

# Package: kazaam (via r-universe)

November 1, 2024

**Type** Package

**Title** Tools for Tall Distributed Matrices

**Version** 0.1-0

**Description** Many data science problems reduce to operations on very tall, skinny matrices. However, sometimes these matrices can be so tall that they are difficult to work with, or do not even fit into main memory. One strategy to deal with such objects is to distribute their rows across several processors. To this end, we offer an 'S4' class for tall, skinny, distributed matrices, called the 'shaq'. We also provide many useful numerical methods and statistics operations for operating on these distributed objects. The naming is a bit ``tongue-in-cheek'', with the class a play on the fact that 'Shaquille' 'ONeal' ('Shaq') is very tall, and he starred in the film 'Kazaam'.

**License** BSD 2-clause License + file LICENSE

**Depends** R (>= 3.0.0), pbdMPI (>= 0.3-0)

**Imports** methods, stats

**ByteCompile** yes

**URL** <http://r-pbd.org/>

**BugReports** <http://group.r-pbd.org/>

**MailingList** Please send questions and comments regarding pbdR to RBigData@gmail.com

**Maintainer** Drew Schmidt <wrathematics@gmail.com>

**RoxygenNote** 6.0.1

**NeedsCompilation** yes

**Author** Drew Schmidt [aut, cre], Wei-Chen Chen [aut], Mike Matheson [aut], George Ostrouchov [aut], ORNL [cph]

**Date/Publication** 2017-06-29 13:19:58 UTC

**Repository** <https://wrathematics.r-universe.dev>

**RemoteUrl** <https://github.com/cran/kazaam>

**RemoteRef** HEAD

**RemoteSha** 4371c4c509f984d5cb97180ca9b93d37a4475901

## Contents

kazaam-package . . . . .	2
arithmetic . . . . .	3
bracket . . . . .	4
cbind.shaq . . . . .	5
collapse . . . . .	6
col_ops . . . . .	7
cov . . . . .	8
crossprod . . . . .	9
expand . . . . .	10
getters . . . . .	11
glms . . . . .	12
is.shaq . . . . .	13
lm_coefs . . . . .	14
matmult . . . . .	15
norm . . . . .	16
prcomp . . . . .	17
print . . . . .	18
qr . . . . .	19
ranshaq . . . . .	20
scale . . . . .	21
setters . . . . .	22
shaq . . . . .	23
shaq-class . . . . .	24
svd . . . . .	24
<b>Index</b>	<b>26</b>

---

kazaam-package	<i>Tall Matrices</i>
----------------	----------------------

---

## Description

Many data science problems reduce to operations on very tall, skinny matrices. However, sometimes these matrices can be so tall that they are difficult to work with, or do not even fit into main memory. One strategy to deal with such objects is to distribute their rows across several processors. To this end, we offer an 'S4' class for tall, skinny, distributed matrices, called the 'shaq'. We also provide many useful numerical methods and statistics operations for operating on these distributed objects. The naming is a bit "tongue-in-cheek", with the class a play on the fact that 'Shaquille' 'O'Neal ('Shaq') is very tall, and he starred in the film 'Kazaam'.

**Author(s)**

Drew Schmidt <wrathematics@gmail.com>, Wei-Chen Chen, Mike Matheson, and George Ostrouchov.

**References**

Programming with Big Data in R Website: <http://r-pbd.org/>

---

arithmetic	<i>Arithmetic Operators</i>
------------	-----------------------------

---

**Description**

Some binary arithmetic operations for shaqs. All operations are vector-shaq or shaq-vector, but not shaq-shaq. See details section for more information.

**Usage**

```
## S4 method for signature 'shaq,shaq'  
e1 + e2  
  
## S4 method for signature 'shaq,numeric'  
e1 + e2  
  
## S4 method for signature 'numeric,shaq'  
e1 + e2  
  
## S4 method for signature 'shaq,shaq'  
e1 - e2  
  
## S4 method for signature 'shaq,numeric'  
e1 - e2  
  
## S4 method for signature 'numeric,shaq'  
e1 - e2  
  
## S4 method for signature 'shaq,shaq'  
e1 * e2  
  
## S4 method for signature 'shaq,numeric'  
e1 * e2  
  
## S4 method for signature 'numeric,shaq'  
e1 * e2  
  
## S4 method for signature 'shaq,shaq'  
e1 / e2
```

```
## S4 method for signature 'shaq,numeric'  
e1 / e2  
  
## S4 method for signature 'numeric,shaq'  
e1 / e2
```

### Arguments

e1, e2            A shaq or a numeric vector.

### Details

For binary operations involving two shaqs, they must be distributed *identically*.

### Value

A shaq.

### Communication

Each operation is completely local.

### Examples

```
## Not run:  
library(kazaam)  
x = ranshaq(runif, 10, 3)  
y = ranshaq(runif, 10, 3)  
  
x + y  
x / 2  
y + 1  
  
finalize()  
  
## End(Not run)
```

---

bracket

*subsetting*

---

### Description

Subsetting via `[` for shaq objects.

**Usage**

```
## S4 method for signature 'shaq'
x[i, j]

## S4 replacement method for signature 'shaq'
x[i, j, ...] <- value
```

**Arguments**

x	A shaq.
i, j	Indices. NOTE currently only implemented for j values.
...	Ignored.
value	Replacement value(s) for the [ <code>&lt;-</code> ] method. This can either be an appropriately sized numeric value or a shaq. See the details section for more information.

**Value**

A shaq.

**Communication**

Each operation is completely local.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)
y = x[, -1]
y

finalize()

## End(Not run)
```

---

cbind.shaq

*cbind*


---

**Description**

Column binding for shaqs.

**Usage**

```
## S3 method for class 'shaq'
cbind(..., deparse.level = 1)
```

**Arguments**

... A collection of shaqs.  
deparse.level Ignored.

**Details**

All shaqs should have the same number of rows. Additionally, all shaqs should be distributed in identical fashion.

**Value**

A shaq.

**Communication**

The operation is completely local.

**Examples**

```
## Not run:  
library(kazaam)  
x = ranshaq(runif, 10, 3)  
y = ranshaq(runif, 10, 1)  
  
cbind(x, y)  
  
finalize()  
  
## End(Not run)
```

---

collapse

*collapse*

---

**Description**

Collapse a shaq into a regular matrix.

**Usage**

```
collapse(x)
```

**Arguments**

x A shaq.

**Details**

Only rank 0 will own the matrix on return.

**Value**

A regular matrix (rank 0) or NULL (everyone else).

**Communication**

Short answer: quite a bit. Each local submatrix has to be sent to rank 0.

**Examples**

```
## Not run:  
library(kazaam)  
dx = ranshaq(runif, 10, 3)  
  
x = collapse(dx)  
comm.print(x)  
  
finalize()  
  
## End(Not run)
```

---

col\_ops

*Column Operations*

---

**Description**

Column operations (currently sums/means) for shaq objects.

**Usage**

```
## S4 method for signature 'shaq'  
colSums(x, na.rm = FALSE, dims = 1L)  
  
## S4 method for signature 'shaq'  
colMeans(x, na.rm = FALSE, dims = 1L)
```

**Arguments**

x	A shaq.
na.rm	Should NA's be removed?
dims	Ignored.

**Value**

A regular vector.

**Communication**

The operation consists of a local column sum operation, followed by an `allreduce()` call, quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)
cs = colSums(x)
comm.print(cs)

finalize()

## End(Not run)
```

---

 cov

*Covariance and Correlation*


---

**Description**

Covariance and (pearson) correlation.

**Usage**

```
## S4 method for signature 'shaq'
cov(x, y = NULL, use = "everything", method = "pearson")

## S4 method for signature 'shaq'
cor(x, y = NULL, use = "everything", method = "pearson")
```

**Arguments**

<code>x</code>	A shaq.
<code>y</code>	At this time, this must be NULL.
<code>use</code>	NA handling rules, as with R's <code>cov/cor</code> functions. At this time, only "everything" is supported.
<code>method</code>	The <code>cov/cor</code> method. Currently only "pearson" is available.

**Value**

A regular matrix.

**Communication**

The operation is completely local except for forming the crossproduct, which is an `allreduce()` call, quadratic on the number of columns.



**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

cov(x)
cor(x)

finalize()

## End(Not run)
```

---

crossprod	<i>Matrix Multiplication</i>
-----------	------------------------------

---

**Description**

Conceptually, this computes  $t(x) \%*\% x$  for a shaq  $x$ .

**Usage**

```
## S4 method for signature 'shaq'
crossprod(x, y = NULL)
```

**Arguments**

x	A shaq.
y	Must be NULL.

**Value**

A regular matrix.

**Communication**

The operation consists of a local crossproduct, followed by an `allreduce()` call, quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

cp = crossprod(x)
comm.print(cp)
```

```
finalize()
## End(Not run)
```

---

expand

*expand*

---

### Description

Expand a regular matrix owned on MPI rank 0 into a shaq.

### Usage

```
expand(x)
```

### Arguments

x                    A regular matrix.

### Value

A shaq.

### Communication

Short answer: quite a bit. Each local submatrix has to be received from rank 0.

### Examples

```
## Not run:
library(kazaam)
if (comm.rank() == 0){
  x = matrix(runif(30), 10, 3)
} else {
  x = NULL
}

dx = expand(x)
dx

finalize()

## End(Not run)
```

---

getters	<i>getters</i>
---------	----------------

---

**Description**

Getters for shaq objects.

**Usage**

```
## S4 method for signature 'shaq'  
nrow(x)
```

```
## S4 method for signature 'shaq'  
NROW(x)
```

```
nrow.local(x)
```

```
## S4 method for signature 'shaq'  
nrow.local(x)
```

```
## S4 method for signature 'shaq'  
ncol(x)
```

```
## S4 method for signature 'shaq'  
NCOL(x)
```

```
ncol.local(x)
```

```
## S4 method for signature 'shaq'  
ncol.local(x)
```

```
## S4 method for signature 'shaq'  
length(x)
```

```
Data(x)
```

```
## S4 method for signature 'shaq'  
Data(x)
```

**Arguments**

x                    A shaq.

**Details**

Functions to return the number of rows (`nrow()` and `NROW()`), the number of columns (`ncol()` and `NCOL()`), the length - or product of the number of rows and cols - (`length()`), and the local submatrix (`Data()`).

**Communication**

Each operation is completely local.

**See Also**

[setters](#)

---

glms

*Generalized Linear Model Fitters*

---

**Description**

Linear regression (Gaussian GLM), logistic regression, and poisson regression model fitters.

**Usage**

```
reg.fit(x, y, maxiter = 100)
```

```
logistic.fit(x, y, maxiter = 100)
```

```
poisson.fit(x, y, maxiter = 100)
```

**Arguments**

<code>x, y</code>	The input data <code>x</code> and response <code>y</code> . Each must be a <code>shaq</code> , and each must be distributed in an identical fashion. See the details section for more information.
<code>maxiter</code>	The maximum number of iterations.

**Details**

Each function is implemented with gradient descent using the conjugate gradients method ("CG") of the `optim()` function.

Both of `x` and `y` must be distributed in an identical fashion. This means that the number of rows owned by each MPI rank should match, and the data rows `x` and response rows `y` should be aligned. Additionally, each MPI rank should own at least one row. Ideally they should be load balanced, so that each MPI rank owns roughly the same amount of data.

**Value**

The return is the output of an `optim()` call.

**Communication**

The communication consists of an allreduce of 1 double (the local cost/objective function value) at each iteration of the optimization.

**References**

McCullagh, P. and Nelder, J.A., 1989. Generalized Linear Models, no. 37 in Monograph on Statistics and Applied Probability.

Duda, R.O., Hart, P.E. and Stork, D.G., 1973. Pattern classification (pp. 526-528). Wiley, New York.

**See Also**

[lm\\_coefs](#)

**Examples**

```
## Not run:
library(kazaam)
comm.set.seed(1234, diff=TRUE)

x = ranshaq(rnorm, 10, 3)
y = ranshaq(function(i) sample(0:1, size=i, replace=TRUE), 10)

fit = logistic.fit(x, y)
comm.print(fit)

finalize()

## End(Not run)
```

---

is.shaq

*is.shaq*

---

**Description**

Test if an object is a shaq.

**Usage**

```
is.shaq(x)
```

**Arguments**

x                    An R object.

**Value**

A logical value, indicating whether or not the input is a shaq.

**Communication**

The operation is completely local.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

comm.print(is.shaq(x))

finalize()

## End(Not run)
```

---

`lm_coefs`*Linear Model Coefficients*

---

**Description**

Coefficients of the linear model.

**Usage**

```
lm_coefs(x, y, tol = 1e-07)
```

**Arguments**

<code>x, y</code>	The input data <code>x</code> and response <code>y</code> . Each must be a shaq, and each must be distributed in an identical fashion. See the details section for more information.
<code>tol</code>	Numerical tolerance for deciding rank.

**Details**

The model is fit using a QR factorization of the input `x`. At this time, that means

Both of `x` and `y` must be distributed in an identical fashion. This means that the number of rows owned by each MPI rank should match, and the data rows `x` and labels `y` should be aligned. Additionally, each MPI rank should own at least one row. Ideally they should be load balanced, so that each MPI rank owns roughly the same amount of data.

**Value**

A regular vector.

**Communication**

The operation has the same communication as

**See Also**

[glm](#)

**Examples**

```
## Not run:
library(kazaam)
comm.set.seed(1234, diff=TRUE)

x = ranshaq(rnorm, 10, 3)
y = ranshaq(runif, 10)

fit = lm_coefs(x, y)
comm.print(fit)

finalize()

## End(Not run)
```

---

matmult

*Matrix Multiplication*

---

**Description**

Multiplies two distributed matrices, if they are conformable.

**Usage**

```
## S4 method for signature 'shaq,matrix'
x %**% y
```

**Arguments**

x	A shaq.
y	A regular matrix, globally owned on all ranks. Since the number of columns of a shaq should be small, this matrix should be small as well.

**Details**

The two shaqs must be distributed *identically*.

**Value**

A shaq.

**Communication**

The operation is completely local.

## Examples

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)
y = matrix(1:9, 3, 3)

x %*% y

finalize()

## End(Not run)
```

---

norm

*norm*

---

## Description

Implementation of R's `norm()` function for shaq objects.

## Usage

```
## S4 method for signature 'shaq,ANY'
norm(x, type = c("O", "I", "F", "M", "2"))
```

## Arguments

x	A shaq
type	The type of norm: one, infinity, frobenius, max-modulus, and spectral.

## Details

If `type == "O"` then the norm is calculated as the maximum of the column sums.

If `type == "I"` then the norm is calculated as the maximum absolute value of the row sums.

If `type == "F"` then the norm is calculated as the square root of the sum of the square of the values of the matrix.

If `type == "M"` then the norm is calculated as the max of the absolute value of the values of the matrix.

If `type == "2"` then the norm is calculated as the largest singular value.

## Value

A number (length 1 regular vector).



**Communication**

If type == "O" then the communication consists of an allreduce, quadratic on the number of columns.

If type == "I" then the communication consists of an allgather.

If type == "F" then the communication is an allreduce, quadratic on the number of columns.

If type == "M" then the communication consists of an allgather.

If type == "2" then the communication consists of the same as that of an svd() call: an allreduce, quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

nm = norm(x)
comm.print(nm)

finalize()

## End(Not run)
```

---

prcomp

*Principal Components Analysis*


---

**Description**

Performs the principal components analysis.

**Usage**

```
## S3 method for class 'shaq'
prcomp(x, retx = TRUE, center = TRUE, scale. = FALSE,
       tol = NULL, ...)
```

**Arguments**

x	A shaq.
retx	Should the rotated variables be returned?
center	Should columns are zero centered?
scale.	Should columns are rescaled to unit variance?
tol	Ignored.
...	Ignored.

**Details**

`prcomp()` performs the principal components analysis on the data matrix by taking the SVD. Sometimes core R and kazaam will disagree slightly in what the rotated variables are because of how the SVD is calculated.

**Value**

A list of elements `sdev`, `rotation`, `center`, `scale`, and `x`, as with R's own `prcomp()`. The elements are, respectively, a regular vector, a regular matrix, a regular vector, a regular vector, and a shaq.

**Communication**

The communication is an `allreduce()` call, quadratic on the number of columns. Most of the run time should be dominated by relatively expensive local operations.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)
pca = prcomp(x)

comm.print(pca)

finalize()

## End(Not run)
```

---

`print`

*print*

---

**Description**

Print method for a shaq.

**Usage**

```
## S4 method for signature 'shaq'
print(x, ...)

## S4 method for signature 'shaq'
show(object)
```

**Arguments**

<code>x, object</code>	A shaq.
<code>...</code>	Ignored

**Communication**

The operation is completely local.

**Examples**

```
## Not run:  
library(kazaam)  
x = shaq(1, 10, 3)  
  
x # same as print(x) or comm.print(x)  
  
finalize()  
  
## End(Not run)
```

---

qr

*QR Decomposition Methods*

---

**Description**

QR factorization.

**Usage**

```
qr_R(x)  
  
qr_Q(x, R)
```

**Arguments**

x	A shaq.
R	A regular matrix. This argument is optional, in that if it is not supplied explicitly, then it will be computed in the background. But if have already computed R, supplying it here will improve performance (by avoiding needlessly recomputing it).

**Details**

$R$  is formed by first forming the crossproduct  $X^T X$  and taking its Cholesky factorization. But then  $Q = X R^{-1}$ . Inverting  $R$  is handled by an efficient triangular inverse routine.

**Value**

Q (a shaq) or R (a regular matrix).

**Communication**

The operation is completely local except for forming the crossproduct, which is an `allreduce()` call, quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

R = qr_R(x)
comm.print(R)

Q = qr_Q(x, R)
Q

finalize()

## End(Not run)
```

---

ranshaq

*ranshaq*


---

**Description**

Generate a random shaq object.

**Usage**

```
ranshaq(generator, nrows, ncols, local = FALSE, ...)
```

**Arguments**

generator	A function, such as <code>runif()</code> or <code>rnorm()</code> (passed without parens). See examples for a demonstration of usage.
nrows, ncols	The number of rows
local	Is the problem size <code>nrows*ncols</code> specifying the local or global problem size?
...	Additional arguments passed to the generator.

**Value**

A shaq.

**Communication**

The operation is entirely local.

**Examples**

```
## Not run:
library(kazaam)

# a 10x3 shaq with random uniform data
x = ranshaq(runif, 10, 3)
x

# a (comm.size() * 10)x3 shaq with random normal data
y = ranshaq(rnorm, 10, 3, local=TRUE)
y

finalize()

## End(Not run)
```

---

scale

*Scale*


---

**Description**

Centers and/or scales the columns of a distributed matrix.

**Usage**

```
scale(x, center = TRUE, scale = TRUE)

## S4 method for signature 'shaq,logical,logical'
scale(x, center = TRUE, scale = TRUE)
```

**Arguments**

x	A shaq.
center	logical value, determines whether or not columns are zero centered
scale	logical value, determines whether or not columns are rescaled to unit variance

**Value**

A shaq.

**Communication**

The communication consists of two allreduce calls, each quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(rnorm, 10, 3, mean=30, sd=10)

x
scale(x)

finalize()

## End(Not run)
```

---

 setters

 setters
 

---

**Description**

Setter functions for shaq objects. Generally not recommended unless you are sure you know what you're doing.

**Usage**

```
Data(x) <- value

## S4 replacement method for signature 'shaq'
Data(x) <- value

DATA(x) <- value

## S4 replacement method for signature 'shaq'
DATA(x) <- value
```

**Arguments**

x	A shaq.
value	The new data.

**Details**

Data<- will perform checks on the inserted data and ensure that the number of columns match across processors (requiring communication). It will also update the number of rows as necessary.

DATA<- will perform no checks, so use only if you're really sure that you know what you're doing.

**Communication**

With Data<-, a check on the global number of rows is performed. This amounts to an allgather operation on a logical value (the local dimension check).

**See Also**

[getters](#), [bracket](#)

---

shaq

*shaq*

---

**Description**

Constructor for shaq objects.

**Usage**

```
shaq(Data, nrows, ncols, checks = TRUE)

## S3 method for class 'matrix'
shaq(Data, nrows, ncols, checks = TRUE)

## S3 method for class 'numeric'
shaq(Data, nrows, ncols, checks = TRUE)
```

**Arguments**

Data	The local submatrix.
nrows, ncols	The GLOBAL number of rows and columns.
checks	Logical. Should some basic dimension checks be performed? Note that these require communication, and with many MPI ranks, could be expensive.

**Details**

If `nrows` and/or `ncols` is missing, then it will be imputed. This means one must be especially careful to manually provide `ncols` if some of ranks have "placeholder data" (a 0x0 matrix), which is typical when reading from a subset of processors and then broadcasting out to the remainder.

**Communication**

If `checks=TRUE`, a check on the global number of rows is performed. This amounts to an allgather operation on a logical value (the local dimension check).

**See Also**

[shaq-class](#)

---

 shaq-class

 Class *shaq*


---

### Description

An S4 container for a distributed tall/skinny matrix.

### Details

The (conceptual) global (non-distributed) matrix should be distributed by row, meaning that each submatrix should own all of the columns of the global matrix. Most methods assume no other real structure, however for best performance (and for the methods which require it), one should try to organize their distributed data in a particular way.

First, adjacent MPI ranks should hold adjacent rows. So if the last row that rank  $k$  owns is  $i$ , then the first row that rank  $k+1$  owns should be row  $i+1$ . Additionally, any method that operates on two (or more) shaq objects, the two shaqs should be distributed identically. By this we mean that if the number of rows shaq A owns on rank  $k$  is  $k_i$ , then the number of rows shaq B owns on rank  $k$  should also be  $k_i$ .

Finally, for best performance, one should generally try to keep the number of rows "balanced" (roughly equal) across processes, with perhaps the last "few" having one less row than the others.

### Slots

DATA The local submatrix.

nrows, ncols The global matrix dimension.

### See Also

[shaq](#)

---

 svd

*svd*


---

### Description

Singular value decomposition.

### Usage

```
## S4 method for signature 'shaq'
svd(x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)
```



**Arguments**

x	A shaq.
nu	number of left singular vectors to return.
nv	number of right singular vectors to return.
LINPACK	Ignored.

**Details**

The factorization works by first forming the crossproduct  $X^T X$  and then taking its eigenvalue decomposition. In this case, the square root of the eigenvalues are the singular values. If the left/right singular vectors  $U$  or  $V$  are desired, then in either case,  $V$  is computed (the eigenvectors). From these,  $U$  can be reconstructed, since if  $X = U\Sigma V^T$ , then  $U = XV\Sigma^{-1}$ .

**Value**

A list of elements d, u, and v, as with R's own `svd()`. The elements are, respectively, a regular vector, a shaq, and a regular matrix.

**Communication**

The operation is completely local except for forming the crossproduct, which is an `allreduce()` call, quadratic on the number of columns.

**Examples**

```
## Not run:
library(kazaam)
x = ranshaq(runif, 10, 3)

svd = svd(x)
comm.print(svd$d) # a globally owned vector
svd$u # a shaq
comm.print(svd$v) # a globally owned matrix

finalize()

## End(Not run)
```

# Index

## \* Package

- kazaam-package, 2
- \*, numeric, shaq-method (arithmetic), 3
- \*, shaq, numeric-method (arithmetic), 3
- \*, shaq, shaq-method (arithmetic), 3
- +, numeric, shaq-method (arithmetic), 3
- +, shaq, numeric-method (arithmetic), 3
- +, shaq, shaq-method (arithmetic), 3
- , numeric, shaq-method (arithmetic), 3
- , shaq, numeric-method (arithmetic), 3
- , shaq, shaq-method (arithmetic), 3
- /, numeric, shaq-method (arithmetic), 3
- /, shaq, numeric-method (arithmetic), 3
- /, shaq, shaq-method (arithmetic), 3
- [, shaq-method (bracket), 4
- [<-, shaq-method (bracket), 4
- %%, shaq, matrix-method (matmult), 15
  
- arithmetic, 3
  
- bracket, 4, 23
  
- cbind.shaq, 5
- col\_ops, 7
- collapse, 6
- colMeans, shaq-method (col\_ops), 7
- colSums, shaq-method (col\_ops), 7
- cor, shaq-method (cov), 8
- cov, 8
- cov, shaq-method (cov), 8
- crossprod, 9
- crossprod, shaq-method (crossprod), 9
  
- Data (getters), 11
- Data, shaq-method (getters), 11
- DATA<- (setters), 22
- Data<- (setters), 22
- DATA<-, shaq-method (setters), 22
- Data<-, shaq-method (setters), 22
  
- expand, 10
  
- getters, 11, 23
- glms, 12, 14
  
- is.shaq, 13
  
- kazaam-package, 2
  
- length, shaq-method (getters), 11
- lm\_coefs, 13, 14
- logistic.fit (glms), 12
  
- matmult, 15
  
- NCOL, shaq-method (getters), 11
- ncol, shaq-method (getters), 11
- ncol.local (getters), 11
- ncol.local, shaq-method (getters), 11
- norm, 16
- norm, shaq, ANY-method (norm), 16
- NROW, shaq-method (getters), 11
- nrow, shaq-method (getters), 11
- nrow.local (getters), 11
- nrow.local, shaq-method (getters), 11
  
- poisson.fit (glms), 12
- prcomp, 17
- print, 18
- print, shaq-method (print), 18
  
- qr, 19
- qr\_Q (qr), 19
- qr\_R (qr), 19
  
- ranshaq, 20
- reg.fit (glms), 12
  
- scale, 21
- scale, shaq, logical, logical-method (scale), 21
- setters, 12, 22
- shaq, 23, 24

shaq-class, [24](#)  
show, shaq-method (print), [18](#)  
svd, [24](#)  
svd, shaq-method (svd), [24](#)