JuliaFinMetriX Documentation

Release 0.0.1

Christian Groll

January 09, 2017

Contents

This documentation lists the most important functions of all JuliaFinMetriX packages.

Packages:

CHAPTER 1

Copulas

1.1 ParamPC_MAT

getVineCPPId (*nam::Symbol*)
Get VineCPP copula ID for copula name.

getVineCPPId (*cop::ParamPC_MAT*)
Get VineCPP copula ID for `ParamPC_MAT` copula object.

getCopNam (*ii::Int*)
Get VineCPP copula name as `Symbol` for given VineCPP ID.

getCopType (*ii::Int*)
Get VineCPP copula type for given VineCPP ID.

1.2 Utilities

checkSameLength (*u1::FloatVec*, *u2::FloatVec*)
Check whether two `Array{Float, 1}` have the same length.

getFamAndParams (*cop::ParamPC_MAT*)
Get the VineCPP copula ID and the parameters corresponding to a `ParamPC_MAT` copula. Both parameters and ID are transformed to `Array{Float64, 1}` and `Float64` respectively.

params (*cop::ParamPC_MAT*)
Get the parameters of a `ParamPC_MAT` copula.

CHAPTER 2

EconDatasets

2.1 Load data

dataset (*dataset_name::String*)
Load data set from disk.

2.2 Download data

getDataset (*dataset_name::String*)
Download data set from the internet.

2.3 Read in data from web

readFamaFrenchRaw (*url::ASCIIString*)
Access data from Kenneth R. French's data library.

CHAPTER 3

Econometrics

3.1 Return calculation

disc2log (*tm::AbstractTimenum; percent = false*)
Convert discrete net returns to logarithmic returns.

log2disc (*tm::AbstractTimenum; percent = false*)
Convert logarithmic returns to discrete net returns.

price2ret (*tm::AbstractTimematr; log = true*)
Convert prices to returns.

price2ret (*tn::AbstractTimenum; log = true*)
Convert prices to returns with possibly occurring missing observations (NAs).

ret2price (*tm::AbstractTimematr; log = true*)
Convert returns to prices.

ret2price (*tn::AbstractTimenum; log = true*)
Convert returns to prices with possibly occurring missing observations (NAs).

CHAPTER 4

TimeData

4.1 Access metadata

idx (*td::AbstractTimedata*)
Get time index as array.

names (*td::AbstractTimedata*)
Get column names.

idxtype (*td::AbstractTimedata*)
Get type of `idx`.

4.2 Access data as Array

get (*td::AbstractTimedata*, *idx1::Int*, *idx2::Int*)
Get single entry [idx1, idx2] of TimeData object.

get (*td::AbstractTimedata*)
Get all entries of TimeData object as `Array` through comprehension.

core (*td::AbstractTimedata*)
Equal to `get` except for `Timematr`.

core (*tm::Timematr*)
Return data as `Array{Float64}`.

getAs (*tn::AbstractTimedata*, *typ::Type=Any*, *replaceNA=NA*)
Get all entries as `Array`. In order to avoid `Array{Any, 2}` as outcome, the entries can be promoted to a given `Type` through argument `typ`. Missing values can be dealt with through a third argument `replaceNA`, which replaces occurring `NAs`. For example, `Timenum tm` can be transformed to `Array{Float64, 2}` through: `getAs(tm, Float64, NaN)`.

4.3 Access data as DataFrame

convert (*::DataFrame*, *td::AbstractTimedata*)
Convert `Timedata` object to `DataFrame` by prepending index column as column *:idx*.

4.4 Dealing with NAs

complete_cases (*td::AbstractTimedata*)
Return `Array{Bool}` with `true` for rows without `NA` values.

narm (*td::AbstractTimedata*)
Get complete cases: return copy of td with all rows removed that were containing `NA`.

rmDatesOnlyNAs (*tn::AbstractTimedata*)
Remove all dates that contain strictly missing values `NA`. Required format, for example, for plotting with Gadfly.

4.5 Show entries

Accessing entries through `getindex` methods will always preserve a rectangular table data structure: the output is an intersection of a subset of indices with a subset of columns. In contrast, `showEntries` methods allow to access data without rectangular structure as stacked data. This way, for example, entry (1,2) and entry (2,1) could be jointly accessed, without simultaneously returning entries (1,1) and (2,2). The output of `showEntries` always is of type `Timedata`, with columns `variable` and `value`.

showEntries (*td::AbstractTimedata*, *f::Function; sort="dates"*)
Show all entries where function f returns `true`. By default, return values in row major order: for each date try all variables. Column major order can be achieved through `sort="variables"`.

showEntries (*td::AbstractTimedata*, *singleInd::Array{Int}*)
Show entries given by linear indexing.

showEntries (*td::AbstractTimedata*, *rowInds::Array{Int}*, *colInds::Array{Int}*)
Show entries given by subscript indexing.

showEntries (*td::AbstractTimedata*, *td2::AbstractTimedata*)
Show entries by element-wise logical indexing.

4.6 Editing entries

setNA!(td::AbstractTimedata, rowIdx::Int, colIdx::Int)
Set a given entry to `NA`. Could require change of column type to `DataArray`. Throws error for `Timematr`.

setindex!(td::Timedata, value::Any, rowIdx::Int, colIdx::Int)
Set entry given by subscript indexing to a given value.

setindex!(td::AbstractTimenum, value::Any, rowIdx::Int, colIdx::Int)
Set entry given by subscript indexing to a given value.

impute!(td::AbstractTimedata, with="last")
Replace `NA` with some value. Implemented options are `last` to use the last available observation, `next` to use the next available option, `zero` to insert a value of 0 for each `NA`. `single last` only uses the last observation if there is a single `NA` in succession. For two or more successive values of `NA` no imputation occurs. A single `NA` in the last row will be treated as if observations would follow, so that it gets replaced. Consecutive occurrences of `NA` at the beginning of the sample will be left untouched with options `last` and `single last`. Same holds for consecutive occurrences of `NA` at the end of the sample for option `next`.

4.7 Basic functions

size (*tn::AbstractTimedata*)

size (*tn::AbstractTimedata*, *ind::Int*)

ndims (*tn::AbstractTimedata*)

4.8 Testing object properties

isequal (*tn::AbstractTimedata*, *tn2::AbstractTimedata*)
Test for equal indices, names, types and values. `NA` is equal to `NA`. Output is a single value of type `Bool`.

isequalElw (*tn::AbstractTimedata*, *tn2::AbstractTimedata*)
Element-wise comparison with `isequal`. Returns `Timedata` with boolean values.

==(tn::AbstractTimedata, tn2::AbstractTimedata)
Test for equal indices, names, types and values. `NA` is not counted as equal to `NA`. Output is a single value of type `Bool`.

.==(tn::AbstractTimedata, tn2::AbstractTimedata)
Element-wise test for equal values: `NA` is not counted as equal to `NA`. Returns `Timedata` with boolean values, or error if meta-data is not matching.

isapprox (*tn::AbstractTimedata*, *tn2::AbstractTimedata*)
Test for equal indices, names, types and approximately equal values. Alleviates unit tests for values of type `Float`. Output is a single value of type `Bool`.

equMeta (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Test for equal meta-data: type, column names and indices. Output is a single value of type `Bool`.

equColMeta (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Test for equal meta-data of columns (dates are left unconsidered): type and column names. Output is a single value of type `Bool`.

isnaElw (*td::AbstractTimedata*)
Element-wise testing for `NA`. Returns boolean values as Timedata object.

4.9 Date formatting functions

datesAsStrings (*dats::Array{Date, 1}*)
Convert vector of dates into `Array{ASCIIString, n}`.

datesAsStrings (*tm::AbstractTimedata*)
Take `TimeData` object and convert its vector of dates into `Array(ASCIIString, n)`.

datesAsNumbers (*dats::Array{Date, 1}*)
Convert vector of dates into `Array{Float64, n}`.

datesAsNumbers (*tm::AbstractTimedata*)
Take `TimeData` object and convert its vector of dates into numbers: `Array{Float64, n}` for `Date` and `DateTime` entries.

4.10 Type preserving functions

hcat (*inst::AbstractTimedata*, *inst2::AbstractTimedata*)
Horizontal concatenation of TimeData objects. Requires objects to be of equal type with completely equal time indices. Result will be of same type as input arguments.

hcat (*inst::AbstractTimedata...*)
Variable argument extension of `hcat`.

vcat (*inst::AbstractTimedata*, *inst2::AbstractTimedata*)
Vertical concatenation of TimeData objects. Requires objects to be of equal type with equal column names and equal time index types. Result will be of same type as input arguments.

vcat (*inst::AbstractTimedata...*)
Variable argument extension of `vcat`.

flipud (*inst::AbstractTimedata*)
Flip `TimeData` object upside down.

4.11 Conversion functions

asArrayOfEqualDimensions (*arr::Array*, *td::AbstractTimedata*)
Extend row or column vector to two-dimensional array through copying values.

asTd (*arr::Array*, *td::Timedata*)
Extend row or column vector to size of `Timedata` object similar to `repmat` and return it as `Timedata` object with equal index and names.

asTn (*arr::Array*, *td::Timenum*)
Extend row or column vector to size of `Timenum` object similar to `repmat` and return it as `Timenum` object with equal index and names.

asTm (*arr::Array*, *td::Timematr*)
Extend row or column vector to size of `Timematr` object similar to `repmat` and return it as `Timematr` object with equal index and names.

convert (*::Type{AbstractTimedata}*, *ta::TimeArray*)
Convert `TimeArray` to `TimeData` object.

4.12 DataFrame extensions

composeDataFrame (*vals*, *nams*)
Compose DataFrame from Array and column names.

round (*df::DataFrame*, *nDgts::Int*)
Return DataFrame with rounded values. DataFrame entries must be numeric.

round (*df::DataFrame*)
Return DataFrame with values rounded to two significant digits. DataFrame entries must be numeric.

@roundDf(expr::Expr)
Display rounded DataFrame. Works with non numeric values also.

4.13 Display functions

display(*tn::AbstractTimedata*)
Timedata display function in standard REPL.

writemime(*io::IO*, *::MIME"text/html"*, *td::AbstractTimedata*)
Timedata display function in ijulia.

writemime(*io::IO*, *::MIME"text/html"*, *tm::AbstractTimematr*)
Timematr display function in ijulia. Values are rounded due to parsimony.

@table(title::String, expr::Union(Expr, Symbol))
Display expression or symbol in HTML with blue title header.

str(*tn::AbstractTimedata*)
More detailled display function similar to R syntax.

4.14 Statistics functions

mean(*tm::AbstractTimematr*, *dim::Int = 1*)
Return mean column values as DataFrame.

rowmeans(*tm::AbstractTimematr*)
Return mean row values as Timematr.

prod(*tm::AbstractTimematr*, *dim::Int = 1*)
Return product of column values as DataFrame.

rowprods(*tm::AbstractTimematr*)
Return product of row values as Timematr.

sum(*tm::AbstractTimematr*, *dim::Int = 1*)
Return sum of columns as DataFrame.

rowsums(*tm::AbstractTimematr*)
Return sum of rows as Timematr.

cov(*tm::AbstractTimematr*)
Return covariance matrix as DataFrame.

cor(*tm::AbstractTimematr*)
Return correlation matrix as DataFrame.

std(*tm::AbstractTimematr*)
Return empirical standard deviation for each column as DataFrame.

std(*tm::AbstractTimematr*, *dim::Integer*)
Return empirical standard deviation for each column as DataFrame.

minimum(*tm::AbstractTimematr*)
Return minimum value as single value.

minimum(*tm::AbstractTimematr*, *dim::Integer*)
Return minimum values of each column as DataFrame.

cumsum(*tm::AbstractTimematr*, *dim::Integer*)
Calculate cumulative sums column-wise and return result as Timematr.

cumprod(*tm::AbstractTimematr*, *dim::Integer*)
Calculate cumulative products column-wise and return result as Timematr.

rowstds (*tm::AbstractTimematr*)
Return empirical standard deviation for each row as Timematr.

geomMean (*x::AbstractTimematr; percent = true*)
Calculate geometric mean for AbstractTimedata.

geomMean (*x; percent = true*)
Calculate geometric mean for Array.

movAvg (*tm::AbstractTimematr*, *nPeriods::Integer*)
Calculate moving average.

4.15 I/O

readTimedata (*filename::String*)
Load csv and parse date column as `idx`.

writeTimedata (*filename::String*, *td::AbstractTimedata*)
Write TimeData object to csv file.

4.16 Join functions

For the case of monotonically increasing index values, join operations can be speeded up. The following join implementations exist. All return a Timedata object.

joinSortedIdx_inner (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Inner join of object indices.

joinSortedIdx_left (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Left join of object indices.

joinSortedIdx_right (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Right join of object indices.

joinSortedIdx_outer (*td1::AbstractTimedata*, *td2::AbstractTimedata*)
Outer join of object indices.

CHAPTER 5

Vines

`Vines` are basically an array of `Trees`, stored in a more concise notation as one single matrix (each tree is given as column vector in parent notation). `Trees` also do not need to be fully specified: they do not necessarily need to comprise all variables. For example, when successively building `Vines`, `Trees` are built up only gradually. If unfinished `Trees` get converted to `Vines`, missing variables will have undefined values (this might be unstable, however, in its current version).

Basically, `Vines` are a memory efficient way of storing multiple trees, while any analysis needs to be built on actual `Trees` anyways.

5.1 Tree lengths

`length` (*tr::Tree*)
Number of paths of the tree.

`maxLen` (*tr::Tree*)
Number of nodes of longest path.

`getLengths` (*tr::Tree*)
Length of each path.

5.2 Tree entries

`getindex` (*tP::Tree*, *pathInd::Int*, *pathNode::Int*)
Get single node.

`getindex` (*tP::Tree*, *pathInd::Int*)
Get single path.

`getindex` (*tP::Tree*, *pathInds::Array{Int, 1}*, *pathNodes::Array{Int, 1}*)
Get unique values of multiple entries. Values are sorted.

`allVals` (*tP::Tree*)
Get all values occuring in tree together with root node.

`allPathVals` (*tP::Tree*)
Get all values occuring in tree paths without root node.

5.3 Tree / vine interfaces

par2tree (*parNot::Array{Int, 1}*)
Transform single parent notation column to tree.

par2tree (*parNot::Array{Int, 2}*)
Transform parent notation matrix to array of trees.

tree2par (*tP::Tree*, *nVars::Int*)
Transform single tree to parent notation vector.

tree2par (*tPs::Array{Tree, 1}*, *nVars::Int*)
Transform array of trees to parent notation matrix.

vine2trees (*vn::Vine*)
Transform `Vine` to array of `Trees`.

trees2vine (*trs::Array{Tree, 1}*, *nVars::Int*)
Transform array of `Trees` to `Vine`.

CHAPTER 6

Conditioning trees

There are a series of different representations of .. _conditioning trees: http://cgroll.github.io/copula_theory/rvines/condTrees.html.

6.1 CTree types

Representation as array of individual paths to leaf nodes.

CTreePaths:

root *Int*

paths *Array{Array{Int, 1}, 1}*

Parent reference notation, where entry *ii* denotes the single parent of node *ii*.

CTreeParRef:

tree *Array{Int, 1}*

CTreeAdja

6.2 Tree dimensions

`width` (*tr::CTree*)
Number of paths / leaf nodes of tree.

`maxDepth` (*tr::CTree*)
Number of nodes of longest path.

`getDepths` (*tr::CTree*)
Length of each path to leaf nodes.

6.3 Tree entries

`getindex` ()
Yet to be defined correctly! Differs for different implementations.

`allNodes` (*tr::CTree*)
Array{Int, 1} of all occurring nodes in sorted order.

allPathNodes(*tr::CTree*)
Array{Int, 1} of all occurring nodes except root in sorted order.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

P

R

S

T

V

W