — *Method.*

```
deploydocs(
    root   = "<current-directory>",
    target = "site",
    repo   = "<required>",
    branch = "gh-pages",
    latest = "master",
    osname = "linux",
    julia  = "nightly",
    deps   = <Function>,
    make   = <Function>,
)
```

Converts markdown files generated by `makedocs` to HTML and pushes them to `repo`. This function should be called from within a package's `docs/make.jl` file after the call to `makedocs`, like so

[] using Documenter, PACKAGE$_N AMEmakedocs(options...)deploydocs(repo =$"$*github.com/...*$")$

Keywords

`root` has the same purpose as the `root` keyword for `makedocs`.

`target` is the directory, relative to `root`, where generated HTML content should be written to. This directory **must** be added to the repository's `.gitignore` file. The default value is `"site"`.

`repo` is the remote repository where generated HTML content should be pushed to. Do not specify any protocol - "https://" or "git@" should not be present. This keyword *must* be set and will throw an error when left undefined. For example this package uses the following `repo` value:

[] repo = "github.com/JuliaDocs/Documenter.jl.git"

`branch` is the branch where the generated documentation is pushed. If the branch does not exist, a new orphaned branch is created automatically. It defaults to `"gh-pages"`.

`latest` is the branch that "tracks" the latest generated documentation. By default this value is set to `"master"`.

`osname` is the operating system which will be used to deploy generated documentation. This defaults to `"linux"`. This value must be one of those specified in the `os:` section of the `.travis.yml` configuration file.

`julia` is the version of Julia that will be used to deploy generated documentation. This defaults to `"nightly"`. This value must be one of those specified in the `julia:` section of the `.travis.yml` configuration file.

`deps` is the function used to install any dependencies needed to build the documentation. By default this function installs `pygments` and `mkdocs` using the `Deps.pip` function:

[] deps = Deps.pip("pygments", "mkdocs")

`make` is the function used to convert the markdown files to HTML. By default this just runs `mkdocs build` which populates the `target` directory.

See Also

The Hosting Documentation section of the manual provides a step-by-step guide to using the `deploydocs` function to automatically generate docs and push then to GitHub.

source

18.2 Documenter.hide

`Documenter.hide` — *Method.*

[] hide(root, children)

Allows a subsection of pages to be hidden from the navigation menu. `root` will be linked to in the navigation menu, with the title determined as usual. `children` should be a list of pages (note that it **can not** be hierarchical).

Usage

[] makedocs(..., pages = [..., hide("Hidden section" =¿ "hidden*i*ndex.md", ["*hidden*1.*md*","*Hidden*

source

18.3 Documenter.hide

`Documenter.hide` — *Method.*

[] hide(page)

Allows a page to be hidden in the navigation menu. It will only show up if it happens to be the current page. The hidden page will still be present in the linear page list that can be accessed via the previous and next page links. The title of the hidden page can be overriden using the `=>` operator as usual.

Usage

[] makedocs(..., pages = [..., hide("page1.md"), hide("Title" =¿ "page2.md")])

source

18.4 Documenter.makedocs

`Documenter.makedocs` — *Method*.

```
makedocs(
    root    = "<current-directory>",
    source  = "src",
    build   = "build",
    clean   = true,
    doctest = true,
    modules = Module[],
    repo    = "",
)
```

Combines markdown files and inline docstrings into an interlinked document. In most cases `makedocs` should be run from a `make.jl` file:

[] using Documenter makedocs(keywords...)

which is then run from the command line with:

[] *juliamake.jl*

The folder structure that `makedocs` expects looks like:

```
docs/
    build/
    src/
    make.jl
```

Keywords

`root` is the directory from which `makedocs` should run. When run from a `make.jl` file this keyword does not need to be set. It is, for the most part, needed when repeatedly running `makedocs` from the Julia REPL like so:

```
julia> makedocs(root = Pkg.dir("MyPackage", "docs"))
```

`source` is the directory, relative to `root`, where the markdown source files are read from. By convention this folder is called `src`. Note that any non-markdown files stored in `source` are copied over to the build directory when `makedocs` is run.

`build` is the directory, relative to `root`, into which generated files and folders are written when `makedocs` is run. The name of the build directory is, by convention, called `build`, though, like with `source`, users are free to change this to anything else to better suit their project needs.

`clean` tells `makedocs` whether to remove all the content from the `build` folder prior to generating new content from `source`. By default this is set to `true`.

`doctest` instructs `makedocs` on whether to try to test Julia code blocks that are encountered in the generated document. By default this keyword is set to `true`. Doctesting should only ever be disabled when initially setting up a newly developed package where the developer is just trying to get their package and documentation structure correct. After that, it's encouraged to always make sure that documentation examples are runnable and produce the expected results. See the Doctests manual section for details about running doctests.

`modules` specifies a vector of modules that should be documented in `source`. If any inline docstrings from those modules are seen to be missing from the generated content then a warning will be printed during execution of `makedocs`. By default no modules are passed to `modules` and so no warnings will appear. This setting can be used as an indicator of the "coverage" of the generated documentation. For example Documenter's `make.jl` file contains:

[] makedocs(modules = [Documenter], ...)

and so any docstring from the module `Documenter` that is not spliced into the generated documentation in `build` will raise a warning.

`repo` specifies a template for the "link to source" feature. If you are using GitHub, this is automatically generated from the remote. If you are using a different host, you can use this option to tell Documenter how URLs should be generated. The following placeholders will be replaced with the respective value of the generated link:

- {`commit`} Git branch or tag name, or commit hash
- {`path`} Path to the file in the repository
- {`line`} Line (or range of lines) in the source file

For example if you are using GitLab.com, you could use

[] makedocs(repo = "https://gitlab.com/user/project/blob/{commit}{path}L{line}")

Experimental keywords

In addition to standard arguments there is a set of non-finalized experimental keyword arguments. The behaviour of these may change or they may be removed without deprecation when a minor version changes (i.e. except in patch releases).

`checkdocs` instructs `makedocs` to check whether all names within the modules defined in the `modules` keyword that have a docstring attached have the docstring also listed in the manual (e.g. there's a `@docs` blocks with that docstring). Possible values are `:all` (check all names) and `:exports` (check only exported names). The default value is `:none`, in which case no checks are performed. If `strict` is also enabled then the build will fail if any missing docstrings are encountered.

`linkcheck` – if set to `true` `makedocs` uses `curl` to check the status codes of external-pointing links, to make sure that they are up-to-date. The links and their status codes are printed to the standard output. If `strict` is also enabled then the build will fail if there are any broken (400+ status code) links. Default: `false`.

`linkcheck_ignore` allows certain URLs to be ignored in `linkcheck`. The values should be a list of strings (which get matched exactly) or `Regex` objects. By default nothing is ignored.

`strict` – `makedocs` fails the build right before rendering if it encountered any errors with the document in the previous build phases.

Non-MkDocs builds

Documenter also has (experimental) support for native HTML and LaTeX builds. These can be enabled using the `format` keyword and they generally require additional keywords be defined, depending on the format. These keywords are also currently considered experimental.

`format` allows the output format to be specified. Possible values are `:html`, `:latex` and `:markdown` (default).

Other keywords related to non-MkDocs builds (`assets`, `sitename`, `analytics`, `authors`, `pages`, `version`) should be documented at the respective `*Writer` modules (`Writers.HTMLWriter`, `Writers.LaTeXWriter`).

See Also

A guide detailing how to document a package using Documenter's `makedocs` is provided in the Usage section of the manual.

source

Chapter 19

ColorTypes

19.1 ColorTypes.alpha

`ColorTypes.alpha` — *Method*.

`alpha(p)` extracts the alpha component of a color. For a color without an alpha channel, it will always return 1.

source

19.2 ColorTypes.alphacolor

`ColorTypes.alphacolor` — *Function*.

`alphacolor(RGB)` returns `ARGB`, i.e., the corresponding transparent color type with storage order (alpha, color).

source

19.3 ColorTypes.base_color_type

`ColorTypes.base_color_type` — *Method*.

`base_color_type` is similar to `color_type`, except it "strips off" the element type. For example,

```
color_type(RGB{N0f8})      == RGB{N0f8}
base_color_type(RGB{N0f8}) == RGB
```

This can be very handy if you want to switch element types. For example:

```
c64 = base_color_type(c){Float64}(color(c))
```

converts `c` into a `Float64` representation (potentially discarding any alpha-channel information).

source

19.4 ColorTypes.base_colorant_type

`ColorTypes.base_colorant_type` — *Method*.

`base_colorant_type` is similar to `base_color_type`, but it preserves the "alpha" portion of the type.

For example,

```
base_color_type(ARGB{N0f8})  == RGB
base_colorant_type(ARGB{N0f8})  == ARGB
```

If you just want to switch element types, this is the safest default and the easiest to use:

```
c64 = base_colorant_type(c){Float64}(c)
```

source

19.5 ColorTypes.blue

`ColorTypes.blue` — *Method*.

`blue(c)` returns the blue component of an `AbstractRGB` opaque or transparent color.

source

19.6 ColorTypes.ccolor

`ColorTypes.ccolor` — *Method*.

`ccolor` ("concrete color") helps write flexible methods. The idea is that users may write `convert(HSV, c)` or even `convert(Array{HSV}, A)` without specifying the element type explicitly (e.g., `convert(Array{HSV{Float32}}, A)`). `ccolor` implements the logic "choose the user's eltype if specified, otherwise retain the eltype of the source object." However, when the source object has FixedPoint element type, and the destination only supports AbstractFloat, we choose Float32.

Usage:

```
ccolor(desttype, srctype) -> concrete desttype
```

Example:

```
convert{C<:Colorant}(::Type{C}, p::Colorant) = cnvt(ccolor(C,typeof(p)), p)
```

where `cnvt` is the function that performs explicit conversion.

source

19.7 ColorTypes.color

`ColorTypes.color` — *Method*.

`color(c)` extracts the opaque color component from a Colorant (e.g., omits the alpha channel, if present).

source

19.8 ColorTypes.color_type

`ColorTypes.color_type` — *Method*.

`color_type(c)` or `color_type(C)` (`c` being a color instance and `C` being the type) returns the type of the Color object (without alpha channel). This, and related functions like `base_color_type`, `base_colorant_type`, and `ccolor` are useful for manipulating types for writing generic code.

For example,

```
color_type(RGB)           == RGB
color_type(RGB{Float32}) == RGB{Float32}
color_type(ARGB{N0f8})     == RGB{N0f8}
```

source

19.9 ColorTypes.coloralpha

`ColorTypes.coloralpha` — *Function*.

`coloralpha(RGB)` returns `RGBA`, i.e., the corresponding transparent color type with storage order (color, alpha).

source

19.10 ColorTypes.comp1

`ColorTypes.comp1` — *Method*.

`comp1(c)` extracts the first component you'd pass to the constructor of the corresponding object. For most color types without an alpha channel, this is just the first field, but for types like `BGR` that reverse the internal storage order this provides the value that you'd use to reconstruct the color.

Specifically, for any `Color{T,3}`,

```
c == typeof(c)(comp1(c), comp2(c), comp3(c))
```

returns true.

source

19.11 ColorTypes.comp2

`ColorTypes.comp2` — *Method*.

`comp2(c)` extracts the second constructor argument (see `comp1`).

source

19.12 ColorTypes.comp3

`ColorTypes.comp3` — *Method*.

`comp3(c)` extracts the third constructor argument (see `comp1`).

source

19.13 ColorTypes.gray

`ColorTypes.gray` — *Method*.

`gray(c)` returns the gray component of a grayscale opaque or transparent color.

source

19.14 ColorTypes.green

`ColorTypes.green` — *Method*.

`green(c)` returns the green component of an `AbstractRGB` opaque or transparent color.

source

19.15 ColorTypes.mapc

`ColorTypes.mapc` — *Method*.

```
mapc(f, rgb) -> rgbf
mapc(f, rgb1, rgb2) -> rgbf
```

`mapc` applies the function `f` to each color channel of the input color(s), returning an output color in the same colorspace.

Examples:

```
julia> mapc(x->clamp(x,0,1), RGB(-0.2,0.3,1.2))
RGB{Float64}(0.0,0.3,1.0)

julia> mapc(max, RGB(0.1,0.8,0.3), RGB(0.5,0.5,0.5))
RGB{Float64}(0.5,0.8,0.5)

julia> mapc(+, RGB(0.1,0.8,0.3), RGB(0.5,0.5,0.5))
RGB{Float64}(0.6,1.3,0.8)
```

source

19.16 ColorTypes.mapreducec

`ColorTypes.mapreducec` — *Method*.

```
mapreducec(f, op, v0, c)
```

Reduce across color channels of `c` with the binary operator `op`, first applying `f` to each channel. `v0` is the neutral element used to initiate the reduction. For grayscale,

```
mapreducec(f, op, v0, c::Gray) = op(v0, f(comp1(c)))
```

whereas for RGB

```
mapreducec(f, op, v0, c::RGB) = op(f(comp3(c)), op(f(comp2(c)), op(v0, f(comp1(c)))))
```

If `c` has an alpha channel, it is always the last one to be folded into the reduction.

source

19.17 ColorTypes.red

`ColorTypes.red` — *Method*.

`red(c)` returns the red component of an `AbstractRGB` opaque or transparent color.

source

19.18 ColorTypes.reducec

`ColorTypes.reducec` — *Method*.

```
reducec(op, v0, c)
```

Reduce across color channels of `c` with the binary operator `op`. `v0` is the neutral element used to initiate the reduction. For grayscale,

```
reducec(op, v0, c::Gray) = op(v0, comp1(c))
```

whereas for RGB

```
reducec(op, v0, c::RGB) = op(comp3(c), op(comp2(c), op(v0, comp1(c))))
```

If `c` has an alpha channel, it is always the last one to be folded into the reduction.

source

Chapter 20

Primes

20.1 Primes.factor

`Primes.factor` — *Method*.

```
factor(ContainerType, n::Integer) -> ContainerType
```

Return the factorization of `n` stored in a `ContainerType`, which must be a subtype of `Associative` or `AbstractArray`, a `Set`, or an `IntSet`.

[] julia¿ factor(DataStructures.SortedDict, 100) DataStructures.SortedDict{Int64,Int64,Base.Order.ForwardOrder with 2 entries: 2 =¿ 2 5 =¿ 2

When `ContainerType <: AbstractArray`, this returns the list of all prime factors of `n` with multiplicities, in sorted order.

[] julia¿ factor(Vector, 100) 4-element Array{Int64,1}: 2 2 5 5

julia¿ prod(factor(Vector, 100)) == 100 true

When `ContainerType == Set`, this returns the distinct prime factors as a set.

[] julia¿ factor(Set, 100) Set([2,5])

source

20.2 Primes.factor

`Primes.factor` — *Method*.

```
factor(n::Integer) -> Primes.Factorization
```

Compute the prime factorization of an integer `n`. The returned object, of type `Factorization`, is an associative container whose keys correspond to the factors, in sorted order. The value associated with each key indicates the multiplicity (i.e. the number of times the factor appears in the factorization).

[] julia¿ factor(100) 2252

For convenience, a negative number `n` is factored as `-1*(-n)` (i.e. `-1` is considered to be a factor), and `0` is factored as `0^1`:

[] julia¿ factor(-9) -1 32

julia¿ factor(0) 0

julia¿ collect(factor(0)) 1-element Array{Pair{Int64,Int64},1}: 0=¿1

source

20.3 Primes.ismersenneprime

`Primes.ismersenneprime` — *Method.*

```
ismersenneprime(M::Integer; [check::Bool = true]) -> Bool
```

Lucas-Lehmer deterministic test for Mersenne primes. `M` must be a Mersenne number, i.e. of the form `M = 2^p - 1`, where `p` is a prime number. Use the keyword argument `check` to enable/disable checking whether `M` is a valid Mersenne number; to be used with caution. Return `true` if the given Mersenne number is prime, and `false` otherwise.

```
julia> ismersenneprime(2^11 - 1)
false

julia> ismersenneprime(2^13 - 1)
true
```

source

20.4 Primes.isprime

`Primes.isprime` — *Function.*

```
isprime(x::BigInt, [reps = 25]) -> Bool
```

Probabilistic primality test. Returns `true` if `x` is prime with high probability (pseudoprime); and `false` if `x` is composite (not prime). The false positive rate is about `0.25^reps`. `reps = 25` is considered safe for cryptographic applications (Knuth, Seminumerical Algorithms).

[] julia¿ isprime(big(3)) true

source

20.5 Primes.isprime

`Primes.isprime` — *Method.*

```
isprime(n::Integer) -> Bool
```

Returns `true` if `n` is prime, and `false` otherwise.

[] julia¿ isprime(3) true

source

20.6 Primes.isrieselprime

`Primes.isrieselprime` — *Method.*

```
isrieselprime(k::Integer, Q::Integer) -> Bool
```

Lucas-Lehmer-Riesel deterministic test for N of the form `N = k * Q`, with `0 < k < Q`, `Q = 2^n - 1` and `n > 0`, also known as Riesel primes. Returns `true` if `R` is prime, and `false` otherwise or if the combination of k and n is not supported.

```
julia> isrieselprime(1, 2^11 - 1)  # == ismersenneprime(2^11 - 1)
false

julia> isrieselprime(3, 2^607 - 1)
true
```

source

20.7 Primes.nextprime

`Primes.nextprime` — *Function.*

```
nextprime(n::Integer, i::Integer=1)
```

The `i`-th smallest prime not less than `n` (in particular, `nextprime(p) == p` if `p` is prime). If `i < 0`, this is equivalent to prevprime(n, -i). Note that for `n::BigInt`, the returned number is only a pseudo-prime (the function `isprime` is used internally). See also `prevprime`.

```
julia> nextprime(4)
5

julia> nextprime(5)
5

julia> nextprime(4, 2)
7

julia> nextprime(5, 2)
7
```

source

20.8 Primes.prevprime

`Primes.prevprime` — *Function*.

```
prevprime(n::Integer, i::Integer=1)
```

The `i`-th largest prime not greater than `n` (in particular `prevprime(p) == p` if `p` is prime). If `i < 0`, this is equivalent to `nextprime(n, -i)`. Note that for `n::BigInt`, the returned number is only a pseudo-prime (the function `isprime` is used internally). See also `nextprime`.

```
julia> prevprime(4)
3

julia> prevprime(5)
5

julia> prevprime(5, 2)
3
```

source

20.9 Primes.prime

`Primes.prime` — *Method*.

```
prime{T}(::Type{T}=Int, i::Integer)
```

The `i`-th prime number.

```
julia> prime(1)
2

julia> prime(3)
5
```

source

20.10 Primes.primes

`Primes.primes` — *Method*.

```
primes([lo,] hi)
```

Returns a collection of the prime numbers (from `lo`, if specified) up to `hi`.
source

20.11 Primes.primesmask

`Primes.primesmask` — *Method.*

```
primesmask([lo,] hi)
```

Returns a prime sieve, as a `BitArray`, of the positive integers (from `lo`, if specified) up to `hi`. Useful when working with either primes or composite numbers.

source

20.12 Primes.prodfactors

`Primes.prodfactors` — *Function.*

```
prodfactors(factors)
```

Compute `n` (or the radical of `n` when `factors` is of type `Set` or `IntSet`) where `factors` is interpreted as the result of `factor(typeof(factors), n)`. Note that if `factors` is of type `AbstractArray` or `Primes.Factorization`, then `prodfactors` is equivalent to `Base.prod`.

```
julia> prodfactors(factor(100))
100
```

source

20.13 Primes.radical

`Primes.radical` — *Method.*

```
radical(n::Integer)
```

Compute the radical of `n`, i.e. the largest square-free divisor of `n`. This is equal to the product of the distinct prime numbers dividing `n`.

```
julia> radical(2*2*3)
6
```

source

20.14 Primes.totient

`Primes.totient` — *Method.*

```
totient(n::Integer) -> Integer
```

Compute the Euler totient function (n), which counts the number of positive integers less than or equal to n that are relatively prime to n (that is, the number of positive integers `m  n` with `gcd(m, n) == 1`). The totient function of `n` when `n` is negative is defined to be `totient(abs(n))`.

source

20.15 Primes.totient

`Primes.totient` — *Method.*

```
totient(f::Factorization{T}) -> T
```

Compute the Euler totient function of the number whose prime factorization is given by `f`. This method may be preferable to `totient(::Integer)` when the factorization can be reused for other purposes.

source

Chapter 21

Roots

21.1 Roots.D

`Roots.D` — *Function*.

Take derivative of order `k` of a function.
Arguments:

- `f::Function`: a mathematical function from R to R.
- `k::Int=1`: A non-negative integer specifying the order of the derivative. Values larger than 8 can be slow to compute.

Wrapper around `derivative` function in `ForwardDiff`
source

21.2 Roots.find_zero

`Roots.find_zero` — *Method*.

Find a zero of a univariate function using one of several different methods.
Positional arugments:

- `f` a function, callable object, or tuple of same. A tuple is used to pass in derivatives, as desired. Most methods are derivative free. Some (`Newton`, `Halley`) may have derivative(s) computed using the `ForwardDiff` pacakge.
- `x0` an initial starting value. Typically a scalar, but may be a two-element tuple or array for bisection methods. The value `float.(x0)` is passed on.
- `method` one of several methods, see below.

Keyword arguments:

- `xabstol=zero()`: declare convergence if |x_n - x_{n-1}| <= max(xabstol, max(1, |x_n|) * xreltol)

- `xreltol=eps()`:
- `abstol=zero()`: declare convergence if |f(x_n)| <= max(abstol, max(1, |x_n|) * reltol)
- `reltol`:
- `bracket`: Optional. A bracketing interval for the sought after root. If given, a hybrid algorithm may be used where bisection is utilized for steps that would go out of bounds. (Using a `FalsePosition` method instead would be suggested.)
- `maxevals::Int=40`: stop trying after `maxevals` steps
- `maxfnevals::Int=typemax(Int)`: stop trying after `maxfnevals` function evaluations
- `verbose::Bool=false`: If `true` show information about algorithm and a trace.

Returns:

Returns `xn` if the algorithm converges. If the algorithm stops, returns `xn` if |f(xn)| ^(2/3), where = reltol, otherwise a `ConvergenceFailed` error is thrown.

Exported methods:

`Bisection()`; `Order0()` (heuristic, slow more robust); `Order1()` (also `Secant()`); `Order2()` (also `Steffensen()`); `Order5()` (KSS); `Order8()` (Thukral); `Order16()` (Thukral); `FalsePosition(i)` (false position, i in 1..12);

Not exported:

`Secant()`, use `Order1()` `Steffensen()` use `Order2()` `Newton()` (use `newton()` function) `Halley()` (use `halley()` function)

The order 0 method is more robust to the initial starting point, but can utilize many more function calls. The higher order methods may be of use when greater precision is desired.‘

Examples:

```
f(x) = x^5 - x - 1
find_zero(f, 1.0, Order5())
find_zero(f, 1.0, Steffensen()) # also Order2()
find_zero(f, (1.0, 2.0), FalsePosition())
```

source

21.3 Roots.fzero

`Roots.fzero` — *Method.*

Find zero of a function within a bracket

Uses a modified bisection method for non `big` arguments

Arguments:

- `f` A scalar function or callable object
- `a` left endpont of interval
- `b` right endpont of interval
- `xtol` optional additional tolerance on sub-bracket size.

For a bracket to be valid, it must be that `f(a)*f(b) < 0`.

For `Float64` values, the answer is such that `f(prevfloat(x)) * f(nextfloat(x)) < 0` unless a non-zero value of `xtol` is specified in which case, it stops when the sub-bracketing produces an bracket with length less than `max(xtol, abs(x1)*xtolrel)`.

For `Big` values, which defaults to the algorithm of Alefeld, Potra, and Shi, a default tolerance is used for the sub-bracket length that can be enlarged with `xtol`.

If `a==-Inf` it is replaced with `nextfloat(a)`; if `b==Inf` it is replaced with `prevfloat(b)`.

Example:

```
‘fzero(sin, 3, 4)‘ # find pi
‘fzero(sin, [big(3), 4]) find pi with more digits
```

source

21.4 Roots.fzero

`Roots.fzero` — *Method.*

Find zero of a function using an iterative algorithm

- `f`: a scalar function or callable object
- `x0`: an initial guess, finite valued.

Keyword arguments:

- `ftol`: tolerance for a guess `abs(f(x)) < ftol`
- `xtol`: stop if `abs(xold - xnew) <= xtol + max(1, |xnew|)*xtolrel`
- `xtolrel`: see `xtol`
- `maxeval`: maximum number of steps
- `verbose`: Boolean. Set `true` to trace algorithm
- `order`: Can specify order of algorithm. 0 is most robust, also 1, 2, 5, 8, 16.
- `kwargs...` passed on to different algorithms. There are `maxfneval` when `order` is 1,2,5,8, or 16 and `beta` for orders 2,5,8,16,

This is a polyalgorithm redirecting different algorithms based on the value of `order`.

source

21.5 Roots.fzero

`Roots.fzero` — *Method.*

Find zero using Newton's method.

source

21.6 Roots.fzero

`Roots.fzero` — *Method.*

Find a zero with bracket specified via `[a,b]`, as `fzero(sin, [3,4])`.

source

21.7 Roots.fzero

`Roots.fzero` — *Method.*

Find a zero within a bracket with an initial guess to *possibly* speed things along.

source

21.8 Roots.fzeros

`Roots.fzeros` — *Method.*

`fzeros(f, a, b)`

Attempt to find all zeros of `f` within an interval `[a,b]`.

Simple algorithm that splits `[a,b]` into subintervals and checks each for a root. For bracketing subintervals, bisection is used. Otherwise, a derivative-free method is used. If there are a large number of zeros found relative to the number of subintervals, the number of subintervals is increased and the process is re-run.

There are possible issues with close-by zeros and zeros which do not cross the origin (non-simple zeros). Answers should be confirmed graphically, if possible.

source

21.9 Roots.halley

`Roots.halley` — *Method.*

Implementation of Halley's method. `xn1 = xn - 2f(xn)*f(xn) / (2*f(xn)^2 - f(xn) * f(xn))`

Arguments:

- `f::Function` – function to find zero of
- `fp::Function=D(f)` – derivative of `f`. Defaults to automatic derivative
- `fpp:Function=D(f,2)` – second derivative of `f`.

- `x0::Real` – initial guess

Keyword arguments:

- `ftol`. Stop iterating when |f(xn)| <= max(1, |xn|) * ftol.
- `xtol`. Stop iterating when |xn+1 - xn| <= xtol + max(1, |xn|) * xtolrel
- `xtolrel`. Stop iterating when |xn+1 - xn| <= xtol + max(1, |xn|) * xtolrel
- `maxeval`. Stop iterating if more than this many steps, throw error.
- `verbose::Bool=false` Set to `true` to see trace.

source

21.10 Roots.newton

`Roots.newton` — *Method*.

Implementation of Newton's method: `x_n1 = x_n - f(x_n)/ f(x_n)`
Arguments:

- `f::Function` – function to find zero of
- `fp::Function=D(f)` – derivative of `f`. Defaults to automatic derivative
- `x0::Number` – initial guess. For Newton's method this may be complex.

Keyword arguments:

- `ftol`. Stop iterating when |f(xn)| <= max(1, |xn|) * ftol.
- `xtol`. Stop iterating when |xn+1 - xn| <= xtol + max(1, |xn|) * xtolrel
- `xtolrel`. Stop iterating when |xn+1 - xn| <= xtol + max(1, |xn|) * xtolrel
- `maxeval`. Stop iterating if more than this many steps, throw error.
- `maxfneval`. Stop iterating if more than this many function calls, throw error.
- `verbose::Bool=false` Set to `true` to see trace.

source

21.11 Roots.secant_method

`Roots.secant_method` — *Method*.

```
secant_method(f, x0, x1; [kwargs...])
```

Solve for zero of `f(x) = 0` using the secant method.
Not exported. Use `find_zero` with `Order1()`.
source

Chapter 22

ImageTransformations

22.1 ImageTransformations.imresize

`ImageTransformations.imresize` — *Method.*

```
imresize(img, sz) -> imgr
imresize(img, inds) -> imgr
```

Change `img` to be of size `sz` (or to have indices `inds`). This interpolates the values at sub-pixel locations. If you are shrinking the image, you risk aliasing unless you low-pass filter `img` first. For example:

```
 = map((o,n)->0.75*o/n, size(img), sz)
kern = KernelFactors.gaussian()   # from ImageFiltering
imgr = imresize(imfilter(img, kern, NA()), sz)
```

See also `restrict`.
source

22.2 ImageTransformations.invwarpedview

`ImageTransformations.invwarpedview` — *Method.*

```
invwarpedview(img, tinv, [indices], [degree = Linear()], [fill = NaN]) -> wv
```

Create a view of `img` that lazily transforms any given index `I` passed to `wv[I]` to correspond to `img[inv(tinv)(I)]`. While technically this approach is known as backward mode warping, note that `InvWarpedView` is created by supplying the forward transformation. The given transformation `tinv` must accept a `SVector` as input and support `inv(tinv)`. A useful package to create a wide variety of such transformations is CoordinateTransformations.jl.

When invoking `wv[I]`, values for `img` must be reconstructed at arbitrary locations `inv(tinv)(I)`. `InvWarpedView` serves as a wrapper around `WarpedView`

which takes care of interpolation and extrapolation. The parameters `degree` and `fill` can be used to specify the b-spline degree and the extrapolation scheme respectively.

The optional parameter `indices` can be used to specify the domain of the resulting `wv`. By default the indices are computed in such a way that `wv` contains all the original pixels in `img`.

source

22.3 ImageTransformations.restrict

`ImageTransformations.restrict` — *Method*.

```
restrict(img[, region]) -> imgr
```

Reduce the size of `img` by two-fold along the dimensions listed in `region`, or all spatial coordinates if `region` is not specified. It anti-aliases the image as it goes, so is better than a naive summation over 2x2 blocks.

See also `imresize`.

source

22.4 ImageTransformations.warp

`ImageTransformations.warp` — *Method*.

```
warp(img, tform, [indices], [degree = Linear()], [fill = NaN]) -> imgw
```

Transform the coordinates of `img`, returning a new `imgw` satisfying `imgw[I] = img[tform(I)]`. This approach is known as backward mode warping. The transformation `tform` must accept a `SVector` as input. A useful package to create a wide variety of such transformations is CoordinateTransformations.jl.

Reconstruction scheme

During warping, values for `img` must be reconstructed at arbitrary locations `tform(I)` which do not lie on to the lattice of pixels. How this reconstruction is done depends on the type of `img` and the optional parameter `degree`.

When `img` is a plain array, then on-grid b-spline interpolation will be used. It is possible to configure what degree of b-spline to use with the parameter `degree`. For example one can use `degree = Linear()` for linear interpolation, `degree = Constant()` for nearest neighbor interpolation, or `degree = Quadratic(Flat())` for quadratic interpolation.

In the case `tform(I)` maps to indices outside the original `img`, those locations are set to a value `fill` (which defaults to `NaN` if the element type supports it, and `0` otherwise). The parameter `fill` also accepts extrapolation schemes, such as `Flat()`, `Periodic()` or `Reflect()`.

For more control over the reconstruction scheme — and how beyond-the-edge points are handled — pass `img` as an `AbstractInterpolation` or `AbstractExtrapolation` from Interpolations.jl.

The meaning of the coordinates

The output array `imgw` has indices that would result from applying `inv(tform)` to the indices of `img`. This can be very handy for keeping track of how pixels in `imgw` line up with pixels in `img`.

If you just want a plain array, you can "strip" the custom indices with `parent(imgw)`.

Examples: a 2d rotation (see JuliaImages documentation for pictures)

```
julia> using Images, CoordinateTransformations, TestImages, OffsetArrays

julia> img = testimage("lighthouse");

julia> indices(img)
(Base.OneTo(512),Base.OneTo(768))

# Rotate around the center of ‘img‘
julia> tfm = recenter(RotMatrix(-pi/4), center(img))
AffineMap([0.707107 0.707107; -0.707107 0.707107], [-196.755,293.99])

julia> imgw = warp(img, tfm);

julia> indices(imgw)
(-196:709,-68:837)

# Alternatively, specify the origin in the image itself
julia> img0 = OffsetArray(img, -30:481, -384:383);  # origin near top of image

julia> rot = LinearMap(RotMatrix(-pi/4))
LinearMap([0.707107 -0.707107; 0.707107 0.707107])

julia> imgw = warp(img0, rot);

julia> indices(imgw)
(-293:612,-293:611)

julia> imgr = parent(imgw);

julia> indices(imgr)
(Base.OneTo(906),Base.OneTo(905))
```

source

22.5 ImageTransformations.warpedview

`ImageTransformations.warpedview` — *Method.*

```
warpedview(img, tform, [indices], [degree = Linear()], [fill = NaN]) -> wv
```

Create a view of `img` that lazily transforms any given index `I` passed to `wv[I]` to correspond to `img[tform(I)]`. This approach is known as backward mode warping. The given transformation `tform` must accept a `SVector` as input. A useful package to create a wide variety of such transformations is CoordinateTransformations.jl.

When invoking `wv[I]`, values for `img` must be reconstructed at arbitrary locations `tform(I)` which do not lie on to the lattice of pixels. How this reconstruction is done depends on the type of `img` and the optional parameter `degree`. When `img` is a plain array, then on-grid b-spline interpolation will be used, where the pixel of `img` will serve as the coeficients. It is possible to configure what degree of b-spline to use with the parameter `degree`. The two possible values are `degree = Linear()` for linear interpolation, or `degree = Constant()` for nearest neighbor interpolation.

In the case `tform(I)` maps to indices outside the domain of `img`, those locations are set to a value `fill` (which defaults to `NaN` if the element type supports it, and `0` otherwise). Additionally, the parameter `fill` also accepts extrapolation schemes, such as `Flat()`, `Periodic()` or `Reflect()`.

The optional parameter `indices` can be used to specify the domain of the resulting `WarpedView`. By default the indices are computed in such a way that the resulting `WarpedView` contains all the original pixels in `img`. To do this `inv(tform)` has to be computed. If the given transformation `tform` does not support `inv`, then the parameter `indices` has to be specified manually.

`warpedview` is essentially a non-coping, lazy version of `warp`. As such, the two functions share the same interface, with one important difference. `warpedview` will insist that the resulting `WarpedView` will be a view of `img` (i.e. `parent(warpedview(img, ...)) === img`). Consequently, `warpedview` restricts the parameter `degree` to be either `Linear()` or `Constant()`.

source

Chapter 23

PyCall

23.1 PyCall.PyNULL

`PyCall.PyNULL` — *Method.*

```
PyNULL()
```

Return a `PyObject` that has a `NULL` underlying pointer, i.e. it doesn't actually refer to any Python object.

This is useful for initializing `PyObject` global variables and array elements before an actual Python object is available. For example, you might do `const myglobal = PyNULL()` and later on (e.g. in a module `__init__` function), reassign `myglobal` to point to an actual object with `copy!(myglobal, someobject)`. This procedure will properly handle Python's reference counting (so that the Python object will not be freed until you are done with `myglobal`).

source

23.2 PyCall.PyReverseDims

`PyCall.PyReverseDims` — *Method.*

```
PyReverseDims(array)
```

Passes a Julia `array` to Python as a NumPy row-major array (rather than Julia's native column-major order) with the dimensions reversed (e.g. a 234 Julia array is passed as a 432 NumPy row-major array). This is useful for Python libraries that expect row-major data.

source

23.3 PyCall.PyTextIO

`PyCall.PyTextIO` — *Method.*

```
PyTextIO(io::IO)
PyObject(io::IO)
```

Julia IO streams are converted into Python objects implementing the RawIOBase interface, so they can be used for binary I/O in Python

source

23.4 PyCall.pybuiltin

`PyCall.pybuiltin` — *Method*.

```
pybuiltin(s::AbstractString)
```

Look up a string or symbol `s` among the global Python builtins. If `s` is a string it returns a PyObject, while if `s` is a symbol it returns the builtin converted to `PyAny`.

source

23.5 PyCall.pybytes

`PyCall.pybytes` — *Method*.

```
pybytes(b::Union{String,Vector{UInt8}})
```

Convert `b` to a Python `bytes` object. This differs from the default `PyObject(b)` conversion of `String` to a Python string (which may fail if `b` does not contain valid Unicode), or from the default conversion of a `Vector{UInt8}` to a `bytearray` object (which is mutable, unlike `bytes`).

source

23.6 PyCall.pycall

`PyCall.pycall` — *Method*.

```
pycall(o::Union{PyObject,PyPtr}, returntype::TypeTuple, args...; kwargs...)
```

Call the given Python function (typically looked up from a module) with the given args... (of standard Julia types which are converted automatically to the corresponding Python types if possible), converting the return value to returntype (use a returntype of PyObject to return the unconverted Python object reference, or of PyAny to request an automated conversion)

source

23.7 PyCall.pyeval

`PyCall.pyeval` — *Function.*

```
pyeval(s::AbstractString, returntype::TypeTuple=PyAny, locals=PyDict{AbstractString, Py
                            input_type=Py_eval_input; kwargs...)
```

This evaluates `s` as a Python string and returns the result converted to `rtype` (which defaults to `PyAny`). The remaining arguments are keywords that define local variables to be used in the expression.

For example, `pyeval("x + y", x=1, y=2)` returns 3.

source

23.8 PyCall.pygui

`PyCall.pygui` — *Method.*

```
pygui()
```

Return the current GUI toolkit as a symbol.

source

23.9 PyCall.pygui_start

`PyCall.pygui_start` — *Function.*

```
pygui_start(gui::Symbol = pygui())
```

Start the event loop of a certain toolkit.

The argument `gui` defaults to the current default GUI, but it could be `:wx`, `:gtk`, `:gtk3`, `:tk`, or `:qt`.

source

23.10 PyCall.pygui_stop

`PyCall.pygui_stop` — *Function.*

```
pygui_stop(gui::Symbol = pygui())
```

Stop any running event loop for gui. The `gui` argument defaults to current default GUI.

source

23.11 PyCall.pyimport

`PyCall.pyimport` — *Method.*

```
pyimport(s::AbstractString)
```

Import the Python module `s` (a string or symbol) and return a pointer to it (a `PyObject`). Functions or other symbols in the module may then be looked up by s[name] where name is a string (for the raw PyObject) or symbol (for automatic type-conversion). Unlike the @pyimport macro, this does not define a Julia module and members cannot be accessed with `s.name`

source

23.12 PyCall.pyimport_conda

`PyCall.pyimport_conda` — *Function.*

```
pyimport_conda(modulename, condapkg, [channel])
```

Returns the result of `pyimport(modulename)` if possible. If the module is not found, and PyCall is configured to use the Conda Python distro (via the Julia Conda package), then automatically install `condapkg` via `Conda.add(condapkg)` and then re-try the `pyimport`. Other Anaconda-based Python installations are also supported as long as their `conda` program is functioning.

If PyCall is not using Conda and the `pyimport` fails, throws an exception with an error message telling the user how to configure PyCall to use Conda for automated installation of the module.

The third argument, `channel` is an optional Anaconda "channel" to use for installing the package; this is useful for packages that are not included in the default Anaconda package listing.

source

23.13 PyCall.pytype_mapping

`PyCall.pytype_mapping` — *Method.*

```
pytype_mapping(pytype, jltype)
```

Given a Python type object `pytype`, tell PyCall to convert it to `jltype` in `PyAny(object)` conversions.

source

23.14 PyCall.pytype_query

`PyCall.pytype_query` — *Function.*

```
pytype_query(o::PyObject, default=PyObject)
```

Given a Python object `o`, return the corresponding native Julia type (defaulting to `default`) that we convert `o` to in `PyAny(o)` conversions.

source

23.15 PyCall.pywrap

`PyCall.pywrap` — *Function.*

```
pywrap(o::PyObject)
```

This returns a wrapper `w` that is an anonymous module which provides (read) access to converted versions of o's members as w.member.

For example, `@pyimport module as name` is equivalent to `const name = pywrap(pyimport("module"))`

If the Python module contains identifiers that are reserved words in Julia (e.g. function), they cannot be accessed as `w.member`; one must instead use `w.pymember(:member)` (for the PyAny conversion) or w.pymember("member") (for the raw PyObject).

source

Chapter 24

Gadfly

24.1 Compose.draw

`Compose.draw` — *Method.*

```
draw(backend::Compose.Backend, p::Plot)
```

A convenience version of Compose.draw without having to call render
Args

- backend: The Compose.Backend object
- p: The Plot object

source

24.2 Compose.hstack

`Compose.hstack` — *Method.*

```
hstack(ps::Union{Plot,Context}...)
hstack(ps::Vector)
```

Arrange plots into a horizontal row. Use `context()` as a placeholder for an empty panel. Heterogeneous vectors must be typed. See also `vstack`, `gridstack`, `subplot_grid`.
Examples
“‘ p1 = plot(x=[1,2], y=[3,4], Geom.line); p2 = Compose.context(); hstack(p1, p2) hstack(Union{Plot,Compose.Context}[p1, p2])
source

24.3 Compose.vstack

`Compose.vstack` — *Method*.

```
vstack(ps::Union{Plot,Context}...)
vstack(ps::Vector)
```

Arrange plots into a vertical column. Use `context()` as a placeholder for an empty panel. Heterogeneous vectors must be typed. See also `hstack`, `gridstack`, `subplot_grid`.

Examples

“‘ p1 = plot(x=[1,2], y=[3,4], Geom.line); p2 = Compose.context(); vstack(p1, p2) vstack(Union{Plot,Compose.Context}[p1, p2])

source

24.4 Gadfly.layer

`Gadfly.layer` — *Method*.

```
layer(data_source::@compat(Union{AbstractDataFrame, (@compat Void)}),
               elements::ElementOrFunction...; mapping...)
```

Creates layers based on elements

Args

- data_source: The data source as a dataframe
- elements: The elements
- mapping: mapping

Returns

An array of layers

source

24.5 Gadfly.plot

`Gadfly.plot` — *Method*.

```
function plot(data_source::@compat(Union{(@compat Void), AbstractMatrix, AbstractDataFr
              mapping::Dict, elements::ElementOrFunctionOrLayers...)
```

The old fashioned (pre named arguments) version of plot.

This version takes an explicit mapping dictionary, mapping aesthetics symbols to expressions or columns in the data frame.

Args:

- data_source: Data to be bound to aesthetics.

- mapping: Dictionary of aesthetics symbols (e.g. :x, :y, :color) to names of columns in the data frame or other expressions.
- elements: Geometries, statistics, etc.

Returns:
A Plot object.
source

24.6 Gadfly.plot

`Gadfly.plot` — *Method.*

```
function plot(data_source::@compat(Union{AbstractMatrix, AbstractDataFrame}),
          elements::ElementOrFunctionOrLayers...; mapping...)
```

Create a new plot.

Grammar of graphics style plotting consists of specifying a dataset, one or more plot elements (scales, coordinates, geometries, etc), and binding of aesthetics to columns or expressions of the dataset.

For example, a simple scatter plot would look something like:
plot(my_data, Geom.point, x="time", y="price")
Where "time" and "price" are the names of columns in my_data.
Args:

- data_source: Data to be bound to aesthetics.
- elements: Geometries, statistics, etc.
- mapping: Aesthetics symbols (e.g. :x, :y, :color) mapped to names of columns in the data frame or other expressions.

source

24.7 Gadfly.render

`Gadfly.render` — *Method.*

```
render(plot::Plot)
```

Render a plot based on the Plot object
Args

- plot: Plot to be rendered.

Returns
A Compose context containing the rendered plot.
source

24.8 Gadfly.set_default_plot_format

`Gadfly.set_default_plot_format` — *Method*.

```
set_default_plot_format(fmt::Symbol)
```

Sets the default plot format
source

24.9 Gadfly.set_default_plot_size

`Gadfly.set_default_plot_size` — *Method*.

```
set_default_plot_size(width::Compose.MeasureOrNumber, height::Compose.MeasureOrNumber)
```

Sets preferred canvas size when rendering a plot without an explicit call to draw
source

24.10 Gadfly.spy

`Gadfly.spy` — *Method*.

```
spy(M::AbstractMatrix, elements::ElementOrFunction...; mapping...)
```

Simple heatmap plots of matrices.

It is a wrapper around the `plot()` function using the `rectbin` geometry. It also applies a sane set of defaults to make sure that the plots look nice by default. Specifically

- the aspect ratio of the coordinate system is fixed Coord.cartesian(fixed=true),

so that the rectangles become squares

- the axes run from 0.5 to N+0.5, because the first row/column is drawn to

(0.5, 1.5) and the last one to (N-0.5, N+0.5).

- the y-direction is flipped, so that the [1,1] of a matrix is in the top

left corner, as is customary

- NaNs are not drawn. `spy` leaves "holes" instead into the heatmap.

Args:

- M: A matrix.

Returns:
A plot object.
Known bugs:

- If the matrix is only NaNs, then it throws an `ArgumentError`, because an empty collection gets passed to the `plot` function / `rectbin` geometry.

source

24.11 Gadfly.style

`Gadfly.style` — *Method.*

Set some attributes in the current `Theme`. See `Theme` for available field.
source

24.12 Gadfly.title

`Gadfly.title` — *Method.*

```
title(ctx::Context, str::String, props::Property...) -> Context
```

Add a title string to a group of plots, typically created with `vstack`, `hstack`, or `gridstack`.
Examples

```
p1 = plot(x=[1,2], y=[3,4], Geom.line);
p2 = plot(x=[1,2], y=[4,3], Geom.line);
title(hstack(p1,p2), "my latest data", Compose.fontsize(18pt), fill(colorant"red"))
```

source

Chapter 25

IterTools

25.1 IterTools.chain

`IterTools.chain` — *Method.*

`chain(xs...)`

Iterate through any number of iterators in sequence.

```
julia> for i in chain(1:3, ['a', 'b', 'c'])
           @show i
       end
i = 1
i = 2
i = 3
i = 'a'
i = 'b'
i = 'c'
```

source

25.2 IterTools.distinct

`IterTools.distinct` — *Method.*

`distinct(xs)`

Iterate through values skipping over those already encountered.

```
julia> for i in distinct([1,1,2,1,2,4,1,2,3,4])
           @show i
       end
i = 1
```

```
i = 2
i = 4
i = 3
```

source

25.3 IterTools.groupby

`IterTools.groupby` — *Method*.

```
groupby(f, xs)
```

Group consecutive values that share the same result of applying `f`.

```
julia> for i in groupby(x -> x[1], ["face", "foo", "bar", "book", "baz", "zzz"])
           @show i
       end
i = String["face","foo"]
i = String["bar","book","baz"]
i = String["zzz"]
```

source

25.4 IterTools.imap

`IterTools.imap` — *Method*.

```
imap(f, xs1, [xs2, ...])
```

Iterate over values of a function applied to successive values from one or more iterators.

```
julia> for i in imap(+, [1,2,3], [4,5,6])
            @show i
       end
i = 5
i = 7
i = 9
```

source

25.5 IterTools.iterate

`IterTools.iterate` — *Method*.

```
iterate(f, x)
```

Iterate over successive applications of `f`, as in `x`, `f(x)`, `f(f(x))`, `f(f(f(x)))`, ...

Use `Base.take()` to obtain the required number of elements.

```
julia> for i in take(iterate(x -> 2x, 1), 5)
           @show i
       end
i = 1
i = 2
i = 4
i = 8
i = 16

julia> for i in take(iterate(sqrt, 100), 6)
           @show i
       end
i = 100
i = 10.0
i = 3.1622776601683795
i = 1.7782794100389228
i = 1.333521432163324
i = 1.1547819846894583
```

source

25.6 IterTools.ncycle

`IterTools.ncycle` — *Method.*

```
ncycle(xs, n)
```

Cycle through `iter` `n` times.

```
julia> for i in ncycle(1:3, 2)
           @show i
       end
i = 1
i = 2
i = 3
i = 1
i = 2
i = 3
```

source

25.7 IterTools.nth

`IterTools.nth` — *Method*.

```
nth(xs, n)
```

Return the `n`th element of `xs`. This is mostly useful for non-indexable collections.

```
julia> mersenne = Set([3, 7, 31, 127])
Set([7,31,3,127])

julia> nth(mersenne, 3)
3
```

source

25.8 IterTools.partition

`IterTools.partition` — *Method*.

```
partition(xs, n, [step])
```

Group values into `n`-tuples.

```
julia> for i in partition(1:9, 3)
           @show i
       end
i = (1,2,3)
i = (4,5,6)
i = (7,8,9)
```

If the `step` parameter is set, each tuple is separated by `step` values.

```
julia> for i in partition(1:9, 3, 2)
           @show i
       end
i = (1,2,3)
i = (3,4,5)
i = (5,6,7)
i = (7,8,9)

julia> for i in partition(1:9, 3, 3)
           @show i
       end
i = (1,2,3)
i = (4,5,6)
```

```
i = (7,8,9)

julia> for i in partition(1:9, 2, 3)
           @show i
       end
i = (1,2)
i = (4,5)
i = (7,8)
```

source

25.9 IterTools.peekiter

`IterTools.peekiter` — *Method.*

```
peekiter(xs)
```

Lets you peek at the head element of an iterator without updating the state.

```
julia> it = peekiter(["face", "foo", "bar", "book", "baz", "zzz"])
IterTools.PeekIter{Array{String,1}}(String["face","foo","bar","book","baz","zzz"])

julia> s = start(it)
(2,Nullable{String}("face"))

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("face")
Nullable{String}("face")

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("face")
Nullable{String}("face")

julia> x, s = next(it, s)
("face",(3,Nullable{String}("foo"),false))

julia> @show x
x = "face"
"face"

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("foo")
Nullable{String}("foo")
```

source

25.10 IterTools.product

`IterTools.product` — *Method*.

```
product(xs...)
```

Iterate over all combinations in the Cartesian product of the inputs.

```
julia> for p in product(1:3,4:5)
           @show p
       end
p = (1,4)
p = (2,4)
p = (3,4)
p = (1,5)
p = (2,5)
p = (3,5)
```

source

25.11 IterTools.repeatedly

`IterTools.repeatedly` — *Method*.

```
repeatedly(f, n)
```

Call function `f` `n` times, or infinitely if `n` is omitted.
[] julia¿ t() = (sleep(0.1); Dates.millisecond(now())) t (generic function with 1 method)
julia¿ collect(repeatedly(t, 5)) 5-element Array{Any,1}: 993 97 200 303 408
source

25.12 IterTools.subsets

`IterTools.subsets` — *Method*.

```
subsets(xs)
subsets(xs, k)
```

Iterate over every subset of the collection `xs`. You can restrict the subsets to a specific size `k`.

```
julia> for i in subsets([1, 2, 3])
           @show i
       end
i = Int64[]
i = [1]
```

```
i = [2]
i = [1,2]
i = [3]
i = [1,3]
i = [2,3]
i = [1,2,3]

julia> for i in subsets(1:4, 2)
          @show i
       end
i = [1,2]
i = [1,3]
i = [1,4]
i = [2,3]
i = [2,4]
i = [3,4]
```

source

25.13 IterTools.takenth

`IterTools.takenth` — *Method.*

```
takenth(xs, n)
```

Iterate through every `n`th element of `xs`.

```
julia> collect(takenth(5:15,3))
3-element Array{Int64,1}:
  7
 10
 13
```

source

25.14 IterTools.takestrict

`IterTools.takestrict` — *Method.*

```
takestrict(xs, n::Int)
```

Like `take()`, an iterator that generates at most the first `n` elements of `xs`, but throws an exception if fewer than `n` items are encountered in `xs`.

```
julia> a = :1:2:11
1:2:11

julia> collect(takestrict(a, 3))
3-element Array{Int64,1}:
 1
 3
 5
```

source

Chapter 26

Iterators

26.1 Iterators.chain

`Iterators.chain` — *Method.*

`chain(xs...)`

Iterate through any number of iterators in sequence.

```
julia> for i in chain(1:3, ['a', 'b', 'c'])
           @show i
       end
i = 1
i = 2
i = 3
i = 'a'
i = 'b'
i = 'c'
```

source

26.2 Iterators.distinct

`Iterators.distinct` — *Method.*

`distinct(xs)`

Iterate through values skipping over those already encountered.

```
julia> for i in distinct([1,1,2,1,2,4,1,2,3,4])
           @show i
       end
i = 1
```

```
i = 2
i = 4
i = 3
```

source

26.3 Iterators.groupby

`Iterators.groupby` — *Method*.

```
groupby(f, xs)
```

Group consecutive values that share the same result of applying `f`.

```
julia> for i in groupby(x -> x[1], ["face", "foo", "bar", "book", "baz", "zzz"])
           @show i
       end
i = String["face","foo"]
i = String["bar","book","baz"]
i = String["zzz"]
```

source

26.4 Iterators.imap

`Iterators.imap` — *Method*.

```
imap(f, xs1, [xs2, ...])
```

Iterate over values of a function applied to successive values from one or more iterators.

```
julia> for i in imap(+, [1,2,3], [4,5,6])
            @show i
       end
i = 5
i = 7
i = 9
```

source

26.5 Iterators.iterate

`Iterators.iterate` — *Method*.

```
iterate(f, x)
```

Iterate over successive applications of `f`, as in `f(x)`, `f(f(x))`, `f(f(f(x)))`, ...

Use `Base.take()` to obtain the required number of elements.

```
julia> for i in take(iterate(x -> 2x, 1), 5)
           @show i
       end
i = 1
i = 2
i = 4
i = 8
i = 16

julia> for i in take(iterate(sqrt, 100), 6)
           @show i
       end
i = 100
i = 10.0
i = 3.1622776601683795
i = 1.7782794100389228
i = 1.333521432163324
i = 1.1547819846894583
```

source

26.6 Iterators.ncycle

`Iterators.ncycle` — *Method*.

```
ncycle(xs, n)
```

Cycle through `iter` `n` times.

```
julia> for i in ncycle(1:3, 2)
           @show i
       end
i = 1
i = 2
i = 3
i = 1
i = 2
i = 3
```

source

26.7 Iterators.nth

`Iterators.nth` — *Method*.

```
nth(xs, n)
```

Return the `n`th element of `xs`. This is mostly useful for non-indexable collections.

```
julia> mersenne = Set([3, 7, 31, 127])
Set([7,31,3,127])

julia> nth(mersenne, 3)
3
```

source

26.8 Iterators.partition

`Iterators.partition` — *Method*.

```
partition(xs, n, [step])
```

Group values into `n`-tuples.

```
julia> for i in partition(1:9, 3)
           @show i
       end
i = (1,2,3)
i = (4,5,6)
i = (7,8,9)
```

If the `step` parameter is set, each tuple is separated by `step` values.

```
julia> for i in partition(1:9, 3, 2)
           @show i
       end
i = (1,2,3)
i = (3,4,5)
i = (5,6,7)
i = (7,8,9)

julia> for i in partition(1:9, 3, 3)
           @show i
       end
i = (1,2,3)
i = (4,5,6)
```

```
i = (7,8,9)

julia> for i in partition(1:9, 2, 3)
           @show i
       end
i = (1,2)
i = (4,5)
i = (7,8)
```

source

26.9 Iterators.peekiter

`Iterators.peekiter` — *Method.*

```
peekiter(xs)
```

Lets you peek at the head element of an iterator without updating the state.

```
julia> it = peekiter(["face", "foo", "bar", "book", "baz", "zzz"])
Iterators.PeekIter{Array{String,1}}(String["face","foo","bar","book","baz","zzz"])

julia> s = start(it)
(2,Nullable{String}("face"))

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("face")
Nullable{String}("face")

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("face")
Nullable{String}("face")

julia> x, s = next(it, s)
("face",(3,Nullable{String}("foo"),false))

julia> @show x
x = "face"
"face"

julia> @show peek(it, s)
peek(it,s) = Nullable{String}("foo")
Nullable{String}("foo")
```

source

26.10 Iterators.product

`Iterators.product` — *Method*.

```
product(xs...)
```

Iterate over all combinations in the Cartesian product of the inputs.

```
julia> for p in product(1:3,4:5)
           @show p
       end
p = (1,4)
p = (2,4)
p = (3,4)
p = (1,5)
p = (2,5)
p = (3,5)
```

source

26.11 Iterators.repeatedly

`Iterators.repeatedly` — *Method*.

```
repeatedly(f, n)
```

Call function `f` `n` times, or infinitely if `n` is omitted.
[] julia¿ t() = (sleep(0.1); Dates.millisecond(now())) t (generic function with 1 method)
julia¿ collect(repeatedly(t, 5)) 5-element Array{Any,1}: 993 97 200 303 408
source

26.12 Iterators.subsets

`Iterators.subsets` — *Method*.

```
subsets(xs)
subsets(xs, k)
```

Iterate over every subset of the collection `xs`. You can restrict the subsets to a specific size `k`.

```
julia> for i in subsets([1, 2, 3])
         @show i
       end
i = Int64[]
i = [1]
```

```
i = [2]
i = [1,2]
i = [3]
i = [1,3]
i = [2,3]
i = [1,2,3]

julia> for i in subsets(1:4, 2)
          @show i
       end
i = [1,2]
i = [1,3]
i = [1,4]
i = [2,3]
i = [2,4]
i = [3,4]
```

source

26.13 Iterators.takenth

`Iterators.takenth` — *Method*.

```
takenth(xs, n)
```

Iterate through every `n`th element of `xs`.

```
julia> collect(takenth(5:15,3))
3-element Array{Int64,1}:
  7
 10
 13
```

source

26.14 Iterators.takestrict

`Iterators.takestrict` — *Method*.

```
takestrict(xs, n::Int)
```

Like `take()`, an iterator that generates at most the first `n` elements of `xs`, but throws an exception if fewer than `n` items are encountered in `xs`.

```
julia> a = :1:2:11
1:2:11

julia> collect(takestrict(a, 3))
3-element Array{Int64,1}:
 1
 3
 5
```

source

Chapter 27

Polynomials

27.1 Polynomials.coeffs

`Polynomials.coeffs` — *Method.*

`coeffs(p::Poly)`

Return the coefficient vector `[a_0, a_1, ..., a_n]` of a polynomial `p`.
source

27.2 Polynomials.degree

`Polynomials.degree` — *Method.*

`degree(p::Poly)`

Return the degree of the polynomial `p`, i.e. the highest exponent in the polynomial that has a nonzero coefficient.
source

27.3 Polynomials.poly

`Polynomials.poly` — *Method.*

`poly(r)`

Construct a polynomial from its roots. Compare this to the `Poly` type constructor, which constructs a polynomial from its coefficients.

If `r` is a vector, the constructed polynomial is $(x - r_1)(x - r_2) \cdots (x - r_n)$. If `r` is a matrix, the constructed polynomial is $(x - e_1) \cdots (x - e_n)$, where e_i is the ith eigenvalue of `r`.

Examples

[] julia¿ poly([1, 2, 3]) The polynomial (x - 1)(x - 2)(x - 3) Poly(-6 + 11x - 6x2+x3)

julia¿ poly([1 2; 3 4]) The polynomial (x - 5.37228)(x + 0.37228) Poly(-1.9999999999999998 - 5.0x + 1.0x2)

source

27.4 Polynomials.polyder

`Polynomials.polyder` — *Method.*

```
polyder(p::Poly, k=1)
```

Compute the `k`th derivative of the polynomial `p`.

Examples

[] julia¿ polyder(Poly([1, 3, -1])) Poly(3 - 2x)

julia¿ polyder(Poly([1, 3, -1]), 2) Poly(-2)

source

27.5 Polynomials.polyfit

`Polynomials.polyfit` — *Function.*

```
polyfit(x, y, n=length(x)-1, sym=:x)
```

Fit a polynomial of degree `n` through the points specified by `x` and `y`, where `n <= length(x) - 1`, using least squares fit. When `n=length(x)-1` (the default), the interpolating polynomial is returned. The optional fourth argument can be used to specify the symbol for the returned polynomial.

Examples

[] julia¿ xs = linspace(0, pi, 5);

julia¿ ys = map(sin, xs);

julia¿ polyfit(xs, ys, 2) Poly(-0.004902082150108854 + 1.242031920509868x - 0.39535103925413095x2)

source

27.6 Polynomials.polyint

`Polynomials.polyint` — *Method.*

```
polyint(p::Poly, a::Number, b::Number)
```

Compute the definite integral of the polynomial `p` over the interval `[a,b]`.

Examples

[] julia¿ polyint(Poly([1, 0, -1]), 0, 1) 0.6666666666666667

source

27.7 Polynomials.polyint

`Polynomials.polyint` — *Method*.

```
polyint(p::Poly, k::Number=0)
```

Integrate the polynomial `p` term by term, optionally adding a constant term `k`. The order of the resulting polynomial is one higher than the order of `p`.

Examples

[] julia¿ polyint(Poly([1, 0, -1])) Poly(1.0x - 0.3333333333333333x3)

julia¿ polyint(Poly([1, 0, -1]), 2) Poly(2.0 + 1.0x - 0.3333333333333333x3)

source

27.8 Polynomials.polyval

`Polynomials.polyval` — *Method*.

```
polyval(p::Poly, x::Number)
```

Evaluate the polynomial `p` at `x` using Horner's method. `Poly` objects are callable, using this function.

Examples

[] julia¿ p = Poly([1, 0, -1]) Poly(1 - x2)

julia¿ polyval(p, 1) 0

julia¿ p(1) 0

source

27.9 Polynomials.printpoly

`Polynomials.printpoly` — *Method*.

```
printpoly(io::IO, p::Poly, mimetype = MIME"text/plain"(); descending_powers=false)
```

Print a human-readable representation of the polynomial `p` to `io`. The MIME types "text/plain" (default), "text/latex", and "text/html" are supported. By default, the terms are in order of ascending powers, matching the order in `coeffs(p)`; specifying `descending_powers=true` reverses the order.

Examples

```
julia> printpoly(STDOUT, Poly([1,2,3], :y))
1 + 2*y + 3*y^2
julia> printpoly(STDOUT, Poly([1,2,3], :y), descending_powers=true)
3*y^2 + 2*y + 1
```

source

27.10 Polynomials.roots

`Polynomials.roots` — *Method*.

```
roots(p::Poly)
```

Return the roots (zeros) of `p`, with multiplicity. The number of roots returned is equal to the order of `p`. The returned roots may be real or complex.

Examples

[] julia¿ roots(Poly([1, 0, -1])) 2-element Array{Float64,1}: -1.0 1.0

julia¿ roots(Poly([1, 0, 1])) 2-element Array{Complex{Float64},1}: 0.0+1.0im 0.0-1.0im

julia¿ roots(Poly([0, 0, 1])) 2-element Array{Float64,1}: 0.0 0.0

julia¿ roots(poly([1,2,3,4])) 4-element Array{Float64,1}: 4.0 3.0 2.0 1.0

source

27.11 Polynomials.variable

`Polynomials.variable` — *Method*.

```
variable(p::Poly)
variable([T::Type,] var)
variable()
```

Return the indeterminate of a polynomial, i.e. its variable, as a `Poly` object. When passed no arguments, this is equivalent to `variable(Float64, :x)`.

Examples

[] julia¿ variable(Poly([1, 2], :x)) Poly(x)

julia¿ variable(:y) Poly(1.0y)

julia¿ variable() Poly(1.0x)

julia¿ variable(Float32, :x) Poly(1.0f0x)

source

Chapter 28

Colors

28.1 Base.hex

`Base.hex` — *Method.*

```
hex(c)
```

Print a color as a RGB hex triple, or a transparent paint as an ARGB hex quadruplet.

source

28.2 Colors.MSC

`Colors.MSC` — *Method.*

```
MSC(h)
MSC(h, l)
```

Calculates the most saturated color for any given hue `h` by finding the corresponding corner in LCHuv space. Optionally, the lightness `l` may also be specified.

source

28.3 Colors.colordiff

`Colors.colordiff` — *Method.*

```
colordiff(a, b)
colordiff(a, b, metric)
```

Compute an approximate measure of the perceptual difference between colors `a` and `b`. Optionally, a `metric` may be supplied, chosen among `DE_2000` (the default), `DE_94`, `DE_JPC79`, `DE_CMC`, `DE_BFD`, `DE_AB`, `DE_DIN99`, `DE_DIN99d`, `DE_DIN99o`.

source

28.4 Colors.colormap

`Colors.colormap` — *Function*.

```
colormap(cname, [N; mid, logscale, kvs...])
```

Returns a predefined sequential or diverging colormap computed using the algorithm by Wijffelaars, M., et al. (2008). Sequential colormaps `cname` choices are `Blues`, `Greens`, `Grays`, `Oranges`, `Purples`, and `Reds`. Diverging colormap choices are `RdBu`. Optionally, you can specify the number of colors `N` (default 100). Keyword arguments include the position of the middle point `mid` (default 0.5) and the possibility to switch to log scaling with `logscale` (default false).

source

28.5 Colors.colormatch

`Colors.colormatch` — *Method*.

```
colormatch(wavelength)
colormatch(matchingfunction, wavelength)
```

Evaluate the CIE standard observer color match function.

Args:

- matchingfunction (optional): a type used to specify the matching function. Choices include `CIE1931_CMF` (the default, the CIE 1931 2 matching function), `CIE1964_CMF` (the CIE 1964 10 color matching function), `CIE1931J_CMF` (Judd adjustment to `CIE1931_CMF`), `CIE1931JV_CMF` (Judd-Vos adjustment to `CIE1931_CMF`).
- wavelen: Wavelength of stimulus in nanometers.

Returns: XYZ value of perceived color.

source

28.6 Colors.deuteranopic

`Colors.deuteranopic` — *Method*.

```
deuteranopic(c)
deuteranopic(c, p)
```

Convert a color to simulate deuteranopic color deficiency (lack of the middle-wavelength photopigment). See the description of `protanopic` for detail about the arguments.

source

28.7 Colors.distinguishable_colors

`Colors.distinguishable_colors` — *Method*.

```
distinguishable_colors(n, [seed]; [transform, lchoices, cchoices, hchoices])
```

Generate n maximally distinguishable colors.

This uses a greedy brute-force approach to choose n colors that are maximally distinguishable. Given seed color(s), and a set of possible hue, chroma, and lightness values (in LCHab space), it repeatedly chooses the next color as the one that maximizes the minimum pairwise distance to any of the colors already in the palette.

Args:

- `n`: Number of colors to generate.
- `seed`: Initial color(s) included in the palette. Default is `Vector{RGB{N0f8}}(0)`.

Keyword arguments:

- `transform`: Transform applied to colors before measuring distance. Default is the identity; other choices include `deuteranopic` to simulate color-blindness.
- `lchoices`: Possible lightness values (default `linspace(0,100,15)`)
- `cchoices`: Possible chroma values (default `linspace(0,100,15)`)
- `hchoices`: Possible hue values (default `linspace(0,340,20)`)

Returns: A `Vector` of colors of length `n`, of the type specified in `seed`.

source

28.8 Colors.protanopic

`Colors.protanopic` — *Method*.

```
protanopic(c)
protanopic(c, p)
```

Convert a color to simulate protanopic color deficiency (lack of the long-wavelength photopigment). `c` is the input color; the optional argument `p` is the fraction of photopigment loss, in the range 0 (no loss) to 1 (complete loss).

source

28.9 Colors.tritanopic

`Colors.tritanopic` — *Method*.

```
tritanopic(c)
tritanopic(c, p)
```

Convert a color to simulate tritanopic color deficiency (lack of the short-wavelength photogiment). See `protanopic` for more detail about the arguments.
source

28.10 Colors.weighted_color_mean

`Colors.weighted_color_mean` — *Method*.

```
weighted_color_mean(w1, c1, c2)
```

Returns the color `w1*c1 + (1-w1)*c2` that is the weighted mean of `c1` and `c2`, where `c1` has a weight 0 `w1` 1.
source

28.11 Colors.whitebalance

`Colors.whitebalance` — *Method*.

```
whitebalance(c, src_white, ref_white)
```

Whitebalance a color.

Input a source (adopted) and destination (reference) white. E.g., if you have a photo taken under florencent lighting that you then want to appear correct under regular sunlight, you might do something like `whitebalance(c, WP_F2, WP_D65)`.

Args:

- `c`: An observed color.
- `src_white`: Adopted or source white corresponding to `c`
- `ref_white`: Reference or destination white.

Returns: A whitebalanced color.
source

Chapter 29

FileIO

29.1 Base.info

`Base.info` — *Method.*

`info(fmt)` returns the magic bytes/extension information for `DataFormat fmt`.

source

29.2 FileIO.add_format

`FileIO.add_format` — *Method.*

`add_format(fmt, magic, extention)` registers a new `DataFormat`. For example:

```
add_format(format"PNG", (UInt8[0x4d,0x4d,0x00,0x2b], UInt8[0x49,0x49,0x2a,0x00]), [".t
add_format(format"PNG", [0x89,0x50,0x4e,0x47,0x0d,0x0a,0x1a,0x0a], ".png")
add_format(format"NRRD", "NRRD", [".nrrd",".nhdr"])
```

Note that extensions, magic numbers, and format-identifiers are case-sensitive.

source

29.3 FileIO.add_loader

`FileIO.add_loader` — *Function.*

`add_loader(fmt, :Package)` triggers `using Package` before loading format `fmt`

source

29.4 FileIO.add_saver

`FileIO.add_saver` — *Function*.

`add_saver(fmt, :Package)` triggers `using Package` before saving format `fmt`

source

29.5 FileIO.del_format

`FileIO.del_format` — *Method*.

`del_format(fmt::DataFormat)` deletes `fmt` from the format registry.

source

29.6 FileIO.file_extension

`FileIO.file_extension` — *Method*.

`file_extension(file)` returns the file extension associated with `File file`.

source

29.7 FileIO.file_extension

`FileIO.file_extension` — *Method*.

`file_extension(file)` returns a nullable-string for the file extension associated with `Stream stream`.

source

29.8 FileIO.filename

`FileIO.filename` — *Method*.

`filename(file)` returns the filename associated with `File file`.

source

29.9 FileIO.filename

`FileIO.filename` — *Method*.

`filename(stream)` returns a nullable-string of the filename associated with `Stream stream`.

source

29.10 FileIO.load

`FileIO.load` — *Method*.

- `load(filename)` loads the contents of a formatted file, trying to infer

  the format from `filename` and/or magic bytes in the file.

- `load(strm)` loads from an `IOStream` or similar object. In this case,

  the magic bytes are essential.

- `load(File(format"PNG",filename))` specifies the format directly, and bypasses inference.

- `load(f; options...)` passes keyword arguments on to the loader.

  source

29.11 FileIO.magic

`FileIO.magic` — *Method*.
`magic(fmt)` returns the magic bytes of format `fmt`
source

29.12 FileIO.query

`FileIO.query` — *Method*.
`query(filename)` returns a `File` object with information about the format inferred from the file's extension and/or magic bytes.
source

29.13 FileIO.query

`FileIO.query` — *Method*.
`query(io, [filename])` returns a `Stream` object with information about the format inferred from the magic bytes.
source

29.14 FileIO.save

`FileIO.save` — *Method*.

- `save(filename, data...)` saves the contents of a formatted file,

  trying to infer the format from `filename`.

- `save(Stream(format"PNG",io), data...)` specifies the format directly, and bypasses inference.
- `save(f, data...; options...)` passes keyword arguments on to the saver.

source

29.15 FileIO.skipmagic

`FileIO.skipmagic` — *Method*.

`skipmagic(s)` sets the position of `Stream s` to be just after the magic bytes. For a plain IO object, you can use `skipmagic(io, fmt)`.

source

29.16 FileIO.stream

`FileIO.stream` — *Method*.

`stream(s)` returns the stream associated with `Stream s`

source

29.17 FileIO.unknown

`FileIO.unknown` — *Method*.

`unknown(f)` returns true if the format of `f` is unknown.

source

Chapter 30

Interact

30.1 Interact.button

`Interact.button` — *Method*.

```
button(label; value=nothing, signal)
```

Create a push button. Optionally specify the `label`, the `value` emitted when then button is clicked, and/or the (Reactive.jl) `signal` coupled to this button.

source

30.2 Interact.checkbox

`Interact.checkbox` — *Method*.

```
checkbox(value=false; label="", signal)
```

Provide a checkbox with the specified starting (boolean) `value`. Optional provide a `label` for this widget and/or the (Reactive.jl) `signal` coupled to this widget.

source

30.3 Interact.dropdown

`Interact.dropdown` — *Method*.

```
dropdown(choices; label="", value, typ, icons, tooltips, signal)
```

Create a "dropdown" widget. `choices` can be a vector of options. Optionally specify the starting `value` (defaults to the first choice), the `typ` of elements

in `choices`, supply custom `icons`, provide `tooltips`, and/or specify the (Reactive.jl) `signal` coupled to this widget.

Examples

```
a = dropdown(["one", "two", "three"])
```

To link a callback to the dropdown, use

```
f = dropdown(["turn red"=>colorize_red, "turn green"=>colorize_green])
map(g->g(image), signal(f))
```

source

30.4 Interact.radiobuttons

`Interact.radiobuttons` — *Method*.

radiobuttons: see the help for `dropdown`

source

30.5 Interact.selection

`Interact.selection` — *Method*.

selection: see the help for `dropdown`

source

30.6 Interact.selection_slider

`Interact.selection_slider` — *Method*.

selection_slider: see the help for `dropdown` If the slider has numeric (<:Real) values, and its signal is updated, it will update to the nearest value from the range/choices provided. To disable this behaviour, so that the widget state will only update if an exact match for signal value is found in the range/choice, use `syncnearest=false`.

source

30.7 Interact.set!

`Interact.set!` — *Method*.

`set!(w::Widget, fld::Symbol, val)`

Set the value of a widget property and update all displayed instances of the widget. If `val` is a `Signal`, then updates to that signal will be reflected in widget instances/views.

If `fld` is `:value`, `val` is also `push!`ed to `signal(w)`

source

30.8 Interact.slider

`Interact.slider` — *Method*.

```
slider(range; value, signal, label="", readout=true, continuous_update=true)
```

Create a slider widget with the specified `range`. Optionally specify the starting `value` (defaults to the median of `range`), provide the (Reactive.jl) `signal` coupled to this slider, and/or specify a string `label` for the widget.

source

30.9 Interact.textarea

`Interact.textarea` — *Method*.

```
textarea(value=""; label="", signal)
```

Creates an extended text-entry area. Optionally provide a `label` and/or the (Reactive.jl) `signal` associated with this widget. The `signal` updates when you type.

source

30.10 Interact.textbox

`Interact.textbox` — *Method*.

```
textbox(value=""; label="", typ=typeof(value), range=nothing, signal)
```

Create a box for entering text. `value` is the starting value; if you don't want to provide an initial value, you can constrain the type with `typ`. Optionally provide a `label`, specify the allowed range (e.g., `-10.0:10.0`) for numeric entries, and/or provide the (Reactive.jl) `signal` coupled to this text box.

source

30.11 Interact.togglebutton

`Interact.togglebutton` — *Method*.

```
togglebutton(label=""; value=false, signal)
```

Create a toggle button. Optionally specify the `label`, the initial state (`value=false` is off, `value=true` is on), and/or provide the (Reactive.jl) `signal` coupled to this button.

source

30.12 Interact.togglebuttons

`Interact.togglebuttons` — *Method*.

togglebuttons: see the help for `dropdown`

source

30.13 Interact.vselection_slider

`Interact.vselection_slider` — *Method*.

```
vselection_slider(args...; kwargs...)
```

Shorthand for `selection_slider(args...; orientation="vertical", kwargs...)`

source

30.14 Interact.vslider

`Interact.vslider` — *Method*.

```
vslider(args...; kwargs...)
```

Shorthand for `slider(args...; orientation="vertical", kwargs...)`

source

Chapter 31

FFTW

31.1 Base.DFT.FFTW.dct

`Base.DFT.FFTW.dct` — *Function.*

```
dct(A [, dims])
```

Performs a multidimensional type-II discrete cosine transform (DCT) of the array `A`, using the unitary normalization of the DCT. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod`. See also `plan_dct` for even greater efficiency.

source

31.2 Base.DFT.FFTW.dct!

`Base.DFT.FFTW.dct!` — *Function.*

```
dct!(A [, dims])
```

Same as `dct!`, except that it operates in-place on `A`, which must be an array of real or complex floating-point values.

source

31.3 Base.DFT.FFTW.idct

`Base.DFT.FFTW.idct` — *Function.*

```
idct(A [, dims])
```

Computes the multidimensional inverse discrete cosine transform (DCT) of the array `A` (technically, a type-III DCT with the unitary normalization). The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod`. See also `plan_idct` for even greater efficiency.

source

31.4 Base.DFT.FFTW.idct!

`Base.DFT.FFTW.idct!` — *Function.*

```
idct!(A [, dims])
```

Same as `idct!`, but operates in-place on `A`.

source

31.5 Base.DFT.FFTW.plan_dct

`Base.DFT.FFTW.plan_dct` — *Function.*

```
plan_dct(A [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized discrete cosine transform (DCT), similar to `plan_fft` except producing a function that computes `dct`. The first two arguments have the same meaning as for `dct`.

source

31.6 Base.DFT.FFTW.plan_dct!

`Base.DFT.FFTW.plan_dct!` — *Function.*

```
plan_dct!(A [, dims [, flags [, timelimit]]])
```

Same as `plan_dct`, but operates in-place on `A`.

source

31.7 Base.DFT.FFTW.plan_idct

`Base.DFT.FFTW.plan_idct` — *Function.*

```
plan_idct(A [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `plan_fft` except producing a function that computes `idct`. The first two arguments have the same meaning as for `idct`.

source

31.8 Base.DFT.FFTW.plan_idct!

`Base.DFT.FFTW.plan_idct!` — *Function.*

```
plan_idct!(A [, dims [, flags [, timelimit]]])
```

Same as `plan_idct`, but operates in-place on `A`.
source

31.9 Base.DFT.FFTW.plan_r2r

`Base.DFT.FFTW.plan_r2r` — *Function.*

```
plan_r2r(A, kind [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized r2r transform, similar to `plan_fft` except that the transforms (and the first three arguments) correspond to `r2r` and `r2r!`, respectively.
source

31.10 Base.DFT.FFTW.plan_r2r!

`Base.DFT.FFTW.plan_r2r!` — *Function.*

```
plan_r2r!(A, kind [, dims [, flags [, timelimit]]])
```

Similar to `plan_fft`, but corresponds to `r2r!`.
source

31.11 Base.DFT.FFTW.r2r

`Base.DFT.FFTW.r2r` — *Function.*

```
r2r(A, kind [, dims])
```

Performs a multidimensional real-input/real-output (r2r) transform of type `kind` of the array `A`, as defined in the FFTW manual. `kind` specifies either a discrete cosine transform of various types (`FFTW.REDFT00`, `FFTW.REDFT01`, `FFTW.REDFT10`, or `FFTW.REDFT11`), a discrete sine transform of various types (`FFTW.RODFT00`, `FFTW.RODFT01`, `FFTW.RODFT10`, or `FFTW.RODFT11`), a real-input DFT with halfcomplex-format output (`FFTW.R2HC` and its inverse `FFTW.HC2R`), or a discrete Hartley transform (`FFTW.DHT`). The `kind` argument may be an array or tuple in order to specify different transform types along the different dimensions of `A`; `kind[end]` is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at http://www.fftw.org/doc.

The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. `kind[i]` is then the transform type for `dims[i]`, with `kind[end]` being used for `i > length(kind)`.

See also `plan_r2r` to pre-plan optimized r2r transforms.

source

31.12 Base.DFT.FFTW.r2r!

`Base.DFT.FFTW.r2r!` — *Function.*

```
r2r!(A, kind [, dims])
```

Same as `r2r`, but operates in-place on `A`, which must be an array of real or complex floating-point numbers.

source

Chapter 32

AxisArrays

32.1 AxisArrays.axes

`AxisArrays.axes` — *Method*.

```
axes(A::AxisArray) -> (Axis...)
axes(A::AxisArray, ax::Axis) -> Axis
axes(A::AxisArray, dim::Int) -> Axis
```

Returns the tuple of axis vectors for an AxisArray. If an specific `Axis` is specified, then only that axis vector is returned. Note that when extracting a single axis vector, `axes(A, Axis{1})`) is type-stable and will perform better than `axes(A)[1]`.

For an AbstractArray without `Axis` information, `axes` returns the default axes, i.e., those that would be produced by `AxisArray(A)`.

source

32.2 AxisArrays.axisdim

`AxisArrays.axisdim` — *Method*.

```
axisdim(::AxisArray, ::Axis) -> Int
axisdim(::AxisArray, ::Type{Axis}) -> Int
```

Given an AxisArray and an Axis, return the integer dimension of the Axis within the array.

source

32.3 AxisArrays.axisnames

`AxisArrays.axisnames` — *Method*.

```
axisnames(A::AxisArray)           -> (Symbol...)
axisnames(::Type{AxisArray{...}}) -> (Symbol...)
axisnames(ax::Axis...)            -> (Symbol...)
axisnames(::Type{Axis{...}}...)   -> (Symbol...)
```

Returns the axis names of an AxisArray or list of Axises as a tuple of Symbols.

source

32.4 AxisArrays.axisvalues

`AxisArrays.axisvalues` — *Method*.

```
axisvalues(A::AxisArray)           -> (AbstractVector...)
axisvalues(ax::Axis...)            -> (AbstractVector...)
```

Returns the axis values of an AxisArray or list of Axises as a tuple of vectors.

source

32.5 AxisArrays.collapse

`AxisArrays.collapse` — *Method*.

```
collapse(::Type{Val{N}}, As::AxisArray...) -> AxisArray
collapse(::Type{Val{N}}, labels::Tuple, As::AxisArray...) -> AxisArray
collapse(::Type{Val{N}}, ::Type{NewArrayType}, As::AxisArray...) -> AxisArray
collapse(::Type{Val{N}}, ::Type{NewArrayType}, labels::Tuple, As::AxisArray...) -> AxisArray
```

Collapses `AxisArray`s with `N` equal leading axes into a single `AxisArray`. All additional axes in any of the arrays are collapsed into a single additional axis of type `Axis{:collapsed, CategoricalVector{Tuple}}`.

Arguments

- `::Type{Val{N}}`: the greatest common dimension to share between all input arrays. The remaining axes are collapsed. All `N` axes must be common to each input array, at the same dimension. Values from `0` up to the minimum number of dimensions across all input arrays are allowed.
- `labels::Tuple`: (optional) an index for each array in `As` used as the leading element in the index tuples in the `:collapsed` axis. Defaults to `1:length(As)`.
- `::Type{NewArrayType<:AbstractArray{_, N+1}}`: (optional) the desired underlying array type for the returned `AxisArray`.
- `As::AxisArray...`: `AxisArray`s to be collapsed together.

Examples

```
julia> price_data = AxisArray(rand(10), Axis{:time}(Date(2016,01,01):Day(1):Date(2016,0
1-dimensional AxisArray{Float64,1,...} with axes:
    :time, 2016-01-01:1 day:2016-01-10
And data, a 10-element Array{Float64,1}:
 0.885014
 0.418562
 0.609344
 0.72221
 0.43656
 0.840304
 0.455337
 0.65954
 0.393801
 0.260207

julia> size_data = AxisArray(rand(10,2), Axis{:time}(Date(2016,01,01):Day(1):Date(2016
2-dimensional AxisArray{Float64,2,...} with axes:
    :time, 2016-01-01:1 day:2016-01-10
    :measure, Symbol[:area, :volume]
And data, a 102 Array{Float64,2}:
 0.159434     0.456992
 0.344521     0.374623
 0.522077     0.313256
 0.994697     0.320953
 0.95104      0.900526
 0.921854     0.729311
 0.000922581  0.148822
 0.449128     0.761714
 0.650277     0.135061
 0.688773     0.513845

julia> collapsed = collapse(Val{1}, (:price, :size), price_data, size_data)
2-dimensional AxisArray{Float64,2,...} with axes:
    :time, 2016-01-01:1 day:2016-01-10
    :collapsed, Tuple{Symbol,Vararg{Symbol,N} where N}[(:price,), (:size, :area), (:siz
And data, a 103 Array{Float64,2}:
 0.885014  0.159434     0.456992
 0.418562  0.344521     0.374623
 0.609344  0.522077     0.313256
 0.72221   0.994697     0.320953
 0.43656   0.95104      0.900526
 0.840304  0.921854     0.729311
 0.455337  0.000922581  0.148822
 0.65954   0.449128     0.761714
```

```
 0.393801  0.650277     0.135061
 0.260207  0.688773     0.513845

julia> collapsed[Axis{:collapsed}(:size)] == size_data
true
```

source

Chapter 33

QuadGK

33.1 Base.quadgk

`Base.quadgk` — *Method.*

```
quadgk(f, a,b,c...; reltol=sqrt(eps), abstol=0, maxevals=10^7, order=7, norm=vecnorm)
```

Numerically integrate the function `f(x)` from `a` to `b`, and optionally over additional intervals `b` to `c` and so on. Keyword options include a relative error tolerance `reltol` (defaults to `sqrt(eps)` in the precision of the endpoints), an absolute error tolerance `abstol` (defaults to 0), a maximum number of function evaluations `maxevals` (defaults to `10^7`), and the `order` of the integration rule (defaults to 7).

Returns a pair `(I,E)` of the estimated integral `I` and an estimated upper bound on the absolute error `E`. If `maxevals` is not exceeded then `E <= max(abstol, reltol*norm(I))` will hold. (Note that it is useful to specify a positive `abstol` in cases where `norm(I)` may be zero.)

The endpoints `a` et cetera can also be complex (in which case the integral is performed over straight-line segments in the complex plane). If the endpoints are `BigFloat`, then the integration will be performed in `BigFloat` precision as well.

!!! note It is advisable to increase the integration `order` in rough proportion to the precision, for smooth integrands.

More generally, the precision is set by the precision of the integration endpoints (promoted to floating-point types).

The integrand `f(x)` can return any numeric scalar, vector, or matrix type, or in fact any type supporting `+`, `-`, multiplication by real values, and a `norm` (i.e., any normed vector space). Alternatively, a different norm can be specified by passing a `norm`-like function as the `norm` keyword argument (which defaults to `vecnorm`).

!!! note Only one-dimensional integrals are provided by this function. For multi-dimensional integration (cubature), there are many different algorithms

(often much better than simple nested 1d integrals) and the optimal choice tends to be very problem-dependent. See the Julia external-package listing for available algorithms for multidimensional integration or other specialized tasks (such as integrals of highly oscillatory or singular functions).

The algorithm is an adaptive Gauss-Kronrod integration technique: the integral in each interval is estimated using a Kronrod rule (`2*order+1` points) and the error is estimated using an embedded Gauss rule (`order` points). The interval with the largest error is then subdivided into two intervals and the process is repeated until the desired error tolerance is achieved.

These quadrature rules work best for smooth functions within each interval, so if your function has a known discontinuity or other singularity, it is best to subdivide your interval to put the singularity at an endpoint. For example, if `f` has a discontinuity at `x=0.7` and you want to integrate from 0 to 1, you should use `quadgk(f, 0,0.7,1)` to subdivide the interval at the point of discontinuity. The integrand is never evaluated exactly at the endpoints of the intervals, so it is possible to integrate functions that diverge at the endpoints as long as the singularity is integrable (for example, a `log(x)` or `1/sqrt(x)` singularity).

For real-valued endpoints, the starting and/or ending points may be infinite. (A coordinate transformation is performed internally to map the infinite interval to a finite one.)

source

33.2 QuadGK.gauss

`QuadGK.gauss` — *Method.*

```
gauss([T,] N)
```

Return a pair `(x, w)` of `N` quadrature points `x[i]` and weights `w[i]` to integrate functions on the interval `(-1, 1)`, i.e. `sum(w .* f.(x))` approximates the integral. Uses the method described in Trefethen & Bau, Numerical Linear Algebra, to find the `N`-point Gaussian quadrature in O(`N`) operations.

`T` is an optional parameter specifying the floating-point type, defaulting to `Float64`. Arbitrary precision (`BigFloat`) is also supported.

source

33.3 QuadGK.kronrod

`QuadGK.kronrod` — *Method.*

```
kronrod([T,] n)
```

Compute `2n+1` Kronrod points `x` and weights `w` based on the description in Laurie (1997), appendix A, simplified for `a=0`, for integrating on `[-1,1]`. Since the rule is symmetric, this only returns the `n+1` points with `x <= 0`. The

function Also computes the embedded `n`-point Gauss quadrature weights `gw` (again for `x <= 0`), corresponding to the points `x[2:2:end]`. Returns `(x,w,wg)` in O(`n`) operations.

`T` is an optional parameter specifying the floating-point type, defaulting to `Float64`. Arbitrary precision (`BigFloat`) is also supported.

Given these points and weights, the estimated integral `I` and error `E` can be computed for an integrand `f(x)` as follows:

```
x, w, wg = kronrod(n)
fx = f(x[end])                 # f(0)
x = x[1:end-1]                 # the x < 0 Kronrod points
fx = f.(x) .+ f.((-).(x))    # f(x < 0) + f(x > 0)
I = sum(fx .* w[1:end-1]) + fx * w[end]
if isodd(n)
    E = abs(sum(fx[2:2:end] .* wg[1:end-1]) + fx*wg[end] - I)
else
    E = abs(sum(fx[2:2:end] .* wg[1:end])- I)
end
```

source

Chapter 34

BusinessDays

34.1 BusinessDays.advancebdays

`BusinessDays.advancebdays` — *Method*.

```
advancebdays(calendar, dt, bdays_count)
```

Increments given date `dt` by `bdays_count`. Decrements it if `bdays_count` is negative. `bdays_count` can be a `Int`, `Vector{Int}` or a `UnitRange`.

Computation starts by next Business Day if `dt` is not a Business Day.

source

34.2 BusinessDays.bdayscount

`BusinessDays.bdayscount` — *Method*.

```
bdays(calendar, dt0, dt1)
```

Counts the number of Business Days between `dt0` and `dt1`. Returns instances of `Dates.Day`.

Computation is always based on next Business Day if given dates are not Business Days.

source

34.3 BusinessDays.firstbdayofmonth

`BusinessDays.firstbdayofmonth` — *Method*.

```
firstbdayofmonth(calendar, dt)
firstbdayofmonth(calendar, yy, mm)
```

Returns the first business day of month.

source

34.4 BusinessDays.isbday

`BusinessDays.isbday` — *Method*.

```
isbday(calendar, dt)
```

Returns `false` for weekends or holidays. Returns `true` otherwise.
source

34.5 BusinessDays.isholiday

`BusinessDays.isholiday` — *Method*.

```
isholiday(calendar, dt)
```

Checks if `dt` is a holiday based on a given `calendar` of holidays.
`calendar` can be an instance of `HolidayCalendar`, a `Symbol` or an `AbstractString`.
Returns boolean values.
source

34.6 BusinessDays.isweekday

`BusinessDays.isweekday` — *Method*.

```
isweekday(dt)
```

Returns `true` for Monday to Friday. Returns `false` otherwise.
source

34.7 BusinessDays.isweekend

`BusinessDays.isweekend` — *Method*.

```
isweekend(dt)
```

Returns `true` for Saturdays or Sundays. Returns `false` otherwise.
source

34.8 BusinessDays.lastbdayofmonth

`BusinessDays.lastbdayofmonth` — *Method*.

```
lastbdayofmonth(calendar, dt)
lastbdayofmonth(calendar, yy, mm)
```

Returns the last business day of month.
source

34.9 BusinessDays.listbdays

`BusinessDays.listbdays` — *Method*.

```
listbdays(calendar, dt0::Date, dt1::Date)  Vector{Date}
```

Returns the list of business days between `dt0` and `dt1`.
source

34.10 BusinessDays.listholidays

`BusinessDays.listholidays` — *Method*.

```
listholidays(calendar, dt0::Date, dt1::Date)  Vector{Date}
```

Returns the list of holidays between `dt0` and `dt1`.
source

34.11 BusinessDays.tobday

`BusinessDays.tobday` — *Method*.

```
tobday(calendar, dt; [forward=true])
```

Adjusts `dt` to next Business Day if it's not a Business Day. If `isbday(dt)`, returns `dt`.
source

Chapter 35

NLSolversBase

35.1 Base.LinAlg.gradient

`Base.LinAlg.gradient` — *Method*.

Evaluates the gradient value at `x`
This does *not* update `obj.DF`.
source

35.2 Base.LinAlg.gradient

`Base.LinAlg.gradient` — *Method*.

Get the `i`th element of the most recently evaluated gradient of `obj`.
source

35.3 Base.LinAlg.gradient

`Base.LinAlg.gradient` — *Method*.

Get the most recently evaluated gradient of `obj`.
source

35.4 NLSolversBase.complex_to_real

`NLSolversBase.complex_to_real` — *Method*.

Convert a complex array of size dims to a real array of size 2 x dims
source

35.5 NLSolversBase.gradient!

`NLSolversBase.gradient!` — *Method*.

Evaluates the gradient value at `x`.
Stores the value in `obj.DF`.
source

35.6 NLSolversBase.hessian

`NLSolversBase.hessian` — *Method*.
Get the most recently evaluated Hessian of `obj`
source

35.7 NLSolversBase.jacobian

`NLSolversBase.jacobian` — *Method*.
Get the most recently evaluated Jacobian of `obj`.
source

35.8 NLSolversBase.real_to_complex

`NLSolversBase.real_to_complex` — *Method*.
Convert a real array of size 2 x dims to a complex array of size dims
source

35.9 NLSolversBase.value!!

`NLSolversBase.value!!` — *Method*.
Force (re-)evaluation of the objective value at `x`.
Returns `f(x)` and stores the value in `obj.F`
source

35.10 NLSolversBase.value!

`NLSolversBase.value!` — *Method*.
Evaluates the objective value at `x`.
Returns `f(x)` and stores the value in `obj.F`
source

35.11 NLSolversBase.value

`NLSolversBase.value` — *Method*.
Evaluates the objective value at `x`.
Returns `f(x)`, but does *not* store the value in `obj.F`
source

35.12 NLSolversBase.value

`NLSolversBase.value` — *Method*.

Get the most recently evaluated objective value of `obj`.

source

Chapter 36

Dagger

36.1 Dagger.alignfirst

`Dagger.alignfirst` — *Method*.

```
alignfirst(a)
```

Make a subdomain a standalone domain. For example,

```
alignfirst(ArrayDomain(11:25, 21:100))
# => ArrayDomain((1:15), (1:80))
```

source

36.2 Dagger.cached_stage

`Dagger.cached_stage` — *Method*.

A memoized version of stage. It is important that the tasks generated for the same DArray have the same identity, for example:

```
A = rand(Blocks(100,100), Float64, 1000, 1000)
compute(A+A')
```

must not result in computation of A twice.
source

36.3 Dagger.compute

`Dagger.compute` — *Method*.

A DArray object may contain a thunk in it, in which case we first turn it into a Thunk object and then compute it.
source

36.4 Dagger.compute

`Dagger.compute` — *Method*.

Compute a Thunk - creates the DAG, assigns ranks to nodes for tie breaking and runs the scheduler.

source

36.5 Dagger.domain

`Dagger.domain` — *Function*.

```
domain(x::T)
```

Returns metadata about `x`. This metadata will be in the `domain` field of a Chunk object when an object of type `T` is created as the result of evaluating a Thunk.

source

36.6 Dagger.domain

`Dagger.domain` — *Method*.

The domain of an array is a ArrayDomain

source

36.7 Dagger.domain

`Dagger.domain` — *Method*.

If no `domain` method is defined on an object, then we use the `UnitDomain` on it. A `UnitDomain` is indivisible.

source

36.8 Dagger.load

`Dagger.load` — *Method*.

```
load(ctx, file_path)
```

Load an Union{Chunk, Thunk} from a file.

source

36.9 Dagger.load

`Dagger.load` — *Method*.

```
load(ctx, ::Type{Chunk}, fpath, io)
```

Load a Chunk object from a file, the file path is required for creating a FileReader object
source

36.10 Dagger.save

`Dagger.save` — *Method*.

```
save(io::IO, val)
```

Save a value into the IO buffer. In the case of arrays and sparse matrices, this will save it in a memory-mappable way.
`load(io::IO, t::Type, domain)` will load the object given its domain
source

36.11 Dagger.save

`Dagger.save` — *Method*.

```
save(ctx, chunk::Union{Chunk, Thunk}, file_path::AbsractString)
```

Save a chunk to a file at `file_path`.
source

36.12 Dagger.save

`Dagger.save` — *Method*.
special case distmem writing - write to disk on the process with the chunk.
source

Chapter 37

ShowItLikeYouBuildIt

37.1 ShowItLikeYouBuildIt.showarg

`ShowItLikeYouBuildIt.showarg` — *Method.*

```
showarg(stream::IO, x)
```

Show `x` as if it were an argument to a function. This function is used in the printing of "type summaries" in terms of sequences of function calls on objects.

The fallback definition is to print `x` as `::$(typeof(x))`, representing argument `x` in terms of its type. However, you can specialize this function for specific types to customize printing.

Example

A SubArray created as `view(a, :, 3, 2:5)`, where `a` is a 3-dimensional Float64 array, has type

```
SubArray{Float64,2,Array{Float64,3},Tuple{Colon,Int64,UnitRange{Int64}},false}
```

and this type will be printed in the summary. To change the printing of this object to

```
view(::Array{Float64,3}, Colon(), 3, 2:5)
```

you could define

```
function ShowItLikeYouBuildIt.showarg(io::IO, v::SubArray)
    print(io, "view(")
    showarg(io, parent(v))
    print(io, ", ", join(v.indexes, ", "))
    print(io, ')')
end
```

Note that we're calling `showarg` recursively for the parent array type. Printing the parent as `::Array{Float64,3}` is the fallback behavior, assuming no specialized method for `Array` has been defined. More generally, this would display as

```
view(<a>, Colon(), 3, 2:5)
```

where <a> is the output of `showarg` for `a`.

This printing might be activated any time `v` is a field in some other container, or if you specialize `Base.summary` for `SubArray` to call `summary_build`.

See also: summary_build.

source

37.2 ShowItLikeYouBuildIt.summary_build

`ShowItLikeYouBuildIt.summary_build` — *Function.*

```
summary_build(A::AbstractArray, [cthresh])
```

Return a string representing `A` in terms of the sequence of function calls that might be used to create `A`, along with information about `A`'s size or indices and element type. This function should never be called directly, but instead used to specialize `Base.summary` for specific `AbstractArray` subtypes. For example, if you want to change the summary of SubArray, you might define

```
Base.summary(v::SubArray) = summary_build(v)
```

This function goes hand-in-hand with `showarg`. If you have defined a `showarg` method for `SubArray` as in the documentation for `showarg`, then the summary of a SubArray might look like this:

```
34 view(::Array{Float64,3}, :, 3, 2:5) with element type Float64
```

instead of this:

```
34 SubArray{Float64,2,Array{Float64,3},Tuple{Colon,Int64,UnitRange{Int64}},false}
```

The optional argument `cthresh` is a "complexity threshold"; objects with type descriptions that are less complex than the specified threshold will be printed using the traditional type-based summary. The default value is `n+1`, where `n` is the number of parameters in `typeof(A)`. The complexity is calculated with `type_complexity`. You can choose a `cthresh` of 0 if you want to ensure that your `showarg` version is always used.

See also: showarg, type_complexity.

source

37.3 ShowItLikeYouBuildIt.type_complexity

`ShowItLikeYouBuildIt.type_complexity` — *Method.*

```
type_complexity(T::Type) -> c
```

Return an integer `c` representing a measure of the "complexity" of type `T`. For unnested types, `c = n+1` where `n` is the number of parameters in type `T`. However, `type_complexity` calls itself recursively on the parameters, so if the parameters have their own parameters then the complexity of the type will be (potentially much) higher than this.

Examples

```
# Array{Float64,2}
julia> a = rand(3,5);

julia> type_complexity(typeof(a))
3

julia> length(typeof(a).parameters)+1
3

# Create an object that has type:
#    SubArray{Int64,2,Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}},Tuple{StepRa
julia> r = reshape(1:9, 3, 3);

julia> v = view(r, 1:2:3, :);

julia> type_complexity(typeof(v))
15

julia> length(typeof(v).parameters)+1
6
```

The second example indicates that the total complexity of `v`'s type is considerably higher than the complexity of just its "outer" SubArray type.

source

Chapter 38

CoordinateTransformations

38.1 CoordinateTransformations.cameramap

`CoordinateTransformations.cameramap` — *Method*.

```
cameramap()
cameramap(scale)
cameramap(scale, offset)
```

Create a transformation that takes points in real space (e.g. 3D) and projects them through a perspective transformation onto the focal plane of an ideal (pinhole) camera with the given properties.

The `scale` sets the scale of the screen. For a standard digital camera, this would be `scale = focal_length / pixel_size`. Non-square pixels are supported by providing a pair of scales in a tuple, `scale = (scale_x, scale_y)`. Positive scales represent a camera looking in the +z axis with a virtual screen in front of the camera (the x,y coordinates are not inverted compared to 3D space). Note that points behind the camera (with negative z component) will be projected (and inverted) onto the image coordinates and it is up to the user to cull such points as necessary.

The `offset = (offset_x, offset_y)` is used to define the origin in the imaging plane. For instance, you may wish to have the point (0,0) represent the top-left corner of your imaging sensor. This measurement is in the units after applying `scale` (e.g. pixels).

(see also `PerspectiveMap`)

source

38.2 CoordinateTransformations.compose

`CoordinateTransformations.compose` — *Method*.

```
compose(trans1, trans2)
```

`trans1  trans2`

Take two transformations and create a new transformation that is equivalent to successively applying `trans2` to the coordinate, and then `trans1`. By default will create a `ComposedTransformation`, however this method can be overloaded for efficiency (e.g. two affine transformations naturally compose to a single affine transformation).

source

38.3 CoordinateTransformations.recenter

`CoordinateTransformations.recenter` — *Method*.

```
recenter(trans::Union{AbstractMatrix,Transformation}, origin::AbstractVector) -> ctrans
```

Return a new transformation `ctrans` such that point `origin` serves as the origin-of-coordinates for `trans`. Translation by `origin` occurs both before and after applying `trans`, so that if `trans` is linear we have

```
ctrans(origin) == origin
```

As a consequence, `recenter` only makes sense if the output space of `trans` is isomorphic with the input space.

For example, if `trans` is a rotation matrix, then `ctrans` rotates space around `origin`.

source

38.4 CoordinateTransformations.transform_deriv

`CoordinateTransformations.transform_deriv` — *Method*.

```
transform_deriv(trans::Transformation, x)
```

A matrix describing how differentials on the parameters of `x` flow through to the output of transformation `trans`.

source

38.5 CoordinateTransformations.transform_deriv_params

`CoordinateTransformations.transform_deriv_params` — *Method*.

```
transform_deriv_params(trans::AbstractTransformation, x)
```

A matrix describing how differentials on the parameters of `trans` flow through to the output of transformation `trans` given input `x`.

source

Chapter 39

Graphics

39.1 Graphics.aspect_ratio

`Graphics.aspect_ratio` — *Method*.

```
aspect_ratio(bb::BoundingBox) -> r
```

Compute the ratio `r` of the height and width of `bb`.
source

39.2 Graphics.deform

`Graphics.deform` — *Method*.

```
deform(bb::BoundingBox, l, r, t, b) -> bbnew
```

Add `l` (left), `r` (right), `t` (top), and `b` (bottom) to the edges of a BoundingBox.
source

39.3 Graphics.inner_canvas

`Graphics.inner_canvas` — *Method*.

```
inner_canvas(c::GraphicsContext, device::BoundingBox, user::BoundingBox)
inner_canvas(c::GraphicsContext, x, y, w, h, l, r, t, b)
```

Create a rectangular drawing area inside `device` (represented in device-coordinates), giving it user-coordinates `user`. Any drawing that occurs outside this box is clipped.

`x`, `y`, `w`, and `h` are an alternative parametrization of `device`, and `l`, `r`, `t`, `b` parametrize `user`.

See also: `set_coordinates`.
source

39.4 Graphics.isinside

`Graphics.isinside` — *Method*.

```
isinside(bb::BoundingBox, p::Point) -> tf::Bool
isinside(bb::BoundingBox, x, y) -> tf::Bool
```

Determine whether the point lies within `bb`.
source

39.5 Graphics.rotate

`Graphics.rotate` — *Method*.

```
rotate(bb::BoundingBox, angle, o) -> bbnew
```

Rotate `bb` around `o` by `angle`, returning the BoundingBox that encloses the vertices of the rotated box.
source

39.6 Graphics.rotate

`Graphics.rotate` — *Method*.

```
rotate(p::Vec2, angle::Real, o::Vec2) -> pnew
```

Rotate `p` around `o` by `angle`.
source

39.7 Graphics.set_coordinates

`Graphics.set_coordinates` — *Method*.

```
set_coordinates(c::GraphicsContext, device::BoundingBox, user::BoundingBox)
set_coordinates(c::GraphicsContext, user::BoundingBox)
```

Set the device->user coordinate transformation of `c` so that `device`, expressed in "device coordinates" (pixels), is equivalent to `user` as expressed in "user coordinates". If `device` is omitted, it defaults to the full span of `c`, `BoundingBox(0, width(c), 0, height(c))`.
See also `get_matrix`, `set_matrix`.
source

39.8 Graphics.shift

`Graphics.shift` — *Method.*

```
shift(bb::BoundingBox, x, y) -> bbnew
```

Shift center by (x,y), keeping width & height fixed.
source

Chapter 40

MacroTools

40.1 MacroTools.inexpr

`MacroTools.inexpr` — *Method.*

```
inexpr(expr, x)
```

Simple expression match; will return `true` if the expression `x` can be found inside `expr`.

```
inexpr(:(2+2), 2) == true
```

source

40.2 MacroTools.isdef

`MacroTools.isdef` — *Method.*

Test for function definition expressions.

source

40.3 MacroTools.isexpr

`MacroTools.isexpr` — *Method.*

```
isexpr(x, ts...)
```

Convenient way to test the type of a Julia expression. Expression heads and types are supported, so for example you can call

```
isexpr(expr, String, :string)
```

to pick up on all string-like expressions.

source

40.4 MacroTools.namify

`MacroTools.namify` — *Method*.

An easy way to get pull the (function/type) name out of expressions like `foo{T}` or `Bar{T} <: Vector{T}`.

source

40.5 MacroTools.prettify

`MacroTools.prettify` — *Method*.

```
prettify(ex)
```

Makes generated code generaly nicer to look at.

source

40.6 MacroTools.rmlines

`MacroTools.rmlines` — *Method*.

```
rmlines(x)
```

Remove the line nodes from a block or array of expressions.

Compare `quote end` vs `rmlines(quote end)`

source

40.7 MacroTools.splitarg

`MacroTools.splitarg` — *Method*.

```
splitarg(arg)
```

Match function arguments (whether from a definition or a function call) such as `x::Int=2` and return `(arg_name, arg_type, is_splat, default)`. `arg_name` and `default` are `nothing` when they are absent. For example:

[] ¿ map(splitarg, (:(f(a=2, x::Int=nothing, y, args...))).args[2:end]) 4-element Array{Tuple{Symbol,Symbol,Bool,Any},1}: (:a, :Any, false, 2) (:x, :Int, false, :nothing) (:y, :Any, false, nothing) (:args, :Any, true, nothing)

source

40.8 MacroTools.splitdef

`MacroTools.splitdef` — *Method*.

```
splitdef(fdef)
```

Match any function definition
[] function name{params}(args; kwargs)::rtype where {whereparams} body end
and return `Dict(:name=>..., :args=>..., etc.)`. The definition can be rebuilt by calling `MacroTools.combinedef(dict)`, or explicitly with

```
rtype = get(dict, :rtype, :Any)
all_params = [get(dict, :params, [])..., get(dict, :whereparams, [])...]
:(function $(dict[:name]){$(all_params...)}($(dict[:args]...);
                                            $(dict[:kwargs]...))::$rtype
      $(dict[:body])
  end)
```

source

40.9 MacroTools.unblock

`MacroTools.unblock` — *Method*.

```
unblock(expr)
```

Remove outer `begin` blocks from an expression, if the block is redundant (i.e. contains only a single expression).
source

Chapter 41

ImageMorphology

41.1 ImageMorphology.bothat

`ImageMorphology.bothat` — *Function.*

`imgbh = bothat(img, [region])` performs `bottom hat` of an image, which is defined as its morphological closing minus the original image. `region` allows you to control the dimensions over which this operation is performed.

source

41.2 ImageMorphology.closing

`ImageMorphology.closing` — *Function.*

`imgc = closing(img, [region])` performs the `closing` morphology operation, equivalent to `erode(dilate(img))`. `region` allows you to control the dimensions over which this operation is performed.

source

41.3 ImageMorphology.dilate

`ImageMorphology.dilate` — *Function.*

```
imgd = dilate(img, [region])
```

perform a max-filter over nearest-neighbors. The default is 8-connectivity in 2d, 27-connectivity in 3d, etc. You can specify the list of dimensions that you want to include in the connectivity, e.g., `region = [1,2]` would exclude the third dimension from filtering.

source

41.4 ImageMorphology.erode

`ImageMorphology.erode` — *Function*.

```
imge = erode(img, [region])
```

perform a min-filter over nearest-neighbors. The default is 8-connectivity in 2d, 27-connectivity in 3d, etc. You can specify the list of dimensions that you want to include in the connectivity, e.g., `region = [1,2]` would exclude the third dimension from filtering.

source

41.5 ImageMorphology.morphogradient

`ImageMorphology.morphogradient` — *Function*.

`imgmg = morphogradient(img, [region])` returns morphological gradient of the image, which is the difference between the dilation and the erosion of a given image. `region` allows you to control the dimensions over which this operation is performed.

source

41.6 ImageMorphology.morpholaplace

`ImageMorphology.morpholaplace` — *Function*.

`imgml = morpholaplace(img, [region])` performs `Morphological Laplacian` of an image, which is defined as the arithmetic difference between the internal and the external gradient. `region` allows you to control the dimensions over which this operation is performed.

source

41.7 ImageMorphology.opening

`ImageMorphology.opening` — *Function*.

`imgo = opening(img, [region])` performs the `opening` morphology operation, equivalent to `dilate(erode(img))`. `region` allows you to control the dimensions over which this operation is performed.

source

41.8 ImageMorphology.tophat

`ImageMorphology.tophat` — *Function*.

`imgth = tophat(img, [region])` performs `top hat` of an image, which is defined as the image minus its morphological opening. `region` allows you to control the dimensions over which this operation is performed.

source

Chapter 42

Contour

42.1 Contour.contour

`Contour.contour` — *Method*.

`contour(x, y, z, level::Number)` Trace a single contour level, indicated by the argument `level`.

You'll usually call `lines` on the output of `contour`, and then iterate over the result.

source

42.2 Contour.contours

`Contour.contours` — *Method*.

`contours(x,y,z,levels)` Trace the contour levels indicated by the `levels` argument.

source

42.3 Contour.contours

`Contour.contours` — *Method*.

`contours` returns a set of isolines.

You'll usually call `levels` on the output of `contours`.

source

42.4 Contour.contours

`Contour.contours` — *Method*.

`contours(x,y,z,Nlevels::Integer)` Trace `Nlevels` contour levels at heights chosen by the library (using the `contourlevels` function).

source

42.5 Contour.contours

`Contour.contours` — *Method*.

`contours(x,y,z)` Trace 10 automatically chosen contour levels.

source

42.6 Contour.coordinates

`Contour.coordinates` — *Method*.

`coordinates(c)` Returns the coordinates of the vertices of the contour line as a tuple of lists.

source

42.7 Contour.level

`Contour.level` — *Method*.

`level(c)` Indicates the `z`-value at which the contour level `c` was traced.

source

42.8 Contour.levels

`Contour.levels` — *Method*.

Turns the output of `contours` into an iterable with each of the traced contour levels. Each of the objects support `level` and `coordinates`.

source

42.9 Contour.lines

`Contour.lines` — *Method*.

`lines(c)` Extracts an iterable collection of isolines from a contour level. Use `coordinates` to get the coordinates of a line.

source

Chapter 43

Compose

43.1 Compose.circle

`Compose.circle` — *Function.*

```
circle(xs, ys, rs)
```

Arguments can be passed in arrays in order to perform multiple drawing operations.

source

43.2 Compose.circle

`Compose.circle` — *Function.*

```
circle(x, y, r)
```

Define a circle with its center at (`x`,`y`) and a radius of `r`.

source

43.3 Compose.circle

`Compose.circle` — *Method.*

```
circle()
```

Define a circle in the center of the current context with a diameter equal to the width of the context.

source

43.4 Compose.polygon

`Compose.polygon` — *Method.*

```
polygon(points)
```

Define a polygon. `points` is an array of `(x,y)` tuples that specify the corners of the polygon.
source

43.5 Compose.rectangle

`Compose.rectangle` — *Function.*

```
rectangle(x0s, y0s, widths, heights)
```

Arguments can be passed in arrays in order to perform multiple drawing operations at once.
source

43.6 Compose.rectangle

`Compose.rectangle` — *Function.*

```
rectangle(x0, y0, width, height)
```

Define a rectangle of size `width`x`height` with its top left corner at the point (`x`, `y`).
source

43.7 Compose.rectangle

`Compose.rectangle` — *Method.*

```
rectangle()
```

Define a rectangle that fills the current context completely.
source

43.8 Compose.text

`Compose.text` — *Function.*

```
text(x, y, value [,halign::HAlignment [,valign::VAlignment [,rot::Rotation]]])
```

Draw the text `value` at the position (`x`,`y`) relative to the current context.

The default alignment of the text is `hleft vbottom`. The vertical and horizontal alignment is specified by passing `hleft`, `hcenter` or `hright` and `vtop`, `vcenter` or `vbottom` as values for `halgin` and `valgin` respectively.

source

43.9 Compose.text

`Compose.text` — *Function.*

```
text(xs, ys, values [,haligns::HAlignment [,valigns::VAlignment [,rots::Rotation]]])
```

Arguments can be passed in arrays in order to perform multiple drawing operations at once.

source

Chapter 44

CoupledFields

44.1 CoupledFields.CVfn

`CoupledFields.CVfn` — *Method.*

```
CVfn{T<:Matrix{Float64}}(parm::T, X::T, Y::T, modelfn::Function, kerneltype::DataType;
```

Cross-validation function
source

44.2 CoupledFields.Rsq_adj

`CoupledFields.Rsq_adj` — *Method.*

```
Rsq_adj{T<:Array{Float64}}(Tx::T, Ty::T, df::Int):
```

Cross-validation metric
source

44.3 CoupledFields.bf

`CoupledFields.bf` — *Method.*

```
bf(x::Vector{Float64}, df::Int):
```

Compute a piecewise linear basis matrix for the vector x.
source

44.4 CoupledFields.cca

`CoupledFields.cca` — *Method*.

```
cca{T<:Matrix{Float64}}(v::Array{Float64}, X::T,Y::T):
```

Regularized Canonical Correlation Analysis using SVD.
source

44.5 CoupledFields.gKCCA

`CoupledFields.gKCCA` — *Method*.

```
gKCCA(par::Array{Float64}, X::Matrix{Float64}, Y::Matrix{Float64}, kpars::KernelParameters):
```

Compute the projection matrices and components for gKCCA.
source

44.6 CoupledFields.gradvecfield

`CoupledFields.gradvecfield` — *Method*.

```
gradvecfield{N<:Float64, T<:Matrix{Float64}}(par::Array{N}, X::T, Y::T, kpars::KernelParameters )
```

Compute the gradient vector or gradient matrix at each instance of the `X` and `Y` fields, by making use of a kernel feature space.
source

44.7 CoupledFields.whiten

`CoupledFields.whiten` — *Method*.

```
whiten(x::Matrix{Float64}, d::Float64; lat=nothing): Whiten matrix
```

`d` (0-1) Percentage variance of components to retain.
`lat` Latitudinal area-weighting.
source

Chapter 45

AxisAlgorithms

45.1 AxisAlgorithms.A_ldiv_B_md!

`AxisAlgorithms.A_ldiv_B_md!` — *Method*.

`A_ldiv_B_md!(dest, F, src, dim)` solves a tridiagonal system along dimension `dim` of `src`, storing the result in `dest`. Currently, `F` must be an LU-factorized tridiagonal matrix. If desired, you may safely use the same array for both `src` and `dest`, so that this becomes an in-place algorithm.

source

45.2 AxisAlgorithms.A_ldiv_B_md

`AxisAlgorithms.A_ldiv_B_md` — *Method*.

`A_ldiv_B_md(F, src, dim)` solves `F` for slices `b` of `src` along dimension `dim`, storing the result along the same dimension of the output. Currently, `F` must be an LU-factorized tridiagonal matrix or a Woodbury matrix.

source

45.3 AxisAlgorithms.A_mul_B_md!

`AxisAlgorithms.A_mul_B_md!` — *Method*.

`A_mul_B_md!(dest, M, src, dim)` computes `M*x` for slices `x` of `src` along dimension `dim`, storing the result in `dest`. `M` must be an `AbstractMatrix`. This uses an in-place naive algorithm.

source

45.4 AxisAlgorithms.A_mul_B_md

`AxisAlgorithms.A_mul_B_md` — *Method*.

`A_mul_B_md(M, src, dim)` computes `M*x` for slices `x` of `src` along dimension `dim`, storing the resulting vector along the same dimension of the output. `M` must be an `AbstractMatrix`. This uses an in-place naive algorithm.

source

45.5 AxisAlgorithms.A_mul_B_perm!

`AxisAlgorithms.A_mul_B_perm!` — *Method.*

`A_mul_B_perm!(dest, M, src, dim)` computes `M*x` for slices `x` of `src` along dimension `dim`, storing the result in `dest`. `M` must be an `AbstractMatrix`. This uses `permutedims` to make dimension `dim` into the first dimension, performs a standard matrix multiplication, and restores the original dimension ordering. In many cases, this algorithm exhibits the best cache behavior.

source

45.6 AxisAlgorithms.A_mul_B_perm

`AxisAlgorithms.A_mul_B_perm` — *Method.*

`A_mul_B_perm(M, src, dim)` computes `M*x` for slices `x` of `src` along dimension `dim`, storing the resulting vector along the same dimension of the output. `M` must be an `AbstractMatrix`. This uses `permutedims` to make dimension `dim` into the first dimension, performs a standard matrix multiplication, and restores the original dimension ordering. In many cases, this algorithm exhibits the best cache behavior.

source

Chapter 46

Libz

46.1 Libz.ZlibDeflateInputStream

`Libz.ZlibDeflateInputStream` — *Method.*

```
ZlibDeflateInputStream(input[; <keyword arguments>])
```

Construct a zlib deflate input stream to compress gzip/zlib data.
Arguments

- `input`: a byte vector, IO object, or BufferedInputStream containing data to compress.
- `bufsize::Integer=8192`: input and output buffer size.
- `gzip::Bool=true`: if true, data is gzip compressed; if false, zlib compressed.
- `level::Integer=6`: compression level in 1-9.
- `mem_level::Integer=8`: memory to use for compression in 1-9.
- `strategy=Z_DEFAULT_STRATEGY`: compression strategy; see zlib documentation.

source

46.2 Libz.ZlibDeflateOutputStream

`Libz.ZlibDeflateOutputStream` — *Method.*

```
ZlibDeflateOutputStream(output[; <keyword arguments>])
```

Construct a zlib deflate output stream to compress gzip/zlib data.
Arguments

- `output`: a byte vector, IO object, or BufferedInputStream to which compressed data should be written.
- `bufsize::Integer=8192`: input and output buffer size.
- `gzip::Bool=true`: if true, data is gzip compressed; if false, zlib compressed.
- `level::Integer=6`: compression level in 1-9.
- `mem_level::Integer=8`: memory to use for compression in 1-9.
- `strategy=Z_DEFAULT_STRATEGY`: compression strategy; see zlib documentation.

source

46.3 Libz.ZlibInflateInputStream

`Libz.ZlibInflateInputStream` — *Method*.

```
ZlibInflateInputStream(input[; <keyword arguments>])
```

Construct a zlib inflate input stream to decompress gzip/zlib data.
Arguments

- `input`: a byte vector, IO object, or BufferedInputStream containing compressed data to inflate.
- `bufsize::Integer=8192`: input and output buffer size.
- `gzip::Bool=true`: if true, data is gzip compressed; if false, zlib compressed.
- `reset_on_end::Bool=true`: on stream end, try to find the start of another stream.

source

46.4 Libz.ZlibInflateOutputStream

`Libz.ZlibInflateOutputStream` — *Method*.

```
ZlibInflateOutputStream(output[; <keyword arguments>])
```

Construct a zlib inflate output stream to decompress gzip/zlib data.
Arguments

- `output`: a byte vector, IO object, or BufferedInputStream to which decompressed data should be written.

- `bufsize::Integer=8192`: input and output buffer size.
- `gzip::Bool=true`: if true, data is gzip compressed; if false, zlib compressed.

source

46.5 Libz.adler32

`Libz.adler32` — *Function*.

```
adler32(data)
```

Compute the Adler-32 checksum over the `data` input. `data` can be `BufferedInputStream` or `Vector{UInt8}`.

source

46.6 Libz.crc32

`Libz.crc32` — *Function*.

```
crc32(data)
```

Compute the CRC-32 checksum over the `data` input. `data` can be `BufferedInputStream` or `Vector{UInt8}`.

source

Chapter 47

NullableArrays

47.1 NullableArrays.dropnull!

`NullableArrays.dropnull!` — *Method.*

```
dropnull!(X::NullableVector)
```

Remove null entries of `X` in-place and return a `Vector` view of the unwrapped `Nullable` entries.

source

47.2 NullableArrays.dropnull!

`NullableArrays.dropnull!` — *Method.*

```
dropnull!(X::AbstractVector)
```

Remove null entries of `X` in-place and return a `Vector` view of the unwrapped `Nullable` entries. If no nulls are present, this is a no-op and `X` is returned.

source

47.3 NullableArrays.dropnull

`NullableArrays.dropnull` — *Method.*

```
dropnull(X::AbstractVector)
```

Return a vector containing only the non-null entries of `X`, unwrapping `Nullable` entries. A copy is always returned, even when `X` does not contain any null values.

source

47.4 NullableArrays.nullify!

`NullableArrays.nullify!` — *Method*.

```
nullify!(X::NullableArray, I...)
```

This is a convenience method to set the entry of `X` at index `I` to be null
source

47.5 NullableArrays.padnull!

`NullableArrays.padnull!` — *Method*.

```
padnull!(X::NullableVector, front::Integer, back::Integer)
```

Insert `front` null entries at the beginning of `X` and add `back` null entries at the end of `X`. Returns `X`.
source

47.6 NullableArrays.padnull

`NullableArrays.padnull` — *Method*.

```
padnull(X::NullableVector, front::Integer, back::Integer)
```

return a copy of `X` with `front` null entries inserted at the beginning of the copy and `back` null entries inserted at the end.
source

Chapter 48

ImageMetadata

48.1 ImageAxes.data

`ImageAxes.data` — *Method*.

```
data(img::ImageMeta) -> array
```

Extract the data from `img`, omitting the properties dictionary. `array` shares storage with `img`, so changes to one affect the other.

See also: `properties`.

source

48.2 ImageMetadata.copyproperties

`ImageMetadata.copyproperties` — *Method*.

```
copyproperties(img::ImageMeta, data) -> imgnew
```

Create a new "image," copying the properties dictionary of `img` but using the data of the AbstractArray `data`. Note that changing the properties of `imgnew` does not affect the properties of `img`.

See also: `shareproperties`.

source

48.3 ImageMetadata.properties

`ImageMetadata.properties` — *Method*.

```
properties(imgmeta) -> props
```

Extract the properties dictionary `props` for `imgmeta`. `props` shares storage with `img`, so changes to one affect the other.

See also: `data`.

source

48.4 ImageMetadata.shareproperties

`ImageMetadata.shareproperties` — *Method*.

```
shareproperties(img::ImageMeta, data) -> imgnew
```

Create a new "image," reusing the properties dictionary of `img` but using the data of the AbstractArray `data`. The two images have synchronized properties; modifying one also affects the other.

See also: `copyproperties`.

source

48.5 ImageMetadata.spatialproperties

`ImageMetadata.spatialproperties` — *Method*.

```
spatialproperties(img)
```

Return a vector of strings, containing the names of properties that have been declared "spatial" and hence should be permuted when calling `permutedims`. Declare such properties like this:

```
img["spatialproperties"] = ["spacedirections"]
```

source

Chapter 49

LegacyStrings

49.1 LegacyStrings.utf16

`LegacyStrings.utf16` — *Function.*

```
utf16(::Union{Ptr{UInt16},Ptr{Int16}} [, length])
```

Create a string from the address of a NUL-terminated UTF-16 string. A copy is made; the pointer can be safely freed. If `length` is specified, the string does not have to be NUL-terminated.

source

49.2 LegacyStrings.utf16

`LegacyStrings.utf16` — *Method.*

```
utf16(s)
```

Create a UTF-16 string from a byte array, array of `UInt16`, or any other string type. (Data must be valid UTF-16. Conversions of byte arrays check for a byte-order marker in the first two bytes, and do not include it in the resulting string.)

Note that the resulting `UTF16String` data is terminated by the NUL code-point (16-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the `utf16(s)` conversion function. If you have a `UInt16` array `A` that is already NUL-terminated valid UTF-16 data, then you can instead use `UTF16String(A)` to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

source

49.3 LegacyStrings.utf32

`LegacyStrings.utf32` — *Function.*

```
utf32(::Union{Ptr{Char},Ptr{UInt32},Ptr{Int32}} [, length])
```

Create a string from the address of a NUL-terminated UTF-32 string. A copy is made; the pointer can be safely freed. If `length` is specified, the string does not have to be NUL-terminated.

source

49.4 LegacyStrings.utf32

`LegacyStrings.utf32` — *Method.*

```
utf32(s)
```

Create a UTF-32 string from a byte array, array of `Char` or `UInt32`, or any other string type. (Conversions of byte arrays check for a byte-order marker in the first four bytes, and do not include it in the resulting string.)

Note that the resulting `UTF32String` data is terminated by the NUL codepoint (32-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the `utf32(s)` conversion function. If you have a `Char` or `UInt32` array `A` that is already NUL-terminated UTF-32 data, then you can instead use `UTF32String(A)` to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

source

Chapter 50

BufferedStreams

50.1 BufferedStreams.anchor!

`BufferedStreams.anchor!` — *Method*.

Set the buffer's anchor to its current position.

source

50.2 BufferedStreams.fillbuffer!

`BufferedStreams.fillbuffer!` — *Method*.

Refill the buffer, optionally moving and retaining part of the data.

source

50.3 BufferedStreams.isanchored

`BufferedStreams.isanchored` — *Method*.

Return true if the stream is anchored.

source

50.4 BufferedStreams.peek

`BufferedStreams.peek` — *Method*.

Return the next byte from the input stream without advancing the position.

source

50.5 BufferedStreams.peekbytes!

`BufferedStreams.peekbytes!` — *Function*.

Fills `buffer` with bytes from `stream`'s buffer without advancing the position.

Unless the buffer is empty, we do not re-fill it. Therefore the number of bytes read is limited to the minimum of `nb` and the remaining bytes in the buffer.

source

50.6 BufferedStreams.takeanchored!

`BufferedStreams.takeanchored!` — *Method*.

Copy and return a byte array from the anchor up to, but not including the current position, also removing the anchor.

source

50.7 BufferedStreams.upanchor!

`BufferedStreams.upanchor!` — *Method*.

Remove and return a buffer's anchor.

source

Chapter 51

MbedTLS

51.1 MbedTLS.decrypt

`MbedTLS.decrypt` — *Function.*

`decrypt(cipher, key, msg, [iv]) -> Vector{UInt8}`

Decrypt a message using the given cipher. The cipher can be specified as

- a generic cipher (like CIPHER_AES)
- a specific cipher (like CIPHER_AES_256_CBC)
- a Cipher object

`key` is the symmetric key used for cryptography, given as either a String or a `Vector{UInt8}`. It must be the right length for the chosen cipher; for example, CIPHER_AES_256_CBC requires a 32-byte (256-bit) key.

`msg` is the message to be encoded. It should either be convertible to a String or be a `Vector{UInt8}`.

`iv` is the initialization vector, whose size must match the block size of the cipher (eg, 16 bytes for AES) and correspond to the iv used by the encryptor. By default, it will be set to all zeros.

source

51.2 MbedTLS.digest

`MbedTLS.digest` — *Function.*

`digest(kind::MDKind, msg::Vector{UInt8}, [key::Vector{UInt8}]) -> Vector{UInt8}`

Perform a digest of the given type on the given message (a byte array), return a byte array with the digest.

If an optional key is given, perform an HMAC digest.

source

51.3 MbedTLS.digest!

`MbedTLS.digest!` — *Function*.

`digest!(kind::MDKind, msg::Vector{UInt8}, [key::Vector{UInt8}, ], buffer::Vector{UInt8})`

In-place version of `digest` that stores the digest to `buffer`.

It is the user's responsibility to ensure that buffer is long enough to contain the digest. `get_size(kind::MDKind)` returns the appropriate size.

source

51.4 MbedTLS.encrypt

`MbedTLS.encrypt` — *Function*.

`encrypt(cipher, key, msg, [iv]) -> Vector{UInt8}`

Encrypt a message using the given cipher. The cipher can be specified as

- a generic cipher (like CIPHER_AES)
- a specific cipher (like CIPHER_AES_256_CBC)
- a Cipher object

`key` is the symmetric key used for cryptography, given as either a String or a `Vector{UInt8}`. It must be the right length for the chosen cipher; for example, CIPHER_AES_256_CBC requires a 32-byte (256-bit) key.

`msg` is the message to be encoded. It should either be convertible to a String or be a `Vector{UInt8}`.

`iv` is the initialization vector, whose size must match the block size of the cipher (eg, 16 bytes for AES). By default, it will be set to all zeros, which is not secure. For security reasons, it should be set to a different value for each encryption operation.

source

Chapter 52

DataValues

52.1 DataValues.dropna!

`DataValues.dropna!` — *Method.*

`dropna!(X::DataValueVector)`

Remove missing entries of `X` in-place and return a `Vector` view of the unwrapped `DataValue` entries.

source

52.2 DataValues.dropna!

`DataValues.dropna!` — *Method.*

`dropna!(X::AbstractVector)`

Remove missing entries of `X` in-place and return a `Vector` view of the unwrapped `DataValue` entries. If no missing values are present, this is a no-op and `X` is returned.

source

52.3 DataValues.dropna

`DataValues.dropna` — *Method.*

`dropna(X::AbstractVector)`

Return a vector containing only the non-missing entries of `X`, unwrapping `DataValue` entries. A copy is always returned, even when `X` does not contain any missing values.

source

52.4 DataValues.padna!

`DataValues.padna!` — *Method*.

```
padna!(X::DataValueVector, front::Integer, back::Integer)
```

Insert `front` null entries at the beginning of `X` and add `back` null entries at the end of `X`. Returns `X`.

source

52.5 DataValues.padna

`DataValues.padna` — *Method*.

```
padna(X::DataValueVector, front::Integer, back::Integer)
```

return a copy of `X` with `front` null entries inserted at the beginning of the copy and `back` null entries inserted at the end.

source

Chapter 53

OnlineStats

53.1 OnlineStats.mapblocks

`OnlineStats.mapblocks` — *Function.*

```
mapblocks(f::Function, b::Int, data, dim::ObsDimension = Rows())
```

Map `data` in batches of size `b` to the function `f`. If data includes an AbstractMatrix, the batches will be based on rows or columns, depending on `dim`. Most usage is through Julia's `do` block syntax.

Examples

```
s = Series(Mean())
mapblocks(10, randn(100)) do yi
    fit!(s, yi)
    info("nobs: $(nobs(s))")
end

x = [1 2 3 4;
     1 2 3 4;
     1 2 3 4;
     1 2 3 4]
mapblocks(println, 2, x)
mapblocks(println, 2, x, Cols())
```

source

53.2 OnlineStats.series

`OnlineStats.series` — *Method.*

```
series(o::OnlineStat...; kw...)
series(wt::Weight, o::OnlineStat...; kw...)
series(data, o::OnlineStat...; kw...)
series(data, wt::Weight, o::OnlineStat...; kw...)
```

Create a `Series` or `AugmentedSeries` based on whether keyword arguments `filter` and `transform` are present.

Example

```
series(-rand(100), Mean(), Variance(); filter = isfinite, transform = abs)
```

source

53.3 StatsBase.confint

`StatsBase.confint` — *Function*.

```
confint(b::Bootstrap, coverageprob = .95)
```

Return a confidence interval for a Bootstrap `b`.
source

53.4 StatsBase.fit!

`StatsBase.fit!` — *Method*.

```
fit!(s::Series, data)
```

Update a Series with more `data`.

Examples

```
# Univariate Series
s = Series(Mean())
fit!(s, randn(100))

# Multivariate Series
x = randn(100, 3)
s = Series(CovMatrix(3))
fit!(s, x)  # Same as fit!(s, x, Rows())
fit!(s, x', Cols())

# Model Series
x, y = randn(100, 10), randn(100)
s = Series(LinReg(10))
fit!(s, (x, y))
```

source

Chapter 54

NearestNeighbors

54.1 NearestNeighbors.injectdata

`NearestNeighbors.injectdata` — *Method*.

```
injectdata(datafreetree, data) -> tree
```

Returns the `KDTree`/`BallTree` wrapped by `datafreetree`, set up to use `data` for the points data.

source

54.2 NearestNeighbors.inrange

`NearestNeighbors.inrange` — *Method*.

```
inrange(tree::NNTree, points, radius [, sortres=false]) -> indices
```

Find all the points in the tree which is closer than `radius` to `points`. If `sortres = true` the resulting indices are sorted.

source

54.3 NearestNeighbors.knn

`NearestNeighbors.knn` — *Method*.

```
knn(tree::NNTree, points, k [, sortres=false]) -> indices, distances
```

Performs a lookup of the `k` nearest neigbours to the `points` from the data in the `tree`. If `sortres = true` the result is sorted such that the results are in the order of increasing distance to the point. `skip` is an optional predicate to determine if a point that would be returned should be skipped.

source

Chapter 55

IJulia

55.1 IJulia.installkernel

`IJulia.installkernel` — *Method*.

```
installkernel(name, options...; specname=replace(lowercase(name), " ", "-")
```

Install a new Julia kernel, where the given `options` are passed to the `julia` executable, and the user-visible kernel name is given by `name` followed by the Julia version.

Internally, the Jupyter name for the kernel (for the `jupyter kernelspec` command is given by the optional keyword `specname` (which defaults to `name`, converted to lowercase with spaces replaced by hyphens), followed by the Julia version number.

Both the `kernelspec` command (a `Cmd` object) and the new kernel name are returned by `installkernel`. For example:

```
kernelspec, kernelname = installkernel("Julia O3", "-O3")
```

creates a new Julia kernel in which `julia` is launched with the `-O3` optimization flag. The returned `kernelspec` command will be something like `jupyter kernelspec` (perhaps with different path), and `kernelname` will be something like `julia-O3-0.6` (in Julia 0.6). You could uninstall the kernel by running e.g.

```
run(`$kernelspec remove -f $kernelname`)
```

source

55.2 IJulia.notebook

`IJulia.notebook` — *Method*.

```
notebook(; dir=homedir(), detached=false)
```

The `notebook()` function launches the Jupyter notebook, and is equivalent to running `jupyter notebook` at the operating-system command-line. The advantage of launching the notebook from Julia is that, depending on how Jupyter was installed, the user may not know where to find the `jupyter` executable.

By default, the notebook server is launched in the user's home directory, but this location can be changed by passing the desired path in the `dir` keyword argument. e.g. `notebook(dir=pwd())` to use the current directory.

By default, `notebook()` does not return; you must hit ctrl-c or quit Julia to interrupt it, which halts Jupyter. So, you must leave the Julia terminal open for as long as you want to run Jupyter. Alternatively, if you run `notebook(detached=true)`, the `jupyter notebook` will launch in the background, and will continue running even after you quit Julia. (The only way to stop Jupyter will then be to kill it in your operating system's process manager.)

source

Chapter 56

WebSockets

56.1 Base.close

`Base.close` — *Method.*

`close(ws::WebSocket)`

Send a close message.
source

56.2 Base.read

`Base.read` — *Method.*

`read(ws::WebSocket)`

Read one non-control message from a WebSocket. Any control messages that are read will be handled by the handle_control_frame function. This function will not return until a full non-control message has been read. If the other side doesn't ever complete its message, this function will never return. Only the data (contents/body/payload) of the message will be returned from this function.
source

56.3 Base.write

`Base.write` — *Method.*
Write binary data; will be sent as one frame.
source

56.4 Base.write

`Base.write` — *Method*.

Write text data; will be sent as one frame.
source

56.5 WebSockets.send_ping

`WebSockets.send_ping` — *Method*.

Send a ping message, optionally with data.
source

56.6 WebSockets.send_pong

`WebSockets.send_pong` — *Method*.

Send a pong message, optionally with data.
source

Chapter 57

AutoGrad

57.1 AutoGrad.getval

`AutoGrad.getval` — *Method.*

```
getval(x)
```

Unbox `x` if it is a boxed value (`Rec`), otherwise return `x`.
source

57.2 AutoGrad.grad

`AutoGrad.grad` — *Function.*

```
grad(fun, argnum=1)
```

Take a function `fun(X...)->Y` and return another function `gfun(X...)->dXi` which computes its gradient with respect to positional argument number `argnum`. The function `fun` should be scalar-valued. The returned function `gfun` takes the same arguments as `fun`, but returns the gradient instead. The gradient has the same type and size as the target argument which can be a Number, Array, Tuple, or Dict.
source

57.3 AutoGrad.gradcheck

`AutoGrad.gradcheck` — *Method.*

```
gradcheck(f, w, x...; kwargs...)
```

Numerically check the gradient of `f(w,x...;o...)` with respect to its first argument `w` and return a boolean result.

The argument `w` can be a Number, Array, Tuple or Dict which in turn can contain other Arrays etc. Only the largest 10 entries in each numerical gradient array are checked by default. If the output of f is not a number, gradcheck constructs and checks a scalar function by taking its dot product with a random vector.

Keywords

- `gcheck=10`: number of largest entries from each numeric array in gradient `dw=(grad(f))(w,x...;o...)` compared to their numerical estimates.
- `verbose=false`: print detailed messages if true.
- `kwargs=[]`: keyword arguments to be passed to `f`.
- `delta=atol=rtol=cbrt(eps(w))`: tolerance parameters. See `isapprox` for their meaning.

source

57.4 AutoGrad.gradloss

`AutoGrad.gradloss` — *Function.*

```
gradloss(fun, argnum=1)
```

Another version of `grad` where the generated function returns a (gradient,value) pair.

source

Chapter 58

ComputationalResources

58.1 ComputationalResources.addresource

`ComputationalResources.addresource` — *Method.*

`addresource(T)`

Add `T` to the list of available resources. For example, `addresource(OpenCLLibs)` would indicate that you have a GPU and the OpenCL libraries installed.

source

58.2 ComputationalResources.haveresource

`ComputationalResources.haveresource` — *Method.*

`haveresource(T)`

Returns `true` if `T` is an available resource. For example, `haveresource(OpenCLLibs)` tests whether the `OpenCLLibs` have been added as an available resource. This function is typically used inside a module's `__init__` function.

Example:

[] The $_{init_{function for MyPackage:}}function_{_{init_{()}}...other initialization code, possibly setting the LOAD_PATH if haveresource(OpenC}$

source

58.3 ComputationalResources.rmresource

`ComputationalResources.rmresource` — *Method.*

`rmresource(T)`

Remove `T` from the list of available resources. For example, `rmresource(OpenCLLibs)` would indicate that any future package loads should avoid loading their specializations for OpenCL.

source

Chapter 59

Clustering

59.1 Clustering.dbscan

`Clustering.dbscan` — *Method.*

```
dbscan(points, radius ; leafsize = 20, min_neighbors = 1, min_cluster_size = 1) -> clus
```

Cluster points using the DBSCAN (density-based spatial clustering of applications with noise) algorithm.

Arguments

- `points`: matrix of points
- `radius::Real`: query radius

Keyword Arguments

- `leafsize::Int`: number of points binned in each leaf node in the `KDTree`
- `min_neighbors::Int`: minimum number of neighbors to be a core point
- `min_cluster_size::Int`: minimum number of points to be a valid cluster

Output

- `Vector{DbscanCluster}`: an array of clusters with the id, size core indices and boundary indices

Example:

[] points = randn(3, 10000) clusters = dbscan(points, 0.05, $\min_n eighbors = 3, min_c luster_s ize$ =20)$clusterswithlessthan20pointswillbediscarded$

source

59.2 Clustering.mcl

`Clustering.mcl` — *Method.*

```
mcl(adj::Matrix; [keyword arguments])::MCLResult
```

Identify clusters in the weighted graph using Markov Clustering Algorithm (MCL).

Arguments

- `adj::Matrix{Float64}`: adjacency matrix that defines the weighted graph to cluster
- `add_loops::Bool`: whether edges of weight 1.0 from the node to itself should be appended to the graph (enabled by default)
- `expansion::Number`: MCL expansion constant (2)
- `inflation::Number`: MCL inflation constant (2.0)
- `save_final_matrix::Bool`: save final equilibrium state in the result, otherwise leave it empty; disabled by default, could be useful if MCL doesn't converge
- `max_iter::Integer`: max number of MCL iterations
- `tol::Number`: MCL adjacency matrix convergence threshold
- `prune_tol::Number`: pruning threshold
- `display::Symbol`: `:none` for no output or `:verbose` for diagnostic messages

See original MCL implementation.
Ref: Stijn van Dongen, "Graph clustering by flow simulation", 2001
source

Chapter 60

JuliaWebAPI

60.1 JuliaWebAPI.apicall

`JuliaWebAPI.apicall` — *Method*.

Calls a remote api `cmd` with `args...` and `data...`. The response is formatted as specified by the formatter specified in `conn`.

source

60.2 JuliaWebAPI.fnresponse

`JuliaWebAPI.fnresponse` — *Method*.

extract and return the response data as a direct function call would have returned but throw error if the call was not successful.

source

60.3 JuliaWebAPI.httpresponse

`JuliaWebAPI.httpresponse` — *Method*.

construct an HTTP Response object from the API response

source

60.4 JuliaWebAPI.process

`JuliaWebAPI.process` — *Method*.

start processing as a server

source

60.5 JuliaWebAPI.register

`JuliaWebAPI.register` — *Method*.

Register a function as API call. TODO: validate method belongs to module? source

Chapter 61

DecisionTree

61.1 DecisionTree.apply_adaboost_stumps_proba

`DecisionTree.apply_adaboost_stumps_proba` — *Method*.

```
apply_adaboost_stumps_proba(stumps::Ensemble, coeffs, features, labels::Vector)
```

computes P(L=label|X) for each row in `features`. It returns a `N_row x n_labels` matrix of probabilities, each row summing up to 1.

`col_labels` is a vector containing the distinct labels (eg. ["versicolor", "virginica", "setosa"]). It specifies the column ordering of the output matrix.

source

61.2 DecisionTree.apply_forest_proba

`DecisionTree.apply_forest_proba` — *Method*.

```
apply_forest_proba(forest::Ensemble, features, col_labels::Vector)
```

computes P(L=label|X) for each row in `features`. It returns a `N_row x n_labels` matrix of probabilities, each row summing up to 1.

`col_labels` is a vector containing the distinct labels (eg. ["versicolor", "virginica", "setosa"]). It specifies the column ordering of the output matrix.

source

61.3 DecisionTree.apply_tree_proba

`DecisionTree.apply_tree_proba` — *Method*.

```
apply_tree_proba(::Node, features, col_labels::Vector)
```

computes P(L=label|X) for each row in `features`. It returns a `N_row x n_labels` matrix of probabilities, each row summing up to 1.

`col_labels` is a vector containing the distinct labels (eg. ["versicolor", "virginica", "setosa"]). It specifies the column ordering of the output matrix.

source

Chapter 62

Blosc

62.1 Blosc.compress

`Blosc.compress` — *Function*.

```
compress(data; level=5, shuffle=true, itemsize)
```

Return a `Vector{UInt8}` of the Blosc-compressed `data`, where `data` is an array or a string.

The `level` keyword indicates the compression level (between `0`=no compression and `9`=max), `shuffle` indicates whether to use Blosc's shuffling preconditioner, and the shuffling preconditioner is optimized for arrays of binary items of size (in bytes) `itemsize` (defaults to `sizeof(eltype(data))` for arrays and the size of the code units for strings).

source

62.2 Blosc.compress!

`Blosc.compress!` — *Function*.

```
compress!(dest::Vector{UInt8}, src; kws...)
```

Like `compress(src; kws...)`, but writes to a pre-allocated array `dest` of bytes. The return value is the size in bytes of the data written to `dest`, or `0` if the buffer was too small.

source

62.3 Blosc.decompress!

`Blosc.decompress!` — *Method*.

```
decompress!(dest::Vector{T}, src::Vector{UInt8})
```

Like `decompress`, but uses a pre-allocated destination buffer `dest`, which is resized as needed to store the decompressed data from `src`.

source

62.4 Blosc.decompress

`Blosc.decompress` — *Method.*

```
decompress(T::Type, src::Vector{UInt8})
```

Return the compressed buffer `src` as an array of element type `T`.

source

Chapter 63

Missings

63.1 Missings.allowmissing

`Missings.allowmissing` — *Method.*

```
allowmissing(x::AbstractArray)
```

Return an array equal to `x` allowing for `missing` values, i.e. with an element type equal to `Union{eltype(x), Missing}`.

When possible, the result will share memory with `x` (as with `convert`).

See also: `disallowmissing`

source

63.2 Missings.disallowmissing

`Missings.disallowmissing` — *Method.*

```
disallowmissing(x::AbstractArray)
```

Return an array equal to `x` not allowing for `missing` values, i.e. with an element type equal to `Missings.T(eltype(x))`.

When possible, the result will share memory with `x` (as with `convert`). If `x` contains missing values, a `MethodError` is thrown.

See also: `allowmissing`

source

63.3 Missings.levels

`Missings.levels` — *Method.*

```
levels(x)
```

Return a vector of unique values which occur or could occur in collection `x`, omitting `missing` even if present. Values are returned in the preferred order for the collection, with the result of `sort` as a default.

Contrary to `unique`, this function may return values which do not actually occur in the data, and does not preserve their order of appearance in `x`.

source

Chapter 64

Parameters

64.1 Parameters.reconstruct

`Parameters.reconstruct` — *Method*.

Make a new instance of a type with the same values as the input type except for the fields given in the associative second argument or as keywords.

[] struct A; a; b end a = A(3,4) b = reconstruct(a, [(:b, 99)]) ==A(3,99)

source

64.2 Parameters.type2dict

`Parameters.type2dict` — *Method*.

Transforms a type-instance into a dictionary.

```
julia> type T
           a
           b
       end

julia> type2dict(T(4,5))
Dict{Symbol,Any} with 2 entries:
  :a => 4
  :b => 5
```

source

64.3 Parameters.with_kw

`Parameters.with_kw` — *Function*.

This function is called by the `@with_kw` macro and does the syntax transformation from:

[] @with$_k$*wstructMM*{*R*}*r* :: *R* =1000.*a* :: *Rend*
into
[] struct MM{R} r::R a::R MM{R}(r,a) where {R} = new(r,a) MM{R}(;r=1000., a=error("no default for a")) where {R} = MM{R}(r,a) inner kw, type-paras are required when calling end MM(r::R,a::R) where {R} = MM{R}(r,a) default outer positional constructor MM(;r=1000,a=error("no default for a")) = MM(r,a) outer kw, so no type-paras are needed when calling MM(m::MM; kws...) = reconstruct(mm,kws) MM(m::MM, di::Union{Associative, Tuple{Symbol,Any}}) = reconstruct(mm, di) macro unpack$_M$*M*(*varname*)*esc*(*quoter* = *varname.ra* = *varname.aend*)*endmacropack*$_M$*M*(

source

Chapter 65

HDF5

65.1 HDF5.h5open

`HDF5.h5open` — *Function*.

```
h5open(filename::AbstractString, mode::AbstractString="r"; swmr=false)
```

Open or create an HDF5 file where `mode` is one of:

- "r" read only
- "r+" read and write
- "w" read and write, create a new file (destroys any existing contents)

Pass `swmr=true` to enable (Single Writer Multiple Reader) SWMR write access for "w" and "r+", or SWMR read access for "r".

source

65.2 HDF5.h5open

`HDF5.h5open` — *Method*.

```
function h5open(f::Function, args...; swmr=false)
```

Apply the function f to the result of `h5open(args...;kwargs...)` and close the resulting `HDF5File` upon completion. For example with a `do` block:

```
h5open("foo.h5","w") do h5
    h5["foo"]=[1,2,3]
end
```

source

65.3 HDF5.ishdf5

`HDF5.ishdf5` — *Method.*

```
ishdf5(name::AbstractString)
```

Returns `true` if `name` is a path to a valid hdf5 file, `false` otherwise.
source

65.4 HDF5.set_dims!

`HDF5.set_dims!` — *Method.*

```
set_dims!(dset::HDF5Dataset, new_dims::Dims)
```

Change the current dimensions of a dataset to `new_dims`, limited by `max_dims = get_dims(dset)[2]`. Reduction is possible and leads to loss of truncated data.
source

Chapter 66

HttpServer

66.1 Base.run

`Base.run` — *Method.*

`run` starts `server`
Functionality:

- Accepts incoming connections and instantiates each `Client`.
- Manages the `client.id` pool.
- Spawns a new `Task` for each connection.
- Blocks forever.

Method accepts following keyword arguments:

- host - binding address
- port - binding port
- ssl - SSL configuration. Use this argument to enable HTTPS support.

Can be either an MbedTLS.SSLConfig object that already has associated certificates, or a tuple of an MbedTLS.CRT (certificate) and MbedTLS.PKContext (private key)). In the latter case, a configuration with reasonable defaults will be used.

- socket - named pipe/domain socket path. Use this argument to enable Unix socket support.

It's available only on Unix. Network options are ignored.
Compatibility:

- for backward compatibility use `run(server::Server, port::Integer)`

Example:

```
server = Server() do req, res
  "Hello world"
end

# start server on localhost
run(server, host=IPv4(127,0,0,1), port=8000)
# or
run(server, 8000)
```

source

66.2 Base.write

`Base.write` — *Method*.

Converts a `Response` to an HTTP response string

source

66.3 HttpServer.setcookie!

`HttpServer.setcookie!` — *Function*.

Sets a cookie with the given name, value, and attributes on the given response object.

source

Chapter 67

MappedArrays

67.1 MappedArrays.mappedarray

`MappedArrays.mappedarray` — *Method*.

```
mappedarray(f, A)
```

creates a view of the array `A` that applies `f` to every element of `A`. The view is read-only (you can get values but not set them).

source

67.2 MappedArrays.mappedarray

`MappedArrays.mappedarray` — *Method*.

```
mappedarray((f, finv), A)
```

creates a view of the array `A` that applies `f` to every element of `A`. The inverse function, `finv`, allows one to also set values of the view and, correspondingly, the values in `A`.

source

67.3 MappedArrays.of_eltype

`MappedArrays.of_eltype` — *Method*.

```
of_eltype(T, A)
of_eltype(val::T, A)
```

creates a view of `A` that lazily-converts the element type to `T`.

source

Chapter 68

TextParse

68.1 TextParse.csvread

`TextParse.csvread` — *Function.*

`csvread(file::Union{String,IO}, delim=','; <arguments>...)`

Read CSV from `file`. Returns a tuple of 2 elements:

1. A tuple of columns each either a `Vector`, `DataValueArray` or `PooledArray`
2. column names if `header_exists=true`, empty array otherwise

Arguments:

- `file`: either an IO object or file name string
- `delim`: the delimiter character
- `spacedelim`: (Bool) parse space-delimited files. `delim` has no effect if true.
- `quotechar`: character used to quote strings, defaults to `"`
- `escapechar`: character used to escape quotechar in strings. (could be the same as quotechar)
- `pooledstrings`: whether to try and create PooledArray of strings
- `nrows`: number of rows in the file. Defaults to `0` in which case we try to estimate this.
- `skiplines_begin`: skips specified number of lines at the beginning of the file
- `header_exists`: boolean specifying whether CSV file contains a header

- `nastrings`: strings that are to be considered NA. Defaults to `TextParse.NA_STRINGS`
- `colnames`: manually specified column names. Could be a vector or a dictionary from Int index (the column) to String column name.
- `colparsers`: Parsers to use for specified columns. This can be a vector or a dictionary from column name / column index (Int) to a "parser". The simplest parser is a type such as Int, Float64. It can also be a `dateformat"..."`, see CustomParser if you want to plug in custom parsing behavior
- `type_detect_rows`: number of rows to use to infer the initial `colparsers` defaults to 20.

source

Chapter 69

LossFunctions

69.1 LearnBase.scaled

`LearnBase.scaled` — *Method.*

```
scaled(loss::SupervisedLoss, K)
```

Returns a version of `loss` that is uniformly scaled by `K`. This function dispatches on the type of `loss` in order to choose the appropriate type of scaled loss that will be used as the decorator. For example, if `typeof(loss) <: DistanceLoss` then the given `loss` will be boxed into a `ScaledDistanceLoss`.

Note: If `typeof(K) <: Number`, then this method will poison the type-inference of the calling scope. This is because `K` will be promoted to a type parameter. For a typestable version use the following signature: `scaled(loss, Val{K})`

source

69.2 LossFunctions.weightedloss

`LossFunctions.weightedloss` — *Method.*

```
weightedloss(loss, weight)
```

Returns a weighted version of `loss` for which the value of the positive class is changed to be `weight` times its original, and the negative class `1 - weight` times its original respectively.

Note: If `typeof(weight) <: Number`, then this method will poison the type-inference of the calling scope. This is because `weight` will be promoted to a type parameter. For a typestable version use the following signature: `weightedloss(loss, Val{weight})`

source

Chapter 70

LearnBase

70.1 LearnBase.grad

`LearnBase.grad` — *Function*.

Return the gradient of the learnable parameters w.r.t. some objective

source

70.2 LearnBase.grad!

`LearnBase.grad!` — *Function*.

Do a backward pass, updating the gradients of learnable parameters and/or inputs

source

70.3 LearnBase.inputdomain

`LearnBase.inputdomain` — *Function*.

Returns an AbstractSet representing valid input values

source

70.4 LearnBase.targetdomain

`LearnBase.targetdomain` — *Function*.

Returns an AbstractSet representing valid output/target values

source

70.5 LearnBase.transform!

`LearnBase.transform!` — *Function*.

Do a forward pass, and return the output
source

Chapter 71

Juno

71.1 Juno.clearconsole

`Juno.clearconsole` — *Method*.

```
clearconsole()
```

Clear the console if Juno is used; does nothing otherwise.
source

71.2 Juno.input

`Juno.input` — *Function*.

```
input(prompt = "") -> "..."
```

Prompt the user to input some text, and return it. Optionally display a prompt.
source

71.3 Juno.selector

`Juno.selector` — *Method*.

```
selector([xs...]) -> x
```

Allow the user to select one of the `xs`.
`xs` should be an iterator of strings. Currently there is no fallback in other environments.
source

71.4 Juno.structure

`Juno.structure` — *Method*.

```
structure(x)
```

Display `x`'s underlying representation, rather than using its normal display method.

For example, `structure(:(2x+1))` displays the `Expr` object with its `head` and `args` fields instead of printing the expression.

source

Chapter 72

HttpParser

72.1 HttpParser.http_method_str

`HttpParser.http_method_str` — *Method*.

Returns a string version of the HTTP method.

source

72.2 HttpParser.http_parser_execute

`HttpParser.http_parser_execute` — *Method*.

Run a request through a parser with specific callbacks on the settings instance.

source

72.3 HttpParser.http_parser_init

`HttpParser.http_parser_init` — *Function*.

Intializes the Parser object with the correct memory.

source

72.4 HttpParser.parse_url

`HttpParser.parse_url` — *Method*.

Parse a URL

source

Chapter 73

ImageAxes

73.1 ImageAxes.istimeaxis

`ImageAxes.istimeaxis` — *Method*.

```
istimeaxis(ax)
```

Test whether the axis `ax` corresponds to time.
source

73.2 ImageAxes.timeaxis

`ImageAxes.timeaxis` — *Method*.

```
timeaxis(A)
```

Return the time axis, if present, of the array `A`, and `nothing` otherwise.
source

73.3 ImageAxes.timedim

`ImageAxes.timedim` — *Method*.

```
timedim(img) -> d::Int
```

Return the dimension of the array used for encoding time, or 0 if not using an axis for this purpose.

Note: if you want to recover information about the time axis, it is generally better to use `timeaxis`.
source

Chapter 74

Flux

74.1 Flux.GRU

`Flux.GRU` — *Method.*

```
GRU(in::Integer, out::Integer,  = tanh)
```

Gated Recurrent Unit layer. Behaves like an RNN but generally exhibits a longer memory span over sequences.

See this article for a good overview of the internals.

source

74.2 Flux.LSTM

`Flux.LSTM` — *Method.*

```
LSTM(in::Integer, out::Integer,  = tanh)
```

Long Short Term Memory recurrent layer. Behaves like an RNN but generally exhibits a longer memory span over sequences.

See this article for a good overview of the internals.

source

74.3 Flux.RNN

`Flux.RNN` — *Method.*

```
RNN(in::Integer, out::Integer,  = tanh)
```

The most basic recurrent layer; essentially acts as a `Dense` layer, but with the output fed back into the input each time step.

source

Chapter 75

IntervalSets

75.1 IntervalSets.:..

`IntervalSets.:..` — *Method.*

```
iv = l..r
```

Construct a ClosedInterval `iv` spanning the region from `l` to `r`.
source

75.2 IntervalSets.:

`IntervalSets.:` — *Method.*

```
iv = centerhalfwidth
```

Construct a ClosedInterval `iv` spanning the region from `center - halfwidth` to `center + halfwidth`.
source

75.3 IntervalSets.width

`IntervalSets.width` — *Method.*

```
w = width(iv)
```

Calculate the width (max-min) of interval `iv`. Note that for integers `l` and `r`, `width(l..r) = length(l:r) - 1`.
source

Chapter 76

Media

76.1 Media.getdisplay

`Media.getdisplay` — *Method*.

```
getdisplay(T)
```

Find out what output device `T` will display on.
source

76.2 Media.media

`Media.media` — *Method*.
`media(T)` gives the media type of the type `T`. The default is `Textual`.

```
media(Gadfly.Plot) == Media.Plot
```

source

76.3 Media.setdisplay

`Media.setdisplay` — *Method*.

```
setdisplay([input], T, output)
```

Display `T` objects using `output` when produced by `input`.
`T` is an object type or media type, e.g. `Gadfly.Plot` or `Media.Graphical`.

```
display(Editor(), Image, Console())
```

source

Chapter 77

Rotations

77.1 Rotations.isrotation

`Rotations.isrotation` — *Method*.

```
isrotation(r)
isrotation(r, tol)
```

Check whether `r` is a 33 rotation matrix, where `r * r` is within `tol` of the identity matrix (using the Frobenius norm). (`tol` defaults to `1000 * eps(eltype(r))`.)

source

77.2 Rotations.rotation_between

`Rotations.rotation_between` — *Method*.

```
rotation_between(from, to)
```

Compute the quaternion that rotates vector `from` so that it aligns with vector `to`, along the geodesic (shortest path).

source

Chapter 78

Mustache

78.1 Mustache.render

`Mustache.render` — *Method*.

Render a set of tokens with a view, using optional `io` object to print or store.

Arguments

- `io::IO`: Optional `IO` object.
- `tokens`: Either Mustache tokens, or a string to parse into tokens
- `view`: A view provides a context to look up unresolved symbols demarcated by mustache braces. A view may be specified by a dictionary, a module, a composite type, a vector, or keyword arguments.

source

78.2 Mustache.render_from_file

`Mustache.render_from_file` — *Method*.

Renders a template from `filepath` and `view`. If it has seen the file before then it finds the compiled `MustacheTokens` in `TEMPLATES` rather than calling `parse` a second time.

source

Chapter 79

TiledIteration

79.1 TiledIteration.padded_tilesize

`TiledIteration.padded_tilesize` — *Method*.

```
padded_tilesize(T::Type, kernelsize::Dims, [ncache=2]) -> tilesize::Dims
```

Calculate a suitable tile size to approximately maximize the amount of productive work, given a stencil of size `kernelsize`. The element type of the array is `T`. Optionally specify `ncache`, the number of such arrays that you'd like to have fit simultaneously in L1 cache.

This favors making the first dimension larger, since the first dimension corresponds to individual cache lines.

Examples

julia> padded_tilesize(UInt8, (3,3)) (768,18)
julia> padded_tilesize(UInt8, (3,3), 4) (512,12)
julia> padded_tilesize(Float64, (3,3)) (96,18)
julia> padded_tilesize(Float32, (3,3,3)) (64,6,6)
source

Chapter 80

Distances

80.1 Distances.renyi_divergence

`Distances.renyi_divergence` — *Function.*

```
RenyiDivergence(::Real)
renyi_divergence(P, Q, ::Real)
```

Create a Rnyi premetric of order .

Rnyi defined a spectrum of divergence measures generalising the Kullback–Leibler divergence (see `KLDivergence`). The divergence is not a semimetric as it is not symmetric. It is parameterised by a parameter , and is equal to Kullback–Leibler divergence at = 1:

At = 0, $R_0(P|Q) = -log(sum_{i:p_i>0}(q_i))$

At = 1, $R_1(P|Q) = sum_{i:p_i>0}(p_i log(p_i/q_i))$

At = , $R(P|Q) = log(sup_{i:p_i>0}(p_i/q_i))$

Otherwise $R(P|Q) = log(sum_{i:p_i>0}((p_i)/(q_i^{(} - 1)))/(- 1)$

Example:

```
julia> x = reshape([0.1, 0.3, 0.4, 0.2], 2, 2);

julia> pairwise(RenyiDivergence(0), x, x)
22 Array{Float64,2}:
 0.0  0.0
 0.0  0.0

julia> pairwise(Euclidean(2), x, x)
22 Array{Float64,2}:
 0.0       0.577315
 0.655407  0.0
```

source

Chapter 81

HttpCommon

81.1 HttpCommon.escapeHTML

`HttpCommon.escapeHTML` — *Method*.

escapeHTML(i::String)

Returns a string with special HTML characters escaped: &, <, >, “, ’

source

81.2 HttpCommon.parsequerystring

`HttpCommon.parsequerystring` — *Method*.

parsequerystring(query::String)

Convert a valid querystring to a Dict:

```
q = "foo=bar&baz=%3Ca%20href%3D%27http%3A%2F%2Fwww.hackershool.com%27%3Ehello%20world%21%3C%2Fa%3
parsequerystring(q)
# Dict{String,String} with 2 entries:
#   "baz" => "<a href='http://www.hackershool.com'>hello world!</a>"
#   "foo" => "bar"
```

source

Chapter 82

StaticArrays

82.1 StaticArrays.similar_type

`StaticArrays.similar_type` — *Function.*

```
similar_type(static_array)
similar_type(static_array, T)
similar_type(array, ::Size)
similar_type(array, T, ::Size)
```

Returns a constructor for a statically-sized array similar to the input array (or type) `static_array`/`array`, optionally with different element type `T` or size `Size`. If the input `array` is not a `StaticArray` then the `Size` is mandatory.

This differs from `similar()` in that the resulting array type may not be mutable (or define `setindex!()`), and therefore the returned type may need to be *constructed* with its data.

Note that the (optional) size *must* be specified as a static `Size` object (so the compiler can infer the result statically).

New types should define the signature `similar_type{A<:MyType,T,S}(::Type{A},::Type{T},::Siz` if they wish to overload the default behavior.

source

Chapter 83

SweepOperator

83.1 SweepOperator.sweep!

`SweepOperator.sweep!` — *Method.*

Symmetric sweep operator

Symmetric sweep operator of the matrix `A` on element `k`. `A` is overwritten. `inv = true` will perform the inverse sweep. Only the upper triangle is read and swept.

```
sweep!(A, k, inv = false)
```

Providing a Range, rather than an Integer, sweeps on each element in the range.

```
sweep!(A, first:last, inv = false)
```

Example:

[] x = randn(100, 10) xtx = x'x sweep!(xtx, 1) sweep!(xtx, 1, true)

source

Chapter 84

PaddedViews

84.1 PaddedViews.paddedviews

`PaddedViews.paddedviews` — *Method*.

```
Aspad = paddedviews(fillvalue, A1, A2, ....)
```

Pad the arrays `A1`, `A2`, ..., to a common size or set of axes, chosen as the span of axes enclosing all of the input arrays.

Example:

[] julia¿ a1 = reshape([1,2], 2, 1) 21 Array{Int64,2}: 1 2

julia¿ a2 = [1.0,2.0]' 12 Array{Float64,2}: 1.0 2.0

julia¿ a1p, a2p = paddedviews(0, a1, a2);

julia¿ a1p 22 PaddedViews.PaddedView{Int64,2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}},Arra
1 0 2 0

julia¿ a2p 22 PaddedViews.PaddedView{Float64,2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}},A
1.0 2.0 0.0 0.0

source

Chapter 85

SpecialFunctions

85.1 SpecialFunctions.cosint

`SpecialFunctions.cosint` — *Function.*

`cosint(x)`

Compute the cosine integral function of `x`, defined by $\mathrm{Ci}(x) := \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt$ for real `x` $>$ `0`, where γ is the Euler-Mascheroni constant.
source

85.2 SpecialFunctions.sinint

`SpecialFunctions.sinint` — *Function.*

`sinint(x)`

Compute the sine integral function of `x`, defined by $\mathrm{Si}(x) := \int_0^x \frac{\sin t}{t} dt$ for real `x`.
source

Chapter 86

NamedTuples

86.1 NamedTuples.delete

`NamedTuples.delete` — *Method*.

Create a new NamedTuple with the specified element removed.

source

86.2 NamedTuples.setindex

`NamedTuples.setindex` — *Method*.

Create a new NamedTuple with the new value set on it, either overwriting the old value or appending a new value. This copies the underlying data.

source

Chapter 87

Loess

87.1 Loess.loess

`Loess.loess` — *Method*.

```
loess(xs, ys, normalize=true, span=0.75, degreee=2)
```

Fit a loess model.

Args: `xs`: A `n` by `m` matrix with `n` observations from `m` independent predictors `ys`: A length `n` response vector. `normalize`: Normalize the scale of each predicitor. (default true when `m > 1`) `span`: The degree of smoothing, typically in [0,1]. Smaller values result in smaller local context in fitting. `degree`: Polynomial degree.

Returns: A fit `LoessModel`.

source

Chapter 88

Nulls

88.1 Nulls.levels

`Nulls.levels` — *Method.*

`levels(x)`

Return a vector of unique values which occur or could occur in collection `x`, omitting `null` even if present. Values are returned in the preferred order for the collection, with the result of `sort` as a default.

Contrary to `unique`, this function may return values which do not actually occur in the data, and does not preserve their order of appearance in `x`.

source

Chapter 89

WoodburyMatrices

89.1 WoodburyMatrices.liftFactor

`WoodburyMatrices.liftFactor` — *Method*.

```
liftFactor(A)
```

More stable version of inv(A). Returns a function which computs the inverse on evaluation, i.e. `liftFactor(A)(x)` is the same as `inv(A)*x`.

source

Chapter 90

Requests

90.1 Requests.save

`Requests.save` — *Function.*

```
save(r::Response, path=".")
```

Saves the data in the response in the directory `path`. If the path is a directory, then the filename is automatically chosen based on the response headers.

Returns the full pathname of the saved file.

source

Chapter 91

MemPool

91.1 MemPool.savetodisk

`MemPool.savetodisk` — *Method*.

Allow users to specifically save something to disk. This does not free the data from memory, nor does it affect size accounting.

source

Chapter 92

SimpleTraits

92.1 SimpleTraits.istrait

`SimpleTraits.istrait` — *Method*.

This function checks whether a trait is fulfilled by a specific set of types.

```
istrait(Tr1{Int,Float64}) => return true or false
```

source

Chapter 93

BinDeps

93.1 BinDeps.glibc_version

`BinDeps.glibc_version` — *Method*.

```
glibc_version()
```

For Linux-based systems, return the version of glibc in use. For non-glibc Linux and other platforms, returns `nothing`.

source