Package: Rmlx (via r-universe)

October 12, 2026

Type Package

Title R Interface to MLX Arrays (GPU-Accelerated with Metal or CUDA)

Version 0.4.0.9000

Author David Hugh-Jones [aut, cre], Apple Inc. [cph] (MLX library downloaded at install time)

Maintainer David Hugh-Jones <david@hughjones.com>

Description S3 class 'mlx' backed by Apple's MLX library, allowing array operations on Apple Silicon GPUs/CPUs and CUDA-enabled Linux systems through lazy evaluation, shared memory between chips, and automatic differentiation.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1.0)

Imports Rcpp (>= 1.0.10), methods, stats

LinkingTo Rcpp

Suggests testthat (>= 3.0.0), knitr, rmarkdown, bench, dplyr, ggplot2, rprojroot

VignetteBuilder knitr

Config/testthat/edition 3

SystemRequirements MLX (>= 0.32.3); C++20; CMake; BLAS/LAPACK development headers (e.g., Debian/Ubuntu: liblapacke-dev or libopenblas-dev)

Roxygen list(markdown = TRUE)

URL https://hughjonesd.github.io/Rmlx/, https://github.com/hughjonesd/Rmlx

BugReports https://github.com/hughjonesd/Rmlx/issues

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake

Repository https://hughjonesd.r-universe.dev

Date/Publication 2026-10-12 17:35:18 UTC

RemoteUrl https://github.com/hughjonesd/Rmlx

RemoteRef HEAD

RemoteSha f9a8dc240031b07eebb61433143d28b227bf5296

Contents

Rmlx-package . . . 6
[<-.mlx . . . 7
%*%.mlx . . . 8
abind . . . 9
all.equal.mlx . . . 10
as.array.mlx . . . 11
as.matrix.mlx . . . 12
as.vector.mlx . . . 12
as_mlx . . . 14
as_r . . . 15
asplit . . . 16
cbind.mlx . . . 17
chol.mlx . . . 18
chol2inv . . . 19
colMeans . . . 20
colSums . . . 21
crossprod.mlx . . . 22
diag . . . 22
diag.mlx . . . 23
dim.mlx . . . 24
dim<-.mlx . . . 24
drop . . . 25
fft . . . 26
format.mlx_stream . . . 27
is_mlx . . . 27
kronecker . . . 28
length.mlx . . . 29
Math.mlx . . . 29
mean.mlx . . . 30
mlx-dimnames . . . 31
mlx-methods . . . 32
mlx_addmm . . . 33
mlx_allclose . . . 33
mlx_arange . . . 34
mlx_argmax . . . 36
mlx_array . . . 37
mlx_batch_norm . . . 38
mlx_best_device . . . 38
mlx_binary_cross_entropy . . . 39
mlx_broadcast_arrays . . . 40

mlx_broadcast_to . . . 40
mlx_cast . . . 41
mlx_cholesky_inv . . . 42
mlx_clip . . . 43
mlx_compile . . . 43
mlx_contiguous . . . 45
mlx_conv_transpose1d . . . 46
mlx_conv_transpose2d . . . 47
mlx_conv_transpose3d . . . 48
mlx_conv1d . . . 49
mlx_conv2d . . . 50
mlx_conv3d . . . 51
mlx_coordinate_descent . . . 52
mlx_cross . . . 54
mlx_cross_entropy . . . 55
mlx_cumsum . . . 56
mlx_degrees . . . 57
mlx_dequantize . . . 57
mlx_device . . . 59
mlx_dexp . . . 59
mlx_disable_compile . . . 60
mlx_dlnorm . . . 61
mlx_dlogis . . . 62
mlx_dnorm . . . 62
mlx_dropout . . . 63
mlx_dtype . . . 64
mlx_dunif . . . 65
mlx_eig . . . 65
mlx_eigh . . . 66
mlx_eigvals . . . 67
mlx_eigvalsh . . . 68
mlx_embedding . . . 69
mlx_erf . . . 69
mlx_eval . . . 70
mlx_expand_dims . . . 71
mlx_eye . . . 71
mlx_fft . . . 72
mlx_flatten . . . 73
mlx_forward . . . 74
mlx_full . . . 75
mlx_gather . . . 76
mlx_gather_qmm . . . 77
mlx_gelu . . . 78
mlx_grad . . . 79
mlx_hadamard_transform . . . 80
mlx_has_gpu . . . 81
mlx_identity . . . 81
mlx_import_function . . . 82

mlx_inv . . . 83
mlx_isclose . . . 84
mlx_isnan . . . 85
mlx_isposinf . . . 85
mlx_key . . . 86
mlx_key_bits . . . 87
mlx_kron . . . 87
mlx_l1_loss . . . 88
mlx_layer_norm . . . 89
mlx_leaky_relu . . . 89
mlx_linear . . . 90
mlx_linspace . . . 91
mlx_load . . . 92
mlx_load_gguf . . . 92
mlx_load_safetensors . . . 93
mlx_logcumsumexp . . . 93
mlx_logsumexp . . . 94
mlx_lu . . . 95
mlx_matrix . . . 96
mlx_maximum . . . 97
mlx_meshgrid . . . 97
mlx_metal_kernel . . . 98
mlx_minimum . . . 99
mlx_moveaxis . . . 100
mlx_mse_loss . . . 101
mlx_nan_to_num . . . 102
mlx_new_stream . . . 102
mlx_norm . . . 103
mlx_ones . . . 104
mlx_ones_like . . . 105
mlx_optimizer_sgd . . . 106
mlx_pad . . . 106
mlx_param_set_values . . . 107
mlx_param_values . . . 108
mlx_parameters . . . 109
mlx_put_along_axis . . . 109
mlx_qr_gpu . . . 110
mlx_quantile . . . 111
mlx_quantize . . . 113
mlx_quantized_matmul . . . 114
mlx_rand_bernoulli . . . 115
mlx_rand_categorical . . . 116
mlx_rand_gumbel . . . 117
mlx_rand_laplace . . . 117
mlx_rand_multivariate_normal . . . 118
mlx_rand_normal . . . 119
mlx_rand_permutation . . . 120
mlx_rand_randint . . . 121

mlx_rand_truncated_normal . . . 122
mlx_rand_uniform . . . 123
mlx_real . . . 124
mlx_relu . . . 124
mlx_repeat . . . 125
mlx_reshape . . . 126
mlx_roll . . . 126
mlx_save . . . 127
mlx_save_gguf . . . 128
mlx_save_safetensors . . . 128
mlx_scalar . . . 129
mlx_scatter_add_axis . . . 129
mlx_sequential . . . 130
mlx_set_default_stream . . . 131
mlx_set_training . . . 131
mlx_sigmoid . . . 132
mlx_silu . . . 132
mlx_slice_update . . . 133
mlx_softmax . . . 134
mlx_softmax_layer . . . 134
mlx_solve_triangular . . . 135
mlx_sort . . . 136
mlx_split . . . 138
mlx_squeeze . . . 138
mlx_stack . . . 139
mlx_stop_gradient . . . 140
mlx_sum . . . 140
mlx_swapaxes . . . 142
mlx_synchronize . . . 143
mlx_take_along_axis . . . 143
mlx_tanh . . . 145
mlx_tile . . . 145
mlx_topk . . . 146
mlx_trace . . . 147
mlx_train_step . . . 148
mlx_tri . . . 149
mlx_tri_inv . . . 150
mlx_unflatten . . . 151
mlx_vector . . . 151
mlx_where . . . 152
mlx_zeros . . . 153
mlx_zeros_like . . . 154
Ops.mlx . . . 155
outer . . . 155
pinv . . . 156
print.mlx . . . 157
print.mlx_stream . . . 157
qr.mlx . . . 158

rbind.mlx . . . 159
row . . . 160
rowMeans . . . 160
rowSums . . . 161
scale.mlx . . . 162
solve.mlx . . . 163
str.mlx . . . 164
Summary.mlx . . . 164
svd . . . 165
svd.mlx . . . 166
t.mlx . . . 167
tcrossprod.mlx . . . 167
with_device . . . 168

Index **169**

`Rmlx-package` *Rmlx: R Interface to Apple's MLX Arrays*

Description

This package provides an R interface to Apple's MLX (Machine Learning eXchange) library for GPU-accelerated array operations on Apple Silicon.

Key Features

- Lazy evaluation: Operations are not computed until explicitly evaluated
- GPU acceleration: Leverage Metal on Apple Silicon
- Familiar syntax: S3 methods for standard R operations
- Unified memory: Efficient data sharing between CPU and GPU

Lazy Evaluation

MLX arrays use lazy evaluation by default. Operations are recorded but not executed until:

- You call `mlx_eval()`
- You convert to R with e.g. `as.array()` or `as.vector()`
- The result is needed for another computation

The package implements most of the C++ API via calls with the `mlx_` prefix, but it also ships S3 methods for many base generics, so common R matrix operations continue to work on MLX arrays. R conventions are used throughout: for example, indexing is 1-based.

Author(s)

Maintainer: David Hugh-Jones <david@hughjones.com>

Authors:

- David Hugh-Jones <david@hughjones.com>

Other contributors:

- Apple Inc. (MLX library downloaded at install time) [copyright holder]

See Also

Useful links:

- https://hughjonesd.github.io/Rmlx/
- https://github.com/hughjonesd/Rmlx
- Report bugs at https://github.com/hughjonesd/Rmlx/issues

[<-.mlx *Subset MLX array*

Description

MLX subsetting is like base R with a few differences:

Usage

```
## S3 replacement method for class 'mlx'
x[...] <- value

## S3 method for class 'mlx'
x[..., drop = FALSE]
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
...	Indices for each dimension. Provide one per axis; omitted indices select the full extent. Logical indices recycle to the dimension length.
value	Value to assign, typically an mlx or R array. Values recycle in R's column-major order; their length must divide the selection size.
drop	Should dimensions be dropped? (default: FALSE)

Details

- drop = FALSE by default.
- Indices containing NA, zero, or out-of-bounds positions give an error.
- Numeric indices truncate toward zero, as in R. Per-axis numeric indices must be undimensioned vectors; numeric matrices are coordinate indices.
- Logical per-axis indices on matrices and arrays must not exceed the axis length.
- Explicit NULL indices select no elements; omitted indices select the full axis.
- Single indices on a 2D or higher array are only allowed for assignment. For example, if x is a matrix, x[x < 0] <- 0 is OK but subset <- x[x < 0] is not. Use mlx_flatten() explicitly for subsetting.
- There is one exception: as in R, a single numeric matrix index selects individual elements. The number of columns must match the rank of x; each row gives coordinates for one element. The return value from subsetting is a flat mlx vector.
- mlx vectors, logical masks, and matrices behave the same as their R equivalents.
- Duplicate vector assignments like x[c(1,1)] <- 2:3 are undefined behaviour. Coordinate matrix assignments reject duplicate rows.
- Character indices match against the relevant axis dimnames.

Value

The subsetted MLX object.

See Also

mlx.core.take

Examples

```
x <- mlx_matrix(1:9, 3, 3)
x[1, ]
```

%*%.mlx *Matrix multiplication for MLX arrays*

Description

Both operands must be 2D matrices; vectors are not auto-promoted (unlike base R).

Usage

```
## S3 method for class 'mlx'
x %*% y
```

Arguments

x, y	MLX matrices or R matrices coercible via as_mlx().

Value

An mlx object.

See Also

mlx.core.matmul

Examples

```
x <- mlx_matrix(1:6, 2, 3)
y <- mlx_matrix(1:6, 3, 2)
x %*% y
```

---

abind *Bind mlx arrays along an axis*

Description

Bind mlx arrays along an axis

Usage

```
abind(..., along = 1L)
```

Arguments

...	One or more arrays (or a single list of arrays) coercible via as_mlx().
along	Positive integer giving the existing axis (1-indexed) along which to bind the inputs.

Details

This is an MLX-backed alternative to abind::abind(). All inputs must share the same shape on non-bound axes. The along axis must already exist; to create a new axis use mlx_stack().

Value

An mlx array formed by concatenating the inputs along along.

See Also

mlx.core.concatenate

Examples

```
x <- mlx_array(1:12, c(2, 3, 2))
y <- mlx_array(13:24, c(2, 3, 2))
z <- abind(x, y, along = 3)
dim(z)
```

`all.equal.mlx` *Test if two MLX arrays are (nearly) equal*

Description

S3 method for `all.equal` following R semantics. Returns `TRUE` if arrays are close, or a character vector describing differences if they are not.

Usage

```
## S3 method for class 'mlx'
all.equal(target, current, tolerance = sqrt(.Machine$double.eps), ...)
```

Arguments

`target, current`	MLX arrays to compare
`tolerance`	Numeric tolerance for comparison (default: sqrt(.Machine$double.eps))
`...`	Additional arguments; ignored.

Details

This method follows R's `all.equal()` semantics:

- Returns `TRUE` if arrays are close within tolerance
- Returns a character vector describing differences otherwise
- Checks dimensions/shapes before comparing values

The tolerance is converted to MLX's rtol and atol parameters:

- rtol = tolerance
- atol = tolerance

Value

Either `TRUE` or a character vector describing differences.

See Also

`mlx_allclose()`, `mlx_isclose()`

Examples

```
a <- as_mlx(c(1.0, 2.0, 3.0))
b <- as_mlx(c(1.0 + 1e-6, 2.0 + 1e-6, 3.0 + 1e-6))
all.equal(a, b)  # TRUE

c <- as_mlx(c(1.0, 2.0, 10.0))
all.equal(a, c)  # Character vector describing difference
```

---

as.array.mlx *Convert MLX array to R array*

Description

Always returns an R array using the MLX shape. One-dimensional MLX inputs become 1-D arrays (with `dim` set to their length) instead of plain vectors.

Usage

```
## S3 method for class 'mlx'
as.array(x, ...)
```

Arguments

`x`	An mlx array.
`...`	Additional arguments; ignored.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An R array with the same shape as the MLX input.

See Also

`as_r()`, `as.vector.mlx()`, `as.matrix.mlx()`

Examples

```
x <- mlx_matrix(1:8, 2, 4)
as.array(x)

v <- as_mlx(1:3)
as.array(v)  # 1-D array with dim 3
```

as.matrix.mlx *Convert MLX array to R matrix*

Description

MLX arrays with other than 2 dimensions are converted to a 1 column matrix, with a warning.

Usage

```
## S3 method for class 'mlx'
as.matrix(x, ...)
```

Arguments

x	An mlx array.
...	Additional arguments; ignored.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

A vector, matrix or array (numeric or logical depending on dtype).

Examples

```
x <- mlx_matrix(1:4, 2, 2)
as.matrix(x)
```

as.vector.mlx *Convert MLX array to R vector*

Description

Converts an MLX array to an R vector. Multi-dimensional arrays are flattened in column-major order (R's default).

Usage

```
## S3 method for class 'mlx'
as.vector(x, mode = "any")

## S3 method for class 'mlx'
as.logical(x, ...)

## S3 method for class 'mlx'
as.double(x, ...)

## S3 method for class 'mlx'
as.numeric(x, ...)

## S3 method for class 'mlx'
as.integer(x, ...)
```

Arguments

x	An mlx array.
mode	Character string specifying the type of vector to return (passed to `base::as.vector()`)
...	Additional arguments; ignored.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

A vector of the specified mode.

Examples

```
x <- as_mlx(-1:1)
as.vector(x)
as.logical(x)
as.numeric(x)

# Multi-dimensional arrays are flattened
m <- mlx_matrix(1:6, 2, 3)
as.vector(m)  # Flattened in column-major order
```

as_mlx *Create MLX array from R object*

Description

If dtype is not specified, numeric inputs use "float32", logical inputs use "bool", and complex inputs use "complex64".

Usage

```
as_mlx(
  x,
 dtype = c("float32", "float64", "bool", "complex64", "int8", "int16", "int32", "int64",
    "uint8", "uint16", "uint32", "uint64")
)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Details

MLX does not support float64 operations on GPU. When this function creates a float64 array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current mlx_device().

Value

An object of class mlx

Integer types require explicit dtype:

R integer vectors (like 1:10) convert to float32 by default. To create integer MLX arrays, you must explicitly specify dtype:

```
x <- as_mlx(1:10, dtype = "int32")  # Creates int32 array
x <- as_mlx(1:10)                    # Creates float32 array
```

Type precision:

- float64 is supported on CPU only. Use with_device() or local_device() to run float64 work on CPU.

- Integer arithmetic may promote types (e.g., int32 + int32 might → int64)
- Mixed integer/float operations promote to float

Missing values:

MLX does not have an NA sentinel. When you pass numeric NA values from R, they are stored as NaN inside MLX and returned to R as NaN. Use `is.nan()` on MLX arrays if you need to detect them. `is.na()` on mlx objects calls `is.nan()`.

Scalars:

MLX allows scalar values, with a zero-length dimension (`integer(0)`). These are not usually what R users want. `as_mlx()` never returns a scalar; call `[mlx_reshape(x, integer(0))][mlx_reshape()]` to create one explicitly, or use `[mlx_array(..., allow_scalar = TRUE)][mlx_array()]`.

See Also

mlx.core.array

mlx-methods

Examples

```
# Default float32 for numeric
x <- as_mlx(c(1.5, 2.5, 3.5))
mlx_dtype(x)  # "float32"

# R integers also default to float32
x <- as_mlx(1:10)
mlx_dtype(x)  # "float32"

# Explicit integer types
x_int <- as_mlx(1:10, dtype = "int32")
mlx_dtype(x_int)  # "int32"

# Unsigned integers
x_uint <- as_mlx(c(0, 128, 255), dtype = "uint8")

# Logical → bool
mask <- as_mlx(c(TRUE, FALSE, TRUE))
mlx_dtype(mask)  # "bool"
```

---

as_r *Convert MLX array to base R objects*

Description

`as_r()` mirrors base R coercion rules: MLX objects with `dim()` equal to NULL return a plain vector, while higher-dimensional inputs return matrices or arrays.

Usage

```
as_r(x, ...)
```

Arguments

x	An mlx array.
...	Additional arguments; ignored.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

A vector, matrix, or array depending on the dimensions of `x`.

See Also

`as.array.mlx()`, `as.vector.mlx()`, `as.matrix.mlx()`

Examples

```
v <- as_mlx(1:3)
as_r(v)      # numeric vector
```

---

asplit *Split mlx arrays along a margin*

Description

`asplit()` extends base `asplit()` to work with mlx arrays by delegating to `mlx_split()`. When `x` is_mlx the result is a list of mlx arrays; otherwise, the base implementation is used.

Usage

```
asplit(x, MARGIN, drop = FALSE)

## Default S3 method:
asplit(x, MARGIN, drop = FALSE)

## S3 method for class 'mlx'
asplit(x, MARGIN, drop = FALSE)
```

Arguments

x	an array, including a matrix.
MARGIN	a vector giving the margins to split by. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where x has named dimnames, it can be a character vector selecting dimension names.
drop	a logical indicating whether the splits should drop dimensions and dimnames.

Details

Currently only a single MARGIN value is supported for mlx arrays.

Value

For mlx inputs, a list of mlx arrays; otherwise matches base::asplit().

---

cbind.mlx *Column-bind mlx arrays*

Description

Column-bind mlx arrays

Usage

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

Arguments

...	Objects to bind. mlx arrays are kept in MLX; other inputs are coerced via as_mlx().
deparse.level	Compatibility argument accepted for S3 dispatch; ignored.

Details

Unlike base R's cbind(), this function supports arrays with more than 2 dimensions and preserves all dimensions except the second (which is summed across inputs). Base R's cbind() flattens higher-dimensional arrays to matrices before binding.

Value

An mlx array stacked along the second axis.

See Also

mlx.core.concatenate

Examples

```
x <- mlx_matrix(1:4, 2, 2)
y <- mlx_matrix(5:8, 2, 2)
cbind(x, y)
```

`chol.mlx` *Cholesky decomposition for mlx arrays*

Description

If `x` is not symmetric positive semi-definite, "behaviour is undefined" according to the MLX documentation.

Usage

```
## S3 method for class 'mlx'
chol(x, pivot = FALSE, ..., device = NULL)
```

Arguments

`x`	An mlx matrix (2-dimensional array).
`pivot`	Ignored; pivoted decomposition is not supported.
`...`	Additional arguments; ignored.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

Upper-triangular Cholesky factor as an mlx matrix.

See Also

mlx.linalg.cholesky

Examples

```
x <- mlx_matrix(c(4, 1, 1, 3), 2, 2)
chol(x, device = "cpu")
```

chol2inv *Inverse from Cholesky decomposition*

Description

Compute the inverse of a symmetric, positive definite matrix from its Cholesky decomposition. The input `x` should be an upper triangular matrix from `chol()`.

Usage

```
chol2inv(x, size = NCOL(x), ...)

## Default S3 method:
chol2inv(x, size = NCOL(x), ...)

## S3 method for class 'mlx'
chol2inv(x, size = NCOL(x), ..., device = NULL)
```

Arguments

`x`	An mlx matrix (2-dimensional array).
`size`	Ignored; included for compatibility with base R.
`...`	Additional arguments; ignored.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

The inverse of the original matrix (before Cholesky decomposition).

See Also

`chol()`, `solve()`, `mlx_cholesky_inv()`

Examples

```
A <- mlx_matrix(c(4, 1, 1, 3), 2, 2)
U <- chol(A, device = "cpu")
A_inv <- chol2inv(U, device = "cpu")
# Verify: A %*% A_inv should be identity
A %*% A_inv
```

`colMeans` *Column means for mlx arrays*

Description

Column means for mlx arrays

Usage

```
colMeans(x, ...)

## Default S3 method:
colMeans(x, na.rm = FALSE, dims = 1, ...)

## S3 method for class 'mlx'
colMeans(x, na.rm = FALSE, dims = 1, ...)
```

Arguments

`x`	An array or mlx array.
`...`	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
`na.rm`	Logical; currently ignored for mlx arrays.
`dims`	Leading dimensions treated as rows/cols (see `base::rowSums()`).

Value

An mlx array if x is_mlx, otherwise a numeric vector.

See Also

mlx.core.mean

Examples

```
x <- mlx_matrix(1:6, 3, 2)
colMeans(x)
```

colSums *Column sums for mlx arrays*

Description

Column sums for mlx arrays

Usage

```
colSums(x, ...)

## Default S3 method:
colSums(x, na.rm = FALSE, dims = 1, ...)

## S3 method for class 'mlx'
colSums(x, na.rm = FALSE, dims = 1, ...)
```

Arguments

x	An array or mlx array.
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
na.rm	Logical; currently ignored for mlx arrays.
dims	Leading dimensions treated as rows/cols (see base::rowSums()).

Value

An mlx array if x is_mlx, otherwise a numeric vector.

See Also

mlx.core.sum

Examples

```
x <- mlx_matrix(1:6, 3, 2)
colSums(x)
```

crossprod.mlx *Cross product*

Description

Cross product

Usage

```
## S3 method for class 'mlx'
crossprod(x, y = NULL, ...)
```

Arguments

x	An mlx matrix (2-dimensional array).
y	An mlx matrix (default: NULL, uses x)
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.

Value

t(x) %*% y as an mlx object.

See Also

mlx.core.matmul

Examples

```
x <- mlx_matrix(1:6, 2, 3)
crossprod(x)
```

diag *Diagonal matrix extraction and construction*

Description

Generic function for extracting/constructing diagonal matrices.

Usage

```
diag(x = 1, nrow, ncol, names = TRUE)
```

Arguments

x	An object.
nrow, ncol	Optional dimensions for matrix construction.
names	Logical indicating whether to use names.

`diag.mlx` *Extract diagonal or construct diagonal matrix for mlx arrays*

Description

Extract a diagonal from a matrix or construct a diagonal matrix from a vector.

Usage

```
## S3 method for class 'mlx'
diag(x, nrow, ncol, names = TRUE)

mlx_diagonal(x, offset = 0L, axis1 = 1L, axis2 = 2L)
```

Arguments

x	An mlx array. If 1D, creates a diagonal matrix. If 2D or higher, extracts the diagonal.
nrow, ncol	Diagonal offset (nrow only; ncol ignored). `diag.mlx()` is an R interface to `mlx_diagonal()` with the same semantics as `base::diag()`.
names	Logical; when `TRUE`, diagonal extraction may preserve names like `base::diag()`.
offset	Diagonal offset (0 for main diagonal, positive for above, negative for below).
axis1, axis2	For multi-dimensional arrays, which axes define the 2D planes (1-indexed).

Value

An mlx array.

See Also

mlx.core.diagonal

Examples

```
# Extract diagonal
x <- mlx_matrix(1:9, 3, 3)
mlx_diagonal(x)
# (Constructing diagonals from 1D inputs is not yet supported.)
```

dim.mlx *Get dimensions of MLX array*

Description

dim() mirrors base R semantics and returns NULL for 1-D vectors and scalars, while mlx_shape() always returns the raw MLX shape (integers, never NULL). Use mlx_shape() when you need the underlying MLX dimension metadata and dim() when you want R-like behaviour.

Usage

```
## S3 method for class 'mlx'
dim(x)

mlx_shape(x)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().

Value

For dim(), an integer vector of dimensions or NULL for vectors/ scalars. For mlx_shape(), an integer vector (length zero for scalars).

Examples

```
x <- mlx_matrix(1:4, 2, 2)
dim(x)

v <- as_mlx(1:3)
dim(v)        # NULL (matches base R)
mlx_shape(v)  # 3
```

dim<-.mlx *Set dimensions of MLX array*

Description

Reshapes the MLX array to the specified dimensions. The total number of elements must remain the same.

Usage

```
## S3 replacement method for class 'mlx'
dim(x) <- value
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
value	Integer vector of new dimensions

Value

Reshaped mlx object.

See Also

mlx_reshape()

Examples

```
x <- as_mlx(1:12)
dim(x) <- c(3, 4)
dim(x)
```

---

drop *Drop singleton dimensions*

Description

drop() removes axes of length one. For base R objects this dispatches to base::drop(), while drop.mlx() delegates to mlx_squeeze() so that mlx arrays remain on the device.

Usage

```
drop(x)

## Default S3 method:
drop(x)

## S3 method for class 'mlx'
drop(x)
```

Arguments

x	Object to drop dimensions from.

Value

An object with singleton dimensions removed. For mlx inputs the result is another mlx array.

See Also

mlx_squeeze(), base::drop()

`fft` *Fast Fourier Transform*

Description

Extends `stats::fft()` to work with mlx objects while delegating to the standard R implementation for other inputs.

Usage

```
fft(z, inverse = FALSE, ...)

## Default S3 method:
fft(z, inverse = FALSE, ...)

## S3 method for class 'mlx'
fft(z, inverse = FALSE, axis, ...)
```

Arguments

`z`	Input to transform. May be a numeric, complex, or mlx object.
`inverse`	Logical flag; if `TRUE` compute the inverse transform.
`...`	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
`axis`	Single axis (1-indexed). Supply a positive integer between 1 and the array rank. Use `NULL` when the helper interprets it as "all axes" (see individual docs).

Value

For mlx inputs, an mlx object containing complex frequency coefficients; otherwise the base R result.

See Also

`stats::fft()`, `mlx_fft()`, `mlx_fft2()`, `mlx_fftn()`, mlx.core.fft.fft

Examples

```
z <- as_mlx(c(1, 2, 3, 4))
fft(z)
fft(z, inverse = TRUE)
```

format.mlx_stream *Format method for mlx_stream*

Description

Format method for mlx_stream

Usage

```
## S3 method for class 'mlx_stream'
format(x, ...)
```

Arguments

x	An mlx_stream object.
...	Additional arguments; ignored.

Value

A character string.

is_mlx *Test if object is an MLX array*

Description

Test if object is an MLX array

Usage

```
is_mlx(x)
```

Arguments

x	Object to test

Value

Logical scalar.

Examples

```
x <- mlx_matrix(1:4, 2, 2)
is_mlx(x)
```

kronecker *Kronecker product dispatcher*

Description

Wrapper around base::kronecker() that enables S3 dispatch for mlx arrays while delegating to base R for all other inputs.

Ensures the base kronecker() generic can dispatch on S3 mlx objects when S4 dispatch is unavailable.

Usage

```
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)

kronecker.default(X, Y, FUN = "*", make.dimnames = FALSE, ...)

## S4 method for signature 'mlx,mlx'
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)

## S4 method for signature 'mlx,ANY'
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)

## S4 method for signature 'ANY,mlx'
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)

kronecker.mlx(X, Y, FUN = "*", ..., make.dimnames = FALSE)
```

Arguments

X	a vector or array.
Y	a vector or array.
FUN	Must be '*' (other functions are unsupported for MLX tensors).
make.dimnames	logical: provide dimnames that are the product of the dimnames of X and Y.
...	optional arguments to be passed to FUN.

Value

An mlx array.

length.mlx *Get length of MLX array*

Description

Get length of MLX array

Usage

```
## S3 method for class 'mlx'
length(x)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().

Value

Total number of elements.

Examples

```
x <- mlx_matrix(1:6, 2, 3)
length(x)
```

Math.mlx *Math operations for MLX arrays*

Description

Math operations for MLX arrays

Usage

```
## S3 method for class 'mlx'
Math(x, ...)
```

Arguments

x	An mlx array.
...	Additional arguments; ignored.

Value

An mlx object with the result.

See Also

mlx.core.array

Examples

```
x <- mlx_matrix(c(-1, 0, 1), 3, 1)
sin(x)
round(x + 0.4)
```

`mean.mlx` *Mean of MLX array elements*

Description

Mean of MLX array elements

Usage

```
## S3 method for class 'mlx'
mean(x, ...)
```

Arguments

`x`	An mlx array.
`...`	Additional arguments; ignored.

Value

An mlx scalar.

See Also

mlx.core.mean

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mean(x)
```

`mlx-dimnames` *Dimnames and names for MLX arrays*

Description

Since version 0.4.0, Rmlx supports character names and dimnames. These work much like base R: `names()`, `rownames()`, `colnames()`, `dimnames()` and associated setters all work. Dimnames may be `NULL` as a whole, and any individual dimension's names may be `NULL`.

Usage

```
## S3 method for class 'mlx'
dimnames(x)

## S3 replacement method for class 'mlx'
dimnames(x) <- value

## S3 method for class 'mlx'
names(x)

## S3 replacement method for class 'mlx'
names(x) <- value
```

Arguments

`x`	An object.
`value`	Replacement names or dimnames.

Details

Dimnames can be convenient, but they also slow down operations by adding work in base R (mlx can't store character strings). In particular, slowdowns for subsetting can be considerable, maybe 5x or 6x slower. To maximize performance, you can avoid names, or remove them with `unname()`.

Value

The requested names, or `x` with updated metadata for replacement forms.

mlx-methods *Base R generics with mlx methods*

Description

Rmlx provides S3 methods for a number of base R generics so that common operations keep working after converting objects with `as_mlx()`. The main entry points are:

Details

- `%*%` for matrix multiplication
- `[` and `[<-` for extraction and assignment
- `Ops` and `Math` for elementwise arithmetic and math
- `Summary` for reductions such as `sum()` and `max()`; also `mean()`, `length()` and `all.equal()`.
- `diag()`, `dim()` and `dim<-`
- `as_r()`, `as.matrix()`, `as.array()`, and `as.vector()` for conversion back to base R
- `row()` and `col()` for index helpers that play nicely with mlx arrays
- `cbind()` and `rbind()` for binding arrays along rows or columns; there is also an `abind()` function modelled on `abind::abind()`.
- `rowMeans()`, `colMeans()`, `rowSums()`, and `colSums()` for axis-wise summaries
- `aperm()`, `t()`, and `dim<-` for shape manipulation
- `kronecker()`, `outer()`, `crossprod()`, and `tcrossprod()` for linear algebra helpers
- `fft()`, `chol()`, `chol2inv()`, `backsolve()`, and `solve()` for numerical routines
- `scale()` for column-wise centring and scaling that stays on the MLX backend
- `asplit()` to slice arrays along a margin while staying on the MLX backend
- `is.finite()`, `is.infinite()` and `is.nan()`

Most methods return mlx objects. One exception is that `all()` and `any()` return standard R `TRUE` or `FALSE` when used on mlx objects.

See Also

`as_mlx()`

mlx_addmm *Fused matrix multiply and add for MLX arrays*

Description

Computes `beta * input + alpha * (mat1 %*% mat2)` in a single MLX kernel. All operands are promoted to a common dtype prior to evaluation.

Usage

```
mlx_addmm(input, mat1, mat2, alpha = 1, beta = 1)
```

Arguments

`input`	Matrix-like object providing the additive term.
`mat1`	Left matrix operand.
`mat2`	Right matrix operand.
`alpha, beta`	Numeric scalars controlling the fused linear combination.

Value

An `mlx` matrix with the same shape as `input`.

See Also

mlx.core.addmm

Examples

```
input <- as_mlx(diag(3))
mat1 <- as_mlx(matrix(rnorm(9), 3, 3))
mat2 <- as_mlx(matrix(rnorm(9), 3, 3))
mlx_addmm(input, mat1, mat2, alpha = 0.5, beta = 2)
```

mlx_allclose *Test if all elements of two arrays are close*

Description

Returns a boolean scalar indicating whether all elements of two arrays are close within specified tolerances.

Usage

```
mlx_allclose(a, b, rtol = 1e-05, atol = 1e-08, equal_nan = FALSE)
```

Arguments

a, b	MLX arrays or objects coercible to MLX arrays
rtol	Relative tolerance (default: 1e-5)
atol	Absolute tolerance (default: 1e-8)
equal_nan	If TRUE, NaN values are considered equal (default: FALSE)

Details

Two values are considered close if: `abs(a - b) <= (atol + rtol * abs(b))`.

Infinite values with matching signs are considered equal. Supports NumPy-style broadcasting.

Value

An mlx array containing a single boolean value

See Also

`mlx_isclose()`, `all.equal.mlx()`, mlx.core.allclose

Examples

```
a <- as_mlx(c(1.0, 2.0, 3.0))
b <- as_mlx(c(1.0 + 1e-6, 2.0 + 1e-6, 3.0 + 1e-6))
mlx_allclose(a, b)  # TRUE
```

---

mlx_arange *Numerical ranges on MLX devices*

Description

`mlx_arange()` creates evenly spaced values starting at `start`, stepping by `step`, up to and including `stop` (if exactly reachable). This matches R's `base::seq()` behavior.

Usage

```
mlx_arange(
  start,
  stop,
  step = 1,
 dtype = c("float32", "float64", "int8", "int16", "int32", "int64", "uint8", "uint16",
    "uint32", "uint64")
)
```

Arguments

`start`	Starting value.
`stop`	Upper bound (included if exactly reachable by the step sequence).
`step`	Step size (defaults to 1).
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

A 1D mlx array.

Difference from Python/C++

Unlike Python's `range()` and `numpy.arange()` which use an exclusive upper bound, `mlx_arange()` matches R's `base::seq()` by including `stop` only if it's exactly reachable by the step sequence. This is consistent with `mlx_linspace()` and `mlx_slice_update()`, which also follow R conventions.

See Also

mlx.core.arange

Examples

```
mlx_arange(0, 4)        # 0, 1, 2, 3, 4
mlx_arange(1, 5)        # 1, 2, 3, 4, 5
mlx_arange(1, 9, 2)     # 1, 3, 5, 7, 9
mlx_arange(1, 6, 2)     # 1, 3, 5 (6 not reachable)
```

`mlx_argmax` *Argmax and argmin on mlx arrays*

Description

Argmax and argmin on mlx arrays

Usage

```
mlx_argmax(x, axis = NULL, drop = TRUE)

mlx_argmin(x, axis = NULL, drop = TRUE)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`axis`	Single axis (1-indexed). Supply a positive integer between 1 and the array rank. Use `NULL` when the helper interprets it as "all axes" (see individual docs).
`drop`	If `TRUE` (default), drop dimensions of length 1. If `FALSE`, retain all dimensions. Equivalent to `keepdims = TRUE` in underlying mlx functions.

Details

When `axis = NULL`, the array is flattened before computing extrema. Setting `drop = FALSE` retains the reduced axis as length one in the returned indices.

Value

An mlx array of indices. Indices are 1-based to match R's conventions.

See Also

mlx.core.argmax, mlx.core.argmin

Examples

```
x <- as_mlx(matrix(c(1, 5, 3, 2), 2, 2))
mlx_argmax(x)
mlx_argmax(x, axis = 1)
mlx_argmin(x)
```

`mlx_array` *Construct an MLX array from R data*

Description

`mlx_array()` is a low-level constructor that skips `as_mlx()`'s type inference and dimension guessing. Supply the raw payload vector plus an explicit shape and it pipes the data straight into MLX.

Usage

```
mlx_array(data, dim, dtype = NULL, dimnames = NULL)
```

Arguments

`data`	Numeric, logical, or complex vector. `data` is recycled to match dimensions according to R rules (but with an error if it doesn't tile into the dimensions exactly).
`dim`	Integer vector of array dimensions. Set `dim = integer(0)` for a scalar, in which case `data` must be length 1.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.
`dimnames`	Optional list of character vectors naming each dimension.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An `mlx` array with the requested shape.

Examples

```
payload <- runif(6)
mlx_array(payload, dim = c(2, 3))
```

mlx_batch_norm *Batch normalization*

Description

Normalizes inputs across the batch dimension.

Usage

```
mlx_batch_norm(num_features, eps = 1e-05, momentum = 0.1)
```

Arguments

num_features	Number of feature channels.
eps	Small constant for numerical stability (default: 1e-5).
momentum	Momentum for running statistics (default: 0.1).

Value

An mlx_module applying batch normalization.

See Also

mlx.nn.BatchNorm

Examples

```
set.seed(1)
bn <- mlx_batch_norm(4)
x <- as_mlx(matrix(rnorm(12), 3, 4))
mlx_forward(bn, x)
```

mlx_best_device *Get best available device*

Description

Returns "gpu" if available, otherwise "cpu".

Usage

```
mlx_best_device()
```

Value

Character: "gpu" or "cpu".

Examples

```
device <- mlx_best_device()
with_device(device, x <- as_mlx(1:10))
```

`mlx_binary_cross_entropy` *Binary cross-entropy loss*

Description

Computes binary cross-entropy loss between predictions and binary targets.

Usage

```
mlx_binary_cross_entropy(
  predictions,
  targets,
  reduction = c("mean", "sum", "none")
)
```

Arguments

`predictions`	Predicted probabilities as an mlx array (values in [0,1]).
`targets`	Binary target values as an mlx array (0 or 1).
`reduction`	Type of reduction: "mean" (default), "sum", or "none".

Value

An mlx array containing the loss.

See Also

mlx.nn.losses.binary_cross_entropy

Examples

```
preds <- mlx_matrix(c(0.9, 0.2, 0.8), 3, 1)
targets <- mlx_matrix(c(1, 0, 1), 3, 1)
mlx_binary_cross_entropy(preds, targets)
```

mlx_broadcast_arrays *Broadcast multiple arrays to a shared shape*

Description

mlx_broadcast_arrays() mirrors mlx.core.broadcast_arrays(), returning a list of inputs expanded to a common shape.

Usage

```
mlx_broadcast_arrays(...)
```

Arguments

...	One or more arrays (or a single list of arrays) coercible via as_mlx().

Value

A list of broadcast mlx arrays, with each input's dimnames broadcast to the shared shape where possible.

See Also

mlx.core.broadcast_arrays

Examples

```
a <- mlx_matrix(1:3, nrow = 1)
b <- mlx_matrix(1:3, ncol = 1)
outs <- mlx_broadcast_arrays(a, b)
lapply(outs, dim)
```

mlx_broadcast_to *Broadcast an array to a new shape*

Description

mlx_broadcast_to() mirrors mlx.core.broadcast_to(), repeating singleton dimensions without copying data.

Usage

```
mlx_broadcast_to(x, shape)
```

Arguments

x	An mlx array.
shape	Integer vector describing the broadcasted shape.

Value

An mlx array with the requested dimensions. Dimnames from matching or singleton broadcast axes are carried to the result.

See Also

mlx.core.broadcast_to

Examples

```
x <- mlx_matrix(1:3, nrow = 1)
broadcast <- mlx_broadcast_to(x, c(5, 3))
dim(broadcast)
```

---

mlx_cast *Cast an mlx array*

Description

`mlx_cast()` converts an mlx array to a different dtype without changing its shape.

Usage

```
mlx_cast(x, dtype = NULL)
```

Arguments

x	An mlx array.
dtype	Target dtype string. Defaults to the array's current dtype.

Value

An mlx array with the requested dtype.

See Also

`mlx_dtype()`

Examples

```
x <- mlx_vector(1:3, dtype = "int32")
mlx_cast(x, dtype = "float32")
```

`mlx_cholesky_inv` *Compute matrix inverse via Cholesky decomposition*

Description

Computes the inverse of a positive definite matrix from its Cholesky factor. Note: `x` should be the Cholesky factor (L or U), not the original matrix.

Usage

```
mlx_cholesky_inv(x, upper = FALSE, device = NULL)
```

Arguments

`x`	An mlx array.
`upper`	Logical; if `TRUE`, `x` is upper triangular, otherwise lower triangular.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

For a more R-like interface, see `chol2inv()`.

Value

The inverse of the original matrix (A^-1 where A = LL' or A = U'U).

See Also

`chol2inv()`, mlx.core.linalg.cholesky_inv

Examples

```
# Create a positive definite matrix
A <- matrix(rnorm(9), 3, 3)
A <- t(A) %*% A
# Compute Cholesky factor
L <- chol(A, pivot = FALSE, upper = FALSE)
# Get inverse from Cholesky factor
mlx_cholesky_inv(as_mlx(L), device = "cpu")
```

mlx_clip *Clip mlx array values into a range*

Description

Clip mlx array values into a range

Usage

```
mlx_clip(x, min = NULL, max = NULL)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
min, max	Scalar bounds. Use NULL to leave a bound open.

Value

An mlx array with values clipped to [min, max].

See Also

mlx.core.clip

Examples

```
x <- as_mlx(rnorm(4))
mlx_clip(x, min = -1, max = 1)
```

mlx_compile *Compile an MLX Function for Optimized Execution*

Description

Returns a compiled version of a function that traces and optimizes the computation graph on first call, then reuses the compiled graph for subsequent calls with matching input shapes and types.

Usage

```
mlx_compile(f, shapeless = FALSE)
```

Arguments

f	An R function that takes MLX arrays as arguments and returns MLX array(s). The function must be pure (no side effects) and use only MLX operations.
shapeless	Logical. If TRUE, the compiled function won't recompile when input shapes change. However, changing input dtypes or number of dimensions still triggers recompilation. Default: FALSE

Details

How Compilation Works:

When you call `mlx_compile(f)`, it returns a new function immediately without any tracing. The actual compilation happens on the **first call** to the compiled function:

1. **First call**: MLX traces the function with placeholder inputs, builds the computation graph, optimizes it (fusing operations, eliminating redundancy), and caches the result. This is slow.
2. **Subsequent calls**: If inputs have the same shapes and dtypes, MLX reuses the cached compiled graph. This is fast.
3. **Recompilation**: Occurs when input shapes change (unless `shapeless = TRUE`), input dtypes change, or the number of arguments changes.

Requirements for Compiled Functions:

Your function must:

- Accept only MLX arrays as arguments
- Return MLX array(s) - either a single mlx object or a list of mlx objects
- Use only MLX operations (no conversion to R)
- Be pure (no side effects, no external state modification)

Your function **cannot**:

- Print or evaluate arrays during execution (`print()`, `as.matrix()`, `as.numeric()`, `[[` extraction, etc.)
- Use control flow based on array values (`if (x > 0)` where `x` is an array)
- Modify external variables or have other side effects
- Return non-MLX values

Performance Benefits:

- **Operation fusion**: Combines multiple operations into optimized kernels
- **Memory reduction**: Eliminates intermediate allocations
- **Overhead reduction**: Bypasses R/C++ call overhead for fused operations

Typical speedups range from 2-10x for operation-heavy functions.

Shapeless Compilation:

Setting `shapeless = TRUE` allows the compiled function to handle varying input shapes without recompilation:

```
# Regular compilation - recompiles for each new shape
fast_fn <- mlx_compile(matmul_fn)
fast_fn(mlx_zeros(c(10, 64)), weights)  # Compiles for shape (10, 64)
fast_fn(mlx_zeros(c(20, 64)), weights)  # Recompiles for shape (20, 64)

# Shapeless compilation - compiles once
fast_fn <- mlx_compile(matmul_fn, shapeless = TRUE)
fast_fn(mlx_zeros(c(10, 64)), weights)  # Compiles once
fast_fn(mlx_zeros(c(20, 64)), weights)  # No recompilation!
```

Shapeless mode sacrifices some optimization opportunities but avoids recompilation costs. Use it when processing variable-sized batches.

Value

A compiled function with the same signature as `f`. The first call will be slow (tracing and compilation), but subsequent calls will be much faster.

See Also

`mlx_disable_compile()`, `mlx_enable_compile()`

mlx.core.compile

Examples

```
# Simple example
matmul_add <- function(x, w, b) {
  (x %*% w) + b
}

# Compile it (returns immediately, no tracing yet)
fast_fn <- mlx_compile(matmul_add)

# First call: slow (traces and compiles)
x <- mlx_rand_normal(c(32, 128))
w <- mlx_rand_normal(c(128, 256))
b <- mlx_rand_normal(c(256))
result <- fast_fn(x, w, b)  # Compiles during this call

# Subsequent calls: fast (uses cached graph)
batches <- replicate(10, mlx_rand_normal(c(32, 128)), simplify = FALSE)
for (bat in batches) {
  result <- fast_fn(bat, w, b)  # Uses cached graph
}

# Multiple returns
forward_and_norm <- function(x, w) {
  y <- x %*% w
  norm <- sqrt(sum(y * y))
  list(y, norm)  # Return list of mlx objects
}

compiled_fn <- mlx_compile(forward_and_norm)
results <- compiled_fn(x, w)  # Returns list(y, norm)
```

---

mlx_contiguous *Ensure contiguous memory layout*

Description

Returns a copy of `x` with contiguous strides.

Usage

```
mlx_contiguous(x)
```

Arguments

x	An mlx array.

Value

An mlx array backed by contiguous storage.

See Also

https://ml-explore.github.io/mlx/build/html/python/array.html#mlx.core.contiguous

Examples

```
x <- mlx_swapaxes(mlx_matrix(1:4, 2, 2), axis1 = 1, axis2 = 2)
y <- mlx_contiguous(x)
identical(as.array(x), as.array(y))
```

---

mlx_conv_transpose1d *1D Transposed Convolution*

Description

Applies a 1D transposed convolution (also called deconvolution) over an input signal. Transposed convolutions are used to upsample the spatial dimensions of the input.

Usage

```
mlx_conv_transpose1d(
  input,
  weight,
  stride = 1L,
  padding = 0L,
  dilation = 1L,
  output_padding = 0L,
  groups = 1L
)
```

Arguments

input	Input mlx array. Shape depends on dimensionality (see individual functions).
weight	Weight array. Shape depends on dimensionality (see individual functions).
stride	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.

padding	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
dilation	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
output_padding	Additional size added to output shape. Default: 0
groups	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape (batch, length, in_channels) for 'NWC' layout. Weight has shape (out_channels, kernel_size, in

Value

An mlx array with the transposed convolution result

See Also

mlx_conv1d(), mlx_conv_transpose2d(), mlx_conv_transpose3d()

mlx.nn

---

mlx_conv_transpose2d *2D Transposed Convolution*

Description

Applies a 2D transposed convolution (also called deconvolution) over an input signal. Transposed convolutions are commonly used in image generation and upsampling tasks.

Usage

```
mlx_conv_transpose2d(
  input,
  weight,
  stride = c(1L, 1L),
  padding = c(0L, 0L),
  dilation = c(1L, 1L),
  output_padding = c(0L, 0L),
  groups = 1L
)
```

Arguments

input	Input mlx array. Shape depends on dimensionality (see individual functions).
weight	Weight array. Shape depends on dimensionality (see individual functions).
stride	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.

padding	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
dilation	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
output_padding	Additional size added to output shape. Can be a scalar or length-2 vector. Default: c(0, 0)
groups	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape `(batch, height, width, in_channels)` for 'NHWC' layout. Weight has shape `(out_channels, kernel_h, kernel_w, in_channels)`.

Value

An mlx array with the transposed convolution result

See Also

`mlx_conv2d()`, `mlx_conv_transpose1d()`, `mlx_conv_transpose3d()`

mlx.nn

---

mlx_conv_transpose3d *3D Transposed Convolution*

Description

Applies a 3D transposed convolution (also called deconvolution) over an input signal. Useful for 3D volumetric data upsampling, such as in medical imaging or video generation.

Usage

```
mlx_conv_transpose3d(
  input,
  weight,
  stride = c(1L, 1L, 1L),
  padding = c(0L, 0L, 0L),
  dilation = c(1L, 1L, 1L),
  output_padding = c(0L, 0L, 0L),
  groups = 1L
)
```

Arguments

input	Input mlx array. Shape depends on dimensionality (see individual functions).
weight	Weight array. Shape depends on dimensionality (see individual functions).
stride	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
padding	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
dilation	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
output_padding	Additional size added to output shape. Can be a scalar or length-3 vector. Default: c(0, 0, 0)
groups	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape `(batch, depth, height, width, in_channels)` for 'NDHWC' layout. Weight has shape `(out_channels, kernel_d, kernel_h, kernel_w, in_channels)`.

Value

An mlx array with the transposed convolution result

See Also

`mlx_conv3d()`, `mlx_conv_transpose1d()`, `mlx_conv_transpose2d()`

mlx.nn

---

mlx_conv1d *1D Convolution*

Description

Applies a 1D convolution over the input signal.

Usage

```
mlx_conv1d(
  input,
  weight,
  stride = 1L,
  padding = 0L,
  dilation = 1L,
  groups = 1L
)
```

Arguments

`input`	Input mlx array. Shape depends on dimensionality (see individual functions).
`weight`	Weight array. Shape depends on dimensionality (see individual functions).
`stride`	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
`padding`	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
`dilation`	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
`groups`	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape `(N, L, C_in)` where N is batch size, L is sequence length, and C_in is number of input channels. Weight has shape `(C_out, kernel_size, C_in)`.

Value

Convolved output array

See Also

mlx.core.conv1d

`mlx_conv2d` *2D Convolution*

Description

Applies a 2D convolution over the input image.

Usage

```
mlx_conv2d(
  input,
  weight,
  stride = c(1L, 1L),
  padding = c(0L, 0L),
  dilation = c(1L, 1L),
  groups = 1L
)
```

Arguments

input	Input mlx array. Shape depends on dimensionality (see individual functions).
weight	Weight array. Shape depends on dimensionality (see individual functions).
stride	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
padding	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
dilation	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
groups	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape `(N, H, W, C_in)` where N is batch size, H and W are height and width, and C_in is number of input channels. Weight has shape `(C_out, kernel_h, kernel_w, C_in)`.

Value

Convolved output array

See Also

mlx.core.conv2d

Examples

```
# Create a simple 2D convolution
input <- mlx_array(rnorm(1*28*28*3), dim = c(1, 28, 28, 3))  # Batch of 1 RGB image
weight <- mlx_array(rnorm(16*3*3*3), dim = c(16, 3, 3, 3))  # 16 filters, 3x3 kernel
output <- mlx_conv2d(input, weight, stride = c(1, 1), padding = c(1, 1))
```

---

mlx_conv3d *3D Convolution*

Description

Applies a 3D convolution over the input volume.

Usage

```
mlx_conv3d(
  input,
  weight,
  stride = c(1L, 1L, 1L),
  padding = c(0L, 0L, 0L),
  dilation = c(1L, 1L, 1L),
  groups = 1L
)
```

Arguments

`input`	Input mlx array. Shape depends on dimensionality (see individual functions).
`weight`	Weight array. Shape depends on dimensionality (see individual functions).
`stride`	Stride of the convolution. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
`padding`	Amount of zero padding. Can be a scalar or vector (length depends on dimensionality). Default: 0 for 1D, c(0,0) for 2D, c(0,0,0) for 3D.
`dilation`	Spacing between kernel elements. Can be a scalar or vector (length depends on dimensionality). Default: 1 for 1D, c(1,1) for 2D, c(1,1,1) for 3D.
`groups`	Number of blocked connections from input to output channels. Default: 1.

Details

Input has shape `(N, D, H, W, C_in)` where N is batch size, D, H, W are depth, height and width, and C_in is number of input channels. Weight has shape `(C_out, kernel_d, kernel_h, kernel_w, C_in)`.

Value

Convolved output array

See Also

mlx.core.conv3d

`mlx_coordinate_descent` *Coordinate Descent with L1 Regularization*

Description

Minimizes f(beta) + lambda * ||beta||_1 using coordinate descent, where f is a smooth differentiable loss function.

Usage

```
mlx_coordinate_descent(
  loss_fn,
  beta_init,
  lambda = 0,
  ridge_penalty = 0,
  grad_fn = NULL,
  lipschitz = NULL,
  max_iter = 1000,
  tol = 1e-06,
  block_size = 1,
  grad_cache = NULL
)
```

Arguments

loss_fn	Function(beta) -> scalar loss (MLX tensor). Must be smooth and differentiable.
beta_init	Initial parameter vector (p x 1 MLX tensor).
lambda	L1 penalty parameter (scalar, default 0).
ridge_penalty	Optional ridge (L2) penalty term applied per-coordinate when computing gradients. Can be a scalar, numeric vector of length p, or an mlx array with shape compatible with beta_init.
grad_fn	Optional gradient function. If NULL, computed via mlx_grad(loss_fn).
lipschitz	Optional Lipschitz constants for each coordinate (length p vector). If NULL, uses simple constant estimates.
max_iter	Maximum number of iterations (default 1000).
tol	Convergence tolerance (default 1e-6).
block_size	Number of coordinates to update before recomputing the gradient. Set to 1 for strict coordinate descent; larger values trade accuracy for speed.
grad_cache	Optional environment for supplying cached gradient components. Supported fields are type = "gaussian" with entries x, residual, n_obs, and optional ridge_penalty; or type = "binomial" with entries x, eta, mu, residual, y, n_obs, and optional ridge_penalty.

Details

This function implements proximal gradient descent for problems of the form: min_beta f(beta) + lambda * ||beta||_1

where f is smooth. At each iteration, all coordinates are updated via: z = beta - (1/L) * grad_f(beta) beta = soft_threshold(z, lambda/L)

where L are Lipschitz constants for each coordinate.

Value

List with:

- beta: Optimized parameter vector (MLX tensor)
- n_iter: Number of iterations performed
- converged: Whether convergence criterion was met

Examples

```
# Lasso regression: min 0.5*||y - X*beta||^2 + lambda*||beta||_1
n <- 100
p <- 50
X <- as_mlx(matrix(rnorm(n*p), n, p))
y <- as_mlx(matrix(rnorm(n), ncol=1))
beta_init <- mlx_zeros(c(p, 1))

loss_fn <- function(beta) {
  residual <- y - X %*% beta
```

```
  sum(residual^2) / (2*n)
}

result <- mlx_coordinate_descent(
  loss_fn = loss_fn,
  beta_init = beta_init,
  lambda = 0.1,
  block_size = 8
)

# Reuse cached residuals for a Gaussian problem
grad_cache <- new.env(parent = emptyenv())
grad_cache$type <- "gaussian"
grad_cache$x <- X
grad_cache$n_obs <- n
grad_cache$residual <- y - X %*% beta_init
cached <- mlx_coordinate_descent(
  loss_fn = loss_fn,
  beta_init = beta_init,
  lambda = 0.1,
  grad_cache = grad_cache
)
```

`mlx_cross` *Vector cross product with mlx arrays*

Description

Vector cross product with mlx arrays

Usage

```
mlx_cross(a, b, axis = NULL)
```

Arguments

a, b	Input mlx arrays containing 3D vectors.
axis	Axis along which to compute the cross product (1-indexed). Omit the argument to use the trailing dimension.

Value

An mlx array of cross products.

See Also

mlx.linalg.cross

Examples

```
u <- as_mlx(c(1, 0, 0))
v <- as_mlx(c(0, 1, 0))
mlx_cross(u, v)
```

`mlx_cross_entropy` *Cross-entropy loss*

Description

Computes cross-entropy loss for multi-class classification.

Usage

```
mlx_cross_entropy(logits, targets, reduction = c("mean", "sum", "none"))
```

Arguments

`logits`	A matrix of unnormalized predictions (logits), with one row per sample and one column per class, as an mlx array or R matrix.
`targets`	Target class indices as an mlx array or integer vector. Indices are 1-based, with one non-missing index per sample.
`reduction`	Type of reduction: "mean" (default), "sum", or "none".

Details

Gradients can be computed with respect to `logits`. Float64 arrays require the CPU.

Value

A scalar mlx array for `"mean"` or `"sum"`, or a vector with one loss per sample for `"none"`.

See Also

mlx.nn.losses.cross_entropy

Examples

```
# Logits for 3 samples, 4 classes
logits <- mlx_matrix(rnorm(12), 3, 4)
targets <- as_mlx(c(1, 3, 2))
mlx_cross_entropy(logits, targets)
```

`mlx_cumsum` *Cumulative sum and product*

Description

Compute cumulative sums or products along an axis.

Usage

```
mlx_cumsum(x, axis = NULL, reverse = FALSE, inclusive = TRUE)

mlx_cumprod(x, axis = NULL, reverse = FALSE, inclusive = TRUE)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`axis`	Single axis (1-indexed). Supply a positive integer between 1 and the array rank. Use `NULL` when the helper interprets it as "all axes" (see individual docs).
`reverse`	If `TRUE`, compute in reverse order.
`inclusive`	If `TRUE` (default), include the current element in the cumulative operation. If `FALSE`, the cumulative operation is exclusive (starts from identity element).

Details

When `axis` is `NULL` (default), the array is flattened before computing the cumulative result.

Value

An mlx array with cumulative sums or products.

See Also

`cumsum()`, `cumprod()`, mlx.core.cumsum, mlx.core.cumprod

Examples

```
x <- as_mlx(1:5)
mlx_cumsum(x)  # [1, 3, 6, 10, 15]

mat <- mlx_matrix(1:12, 3, 4)
mlx_cumsum(mat, axis = 1)  # cumsum down rows
```

mlx_degrees *Convert between radians and degrees*

Description

mlx_degrees() and mlx_radians() mirror mlx.core.degrees() and mlx.core.radians(), converting angular values elementwise using MLX kernels.

Usage

```
mlx_degrees(x)

mlx_radians(x)
```

Arguments

x	An mlx array.

Value

An mlx array with transformed angular units.

See Also

mlx.core.degrees, mlx.core.radians

Examples

```
x <- as_mlx(pi / 2)
mlx_degrees(x)  # 90
mlx_radians(mlx_vector(c(0, 90, 180)))
```

mlx_dequantize *Dequantize a Matrix*

Description

Reconstructs an approximate floating-point matrix from a quantized representation produced by mlx_quantize().

Usage

```
mlx_dequantize(
  w,
  scales,
  biases = NULL,
  group_size = 64L,
  bits = 4L,
  mode = "affine"
)
```

Arguments

w	An mlx array representing the weight matrix. Accepts either an unquantized matrix (which may be quantized automatically) or a pre-quantized uint32 matrix produced by mlx_quantize().
scales	An optional mlx array of quantization scales. Required when w is already quantized.
biases	An optional mlx array of quantization biases (affine mode); use NULL for symmetric quantization.
group_size	The group size for quantization. Smaller groups improve accuracy at the cost of slightly higher memory. Default: 64.
bits	Number of bits for quantization (typically 4 or 8). Default: 4.
mode	Quantization mode, either "affine" or "mxfp4".

Details

Dequantization unpacks the low-precision quantized weights and applies the scales (and biases) to reconstruct approximate floating-point values. Note that some precision is lost during quantization and cannot be recovered.

Value

An mlx array with the dequantized (approximate) floating-point weights

See Also

mlx_quantize(), mlx_quantized_matmul()

Examples

```
w <- mlx_rand_normal(c(64, 32))
quant <- mlx_quantize(w, group_size = 32)
w_reconstructed <- mlx_dequantize(quant$w_q, quant$scales, quant$biases, group_size = 32)
```

mlx_device *Get or set current MLX device*

Description

Get or set current MLX device

Usage

```
mlx_device(value)
```

Arguments

value	New current device ("gpu" or "cpu"). If missing, returns the current device.

Value

Current device (character).

See Also

mlx.core.default_device

Examples

```
mlx_device()  # Get current device
mlx_device("cpu")  # Set to CPU
if (mlx_has_gpu()) {
  mlx_device("gpu")  # Set back to GPU
  mlx_device()
}
mlx_device("cpu")
```

mlx_dexp *Exponential distribution functions*

Description

Compute density (mlx_dexp), cumulative distribution (mlx_pexp), and quantile (mlx_qexp) functions for the exponential distribution using MLX.

Usage

```
mlx_dexp(x, rate = 1, log = FALSE)

mlx_pexp(x, rate = 1)

mlx_qexp(p, rate = 1)
```

Arguments

x	Vector of quantiles (mlx array or coercible to mlx)
rate	Rate parameter (default: 1)
log	If TRUE, return log density (default: FALSE)
p	Vector of probabilities (mlx array or coercible to mlx)

Value

An mlx array with the computed values.

Examples

```
x <- as_mlx(seq(0, 5, by = 0.5))
mlx_dexp(x)
mlx_pexp(x)

p <- as_mlx(c(0.25, 0.5, 0.75))
mlx_qexp(p)
```

---

mlx_disable_compile *Control Global Compilation Behavior*

Description

- mlx_disable_compile() prevents all compilation globally. Compiled functions will execute without optimization.
- mlx_enable_compile() enables compilation (overrides the MLX_DISABLE_COMPILE environment variable).

Usage

```
mlx_disable_compile()

mlx_enable_compile()
```

Details

These functions control whether MLX compilation is enabled globally.

These are useful for debugging (to check if compilation is causing issues) or benchmarking (to measure compilation overhead vs speedup).

You can also disable compilation by setting the MLX_DISABLE_COMPILE environment variable before loading the package.

Value

Invisibly returns NULL.

Examples

```
demo_fn <- mlx_compile(function(x) x + 1)
x <- mlx_rand_normal(c(4, 4))

# Disable compilation for debugging
mlx_disable_compile()
demo_fn(x)  # Runs without optimization

# Re-enable compilation
mlx_enable_compile()
demo_fn(x)  # Runs with optimization
```

---

mlx_dlnorm *Lognormal distribution functions*

Description

Compute density (`mlx_dlnorm`), cumulative distribution (`mlx_plnorm`), and quantile (`mlx_qlnorm`) functions for the lognormal distribution using MLX.

Usage

```
mlx_dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)

mlx_plnorm(x, meanlog = 0, sdlog = 1)

mlx_qlnorm(p, meanlog = 0, sdlog = 1)
```

Arguments

`x`	Vector of quantiles (mlx array or coercible to mlx)
`meanlog, sdlog`	Mean and standard deviation of distribution on log scale (default: 0, 1)
`log`	If `TRUE`, return log density (default: `FALSE`)
`p`	Vector of probabilities (mlx array or coercible to mlx)

Value

An mlx array with the computed values.

Examples

```
x <- as_mlx(seq(0.1, 3, by = 0.2))
mlx_dlnorm(x)
mlx_plnorm(x)

p <- as_mlx(c(0.25, 0.5, 0.75))
mlx_qlnorm(p)
```

mlx_dlogis *Logistic distribution functions*

Description

Compute density (`mlx_dlogis`), cumulative distribution (`mlx_plogis`), and quantile (`mlx_qlogis`) functions for the logistic distribution using MLX.

Usage

```
mlx_dlogis(x, location = 0, scale = 1, log = FALSE)

mlx_plogis(x, location = 0, scale = 1)

mlx_qlogis(p, location = 0, scale = 1)
```

Arguments

`x`	Vector of quantiles (mlx array or coercible to mlx)
`location, scale`	Location and scale parameters (default: 0, 1)
`log`	If `TRUE`, return log density (default: `FALSE`)
`p`	Vector of probabilities (mlx array or coercible to mlx)

Value

An mlx array with the computed values.

Examples

```
x <- as_mlx(seq(-3, 3, by = 0.5))
mlx_dlogis(x)
mlx_plogis(x)

p <- as_mlx(c(0.25, 0.5, 0.75))
mlx_qlogis(p)
```

mlx_dnorm *Normal distribution functions*

Description

Compute density (`mlx_dnorm`), cumulative distribution (`mlx_pnorm`), and quantile (`mlx_qnorm`) functions for the normal distribution using MLX.

Usage

```
mlx_dnorm(x, mean = 0, sd = 1, log = FALSE)

mlx_pnorm(x, mean = 0, sd = 1)

mlx_qnorm(p, mean = 0, sd = 1)
```

Arguments

x	Vector of quantiles (mlx array or coercible to mlx)
mean	Mean of the distribution (default: 0)
sd	Standard deviation of the distribution (default: 1)
log	If TRUE, return log density (default: FALSE)
p	Vector of probabilities (mlx array or coercible to mlx)

Value

An mlx array with the computed values.

See Also

mlx_erf(), mlx_erfinv(), mlx.core.erf, mlx.core.erfinv

Examples

```
x <- as_mlx(seq(-3, 3, by = 0.5))
mlx_dnorm(x)
mlx_pnorm(x)

p <- as_mlx(c(0.025, 0.5, 0.975))
mlx_qnorm(p)
```

---

mlx_dropout *Dropout layer*

Description

Dropout layer

Usage

```
mlx_dropout(p = 0.5)
```

Arguments

p	Probability of dropping an element (default: 0.5).

Value

An `mlx_module` applying dropout during training.

See Also

mlx.nn.Dropout

Examples

```
set.seed(1)
dropout <- mlx_dropout(p = 0.3)
x <- mlx_matrix(1:12, 3, 4)
mlx_forward(dropout, x)
```

`mlx_dtype` *Get the data type of an MLX array*

Description

Get the data type of an MLX array

Usage

```
mlx_dtype(x)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.

Value

A data type string (see `as_mlx()` for possibilities).

Examples

```
x <- mlx_matrix(1:6, 2, 3)
mlx_dtype(x)
```

mlx_dunif *Uniform distribution functions*

Description

Compute density (`mlx_dunif`), cumulative distribution (`mlx_punif`), and quantile (`mlx_qunif`) functions for the uniform distribution using MLX.

Usage

```
mlx_dunif(x, min = 0, max = 1, log = FALSE)

mlx_punif(x, min = 0, max = 1)

mlx_qunif(p, min = 0, max = 1)
```

Arguments

`x`	Vector of quantiles (mlx array or coercible to mlx)
`min`, `max`	Lower and upper limits of the distribution (default: 0, 1)
`log`	If `TRUE`, return log density (default: `FALSE`)
`p`	Vector of probabilities (mlx array or coercible to mlx)

Value

An mlx array with the computed values.

Examples

```
x <- as_mlx(seq(0, 1, by = 0.1))
mlx_dunif(x)
mlx_punif(x)

p <- as_mlx(c(0.25, 0.5, 0.75))
mlx_qunif(p)
```

mlx_eig *Eigen decomposition for mlx arrays*

Description

Eigen decomposition for mlx arrays

Usage

```
mlx_eig(x, device = NULL)
```

Arguments

x	An mlx matrix (2-dimensional array).
device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside with_device() or local_device(), or pass device = "cpu".

Value

A list with components values and vectors, both mlx arrays.

See Also

mlx.linalg.eig

Examples

```
x <- mlx_matrix(c(2, -1, 0, 2), 2, 2)
eig <- mlx_eig(x, device = "cpu")
eig$values
eig$vectors
```

---

mlx_eigh *Eigen decomposition of Hermitian mlx arrays*

Description

Eigen decomposition of Hermitian mlx arrays

Usage

```
mlx_eigh(x, uplo = c("L", "U"), device = NULL)
```

Arguments

x	An mlx matrix (2-dimensional array).
uplo	Character string indicating which triangle to use ("L" or "U").
device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

A list with components `values` and `vectors`.

See Also

mlx.linalg.eigh

Examples

```
x <- mlx_matrix(c(2, 1, 1, 3), 2, 2)
mlx_eigh(x, device = "cpu")
```

`mlx_eigvals` *Eigenvalues of mlx arrays*

Description

Eigenvalues of mlx arrays

Usage

```
mlx_eigvals(x, device = NULL)
```

Arguments

`x`	An mlx matrix (2-dimensional array).
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

An mlx array containing eigenvalues.

See Also

mlx.linalg.eigvals

Examples

```
x <- mlx_matrix(c(3, 1, 0, 2), 2, 2)
mlx_eigvals(x, device = "cpu")
```

`mlx_eigvalsh` *Eigenvalues of Hermitian mlx arrays*

Description

Eigenvalues of Hermitian mlx arrays

Usage

```
mlx_eigvalsh(x, uplo = c("L", "U"), device = NULL)
```

Arguments

`x`	An mlx matrix (2-dimensional array).
`uplo`	Character string indicating which triangle to use ("L" or "U").
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

An mlx array containing eigenvalues.

See Also

mlx.linalg.eigvalsh

Examples

```
x <- mlx_matrix(c(2, 1, 1, 3), 2, 2)
mlx_eigvalsh(x, device = "cpu")
```

mlx_embedding *Embedding layer*

Description

Maps discrete tokens to continuous vectors.

Usage

```
mlx_embedding(num_embeddings, embedding_dim)
```

Arguments

num_embeddings	Size of vocabulary.
embedding_dim	Dimension of embedding vectors.

Value

An mlx_module for token embeddings.

See Also

mlx.nn.Embedding

Examples

```
set.seed(1)
emb <- mlx_embedding(num_embeddings = 100, embedding_dim = 16)
# Token indices (1-indexed)
tokens <- as_mlx(matrix(c(5, 10, 3, 7), 2, 2))
mlx_forward(emb, tokens)
```

mlx_erf *Error function and inverse error function*

Description

mlx_erf() computes the error function elementwise. mlx_erfinv() computes the inverse error function elementwise.

Usage

```
mlx_erf(x)

mlx_erfinv(x)
```

Arguments

x	An mlx array.

Value

An mlx array with the result.

See Also

mlx.core.erf, mlx.core.erfinv

Examples

```
x <- as_mlx(c(-1, 0, 1))
mlx_erf(x)
p <- as_mlx(c(-0.5, 0, 0.5))
mlx_erfinv(p)
```

---

mlx_eval *Force evaluation of an MLX operations*

Description

By default MLX computations are lazy. `mlx_eval(x)` forces the computations behind `x` to run. You can do the same by calling (e.g.) as.matrix(x).

Usage

```
mlx_eval(x)
```

Arguments

x	An mlx array.

Value

The input object, invisibly.

See Also

mlx.core.eval

Examples

```
system.time(x <- mlx_rand_normal(1e7))
system.time(mlx_eval(x))
```

mlx_expand_dims *Insert singleton dimensions*

Description

Insert singleton dimensions

Usage

```
mlx_expand_dims(x, axes)
```

Arguments

x	An mlx array.
axes	Integer vector of axis positions (1-indexed) where new singleton dimensions should be inserted.

Value

An mlx array with additional dimensions of length one.

See Also

mlx.core.expand_dims

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_expand_dims(x, axes = 1)
```

mlx_eye *Identity-like matrices on MLX devices*

Description

Identity-like matrices on MLX devices

Usage

```
mlx_eye(n, m = n, k = 0L, dtype = c("float32", "float64"))
```

Arguments

n	Number of rows.
m	Optional number of columns (defaults to n).
k	Diagonal index: 0 is the main diagonal, positive values shift upward, negative values shift downward.
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Details

MLX does not support float64 operations on GPU. When this function creates a float64 array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current mlx_device().

Value

An mlx matrix with ones on the selected diagonal and zeros elsewhere.

See Also

mlx.core.eye

Examples

```
mlx_eye(3)
mlx_eye(3, k = 1)
```

---

mlx_fft *Fast Fourier transforms for MLX arrays*

Description

mlx_fft(), mlx_fft2(), and mlx_fftn() wrap the MLX FFT kernels with R-friendly defaults. Inputs are converted with as_mlx() and results are returned as mlx arrays.

Usage

```
mlx_fft(x, axis, inverse = FALSE)

mlx_fft2(x, axes, inverse = FALSE)

mlx_fftn(x, axes = NULL, inverse = FALSE)
```

Arguments

x	Array-like object coercible to mlx.
axis	Optional integer axis (1-indexed) for the one-dimensional transform. Omit the argument to use the last dimension (no negative axes).
inverse	Logical flag; if TRUE, compute the inverse transform. The inverse is un-normalised to match base R's fft(), i.e. results are multiplied by the product of the transformed axis lengths.
axes	Optional integer vector of axes for the multi-dimensional transforms. Supply positive, 1-based axes; omit the argument to use the trailing axes (mlx_fft() defaults to the last axis, mlx_fft2() defaults to the last two axes, and mlx_fftn() defaults to all axes).

Value

An mlx array containing complex frequency coefficients.

See Also

fft(), mlx.fft

Examples

```
x <- as_mlx(c(1, 2, 3, 4))
mlx_fft(x)
mlx_fft(x, inverse = TRUE)
mat <- matrix(1:9, 3, 3)
mlx_fft2(as_mlx(mat))
arr <- mlx_array(1:8, dim = c(2, 2, 2))
mlx_fftn(arr)
```

---

mlx_flatten *Flatten axes of an mlx array*

Description

mlx_flatten() mirrors mlx.core.flatten(), collapsing a contiguous range of axes into a single dimension.

Usage

```
mlx_flatten(x, start_axis = 1L, end_axis = NULL)
```

Arguments

x	An mlx array.
start_axis	First axis (1-indexed) in the flattened range.
end_axis	Last axis (1-indexed) in the flattened range. Omit to use the final dimension.

Value

An mlx array with the selected axes collapsed.

See Also

mlx.core.flatten

Examples

```
x <- mlx_array(1:12, dim = c(2, 3, 2))
mlx_flatten(x)
mlx_flatten(x, start_axis = 2, end_axis = 3)
```

`mlx_forward` *Forward pass utility*

Description

Forward pass utility

Usage

```
mlx_forward(module, x)
```

Arguments

`module`	An `mlx_module`.
`x`	An mlx array.

Value

Output array.

See Also

mlx.nn.Module

Examples

```
set.seed(1)
layer <- mlx_linear(2, 1)
input <- as_mlx(matrix(c(1, 2), 1, 2))
mlx_forward(layer, input)
```

`mlx_full` *Fill an mlx array with a constant value*

Description

Fill an mlx array with a constant value

Usage

```
mlx_full(dim, value, dtype = NULL)
```

Arguments

`dim`	Integer vector specifying array dimensions (shape).
`value`	Scalar value used to fill the array. Numeric, logical, or complex.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An mlx array filled with the supplied value.

See Also

mlx.core.full

Examples

```
filled <- mlx_full(c(2, 2), 3.14)
complex_full <- mlx_full(c(2, 2), 1+2i, dtype = "complex64")
```

`mlx_gather` *Gather elements from an mlx array*

Description

Wraps `mlx.core.gather()` so you can pull elements by axis. Provide one index per axis. Axes must be positive integers (we don't allow negative indices, unlike Python).

Usage

```
mlx_gather(x, indices, axes = NULL)
```

Arguments

`x`	An mlx array.
`indices`	List of numeric/logical vectors or arrays (R or `mlx`). All entries must broadcast to a common shape.
`axes`	Integer vector of axes (1-indexed). Defaults to the first `length(indices)` axes.

Value

An `mlx` array containing the gathered elements.

Element-wise indexing

The output has the same shape as the indices (after broadcasting). Each element `[i, j, ...]`of the output is `x[index_1[i, j, ...], index_2[i, j, ...], ...]` from the corresponding position of each index. See the examples below.

Examples

```
x <- mlx_matrix(1:9, 3, 3)

# Simple cartesian gather:
mlx_gather(x, list(1:2, 1:2))

# Element-wise pairs: grab a custom 2x2 grid of coordinates
row_idx <- matrix(c(1, 1,
                    2, 3), nrow = 2, byrow = TRUE)
col_idx <- matrix(c(1, 3,
                    2, 2), nrow = 2, byrow = TRUE)

# A 2x2 matrix where (e.g.) the bottom right element is x[3, 2]
mlx_gather(x, list(row_idx, col_idx))
```

`mlx_gather_qmm` *Gather-based Quantized Matrix Multiplication*

Description

Performs quantized matrix multiplication with optional gather operations on inputs. This is useful for combining embedding lookups with quantized linear transformations, a common pattern in transformer models.

Usage

```
mlx_gather_qmm(
  x,
  w,
  scales,
  biases = NULL,
  lhs_indices = NULL,
  rhs_indices = NULL,
  transpose = TRUE,
  group_size = 64L,
  bits = 4L,
  mode = "affine",
  sorted_indices = FALSE
)
```

Arguments

`x`	An mlx array.
`w`	An mlx array representing the weight matrix. Accepts either an unquantized matrix (which may be quantized automatically) or a pre-quantized uint32 matrix produced by `mlx_quantize()`.
`scales`	An optional mlx array of quantization scales. Required when `w` is already quantized.
`biases`	An optional mlx array of quantization biases (affine mode); use `NULL` for symmetric quantization.
`lhs_indices`	An optional integer vector/array (1-indexed) or `mlx` tensor of indices for gathering from `x`'s leading (batch) dimension. Default: NULL
`rhs_indices`	An optional integer vector/array (1-indexed) or `mlx` tensor of indices for gathering from `w`'s leading (batch) dimension. Default: NULL
`transpose`	Whether to transpose the weight matrix before multiplication.
`group_size`	The group size for quantization. Smaller groups improve accuracy at the cost of slightly higher memory. Default: 64.
`bits`	Number of bits for quantization (typically 4 or 8). Default: 4.
`mode`	Quantization mode, either `"affine"` or `"mxfp4"`.
`sorted_indices`	Whether supplied indices are sorted (enables optimizations in gather-based kernels).

Details

This function combines gather operations (indexed lookups) with quantized matrix multiplication. When lhs_indices is provided, it performs `x[lhs_indices]` before the multiplication. Similarly, rhs_indices gathers from the weight matrix.

This is particularly efficient for transformer models where you need to look up token embeddings and then apply a quantized linear transformation in one fused operation.

Value

An mlx array with the result of the gather-based quantized matrix multiplication

See Also

`mlx_quantized_matmul()`

mlx.nn

---

mlx_gelu *GELU activation*

Description

Gaussian Error Linear Unit activation function.

Usage

```
mlx_gelu()
```

Value

An `mlx_module` applying GELU activation.

See Also

mlx.nn.GELU

Examples

```
act <- mlx_gelu()
x <- as_mlx(matrix(c(-2, -1, 0, 1, 2), 5, 1))
mlx_forward(act, x)
```

mlx_grad *Automatic differentiation for MLX functions*

Description

`mlx_grad()` computes gradients of an R function that operates on mlx arrays. The function must keep all differentiable computations in MLX (e.g., via `as_mlx()` and MLX operators) and return an mlx object.

Usage

```
mlx_grad(f, ..., argnums = NULL, value = FALSE)

mlx_value_grad(f, ..., argnums = NULL)
```

Arguments

`f`	An R function. Its arguments should be mlx objects, and its return value must be an mlx array (typically a scalar loss; a length-one vector is also OK).
`...`	Arguments to pass to `f`. They will be coerced to mlx if needed.
`argnums`	Indices (1-based) identifying which arguments to differentiate with respect to. Defaults to all arguments.
`value`	Should the function value be returned alongside gradients? Set to `TRUE` to receive a list with components `value` and `grads`.

Details

Keep the differentiated closure inside MLX operations. Coercing arrays back to base R objects (e.g. via `as.matrix()` or `[[` extraction) breaks the gradient tape and results in an error.

Value

When `value = FALSE` (default), a list of mlx arrays containing the gradients in the same order as `argnums`. When `value = TRUE`, a list with elements `value` (the function output as mlx) and `grads`.

See Also

mlx.core.grad, mlx.core.value_and_grad

Examples

```
loss <- function(w, x, y) {
  preds <- x %*% w
  resids <- preds - y
  sum(resids * resids) / length(y)
}
x <- mlx_matrix(1:8, 4, 2)
```

```
y <- mlx_matrix(c(1, 3, 2, 4), 4, 1)
w <- mlx_matrix(0, 2, 1)
mlx_grad(loss, w, x, y)[[1]]
loss <- function(w, x) sum((x %*% w) * (x %*% w))
x <- mlx_matrix(1:4, 2, 2)
w <- mlx_matrix(c(1, -1), 2, 1)
mlx_value_grad(loss, w, x)
```

`mlx_hadamard_transform` *Hadamard transform for MLX arrays*

Description

Multiplies the last dimension of `x` by the Sylvester-Hadamard matrix of the corresponding size. The transform expects the length of the last axis to be a power of two.

Usage

```
mlx_hadamard_transform(x, scale = NULL)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`scale`	Optional numeric scalar applied to the result. MLX defaults to `1 / sqrt(n)` where `n` is the size of the transformed axis; set `scale` to override the factor (for example, `scale = 1` yields the unnormalised Hadamard transform).

Value

An `mlx` array containing the Hadamard-transformed values.

See Also

https://ml-explore.github.io/mlx/build/html/python/array.html#mlx.core.hadamard_transform

Examples

```
x <- as_mlx(c(1, -1))
as.vector(mlx_hadamard_transform(x))
as.vector(mlx_hadamard_transform(x, scale = 1))
```

mlx_has_gpu *Check if GPU backend is available*

Description

Determines whether the GPU backend was compiled and is available.

Usage

```
mlx_has_gpu()
```

Value

Logical: TRUE if GPU is available, FALSE if only CPU.

Examples

```
if (mlx_has_gpu()) {
  mlx_synchronize("gpu")
} else {
  mlx_synchronize("cpu")
}
```

mlx_identity *Identity matrices on MLX devices*

Description

Identity matrices on MLX devices

Usage

```
mlx_identity(n, dtype = c("float32", "float64"))
```

Arguments

n	Size of the square matrix.
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An mlx identity matrix.

See Also

mlx.core.identity

Examples

```
I4 <- mlx_identity(4)
```

---

mlx_import_function *Import an exported MLX function*

Description

Loads a function previously exported with the MLX Python utilities and returns an R callable.

Usage

```
mlx_import_function(path)
```

Arguments

`path`	Path to a `.mlxfn` file created via MLX export utilities.

Details

Imported functions behave like regular R closures:

- Positional arguments are passed first and become the positional inputs the original MLX function expects.
- Named arguments (e.g. `bias = ...`) become MLX keyword arguments and must match the names that were used when exporting.
- Each argument is coerced to `mlx` via `as_mlx()`.
- If the MLX function yields a single array the result is returned as an `mlx` object; multiple outputs are returned as a list in the order MLX produced them.

Because `.mlxfn` files can bundle multiple traces (different shapes or keyword combinations), the imported callable keeps a varargs (`...`) signature. MLX selects the appropriate trace at runtime based on the shapes and keyword names you provide.

Value

An R function. Calling it returns an `mlx` array if the imported function has a single output, or a list of `mlx` arrays otherwise.

Examples

```
fixture_names <- c("add_matrix.mlxfn", "add_matrix_pre_metadata.mlxfn")
fixture_paths <- system.file("extdata", fixture_names, package = "Rmlx")
add_fn <- NULL
for (fixture_path in fixture_paths) {
  add_fn <- tryCatch(
    mlx_import_function(fixture_path),
    error = function(err) NULL
  )
  if (!is.null(add_fn)) break
}
stopifnot(!is.null(add_fn))
x <- mlx_matrix(1:4, 2, 2)
y <- mlx_matrix(5:8, 2, 2)
add_fn(x, y)
```

---

mlx_inv *Compute matrix inverse*

Description

Computes the inverse of a square matrix. Note that as of MLX 0.30.0, this runs on the CPU.

Usage

```
mlx_inv(x, device = NULL)
```

Arguments

`x`	An mlx array.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

The inverse of `x`.

See Also

mlx.core.linalg.inv

Examples

```
A <- mlx_matrix(c(4, 7, 2, 6), 2, 2)
A_inv <- mlx_inv(A, device = "cpu")
# Verify: A %*% A_inv should be identity
A %*% A_inv
```

`mlx_isclose` *Element-wise approximate equality*

Description

Returns a boolean array indicating which elements of two arrays are close within specified tolerances.

Usage

```
mlx_isclose(a, b, rtol = 1e-05, atol = 1e-08, equal_nan = FALSE)
```

Arguments

a, b	MLX arrays or objects coercible to MLX arrays
rtol	Relative tolerance (default: 1e-5)
atol	Absolute tolerance (default: 1e-8)
equal_nan	If TRUE, NaN values are considered equal (default: FALSE)

Details

Two values are considered close if: `abs(a - b) <= (atol + rtol * abs(b))`.

Infinite values with matching signs are considered equal. Supports NumPy-style broadcasting.

Value

An mlx array of booleans with element-wise comparison results

See Also

`mlx_allclose()`, `all.equal.mlx()`, mlx.core.isclose

Examples

```
a <- as_mlx(c(1.0, 2.0, 3.0))
b <- as_mlx(c(1.0 + 1e-6, 2.0 + 1e-6, 3.0 + 1e-3))
mlx_isclose(a, b)  # First two TRUE, last FALSE
```

mlx_isnan *Elementwise NaN and infinity predicates*

Description

`mlx_isnan()`, `mlx_isinf()`, and `mlx_isfinite()` wrap the corresponding MLX elementwise predicates, returning boolean arrays.

Usage

```
mlx_isnan(x)

mlx_isinf(x)

mlx_isfinite(x)
```

Arguments

`x`	An mlx array.

Value

An mlx boolean array.

See Also

mlx.core.isnan, mlx.core.isinf, mlx.core.isfinite

mlx_isposinf *Detect signed infinities in mlx arrays*

Description

`mlx_isposinf()` and `mlx_isneginf()` mirror `mlx.core.isposinf()` and `mlx.core.isneginf()`, returning boolean masks of positive or negative infinities.

Usage

```
mlx_isposinf(x)

mlx_isneginf(x)
```

Arguments

`x`	An mlx array.

Value

An mlx boolean array highlighting infinite entries.

See Also

mlx.core.isposinf, mlx.core.isneginf

Examples

```
vals <- as_mlx(c(-Inf, -1, 0, Inf))
mlx_isposinf(vals)
mlx_isneginf(vals)
```

---

mlx_key *Construct MLX random number generator keys*

Description

`mlx_key()` provides access to MLX's stateless PRNG. Given a 64-bit seed it returns a key that can be passed to other random helpers. Use `mlx_key_split()` to derive multiple independent keys from an existing key.

Usage

```
mlx_key(seed)

mlx_key_split(key, num = 2L)
```

Arguments

`seed`	Integer or numeric seed (converted to unsigned 64-bit).
`key`	An `mlx` key array returned by `mlx_key()`.
`num`	Number of subkeys to produce (default 2L).

Value

An `mlx` array holding the PRNG key.

A list of `num` `mlx` key arrays.

See Also

mlx.core.random.key

Examples

```
k <- mlx_key(42)
subkeys <- mlx_key_split(k, num = 2)
```

mlx_key_bits *Generate raw random bits on MLX arrays*

Description

Generate raw random bits on MLX arrays

Usage

```
mlx_key_bits(dim, width = 4L, key = NULL)
```

Arguments

dim	Integer vector specifying array dimensions (shape).
width	Number of bytes per element (default 4 = 32 bits). Must be positive.
key	Optional mlx key array. If omitted, MLX's default generator is used.

Value

An mlx array of unsigned integers filled with random bits.

See Also

mlx.core.random.bits

Examples

```
k <- mlx_key(12)
raw_bits <- mlx_key_bits(c(4, 4), key = k)
```

mlx_kron *Kronecker product for mlx arrays*

Description

Computes the Kronecker (tensor) product between two mlx arrays. Inputs are automatically cast to a common dtype before evaluation.

Usage

```
mlx_kron(a, b)
```

Arguments

a, b	Objects coercible to mlx.

Value

An `mlx` array representing the Kronecker product.

See Also

mlx.core.kron

Examples

```
A <- mlx_matrix(1:4, 2, 2)
B <- mlx_matrix(c(0, 5, 6, 7), 2, 2)
mlx_kron(A, B)
```

`mlx_l1_loss` *L1 loss (Mean Absolute Error)*

Description

Computes the mean absolute error between predictions and targets.

Usage

```
mlx_l1_loss(predictions, targets, reduction = c("mean", "sum", "none"))
```

Arguments

`predictions`	Predicted values as an mlx array.
`targets`	Target values as an mlx array.
`reduction`	Type of reduction: "mean" (default), "sum", or "none".

Value

An mlx array containing the loss.

See Also

mlx.nn.losses.l1_loss

Examples

```
preds <- mlx_matrix(c(1.5, 2.3, 0.8), 3, 1)
targets <- mlx_matrix(c(1, 2, 1), 3, 1)
mlx_l1_loss(preds, targets)
```

mlx_layer_norm *Layer normalization*

Description

Normalizes inputs across the feature dimension.

Usage

```
mlx_layer_norm(normalized_shape, eps = 1e-05)
```

Arguments

normalized_shape	Size of the feature dimension to normalize.
eps	Small constant for numerical stability (default: 1e-5).

Value

An `mlx_module` applying layer normalization.

See Also

mlx.nn.LayerNorm

Examples

```
set.seed(1)
ln <- mlx_layer_norm(4)
x <- as_mlx(matrix(rnorm(12), 3, 4))
mlx_forward(ln, x)
```

mlx_leaky_relu *Leaky ReLU activation*

Description

Leaky ReLU activation

Usage

```
mlx_leaky_relu(negative_slope = 0.01)
```

Arguments

negative_slope	Slope for negative values (default: 0.01).

Value

An `mlx_module` applying Leaky ReLU activation.

See Also

mlx.nn.LeakyReLU

Examples

```
act <- mlx_leaky_relu(negative_slope = 0.1)
x <- as_mlx(matrix(c(-2, -1, 0, 1, 2), 5, 1))
mlx_forward(act, x)
```

---

mlx_linear *Create a learnable linear transformation*

Description

Create a learnable linear transformation

Usage

```
mlx_linear(in_features, out_features, bias = TRUE)
```

Arguments

`in_features`	Number of input features.
`out_features`	Number of output features.
`bias`	Should a bias term be included?

Value

An object of class `mlx_module`.

See Also

mlx.nn.Linear

Examples

```
set.seed(1)
layer <- mlx_linear(3, 2)
x <- mlx_matrix(1:6, 2, 3)
mlx_forward(layer, x)
```

`mlx_linspace` *Evenly spaced ranges on MLX devices*

Description

`mlx_linspace()` creates `num` evenly spaced values from `start` to `stop`, inclusive. Unlike `mlx_arange()`, you specify how many samples you want rather than the step size.

Usage

```
mlx_linspace(start, stop, num = 50L, dtype = c("float32", "float64"))
```

Arguments

`start`	Starting value.
`stop`	Final value (inclusive).
`num`	Number of samples to generate.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

A 1D mlx array.

See Also

mlx.core.linspace

Examples

```
mlx_linspace(0, 1, num = 5)
```

mlx_load — *Load an MLX array from disk*

Description

Restores an array saved with `mlx_save()`.

Usage

```
mlx_load(file)
```

Arguments

`file`	Path to a `.npy` file. The extension is appended automatically when missing.

Value

An `mlx` array containing the file contents.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.load

mlx_load_gguf — *Load MLX tensors from the GGUF format*

Description

Load MLX tensors from the GGUF format

Usage

```
mlx_load_gguf(file)
```

Arguments

`file`	Path to a `.npy` file. The extension is appended automatically when missing.

Value

A list containing:

`tensors` Named list of `mlx` arrays.

`metadata` Named list where values are `NULL`, character vectors, or `mlx` arrays depending on the GGUF entry type.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.load_gguf

mlx_load_safetensors *Load MLX arrays from the safetensors format*

Description

Load MLX arrays from the safetensors format

Usage

```
mlx_load_safetensors(file)
```

Arguments

file	Path to a .npy file. The extension is appended automatically when missing.

Value

A list containing:

tensors Named list of mlx arrays.

metadata Named character vector with the serialized metadata.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.load_safetensors

mlx_logcumsumexp *Log cumulative sum exponential for mlx arrays*

Description

Log cumulative sum exponential for mlx arrays

Usage

```
mlx_logcumsumexp(x, axis = NULL, reverse = FALSE, inclusive = TRUE)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
axis	Optional axis (single integer) to operate over.
reverse	Logical flag for reverse accumulation.
inclusive	Logical flag controlling inclusivity.

Value

An mlx array.

See Also

mlx.core.logaddexp

Examples

```
x <- as_mlx(1:4)
mlx_logcumsumexp(x)
m <- mlx_matrix(1:6, 2, 3)
mlx_logcumsumexp(m, axis = 2)
```

`mlx_logsumexp` *Log-sum-exp reduction for mlx arrays*

Description

Log-sum-exp reduction for mlx arrays

Usage

```
mlx_logsumexp(x, axes = NULL, drop = TRUE)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`axes`	Integer vector of axes (1-indexed). Supply positive integers between 1 and the array rank. Many helpers interpret `NULL` to mean "all axes"—see the function details for specifics.
`drop`	Logical indicating whether the reduced axes should be dropped (default `TRUE`).

Value

An mlx array containing log-sum-exp results.

See Also

mlx.core.logsumexp

Examples

```
x <- mlx_matrix(1:6, 2, 3)
mlx_logsumexp(x)
mlx_logsumexp(x, axes = 2)
```

`mlx_lu` *LU factorization*

Description

Computes the LU factorization of a matrix.

Usage

```
mlx_lu(x, device = NULL)
```

Arguments

`x`	An mlx array.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

A list with components `p` (pivot indices), `l` (lower triangular), and `u` (upper triangular). The relationship is `A = L[P, ] %*% U`.

See Also

mlx.core.linalg.lu

Examples

```
A <- mlx_matrix(rnorm(16), 4, 4)
lu_result <- mlx_lu(A, device = "cpu")
P <- lu_result$p  # Pivot indices
L <- lu_result$l  # Lower triangular
U <- lu_result$u  # Upper triangular
```

mlx_matrix *Construct MLX matrices efficiently*

Description

`mlx_matrix()` wraps `mlx_array()` for the common 2-D case. It accepts the same style arguments as `base::matrix()` but without recycling, so mistakes surface early. Supply `nrow` or `ncol` (the other may be inferred from `length(data)`).

Usage

```
mlx_matrix(
  data,
  nrow = NULL,
  ncol = NULL,
  byrow = FALSE,
  dtype = NULL,
  dimnames = NULL
)
```

Arguments

`data`	Numeric, logical, or complex vector. `data` is recycled to match dimensions according to R rules (but with an error if it doesn't tile into the dimensions exactly).
`nrow`, `ncol`	Matrix dimensions (positive integers).
`byrow`	Logical; if `TRUE`, fill by rows (same semantics as `base::matrix()`).
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.
`dimnames`	Optional list of character vectors naming each dimension.

Value

An `mlx` matrix with `dim = c(nrow, ncol)`.

Examples

```
mlx_matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

mlx_maximum *Elementwise maximum of two mlx arrays*

Description

Elementwise maximum of two mlx arrays

Usage

```
mlx_maximum(x, y)
```

Arguments

x, y	mlx arrays or objects coercible with as_mlx().

Value

An mlx array containing the elementwise maximum.

See Also

mlx.core.maximum

Examples

```
mlx_maximum(1:3, c(3, 2, 1))
```

mlx_meshgrid *Construct coordinate arrays from input vectors*

Description

mlx_meshgrid() mirrors mlx.core.meshgrid(), returning coordinate arrays suitable for vectorised evaluation on MLX devices.

Usage

```
mlx_meshgrid(..., sparse = FALSE, indexing = c("xy", "ij"))
```

Arguments

...	One or more arrays (or a single list of arrays) coercible via as_mlx().
sparse	Logical flag producing broadcast-friendly outputs when TRUE.
indexing	Either "xy" (Cartesian) or "ij" (matrix) indexing.

Value

A list of mlx arrays matching the number of inputs.

See Also

mlx.core.meshgrid

Examples

```
xs <- as_mlx(1:3)
ys <- as_mlx(1:2)
mlx_meshgrid(xs, ys, indexing = "xy")
```

`mlx_metal_kernel` *Create a JIT-compiled custom Metal kernel*

Description

Wraps MLX's Metal kernel API so R code can define custom GPU kernels while keeping inputs and outputs as `mlx` arrays.

Usage

```
mlx_metal_kernel(
  name,
  input_names,
  output_names,
  source,
  header = "",
  ensure_row_contiguous = TRUE,
  atomic_outputs = FALSE
)
```

Arguments

`name`	Kernel name used in generated Metal code.
`input_names`	Character vector naming the kernel inputs.
`output_names`	Character vector naming the kernel outputs.
`source`	Metal source for the kernel body. MLX generates the function signature automatically.
`header`	Optional Metal source prepended before the generated function.
`ensure_row_contiguous`	Logical. Should MLX make inputs row-contiguous before launching the kernel?
`atomic_outputs`	Logical. Should output buffers use Metal atomic types?

Value

A function that executes the compiled kernel and returns one `mlx` array for a single output or a named list of `mlx` arrays otherwise.

See Also

`mlx_compile()`

mlx.core.fast.metal_kernel

Custom Metal Kernels

Examples

```
## Not run:
add_one <- mlx_metal_kernel(
  name = "add_one",
  input_names = "inp",
  output_names = "out",
  source = "
    uint elem = thread_position_in_grid.x;
    out[elem] = inp[elem] + (T)1;
  "
)

x <- mlx_cast(as_mlx(1:8), "float32")
y <- add_one(
  inputs = list(x),
  output_shapes = list(c(length(x))),
  output_dtypes = "float32",
  grid = c(length(x), 1L, 1L),
  threadgroup = c(length(x), 1L, 1L),
  template = list(T = "float32")
)

## End(Not run)
```

`mlx_minimum` *Elementwise minimum of two mlx arrays*

Description

Elementwise minimum of two mlx arrays

Usage

```
mlx_minimum(x, y)
```

Arguments

`x, y`	mlx arrays or objects coercible with `as_mlx()`.

Value

An mlx array containing the elementwise minimum.

See Also

mlx.core.minimum

Examples

```
a <- mlx_matrix(1:4, 2, 2)
b <- as_mlx(matrix(c(4, 3, 2, 1), 2, 2))
mlx_minimum(a, b)
```

`mlx_moveaxis` *Reorder mlx array axes*

Description

- `mlx_moveaxis()` repositions one or more axes to new locations.
- `aperm.mlx()` provides the familiar R interface, permuting axes according to `perm` via repeated calls to `mlx_moveaxis()`.

Usage

```
mlx_moveaxis(x, source, destination)

## S3 method for class 'mlx'
aperm(a, perm = NULL, resize = TRUE, ...)
```

Arguments

`x`, `a`	An object coercible to mlx via `as_mlx()`.
`source`	Integer vector of axis indices to move (1-indexed).
`destination`	Integer vector giving the target positions for `source` axes (1-indexed). Must be the same length as `source`.
`perm`	Integer permutation describing the desired axis order, matching the semantics of `base::aperm()`.
`resize`	Logical flag from `base::aperm()`. Only `TRUE` is currently supported for mlx arrays.
`...`	Additional arguments; ignored.

Value

An mlx array with axes permuted.

See Also

mlx.core.moveaxis

Examples

```
x <- mlx_array(1:8, dim = c(2, 2, 2))
moved <- mlx_moveaxis(x, source = 1, destination = 3)
permuted <- aperm(x, c(2, 1, 3))
```

---

mlx_mse_loss *Mean squared error loss*

Description

Computes the mean squared error between predictions and targets.

Usage

```
mlx_mse_loss(predictions, targets, reduction = c("mean", "sum", "none"))
```

Arguments

predictions	Predicted values as an mlx array.
targets	Target values as an mlx array.
reduction	Type of reduction: "mean" (default), "sum", or "none".

Value

An mlx array containing the loss.

See Also

mlx.nn.losses.mse_loss

Examples

```
preds <- mlx_matrix(c(1.5, 2.3, 0.8), 3, 1)
targets <- mlx_matrix(c(1, 2, 1), 3, 1)
mlx_mse_loss(preds, targets)
```

mlx_nan_to_num *Replace NaN and infinite values with finite numbers*

Description

`mlx_nan_to_num()` mirrors `mlx.core.nan_to_num()`, filling non-finite entries with user-provided finite substitutes.

Usage

```
mlx_nan_to_num(x, nan = 0, posinf = NULL, neginf = NULL)
```

Arguments

`x`	An mlx array.
`nan`	Replacement for NaN values (default `0`). Use `NULL` to retain MLX's default.
`posinf`	Optional replacement for positive infinity.
`neginf`	Optional replacement for negative infinity.

Value

An mlx array with non-finite values replaced.

See Also

mlx.core.nan_to_num

Examples

```
x <- as_mlx(c(-Inf, -1, NaN, 3, Inf))
mlx_nan_to_num(x, nan = 0, posinf = 10, neginf = -10)
```

mlx_new_stream *MLX streams for asynchronous execution*

Description

Streams provide independent execution queues on a device, allowing overlap of computation and finer control over scheduling.

`mlx_default_stream()` returns the current default stream for a device.

Usage

```
mlx_new_stream(device = mlx_device())

mlx_default_stream(device = mlx_device())
```

Arguments

device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Value

An object of class mlx_stream.

See Also

https://ml-explore.github.io/mlx/build/html/usage/using_streams.html

Examples

```
stream <- mlx_new_stream()
stream
```

---

mlx_norm *Matrix and vector norms for mlx arrays*

Description

Matrix and vector norms for mlx arrays

Usage

```
mlx_norm(x, ord = NULL, axes = NULL, drop = TRUE)
```

Arguments

x	An mlx array.
ord	Numeric or character norm order. Use NULL for the default 2-norm.
axes	Integer vector of axes (1-indexed). Supply positive integers between 1 and the array rank. Many helpers interpret NULL to mean "all axes"—see the function details for specifics.
drop	If TRUE (default), drop dimensions of length 1. If FALSE, retain all dimensions. Equivalent to keepdims = TRUE in underlying mlx functions.

Value

An mlx array containing the requested norm.

See Also

mlx.linalg.norm

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_norm(x)
mlx_norm(x, ord = 2)
mlx_norm(x, axes = 2)
```

---

mlx_ones *Create arrays of ones on MLX devices*

Description

Create arrays of ones on MLX devices

Usage

```
mlx_ones(
  dim,
 dtype = c("float32", "float64", "int8", "int16", "int32", "int64", "uint8", "uint16",
    "uint32", "uint64", "bool", "complex64")
)
```

Arguments

dim	Integer vector specifying array dimensions (shape).
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array filled with ones.

See Also

mlx.core.ones

Examples

```
ones <- with_device("cpu", mlx_ones(c(2, 2), dtype = "float64"))
ones_int <- mlx_ones(c(3, 3), dtype = "int32")
```

`mlx_ones_like` *Ones shaped like an existing mlx array*

Description

`mlx_ones_like()` mirrors `mlx.core.ones_like()`, creating an array of ones with the same shape. Optionally override dtype.

Usage

```
mlx_ones_like(x, dtype = NULL)
```

Arguments

`x`	An mlx array.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An mlx array of ones matching `x`.

See Also

mlx.core.ones_like

Examples

```
base <- mlx_full(c(2, 3), 5)
mlx_ones_like(base)
```

mlx_optimizer_sgd *Stochastic gradient descent optimizer*

Description

Stochastic gradient descent optimizer

Usage

```
mlx_optimizer_sgd(params, lr = 0.01)
```

Arguments

params	List of parameters (from mlx_parameters()).
lr	Learning rate.

Value

An optimizer object with a step() method.

See Also

mlx.optimizers.SGD

Examples

```
set.seed(1)
model <- mlx_linear(2, 1, bias = FALSE)
opt <- mlx_optimizer_sgd(mlx_parameters(model), lr = 0.1)
```

mlx_pad *Pad mlx arrays*

Description

mlx_pad() mirrors the MLX padding primitive, enlarging each axis according to pad_width. Values are added symmetrically (pad_width[i, 1] before, pad_width[i, 2] after) using the specified mode.

Usage

```
mlx_pad(
  x,
  pad_width,
  value = 0,
  mode = c("constant", "edge", "reflect", "symmetric"),
  axes = NULL
)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
pad_width	Padding extents. Supply a single integer, a length-two numeric vector, or a matrix/list with one (before, after) pair per padded axis.
value	Constant fill value used when mode = "constant".
mode	Padding mode passed to MLX (e.g., "constant", "edge", "reflect").
axes	Optional integer vector of axes (1-indexed) to which pad_width applies. Unlisted axes receive zero padding.

Value

An mlx array with the requested padding applied. Named axes are extended according to the padding mode.

See Also

mlx.core.pad, mlx_split()

Examples

```
x <- mlx_matrix(1:4, 2, 2)
padded <- mlx_pad(x, pad_width = 1)
padded_cols <- mlx_pad(x, pad_width = c(0, 1), axes = 2)
```

---

mlx_param_set_values *Assign arrays back to parameters*

Description

Assign arrays back to parameters

Usage

```
mlx_param_set_values(params, values)
```

Arguments

params	A list of mlx_param.
values	A list of arrays.

See Also

mlx.nn.Module.update

Examples

```
set.seed(1)
layer <- mlx_linear(2, 1)
params <- mlx_parameters(layer)
current <- mlx_param_values(params)
mlx_param_set_values(params, current)
```

`mlx_param_values` *Retrieve parameter arrays*

Description

Retrieve parameter arrays

Usage

```
mlx_param_values(params)
```

Arguments

`params`	A list of `mlx_param`.

Value

List of mlx arrays.

See Also

mlx.nn.Module.parameters

Examples

```
set.seed(1)
layer <- mlx_linear(2, 1)
vals <- mlx_param_values(mlx_parameters(layer))
```

mlx_parameters *Collect parameters from modules*

Description

Collect parameters from modules

Usage

```
mlx_parameters(module)
```

Arguments

module	An mlx_module or list of modules.

Value

A list of mlx_param objects.

See Also

mlx.nn.Module.parameters

Examples

```
set.seed(1)
layer <- mlx_linear(2, 1)
mlx_parameters(layer)
```

mlx_put_along_axis *Write values using per-position axis indices*

Description

Mirrors mlx.core.put_along_axis() while accepting 1-based R indices.

Usage

```
mlx_put_along_axis(x, indices, values, axis)
```

Arguments

x	An mlx array.
indices	Integer positions along axis. Must be broadcast-compatible with x except at the selected axis.
values	Replacement values.
axis	Axis to index (1-based).

Value

An updated `mlx` array.

Examples

```
x <- mlx_matrix(1:12, nrow = 3, ncol = 4)
idx <- matrix(c(1L, 4L,
                2L, 3L,
                4L, 1L), nrow = 3, byrow = TRUE)
values <- matrix(c(100, 200,
                   300, 400,
                   500, 600), nrow = 3, byrow = TRUE)
mlx_put_along_axis(x, idx, values, axis = 2L)
```

`mlx_qr_gpu` *GPU QR reduction for tall least-squares problems*

Description

Computes the QR quantities needed by large least-squares fits without materializing the full `Q` matrix. This is intended for tall, full-rank real-valued matrices. It returns the final upper-triangular `R` and, when `y` is supplied, `qty = Q' y`.

Usage

```
mlx_qr_gpu(
  x,
  y = NULL,
  block_rows = NULL,
  tol = 1e-04,
  method = c("cholqr", "tsqr")
)
```

Arguments

`x`	An mlx matrix (2-dimensional array).
`y`	Optional response vector or matrix with `nrow(x)` rows.
`block_rows`	Number of rows reduced by each first-level GPU block for `method = "tsqr"`. The default `NULL` chooses a GPU tile size automatically.
`tol`	Relative tolerance for detecting rank deficiency from `diag(R)`.
`method`	`"cholqr"` for two Cholesky QR passes or `"tsqr"` for the custom Metal tall-skinny QR reduction.

Details

The default "cholqr" method applies two Cholesky QR passes, with the large matrix products on the GPU. The "tsqr" method uses custom Metal kernels for a tiled Householder reduction followed by a tree reduction of the small triangular factors.

GPU work is currently restricted to float32. Integer inputs are cast to float32; float64 and complex inputs are not supported on the GPU path.

method = "tsqr" stores one input tile in Metal threadgroup memory. When block_rows = NULL, the tile height is chosen from p and ncol(y) to fit the 32 KB threadgroup-memory limit and provide enough independent blocks to occupy the GPU.

Cholesky QR checks the orthogonality of its first pass. If that pass is unsafe, it falls back to GPU TSQR when its compact state fits in threadgroup memory, and otherwise to MLX QR on the CPU. For well-conditioned fits with y, an MLX GPU residual-correction pass returns qty_corrected for a more accurate least-squares solve while preserving qty = Q' y.

Value

A list with components R, optional qty, rank, pivot, and block_rows. Well-conditioned Cholesky QR fits with y also return qty_corrected for the coefficient solve.

Examples

```
## Not run:
x <- as_mlx(matrix(rnorm(1000 * 8), 1000, 8))
y <- as_mlx(matrix(rnorm(1000), 1000, 1))
fit <- mlx_qr_gpu(x, y)
coef <- mlx_solve_triangular(fit$R, fit$qty, upper = TRUE, device = "cpu")

## End(Not run)
```

---

mlx_quantile *Compute quantiles of MLX arrays*

Description

Calculates sample quantiles corresponding to given probabilities using linear interpolation (R's type 7 quantiles, the default in stats::quantile()). The S3 method quantile.mlx() provides an interface compatible with the generic stats::quantile().

Usage

```
mlx_quantile(x, probs, axis = NULL, drop = FALSE)

## S3 method for class 'mlx'
quantile(x, probs, ...)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
probs	Numeric vector of probabilities in [0, 1].
axis	Optional integer axis (or vector of axes) along which to compute quantiles. When NULL (default), quantiles are computed over the entire flattened array.
drop	Logical; when TRUE and computing quantiles along an axis with a single probability, removes the quantile dimension of length 1. Defaults to FALSE to match the behavior of other reduction functions.
...	Additional arguments; ignored.

Details

Uses type 7 quantiles (linear interpolation): for probability p and n observations, the quantile is computed as:

- h = (n-1) * p
- Interpolate between floor(h) and ceiling(h)

This matches the default behavior of stats::quantile().

Value

An mlx array containing the requested quantiles. The shape depends on probs, axis, and drop: when axis = NULL, returns a scalar for a single probability or a vector for multiple probabilities. When axis is specified, the quantile dimension replaces the reduced axis (e.g., a (3, 4) matrix with axis = 1 and 2 quantiles gives (2, 4)), unless drop = TRUE with a single probability removes that dimension.

See Also

stats::quantile(), mlx.core.sort

Examples

```
x <- as_mlx(1:10)
mlx_quantile(x, 0.5)  # median
mlx_quantile(x, c(0.25, 0.5, 0.75))  # quartiles

# S3 method:
quantile(x, probs = c(0, 0.25, 0.5, 0.75, 1))

# With axis parameter, quantile dimension replaces the reduced axis:
mat <- mlx_matrix(1:12, 3, 4)  # shape (3, 4)
result <- mlx_quantile(mat, c(0.25, 0.75), axis = 1)  # shape (2, 4)
result <- mlx_quantile(mat, 0.5, axis = 1)  # shape (1, 4)
result <- mlx_quantile(mat, 0.5, axis = 1, drop = TRUE)  # shape (4)
```

`mlx_quantize` *Quantize a Matrix*

Description

Quantizes a weight matrix to low-precision representation (typically 4-bit or 8-bit). This reduces memory usage and enables faster computation during inference.

Usage

```
mlx_quantize(w, group_size = 64L, bits = 4L, mode = "affine")
```

Arguments

`w`	An mlx array representing the weight matrix. Accepts either an unquantized matrix (which may be quantized automatically) or a pre-quantized uint32 matrix produced by `mlx_quantize()`.
`group_size`	The group size for quantization. Smaller groups improve accuracy at the cost of slightly higher memory. Default: 64.
`bits`	Number of bits for quantization (typically 4 or 8). Default: 4.
`mode`	Quantization mode, either `"affine"` or `"mxfp4"`.

Details

Quantization converts floating-point weights to low-precision integers, reducing memory by up to 8x for 4-bit quantization. The scales (and optionally biases) are stored to enable approximate reconstruction of the original values.

Value

A list containing:

`w_q`	The quantized weight matrix (packed as uint32)
`scales`	The quantization scales for dequantization
`biases`	The quantization biases (NULL for symmetric mode)

See Also

`mlx_dequantize()`, `mlx_quantized_matmul()`

Examples

```
w <- mlx_rand_normal(c(64, 32))
quant <- mlx_quantize(w, group_size = 32, bits = 4)
# Use quant$w_q, quant$scales, quant$biases with mlx_quantized_matmul()
```

`mlx_quantized_matmul` *Quantized Matrix Multiplication*

Description

Performs matrix multiplication with a quantized weight matrix. This operation is essential for efficient inference with quantized models, significantly reducing memory usage and computation time while maintaining reasonable accuracy.

Usage

```
mlx_quantized_matmul(
  x,
  w,
  scales = NULL,
  biases = NULL,
  transpose = TRUE,
  group_size = 64L,
  bits = 4L,
  mode = "affine"
)
```

Arguments

`x`	An mlx array.
`w`	An mlx array representing the weight matrix. Accepts either an unquantized matrix (which may be quantized automatically) or a pre-quantized uint32 matrix produced by `mlx_quantize()`.
`scales`	An optional mlx array of quantization scales. Required when `w` is already quantized.
`biases`	An optional mlx array of quantization biases (affine mode); use `NULL` for symmetric quantization.
`transpose`	Whether to transpose the weight matrix before multiplication.
`group_size`	The group size for quantization. Smaller groups improve accuracy at the cost of slightly higher memory. Default: 64.
`bits`	Number of bits for quantization (typically 4 or 8). Default: 4.
`mode`	Quantization mode, either `"affine"` or `"mxfp4"`.

Details

Quantized matrix multiplication uses low-precision representations (typically 4-bit or 8-bit integers) for weights, which reduces memory footprint by up to 8x compared to float32. The scales parameter contains the dequantization factors needed to reconstruct approximate float values during computation.

The group_size parameter controls the granularity of quantization - smaller groups provide better accuracy but slightly higher memory usage.

Automatic Quantization: If only w is provided (without scales), the function will automatically quantize w using `mlx_quantize()` before performing the multiplication. For repeated operations, it's more efficient to pre-quantize weights once using `mlx_quantize()` and reuse them.

Value

An mlx array with the result of the quantized matrix multiplication

See Also

`mlx_quantize()`, `mlx_dequantize()`, `mlx_gather_qmm()`

Examples

```
# Automatic quantization (convenient but slower for repeated use)
x <- mlx_rand_normal(c(4, 64))
w <- mlx_rand_normal(c(128, 64))
result <- mlx_quantized_matmul(x, w, group_size = 32)

# Pre-quantized weights (faster for repeated operations)
quant <- mlx_quantize(w, group_size = 32, bits = 4)
result <- mlx_quantized_matmul(x, quant$w_q, quant$scales, quant$biases, group_size = 32)
```

---

mlx_rand_bernoulli *Sample Bernoulli random variables on mlx arrays*

Description

Sample Bernoulli random variables on mlx arrays

Usage

```
mlx_rand_bernoulli(dim, prob = 0.5)
```

Arguments

dim	Integer vector specifying array dimensions (shape).
prob	Probability of a one.

Value

An mlx boolean array.

See Also

mlx.core.random.bernoulli

Examples

```
mask <- mlx_rand_bernoulli(c(4, 4), prob = 0.3)
```

`mlx_rand_categorical` *Sample from a categorical distribution on mlx arrays*

Description

Samples indices from categorical distributions. Each row (or slice along the specified axis) represents a separate categorical distribution over classes.

Usage

```
mlx_rand_categorical(logits, axis = NULL, num_samples = 1L)
```

Arguments

`logits`	A matrix or mlx array of log-probabilities. The values don't need to be normalized (the function applies softmax internally). For a single distribution over K classes, use a 1×K matrix. For multiple independent distributions, use an N×K matrix where each row is a distribution.
`axis`	Axis (1-indexed) along which to sample. Omit the argument to use the last dimension (typically the class dimension).
`num_samples`	Number of samples to draw from each distribution.

Value

An mlx array of integer indices (1-indexed) sampled from the categorical distributions.

See Also

mlx.core.random.categorical

Examples

```
# Single distribution over 3 classes
logits <- matrix(c(0.5, 0.2, 0.3), 1, 3)
samples <- mlx_rand_categorical(logits, num_samples = 10)

# Multiple distributions
logits <- matrix(c(1, 2, 3,
                   3, 2, 1), nrow = 2, byrow = TRUE)
samples <- mlx_rand_categorical(logits, num_samples = 5)
```

mlx_rand_gumbel *Sample from the Gumbel distribution on mlx arrays*

Description

Sample from the Gumbel distribution on mlx arrays

Usage

```
mlx_rand_gumbel(dim, dtype = c("float32", "float64"))
```

Arguments

dim	Integer vector specifying array dimensions (shape).
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array with Gumbel-distributed entries.

See Also

mlx.core.random.gumbel

Examples

```
samples <- mlx_rand_gumbel(c(2, 3))
```

mlx_rand_laplace *Sample from the Laplace distribution on mlx arrays*

Description

Sample from the Laplace distribution on mlx arrays

Usage

```
mlx_rand_laplace(dim, loc = 0, scale = 1, dtype = c("float32", "float64"))
```

Arguments

dim	Integer vector specifying array dimensions (shape).
loc	Location parameter (mean) of the Laplace distribution.
scale	Scale parameter (diversity) of the Laplace distribution.
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array with Laplace-distributed entries.

See Also

mlx.core.random.laplace

Examples

```
samples <- mlx_rand_laplace(c(2, 3), loc = 0, scale = 1)
```

---

mlx_rand_multivariate_normal

Sample from a multivariate normal distribution on mlx arrays

Description

Sample from a multivariate normal distribution on mlx arrays

Usage

```
mlx_rand_multivariate_normal(
  dim,
  mean,
  cov,
  dtype = c("float32", "float64"),
  device = "cpu"
)
```

Arguments

dim	Integer vector specifying array dimensions (shape).
mean	An mlx array or vector for the mean.
cov	An mlx array or matrix for the covariance.
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.
device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Details

GPU execution is currently unavailable because the covariance factorisation runs on the host.

Value

An mlx array with samples from the multivariate normal.

See Also

mlx.core.random.multivariate_normal

Examples

```
mean <- as_mlx(c(0, 0))
cov <- as_mlx(matrix(c(1, 0, 0, 1), 2, 2))
samples <- with_device("cpu", mlx_rand_multivariate_normal(10, mean, cov))
```

---

mlx_rand_normal *Sample from a normal distribution on mlx arrays*

Description

Sample from a normal distribution on mlx arrays

Usage

```
mlx_rand_normal(dim, mean = 0, sd = 1, dtype = c("float32", "float64"))
```

Arguments

dim	Integer vector specifying array dimensions (shape).
mean	Mean of the normal distribution.
sd	Standard deviation of the normal distribution.
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array with normally distributed entries.

See Also

mlx.core.random.normal

Examples

```
weights <- mlx_rand_normal(c(3, 3), mean = 0, sd = 0.1)
```

---

mlx_rand_permutation *Generate random permutations on mlx arrays*

Description

Generate a random permutation of integers or permute the entries of an array along a specified axis.

Usage

```
mlx_rand_permutation(x, axis = 1L)
```

Arguments

x	Either an integer n (to generate a permutation of 1:n), or an mlx array or matrix to permute.
axis	Axis (1-indexed) along which to permute when x is an array. Default is 1L (permute rows).

Value

An mlx array containing the random permutation.

See Also

mlx.core.random.permutation

Examples

```
# Generate a random permutation of 1:10
perm <- mlx_rand_permutation(10)

# Permute the rows of a matrix
mat <- matrix(1:12, 4, 3)
perm_mat <- mlx_rand_permutation(mat)

# Permute columns instead
perm_cols <- mlx_rand_permutation(mat, axis = 2)
```

---

mlx_rand_randint *Sample random integers on mlx arrays*

Description

Generates random integers uniformly distributed over the interval [low, high).

Usage

```
mlx_rand_randint(
  dim,
  low,
  high,
  dtype = c("int32", "int64", "uint32", "uint64")
)
```

Arguments

dim	Integer vector specifying array dimensions (shape).
low	Lower bound (inclusive).
high	Upper bound (exclusive).
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array of random integers.

See Also

mlx.core.random.randint

Examples

```
# Random integers from 0 to 9
samples <- mlx_rand_randint(c(3, 3), low = 0, high = 10)

# Random integers from -5 to 4
samples <- mlx_rand_randint(c(2, 5), low = -5, high = 5)
```

---

mlx_rand_truncated_normal

Sample from a truncated normal distribution on mlx arrays

Description

Sample from a truncated normal distribution on mlx arrays

Usage

```
mlx_rand_truncated_normal(lower, upper, dim, dtype = c("float32", "float64"))
```

Arguments

lower	Lower bound of the truncated normal.
upper	Upper bound of the truncated normal.
dim	Integer vector specifying array dimensions (shape).
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx array with truncated normally distributed entries.

See Also

mlx.core.random.truncated_normal

Examples

```
samples <- mlx_rand_truncated_normal(-1, 1, c(5, 5))
```

`mlx_rand_uniform` *Sample from a uniform distribution on mlx arrays*

Description

Sample from a uniform distribution on mlx arrays

Usage

```
mlx_rand_uniform(dim, min = 0, max = 1, dtype = c("float32", "float64"))
```

Arguments

`dim`	Integer vector specifying array dimensions (shape).
`min`	Lower bound for the uniform distribution.
`max`	Upper bound for the uniform distribution.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Value

An mlx array whose entries are sampled uniformly.

See Also

mlx.core.random.uniform

Examples

```
noise <- mlx_rand_uniform(c(2, 2), min = -1, max = 1)
```

mlx_real *Complex-valued helpers for mlx arrays*

Description

`mlx_real()`, `mlx_imag()`, and `mlx_conjugate()` expose MLX's complex helpers to extract the real part, imaginary part, or complex conjugate of an `mlx` array. Corresponding S3 methods for `Re()`, `Im()`, and `Conj()` are also provided.

Usage

```
mlx_real(x)

mlx_imag(x)

mlx_conjugate(x)
```

Arguments

x	An mlx array.

Value

An `mlx` array containing the requested component.

See Also

mlx.core.array

Examples

```
z <- as_mlx(1:4 + 1i * (4:1))
mlx_real(z)
Im(z)
```

mlx_relu *Rectified linear activation module*

Description

Rectified linear activation module

Usage

```
mlx_relu()
```

Value

An `mlx_module` applying ReLU.

See Also

mlx.nn.ReLU

Examples

```
act <- mlx_relu()
x <- as_mlx(matrix(c(-1, 0, 2), 3, 1))
mlx_forward(act, x)
```

`mlx_repeat` *Repeat array elements*

Description

Repeat array elements

Usage

```
mlx_repeat(x, repeats, axis = NULL)
```

Arguments

`x`	An mlx array.
`repeats`	Number of repetitions.
`axis`	Optional axis along which to repeat. When `NULL`, the array is flattened before repetition (matching NumPy semantics).

Value

An mlx array with repeated values. Dimnames are repeated on the selected axis when they still describe the result.

See Also

mlx.core.repeat

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_repeat(x, repeats = 2, axis = 2)
```

mlx_reshape *Reshape an mlx array*

Description

Reshape an mlx array

Usage

```
mlx_reshape(x, newshape)
```

Arguments

x	An mlx array.
newshape	Integer vector specifying the new dimensions.

Value

An mlx array with the specified shape.

See Also

mlx.core.reshape

Examples

```
x <- as_mlx(1:12)
mlx_reshape(x, c(3, 4))
mlx_reshape(x, c(2, 6))
```

mlx_roll *Roll array elements*

Description

Roll array elements

Usage

```
mlx_roll(x, shift, axes = NULL)
```

Arguments

x	An mlx array.
shift	Integer vector giving the number of places by which elements are shifted.
axes	Optional integer vector (1-indexed) along which elements are shifted. When NULL, the array is flattened and shifted, then the shape is restored.

Value

An mlx array with elements circularly shifted. Dimnames are rolled with explicit axes; flattening rolls only keep names for vectors.

See Also

mlx.core.roll

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_roll(x, shift = 1, axes = 2)
```

---

mlx_save *Save an MLX array to disk*

Description

Persists an MLX array to a `.npy` file using MLX's native serialization.

Usage

```
mlx_save(x, file)
```

Arguments

`x`	Object coercible to `mlx`.
`file`	Path to the output file. If the file does not end with `.npy`, the extension is appended automatically.

Value

Invisibly returns the full path that was written, including the `.npy` suffix.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.save

Examples

```
path <- tempfile(fileext = ".mlx")
mlx_save(as_mlx(matrix(1:4, 2, 2)), path)
restored <- mlx_load(path)
```

mlx_save_gguf *Save MLX arrays to the GGUF format*

Description

Save MLX arrays to the GGUF format

Usage

```
mlx_save_gguf(file, arrays, metadata = list())
```

Arguments

file	Output path. .safetensors is appended automatically when omitted.
arrays	Named list of objects coercible to mlx.
metadata	Optional named list describing GGUF metadata. Values may be character vectors or mlx arrays.

Value

Invisibly returns the full path that was written.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.save_gguf

mlx_save_safetensors *Save MLX arrays to the safetensors format*

Description

Save MLX arrays to the safetensors format

Usage

```
mlx_save_safetensors(file, arrays, metadata = character())
```

Arguments

file	Output path. .safetensors is appended automatically when omitted.
arrays	Named list of objects coercible to mlx.
metadata	Optional named character vector of metadata entries.

Value

Invisibly returns the full path that was written.

See Also

https://ml-explore.github.io/mlx/build/html/python/io.html#mlx.core.save_safetensors

---

`mlx_scalar` *Construct MLX scalars*

Description

Construct MLX scalars

Usage

```
mlx_scalar(value, dtype = NULL)
```

Arguments

`value`	Single value (numeric, logical, or complex).
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Value

A dimensionless `mlx` scalar.

`mlx_scatter_add_axis` *Add values using per-position axis indices*

Description

Mirrors `mlx.core.scatter_add_axis()` while accepting 1-based R indices.

Usage

```
mlx_scatter_add_axis(x, indices, values, axis)
```

Arguments

`x`	An mlx array.
`indices`	Integer positions along `axis`. Must be broadcast-compatible with `x` except at the selected axis.
`values`	Replacement values.
`axis`	Axis to index (1-based).

Value

An updated `mlx` array after additive scatter.

Examples

```
x <- mlx_matrix(1:12, nrow = 3, ncol = 4)
idx <- matrix(c(1L, 1L,
                2L, 3L,
                4L, 4L), nrow = 3, byrow = TRUE)
values <- matrix(c(10, 20,
                   30, 40,
                   50, 60), nrow = 3, byrow = TRUE)
mlx_scatter_add_axis(x, idx, values, axis = 2L)
```

`mlx_sequential` *Compose modules sequentially*

Description

Compose modules sequentially

Usage

```
mlx_sequential(...)
```

Arguments

`...`	Modules to compose.

Value

An `mlx_module`.

See Also

mlx.nn.Sequential

Examples

```
set.seed(1)
net <- mlx_sequential(mlx_linear(2, 3), mlx_relu(), mlx_linear(3, 1))
input <- as_mlx(matrix(c(1, 2), 1, 2))
mlx_forward(net, input)
```

mlx_set_default_stream *Set the default MLX stream*

Description

Set the default MLX stream

Usage

```
mlx_set_default_stream(stream)
```

Arguments

stream	An object created by mlx_new_stream() or mlx_default_stream().

Value

Invisibly returns stream.

Examples

```
stream <- mlx_new_stream()
old <- mlx_default_stream()
mlx_set_default_stream(stream)
mlx_set_default_stream(old)  # restore
```

mlx_set_training *Toggle training mode for MLX modules*

Description

mlx_set_training() switches modules between training and evaluation modes. Layers that do not implement training-specific behaviour ignore the call.

Usage

```
mlx_set_training(module, mode = TRUE)
```

Arguments

module	An object inheriting from mlx_module.
mode	Logical flag; TRUE for training mode, FALSE for evaluation.

Value

The input module (invisibly).

See Also

https://ml-explore.github.io/mlx/build/html/python/nn.html#mlx.nn.Module.train

Examples

```
model <- mlx_sequential(mlx_linear(2, 4), mlx_dropout(0.5))
mlx_set_training(model, FALSE)
```

---

mlx_sigmoid *Sigmoid activation*

Description

Sigmoid activation

Usage

```
mlx_sigmoid()
```

Value

An `mlx_module` applying sigmoid activation.

See Also

mlx.nn.Sigmoid

Examples

```
act <- mlx_sigmoid()
x <- as_mlx(matrix(c(-2, -1, 0, 1, 2), 5, 1))
mlx_forward(act, x)
```

---

mlx_silu *SiLU (Swish) activation*

Description

Sigmoid Linear Unit, also known as Swish activation.

Usage

```
mlx_silu()
```

Value

An `mlx_module` applying SiLU activation.

See Also

mlx.nn.SiLU

Examples

```
act <- mlx_silu()
x <- as_mlx(matrix(c(-2, -1, 0, 1, 2), 5, 1))
mlx_forward(act, x)
```

`mlx_slice_update` *Update a slice of an mlx array*

Description

Wrapper around `mlx.core.slice_update()` that replaces a contiguous strided region with `value`.

Usage

```
mlx_slice_update(x, value, start, stop, strides = NULL)
```

Arguments

`x`	An mlx array.
`value`	Replacement `mlx` (or coercible) array. Must broadcast to the slice determined by `start`, `stop`, and `strides`.
`start`	Integer vector (1-indexed) giving the inclusive starting index for each axis.
`stop`	Integer vector (1-indexed) giving the inclusive stopping index for each axis.
`strides`	Optional integer vector of strides (defaults to ones).

Value

An `mlx` array with the specified slice replaced.

Difference from Python/C++

Unlike Python's slice notation `array[start:stop]` which uses an exclusive upper bound, `mlx_slice_update()` uses **inclusive** bounds for both `start` and `stop` to match R conventions and to be consistent with `mlx_arange()` and `mlx_linspace()`.

Examples

```
x <- mlx_matrix(1:9, 3, 3)
replacement <- mlx_matrix(100:103, nrow = 2)
updated <- mlx_slice_update(x, replacement, start = c(1L, 2L), stop = c(2L, 3L))
updated
```

mlx_softmax *Softmax for mlx arrays*

Description

Softmax for mlx arrays

Usage

```
mlx_softmax(x, axes = NULL, precise = FALSE)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
axes	Integer vector of axes (1-indexed). Supply positive integers between 1 and the array rank. Many helpers interpret NULL to mean "all axes"—see the function details for specifics.
precise	Logical; compute in higher precision for stability.

Value

An mlx array with normalized probabilities.

See Also

mlx.core.softmax

Examples

```
x <- mlx_matrix(1:6, 2, 3)
sm <- mlx_softmax(x, axes = 2)
rowSums(sm)
```

mlx_softmax_layer *Softmax activation*

Description

Softmax activation

Usage

```
mlx_softmax_layer(axis = NULL)
```

Arguments

axis	Axis (1-indexed) along which to apply softmax. Omit the argument to use the last dimension at runtime.

Value

An mlx_module applying softmax activation.

See Also

mlx.nn.Softmax

Examples

```
act <- mlx_softmax_layer()
x <- mlx_matrix(1:6, 2, 3)
mlx_forward(act, x)
```

---

mlx_solve_triangular *Solve triangular systems with mlx arrays*

Description

Solve triangular systems with mlx arrays

Usage

```
mlx_solve_triangular(a, b, upper = FALSE, device = NULL)

backsolve(r, x, k = NULL, upper.tri = TRUE, transpose = FALSE, ...)

## Default S3 method:
backsolve(r, x, k = NULL, upper.tri = TRUE, transpose = FALSE, ...)

## S3 method for class 'mlx'
backsolve(
  r,
  x,
  k = NULL,
  upper.tri = TRUE,
  transpose = FALSE,
  ...,
  device = NULL
)
```

Arguments

a	An mlx triangular matrix.
b	Right-hand side matrix or vector.
upper	Logical; if TRUE, a is upper triangular, otherwise lower.
device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.
r	Triangular system matrix passed to backsolve().
x	Right-hand side supplied to backsolve().
k	Number of columns of r to use.
upper.tri	Logical; indicates if r is upper triangular.
transpose	Logical; if TRUE, solve t(r) %*% x = b.
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside with_device() or local_device(), or pass device = "cpu".

Value

An mlx array solution.

See Also

mlx.linalg.solve_triangular

Examples

```
a <- mlx_matrix(c(2, 1, 0, 3), 2, 2)
b <- mlx_matrix(c(1, 5), 2, 1)
mlx_solve_triangular(a, b, upper = FALSE, device = "cpu")
```

---

mlx_sort *Sort and argsort for mlx arrays*

Description

mlx_sort() returns sorted values along the specified axis. mlx_argsort() returns the indices that would sort the array.

Usage

```
mlx_sort(x, axis = NULL)

mlx_argsort(x, axis = NULL)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
axis	Single axis (1-indexed). Supply a positive integer between 1 and the array rank. Use NULL when the helper interprets it as "all axes" (see individual docs).

Details

mlx_argsort() returns **1-based indices** that would sort the array in ascending order. This follows R's indexing convention (unlike the underlying MLX library which uses 0-based indexing). The returned indices can be used directly to reorder the original array.

Named vectors keep names on sorted values. For arrays sorted along an axis, the sorted axis drops names because each slice may use a different permutation, while names on untouched axes are kept.

For partial sorting (finding elements up to a certain rank without fully sorting), see mlx_partition() and mlx_argpartition().

Value

An mlx array containing sorted values (for mlx_sort()) or **1-based indices** (for mlx_argsort()). The indices follow R's indexing convention and can be used directly with R's [operator.

See Also

mlx.core.sort, mlx.core.argsort

Examples

```
x <- as_mlx(c(3, 1, 4, 2))
mlx_sort(x)

# Returns 1-based indices
idx <- mlx_argsort(x)
as.integer(as.matrix(idx))  # [1] 2 4 1 3

# Can be used directly with R indexing
original <- c(3, 1, 4, 2)
sorted_idx <- as.integer(as.matrix(mlx_argsort(as_mlx(original))))
original[sorted_idx]  # [1] 1 2 3 4

mlx_sort(mlx_matrix(1:6, 2, 3), axis = 1)
```

mlx_split *Split mlx arrays*

Description

`mlx_split()` divides an array along an axis either into equal sections (`sections` scalar) or at explicit 1-based split points (`sections` list), returning a list of mlx arrays.

Usage

```
mlx_split(x, sections, axis = 1L)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`sections`	Either a single integer (number of equal parts) or a *list* of 1-based split points along `axis`.
`axis`	Axis (1-indexed) to operate on.

Value

A list of mlx arrays split along the chosen axis.

See Also

mlx.core.split, `mlx_pad()`

Examples

```
x <- mlx_matrix(1:4, 2, 2)
parts <- mlx_split(x, sections = 2, axis = 1)
custom_parts <- mlx_split(x, sections = list(1), axis = 2)
```

mlx_squeeze *Remove singleton dimensions*

Description

Remove singleton dimensions

Usage

```
mlx_squeeze(x, axes = NULL)
```

Arguments

x	An mlx array.
axes	Optional integer vector of axes (1-indexed) to remove. When NULL all axes of length one are removed.

Value

An mlx array with the selected axes removed.

See Also

mlx.core.squeeze

Examples

```
x <- mlx_array(1:4, dim = c(1, 2, 2, 1))
mlx_squeeze(x)
mlx_squeeze(x, axes = 1)
```

---

mlx_stack *Stack mlx arrays along a new axis*

Description

Stack mlx arrays along a new axis

Usage

```
mlx_stack(..., axis = 1L)
```

Arguments

...	One or more arrays (or a single list of arrays) coercible via as_mlx().
axis	Position of the new axis (1-indexed). Supply values between 1 and length(mlx_shape(x)) + 1 to insert anywhere along the dimension list.

Value

An mlx array with one additional dimension.

See Also

mlx.core.stack

Examples

```
x <- mlx_matrix(1:4, 2, 2)
y <- mlx_matrix(5:8, 2, 2)
stacked <- mlx_stack(x, y, axis = 1)
```

mlx_stop_gradient *Stop gradient propagation through an mlx array*

Description

Stop gradient propagation through an mlx array

Usage

```
mlx_stop_gradient(x)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().

Value

A new mlx array with identical values but zero gradient.

See Also

mlx.core.stop_gradient

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_stop_gradient(x)
```

mlx_sum *Reduce mlx arrays*

Description

These helpers mirror NumPy-style reductions, optionally collapsing one or more axes. Use drop = FALSE to retain reduced axes with length one (akin to setting drop = FALSE in base R).

Usage

```
mlx_sum(x, axes = NULL, drop = TRUE)

mlx_prod(x, axes = NULL, drop = TRUE)

mlx_all(x, axes = NULL, drop = TRUE)

mlx_any(x, axes = NULL, drop = TRUE)
```

```
mlx_mean(x, axes = NULL, drop = TRUE)

mlx_var(x, axes = NULL, drop = TRUE, ddof = 0L)

mlx_std(x, axes = NULL, drop = TRUE, ddof = 0L)

mlx_sd(x, axes = NULL, drop = TRUE)
```

Arguments

x	An mlx array, or an R array/matrix/vector that will be converted via as_mlx().
axes	Integer vector of axes (1-indexed). Supply positive integers between 1 and the array rank. Many helpers interpret NULL to mean "all axes"—see the function details for specifics.
drop	If TRUE (default), drop dimensions of length 1. If FALSE, retain all dimensions. Equivalent to keepdims = TRUE in underlying mlx functions.
ddof	Non-negative integer delta degrees of freedom for variance or standard deviation reductions.

Details

mlx_all() and mlx_any() return mlx boolean scalars, while the base R reducers all() and any() applied to mlx inputs return plain logical scalars.

The axes argument is the inverse of MARGIN in base R apply(). axes gives the axes which will be reduced; MARGIN gives the axes which an operation will be applied over. See the example. Multiple axes are reduced jointly. For variance and standard deviation, ddof is subtracted once from the total number of elements being reduced.

mlx_sd() is a convenience wrapper that matches the default behaviour of stats::sd(), computing a sample standard deviation with ddof = 1.

Value

An mlx array containing the reduction result.

See Also

mlx.core.sum, mlx.core.prod, mlx.core.all, mlx.core.any, mlx.core.mean, mlx.core.var, mlx.core.std

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_sum(x)
mlx_sum(x, axes = 1)
mlx_prod(x, axes = 2, drop = FALSE)
mlx_all(x > 0)
mlx_any(x > 3)
mlx_mean(x, axes = 1)
mlx_var(x, axes = 2)
mlx_std(x)
```

```
mlx_sd(x)
# for comparison:
stats::sd(as.matrix(x))

a <- array(1:6, dim = 1:3)
ax <- as_mlx(a)
# These are equivalent:
apply(a, 1:2, sum) # leaves dimensions 1-2 intact, sums over dimension 3
mlx_sum(a, 3)      # the same
```

`mlx_swapaxes` *Swap two axes of an mlx array*

Description

`mlx_swapaxes()` mirrors `mlx.core.swapaxes()`, exchanging two dimensions while leaving others intact.

Usage

```
mlx_swapaxes(x, axis1, axis2)
```

Arguments

`x`	An mlx array.
`axis1`, `axis2`	Axes to swap (1-indexed).

Value

An mlx array with the specified axes exchanged.

See Also

mlx.core.swapaxes

Examples

```
x <- mlx_array(1:24, dim = c(2, 3, 4))
swapped <- mlx_swapaxes(x, axis1 = 1, axis2 = 3)
dim(swapped)
```

mlx_synchronize *Synchronize MLX execution*

Description

Waits for outstanding operations on the specified device or stream to complete.

Usage

```
mlx_synchronize(device = mlx_device())
```

Arguments

device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Value

Returns NULL invisibly.

See Also

mlx.core.default_device

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_synchronize("cpu")
if (mlx_has_gpu()) mlx_synchronize("gpu")
stream <- mlx_new_stream()
mlx_synchronize(stream)
```

mlx_take_along_axis *Take values using per-position axis indices*

Description

Mirrors mlx.core.take_along_axis() while accepting 1-based R indices.

Usage

```
mlx_take_along_axis(x, indices, axis)
```

Arguments

x	An mlx array.
indices	Integer positions along axis. Must be broadcast-compatible with x except at the selected axis.
axis	Axis to index (1-based).

Details

If y <- mlx_take_along_axis(x, idx, axis) where x is an m x n matrix and idx is a matrix:

- y will have the same shape as idx, possibly after idx has been broadcast to the dimensions of y for all axes except axis.
- For axis = 1, values of idx give the row, and columns are in order: y[i, j] equals x[idx[i, j], j]. idx must have 1 or n columns. y will have the same number of rows as idx.
- For axis = 2, values of idx give the column, and rows are in order: y[i, j] equals x[i, idx[i, j]]. idx must have 1 or m rows, and y will have the same number of columns as idx.

More generally, for x and idx of d dimensions, and axis = a:

- y[i_1, ..., i_d] equals x[i_1, ..., idx[i_1,...,i_d], ..., i_d] where the idx vector is in position a.

For broadcasting, the simplest rule is that if idx has 1 column, mlx_take_along_axis(x, idx, 1) is the same as x[drop(idx),]; and if idx has 1 row, mlx_take_along_axis(x, idx, 2) is the same as x[, drop(idx)].

Value

An mlx array. Names on the indexed axis are dropped because per-position indices may reorder each slice differently.

Examples

```
x <- outer(1:3, c(0.1, 0.2), "+")
x <- as_mlx(x)
x

idx_cols <- matrix(c(1, 2,
                     2, 2,
                     1, 1), nrow = 3, byrow = TRUE)
mlx_take_along_axis(x, idx_cols, axis = 2)

idx_rows <- matrix(c(1, 2,
                     3, 1), nrow = 2, byrow = TRUE)
mlx_take_along_axis(x, idx_rows, axis = 1)
```

mlx_tanh *Tanh activation*

Description

Tanh activation

Usage

```
mlx_tanh()
```

Value

An `mlx_module` applying hyperbolic tangent activation.

See Also

mlx.nn.Tanh

Examples

```
act <- mlx_tanh()
x <- as_mlx(matrix(c(-2, -1, 0, 1, 2), 5, 1))
mlx_forward(act, x)
```

mlx_tile *Tile an array*

Description

Tile an array

Usage

```
mlx_tile(x, reps)
```

Arguments

`x`	An mlx array.
`reps`	Integer vector giving the number of repetitions for each axis.

Value

An mlx array with tiled content. Existing axis names are tiled with their axes; new leading axes introduced by `reps` are unnamed.

See Also

mlx.core.tile

Examples

```
x <- mlx_matrix(1:4, 2, 2)
mlx_tile(x, reps = c(1, 2))
```

`mlx_topk` *Top-k selection and partitioning on mlx arrays*

Description

`mlx_topk()` returns the largest `k` values. `mlx_partition()` and `mlx_argpartition()` perform partial sorting, rearranging elements so that the element at position `kth` is in its correctly sorted position, with all smaller elements before it and all larger elements after it. This is more efficient than full sorting when you only need elements up to a certain rank.

Usage

```
mlx_topk(x, k, axis = NULL)

mlx_partition(x, kth, axis = NULL)

mlx_argpartition(x, kth, axis = NULL)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`k`	Positive integer specifying the number of elements to select.
`axis`	Single axis (1-indexed). Supply a positive integer between 1 and the array rank. Use `NULL` when the helper interprets it as "all axes" (see individual docs).
`kth`	Zero-based index of the element that should be placed in-order after partitioning.

Details

- `mlx_topk()` returns the largest `k` values along the specified axis.
- `mlx_partition()` rearranges elements so the kth element is correctly positioned.
- `mlx_argpartition()` returns the **1-based indices** that would partition the array. This follows R's indexing convention (unlike the underlying MLX library which uses 0-based indexing).
- Named vectors keep names on partitioned values. For arrays partitioned or selected along an axis, the reordered axis drops names because each slice may use a different permutation, while names on untouched axes are kept.

Use `mlx_argsort()` if you need fully sorted indices.

Value

An mlx array. For `mlx_argpartition()`, returns 1-based indices (following R conventions) showing the partition ordering.

See Also

mlx.core.topk, mlx.core.partition, mlx.core.argpartition

Examples

```
scores <- as_mlx(c(0.7, 0.2, 0.9, 0.4))
mlx_topk(scores, k = 2)
mlx_partition(scores, kth = 1)

# Returns 1-based indices
idx <- mlx_argpartition(scores, kth = 1)
as.integer(as.matrix(idx))  # 1-based indices

mlx_topk(mlx_matrix(1:6, 2, 3), k = 1, axis = 1)
```

`mlx_trace` *Matrix trace for mlx arrays*

Description

Computes the sum of the diagonal elements of a 2D array, or the sum along diagonals of a higher dimensional array.

Usage

```
mlx_trace(x, offset = 0L, axis1 = 1L, axis2 = 2L)
```

Arguments

`x`	An mlx array.
`offset`	Offset of the diagonal (0 for main diagonal, positive for above, negative for below).
`axis1, axis2`	Axes along which the diagonals are taken (1-indexed, default 1 and 2).

Value

An mlx scalar or array containing the trace.

See Also

mlx.core.trace

Examples

```
x <- mlx_matrix(1:9, 3, 3)
mlx_trace(x)
mlx_trace(x, offset = 1)
```

---

mlx_train_step *Single training step helper*

Description

Single training step helper

Usage

```
mlx_train_step(module, loss_fn, optimizer, ...)
```

Arguments

module	An mlx_module.
loss_fn	Function of module and data returning an mlx scalar.
optimizer	Optimizer object from mlx_optimizer_sgd().
...	Additional data passed to loss_fn.

Value

A list with the current loss.

See Also

mlx.optimizers.Optimizer

Examples

```
set.seed(1)
model <- mlx_linear(2, 1, bias = FALSE)
opt <- mlx_optimizer_sgd(mlx_parameters(model), lr = 0.1)
data_x <- as_mlx(matrix(c(1, 2, 3, 4), 2, 2))
data_y <- as_mlx(matrix(c(5, 6), 2, 1))
loss_fn <- function(model, x, y) {
  pred <- model$forward(x)
  mean((pred - y)^2)
}
result <- mlx_train_step(model, loss_fn, opt, data_x, data_y)
```

mlx_tri *Triangular helpers for MLX arrays*

Description

`mlx_tri()` creates a lower-triangular mask (ones on and below a diagonal, zeros elsewhere). `mlx_tril()` and `mlx_triu()` retain only the lower or upper triangular part of an existing array, respectively.

Usage

```
mlx_tri(n, m = NULL, k = 0L, dtype = c("float32", "float64"))

mlx_tril(x, k = 0L)

mlx_triu(x, k = 0L)
```

Arguments

`n`	Number of rows.
`m`	Optional number of columns (defaults to `n` for square output).
`k`	Diagonal offset: `0` selects the main diagonal, positive values move to the upper diagonals, negative values to the lower diagonals.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.
`x`	Object coercible to `mlx`.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An `mlx` array.

See Also

mlx.core.tri

Examples

```
mlx_tri(3)          # 3x3 lower-triangular mask
mlx_tril(diag(3) + 2)  # keep lower part of a matrix
```

`mlx_tri_inv` *Compute triangular matrix inverse*

Description

Computes the inverse of a triangular matrix.

Usage

```
mlx_tri_inv(x, upper = FALSE, device = NULL)
```

Arguments

`x`	An mlx array.
`upper`	Logical; if `TRUE`, `x` is upper triangular, otherwise lower triangular.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

Note: MLX may crash if `x` is not triangular.

Value

The inverse of the triangular matrix `x`.

See Also

mlx.core.linalg.tri_inv

Examples

```
# Lower triangular matrix
L <- mlx_matrix(c(1:3, 0, 4:5, 0, 0, 6), 3, 3)
mlx_tri_inv(L, upper = FALSE, device = "cpu")
```

mlx_unflatten *Unflatten an axis into multiple axes*

Description

The reverse of flattening: expands a single axis into multiple axes with the given shape.

Usage

```
mlx_unflatten(x, axis, shape)
```

Arguments

x	An mlx array.
axis	Which axis to unflatten (1-indexed).
shape	Integer vector specifying the new shape for the unflattened axis.

Value

An mlx array with the axis expanded.

See Also

mlx.core.unflatten

Examples

```
# Flatten and unflatten
x <- mlx_array(1:24, c(2, 3, 4))
x_flat <- mlx_reshape(x, c(2, 12))  # flatten last two dims
mlx_unflatten(x_flat, axis = 2, shape = c(3, 4))  # restore original shape
```

mlx_vector *Construct MLX vectors*

Description

`mlx_vector()` is a convenience around `mlx_array()` for 1-D payloads.

Usage

```
mlx_vector(data, dtype = NULL)
```

Arguments

data	Atomic vector providing the elements (recycling is not allowed).
dtype	Data type string. Supported types include: • Floating point: "float32", "float64" • Integer: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" • Other: "bool", "complex64" Not all functions support all types. See individual function documentation.

Value

An mlx vector with dim = length(data).

---

mlx_where *Elementwise conditional selection*

Description

Elementwise conditional selection

Usage

```
mlx_where(condition, x, y)
```

Arguments

condition	Logical mlx array (non-zero values are treated as TRUE).
x, y	Arrays broadcastable to the shape of condition.

Details

Behaves like ifelse() for arrays, but evaluates both branches.

Value

An mlx array where elements are drawn from x when condition is TRUE, otherwise from y.

See Also

mlx.core.where

Examples

```
cond <- mlx_matrix(c(TRUE, FALSE, TRUE, FALSE), 2, 2)
a <- mlx_matrix(1:4, 2, 2)
b <- mlx_matrix(5:8, 2, 2)
mlx_where(cond, a, b)
```

`mlx_zeros` *Create arrays of zeros on MLX devices*

Description

Create arrays of zeros on MLX devices

Usage

```
mlx_zeros(
  dim,
 dtype = c("float32", "float64", "int8", "int16", "int32", "int64", "uint8", "uint16",
    "uint32", "uint64", "bool", "complex64")
)
```

Arguments

`dim`	Integer vector specifying array dimensions (shape).
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An mlx array filled with zeros.

See Also

mlx.core.zeros

Examples

```
zeros <- mlx_zeros(c(2, 3))
zeros_int <- mlx_zeros(c(2, 3), dtype = "int32")
```

`mlx_zeros_like` *Zeros shaped like an existing mlx array*

Description

`mlx_zeros_like()` mirrors `mlx.core.zeros_like()`: it creates a zero-filled array matching the source array's shape. Optionally override the dtype or dtype.

Usage

```
mlx_zeros_like(x, dtype = NULL)
```

Arguments

`x`	An mlx array.
`dtype`	Data type string. Supported types include: • Floating point: `"float32"`, `"float64"` • Integer: `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"` • Other: `"bool"`, `"complex64"` Not all functions support all types. See individual function documentation.

Details

MLX does not support `float64` operations on GPU. When this function creates a `float64` array or converts one back to R, Rmlx temporarily switches only that internal creation or layout work to CPU. Later operations on the returned array still use the current `mlx_device()`.

Value

An mlx array of zeros matching `x`.

See Also

mlx.core.zeros_like

Examples

```
base <- mlx_ones(c(2, 2))
mlx_zeros_like(base)
```

Ops.mlx — *Arithmetic and comparison operators for MLX arrays*

Description

Arithmetic and comparison operators for MLX arrays

Usage

```
## S3 method for class 'mlx'
Ops(e1, e2 = NULL)
```

Arguments

e1	First operand (mlx or numeric)
e2	Second operand (mlx or numeric)

Value

An mlx object.

See Also

mlx.core.array

Examples

```
x <- mlx_matrix(1:4, 2, 2)
y <- mlx_matrix(5:8, 2, 2)
x + y
x < y
```

outer — *Outer product of two vectors*

Description

Outer product of two vectors

Usage

```
outer(X, Y, FUN = "*", ...)

## S3 method for class 'mlx'
outer(X, Y, FUN = "*", ...)
```

Arguments

X, Y	Numeric vectors or mlx arrays.
FUN	Function to apply (for default method).
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.

Value

For mlx inputs, an mlx matrix. Otherwise delegates to `base::outer`.

See Also

mlx.core.outer

Examples

```
x <- as_mlx(c(1, 2, 3))
y <- as_mlx(c(4, 5))
outer(x, y)
```

---

pinv *Moore-Penrose pseudoinverse for MLX arrays*

Description

Moore-Penrose pseudoinverse for MLX arrays

Usage

```
pinv(x, device = NULL)
```

Arguments

x	An mlx object or coercible matrix.
device	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

An mlx object containing the pseudoinverse.

See Also

mlx.linalg.pinv

Examples

```
x <- mlx_matrix(c(1, 2, 3, 4), 2, 2)
pinv(x, device = "cpu")
```

`print.mlx` *Print MLX array*

Description

Printing an array only evaluates it if it is of small size (less than 100 elements and 2 dimensions)

Usage

```
## S3 method for class 'mlx'
print(x, ...)
```

Arguments

`x`	An mlx array, or an R array/matrix/vector that will be converted via `as_mlx()`.
`...`	Additional arguments; ignored.

Value

`x`, invisibly.

Examples

```
x <- mlx_matrix(1:4, 2, 2)
print(x)
```

`print.mlx_stream` *Print method for mlx_stream*

Description

Print method for mlx_stream

Usage

```
## S3 method for class 'mlx_stream'
print(x, ...)
```

Arguments

x	An mlx_stream object.
...	Additional arguments; ignored.

Value

Returns x invisibly.

---

qr.mlx *QR decomposition for mlx arrays*

Description

QR decomposition for mlx arrays

Usage

```
## S3 method for class 'mlx'
qr(x, tol = 1e-07, LAPACK = FALSE, ..., device = NULL)
```

Arguments

x	An mlx matrix (2-dimensional array).
tol	Ignored; custom tolerances are not supported.
LAPACK	Ignored; set to FALSE.
...	Additional arguments; ignored.
device	Execution target for APIs that expose a one-off device or stream override. Supply "gpu", "cpu", or an mlx_stream created via mlx_new_stream(). Ordinary array operations use the current mlx_device() instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside with_device() or local_device(), or pass device = "cpu".

Value

A list with components Q and R, each an mlx matrix.

See Also

mlx.linalg.qr

Examples

```
x <- mlx_matrix(c(1, 2, 3, 4, 5, 6), 3, 2)
qr(x, device = "cpu")
```

`rbind.mlx` *Row-bind mlx arrays*

Description

Row-bind mlx arrays

Usage

```
## S3 method for class 'mlx'
rbind(..., deparse.level = 1)
```

Arguments

`...`	Objects to bind. mlx arrays are kept in MLX; other inputs are coerced via `as_mlx()`.
`deparse.level`	Compatibility argument accepted for S3 dispatch; ignored.

Details

Unlike base R's `rbind()`, this function supports arrays with more than 2 dimensions and preserves all dimensions except the first (which is summed across inputs). Base R's `rbind()` flattens higher-dimensional arrays to matrices before binding.

Value

An mlx array stacked along the first axis.

See Also

mlx.core.concatenate

Examples

```
x <- mlx_matrix(1:4, 2, 2)
y <- mlx_matrix(5:8, 2, 2)
rbind(x, y)
```

row *Row and column indices for mlx arrays*

Description

Extends base `row()` and `col()` so they also accept mlx arrays. When `as.factor = FALSE` the result stays on the MLX backend, avoiding round-tripping through base R.

Usage

```
row(x, as.factor = FALSE)

## Default S3 method:
row(x, as.factor = FALSE)

## S3 method for class 'mlx'
row(x, as.factor = FALSE)

col(x, as.factor = FALSE)

## Default S3 method:
col(x, as.factor = FALSE)

## S3 method for class 'mlx'
col(x, as.factor = FALSE)
```

Arguments

`x`	a matrix-like object, that is one with a two-dimensional `dim`.
`as.factor`	a logical value indicating whether the value should be returned as a factor of row labels (created if necessary) rather than as numbers.

Value

A matrix or array of row indices (for `row()`) or column indices (for `col()`), matching the base R behaviour.

rowMeans *Row means for mlx arrays*

Description

Row means for mlx arrays

Usage

```
rowMeans(x, ...)

## Default S3 method:
rowMeans(x, na.rm = FALSE, dims = 1, ...)

## S3 method for class 'mlx'
rowMeans(x, na.rm = FALSE, dims = 1, ...)
```

Arguments

x	An array or mlx array.
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
na.rm	Logical; currently ignored for mlx arrays.
dims	Leading dimensions treated as rows/cols (see base::rowSums()).

Value

An mlx array if x is_mlx, otherwise a numeric vector.

See Also

mlx.core.mean

Examples

```
x <- mlx_matrix(1:6, 3, 2)
rowMeans(x)
```

---

rowSums *Row sums for mlx arrays*

Description

Row sums for mlx arrays

Usage

```
rowSums(x, ...)

## Default S3 method:
rowSums(x, na.rm = FALSE, dims = 1, ...)

## S3 method for class 'mlx'
rowSums(x, na.rm = FALSE, dims = 1, ...)
```

Arguments

x	An array or mlx array.
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
na.rm	Logical; currently ignored for mlx arrays.
dims	Leading dimensions treated as rows/cols (see base::rowSums()).

Value

An mlx array if x is_mlx, otherwise a numeric vector.

See Also

mlx.core.sum

Examples

```
x <- mlx_matrix(1:6, 3, 2)
rowSums(x)
```

---

scale.mlx *Scale mlx arrays*

Description

Extends base scale() to handle mlx inputs with MLX reductions and broadcasting. Computed center and scale attributes are mlx arrays. User-supplied attributes keep their input type: mlx inputs stay mlx, while base R vectors stay base R vectors.

Usage

```
## S3 method for class 'mlx'
scale(x, center = TRUE, scale = TRUE)
```

Arguments

x	a numeric matrix(like object).
center	either a logical value or numeric-alike vector of length equal to the number of columns of x, where 'numeric-alike' means that as.numeric(.) will be applied successfully if is.numeric(.) is not true.
scale	either a logical value or a numeric-alike vector of length equal to the number of columns of x.

Value

An mlx array with centred/scaled columns.

`solve.mlx` *Solve a system of linear equations*

Description

Solve a system of linear equations

Usage

```
## S3 method for class 'mlx'
solve(a, b = NULL, ..., device = NULL)
```

Arguments

`a`	An mlx matrix of coefficients.
`b`	An mlx vector or matrix (the right-hand side). If omitted, computes the matrix inverse.
`...`	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.
`device`	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

An mlx object containing the solution.

See Also

mlx.linalg.solve

Examples

```
with_device("cpu", {
  a <- mlx_matrix(c(3, 1, 1, 2), 2, 2)
  b <- as_mlx(c(9, 8))
  solve(a, b)
})
```

str.mlx *Object structure for MLX array*

Description

Object structure for MLX array

Usage

```
## S3 method for class 'mlx'
str(object, ...)
```

Arguments

object	An mlx object
...	Additional arguments; ignored.

Value

NULL invisibly.

Examples

```
x <- mlx_matrix(1:4, 2, 2)
str(x)
```

Summary.mlx *Summary operations for MLX arrays*

Description

S3 group generic for summary functions including sum(), prod(), min(), max(), all(), and any().

Usage

```
## S3 method for class 'mlx'
Summary(x, ..., na.rm = FALSE)
```

Arguments

x	mlx array or object coercible to mlx
...	Additional mlx arrays (for reducing multiple arrays), or named arguments axes (legacy axis) and drop
na.rm	Logical; currently ignored for mlx arrays (generates warning if TRUE)

Value

An mlx array with the summary result.

See Also

mlx.core.array

Examples

```
x <- mlx_matrix(1:6, 2, 3)
sum(x)
any(x > 3)
all(x > 0)
```

`svd` *Singular value decomposition*

Description

Generic function for SVD computation.

Usage

```
svd(x, ...)
```

Arguments

`x`	An object.
`...`	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.

Value

A list with components `d`, `u`, and `v`.

svd.mlx *Singular value decomposition for mlx arrays*

Description

Note that mlx's svd returns "full" SVD, with U and V' both square matrices. This is different from R's implementation.

Usage

```
## S3 method for class 'mlx'
svd(x, nu = min(n, p), nv = min(n, p), ..., device = NULL)
```

Arguments

x	An mlx matrix (2-dimensional array).
nu	Number of left singular vectors to return (0 or `min(dim(x))`).
nv	Number of right singular vectors to return (0 or `min(dim(x))`).
...	Additional arguments; ignored.
device	Execution target for APIs that expose a one-off device or stream override. Supply `"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`. Ordinary array operations use the current `mlx_device()` instead.

Details

As of MLX 0.31.1, this operation only runs on CPU. Run it inside `with_device()` or `local_device()`, or pass `device = "cpu"`.

Value

A list with components `d`, `u`, and `v`.

See Also

mlx.linalg.svd

Examples

```
x <- mlx_matrix(c(1, 0, 0, 2), 2, 2)
svd(x, device = "cpu")
```

t.mlx *Transpose of MLX matrix*

Description

Transpose of MLX matrix

Usage

```
## S3 method for class 'mlx'
t(x)
```

Arguments

x	An mlx matrix (2-dimensional array).

Value

The transposed MLX matrix.

See Also

mlx.core.transpose

Examples

```
x <- mlx_matrix(1:6, 2, 3)
t(x)
```

tcrossprod.mlx *Transposed cross product*

Description

Transposed cross product

Usage

```
## S3 method for class 'mlx'
tcrossprod(x, y = NULL, ...)
```

Arguments

x	An mlx matrix (2-dimensional array).
y	An mlx matrix (default: NULL, uses x)
...	Additional arguments forwarded to the corresponding base R implementation for signature compatibility.

Value

`x %*% t(y)` as an mlx object.

See Also

mlx.core.matmul

Examples

```
x <- mlx_matrix(1:6, 2, 3)
tcrossprod(x)
```

`with_device` *Temporarily set the current MLX device or stream*

Description

Use `local_device()` to temporarily switch devices within the current function.

Usage

```
with_device(device, code)

local_device(device, .local_envir = parent.frame())
```

Arguments

`device`	`"gpu"`, `"cpu"`, or an `mlx_stream` created via `mlx_new_stream()`.
`code`	Expression to evaluate while `device` is active.
`.local_envir`	Environment to bind the restoration to. Defaults to the calling environment.

Value

The result of evaluating `code`.

Invisibly returns the previous default device.

See Also

mlx.core.default_device

Examples

```
with_device("cpu", x <- mlx_vector(1:10))

local_device("cpu")
# code here runs on CPU, then the previous default is restored
```

Index

!.mlx *(Ops.mlx)*, 155
!=.mlx *(Ops.mlx)*, 155
∗ **documentation**
    mlx-methods, 32
*.mlx *(Ops.mlx)*, 155
+.mlx *(Ops.mlx)*, 155
-.mlx *(Ops.mlx)*, 155
/.mlx *(Ops.mlx)*, 155
<.mlx *(Ops.mlx)*, 155
<=.mlx *(Ops.mlx)*, 155
==.mlx *(Ops.mlx)*, 155
>.mlx *(Ops.mlx)*, 155
>=.mlx *(Ops.mlx)*, 155
[.mlx *([<-.mlx)*, 7
[<-.mlx, 7
%/%.mlx *(Ops.mlx)*, 155
%%.mlx *(Ops.mlx)*, 155
&.mlx *(Ops.mlx)*, 155
%*%.mlx, 8
^.mlx *(Ops.mlx)*, 155

abind, 9
abind(), *32*
abind::abind(), *9*
abs.mlx *(Math.mlx)*, 29
acos.mlx *(Math.mlx)*, 29
acosh.mlx *(Math.mlx)*, 29
all(), *141*
all.equal.mlx, 10
all.equal.mlx(), *34*, *84*
all.mlx *(Summary.mlx)*, 164
any(), *141*
any.mlx *(Summary.mlx)*, 164
aperm.mlx *(mlx_moveaxis)*, 100
apply(), *141*
as.array(), *6*
as.array.mlx, 11
as.array.mlx(), *16*
as.double.mlx *(as.vector.mlx)*, 12
as.integer.mlx *(as.vector.mlx)*, 12
as.logical.mlx *(as.vector.mlx)*, 12
as.matrix(), *79*
as.matrix(x), *70*
as.matrix.mlx, 12
as.matrix.mlx(), *11*, *16*
as.numeric, *162*
as.numeric.mlx *(as.vector.mlx)*, 12
as.vector(), *6*
as.vector.mlx, 12
as.vector.mlx(), *11*, *16*
as_mlx, 14
as_mlx(), *7*, *9*, *14*, *24*, *25*, *29*, *32*, *36*, *40*, *43*, *56*, *64*, *72*, *80*, *82*, *93*, *94*, *97*, *99*, *100*, *107*, *112*, *134*, *137–141*, *146*, *157*
as_r, 15
as_r(), *11*
asin.mlx *(Math.mlx)*, 29
asinh.mlx *(Math.mlx)*, 29
asplit, 16
asplit(), *16*
atan.mlx *(Math.mlx)*, 29
atanh.mlx *(Math.mlx)*, 29

backsolve *(mlx_solve_triangular)*, 135
backsolve(), *136*
base::aperm(), *100*
base::as.vector(), *13*
base::asplit(), *17*
base::diag(), *23*
base::drop(), *25*
base::kronecker(), *28*
base::matrix(), *96*
base::rowSums(), *20*, *21*, *161*, *162*
base::seq(), *34*, *35*

cbind.mlx, 17
ceiling.mlx *(Math.mlx)*, 29
chol(), *19*
chol.mlx, 18

chol2inv, 19
chol2inv(), *42*
col *(*row*)*, 160
col(), *160*
colMeans, 20
colnames(), *31*
colSums, 21
Conj(), *124*
cos.mlx *(*Math.mlx*)*, 29
cosh.mlx *(*Math.mlx*)*, 29
cospi.mlx *(*Math.mlx*)*, 29
crossprod.mlx, 22
cummax.mlx *(*Math.mlx*)*, 29
cummin.mlx *(*Math.mlx*)*, 29
cumprod(), *56*
cumprod.mlx *(*Math.mlx*)*, 29
cumsum(), *56*
cumsum.mlx *(*Math.mlx*)*, 29

diag, 22
diag.mlx, 23
dim.mlx, 24
dim<-.mlx, 24
dimnames(), *31*
dimnames.mlx *(*mlx-dimnames*)*, 31
dimnames<-.mlx *(*mlx-dimnames*)*, 31
drop, 25

exp.mlx *(*Math.mlx*)*, 29
expm1.mlx *(*Math.mlx*)*, 29

fft, 26
fft(), *73*
floor.mlx *(*Math.mlx*)*, 29
format.mlx_stream, 27

ifelse(), *152*
Im(), *124*
is.na(), *15*
is.nan(), *15*
is.numeric, *162*
is_mlx, 27

kronecker, 28
kronecker,ANY,mlx-method *(*kronecker*)*, 28
kronecker,mlx,ANY-method *(*kronecker*)*, 28
kronecker,mlx,mlx-method *(*kronecker*)*, 28
kronecker.default *(*kronecker*)*, 28
kronecker.mlx *(*kronecker*)*, 28

length.mlx, 29
local_device *(*with_device*)*, 168
local_device(), *14*, *18*, *19*, *66–68*, *83*, *95*, *136*, *156*, *158*, *163*, *166*
log.mlx *(*Math.mlx*)*, 29
log10.mlx *(*Math.mlx*)*, 29
log1p.mlx *(*Math.mlx*)*, 29
log2.mlx *(*Math.mlx*)*, 29

Math.mlx, 29
max.mlx *(*Summary.mlx*)*, 164
mean.mlx, 30
min.mlx *(*Summary.mlx*)*, 164
mlx-dimnames, 31
mlx-methods, *15*, 32
mlx_addmm, 33
mlx_all *(*mlx_sum*)*, 140
mlx_allclose, 33
mlx_allclose(), *10*, *84*
mlx_any *(*mlx_sum*)*, 140
mlx_arange, 34
mlx_arange(), *133*
mlx_argmax, 36
mlx_argmin *(*mlx_argmax*)*, 36
mlx_argpartition *(*mlx_topk*)*, 146
mlx_argpartition(), *137*
mlx_argsort *(*mlx_sort*)*, 136
mlx_array, 37
mlx_array(), *96*, *151*
mlx_batch_norm, 38
mlx_best_device, 38
mlx_binary_cross_entropy, 39
mlx_broadcast_arrays, 40
mlx_broadcast_to, 40
mlx_cast, 41
mlx_cholesky_inv, 42
mlx_cholesky_inv(), *19*
mlx_clip, 43
mlx_compile, 43
mlx_compile(), *99*
mlx_conjugate *(*mlx_real*)*, 124
mlx_contiguous, 45
mlx_conv1d, 49
mlx_conv1d(), *47*
mlx_conv2d, 50
mlx_conv2d(), *48*
mlx_conv3d, 51
mlx_conv3d(), *49*
mlx_conv_transpose1d, 46

mlx_conv_transpose1d(), *48*, *49*
mlx_conv_transpose2d, 47
mlx_conv_transpose2d(), *47*, *49*
mlx_conv_transpose3d, 48
mlx_conv_transpose3d(), *47*, *48*
mlx_coordinate_descent, 52
mlx_cross, 54
mlx_cross_entropy, 55
mlx_cumprod *(*mlx_cumsum*)*, 56
mlx_cumsum, 56
mlx_default_stream *(*mlx_new_stream*)*, 102
mlx_default_stream(), *131*
mlx_degrees, 57
mlx_dequantize, 57
mlx_dequantize(), *113*, *115*
mlx_device, 59
mlx_device(), *11–14*, *16*, *18*, *19*, *35*, *37*, *42*, *66–68*, *72*, *75*, *82*, *83*, *91*, *95*, *103*, *105*, *119*, *136*, *143*, *149*, *150*, *153*, *154*, *156*, *158*, *163*, *166*
mlx_dexp, 59
mlx_diagonal *(*diag.mlx*)*, 23
mlx_disable_compile, 60
mlx_disable_compile(), *45*
mlx_dlnorm, 61
mlx_dlogis, 62
mlx_dnorm, 62
mlx_dropout, 63
mlx_dtype, 64
mlx_dtype(), *41*
mlx_dunif, 65
mlx_eig, 65
mlx_eigh, 66
mlx_eigvals, 67
mlx_eigvalsh, 68
mlx_embedding, 69
mlx_enable_compile *(*mlx_disable_compile*)*, 60
mlx_enable_compile(), *45*
mlx_erf, 69
mlx_erf(), *63*
mlx_erfinv *(*mlx_erf*)*, 69
mlx_erfinv(), *63*
mlx_eval, 70
mlx_eval(), *6*
mlx_expand_dims, 71
mlx_eye, 71
mlx_fft, 72
mlx_fft(), *26*
mlx_fft2 *(*mlx_fft*)*, 72
mlx_fft2(), *26*
mlx_fftn *(*mlx_fft*)*, 72
mlx_fftn(), *26*
mlx_flatten, 73
mlx_flatten(), *8*
mlx_forward, 74
mlx_full, 75
mlx_gather, 76
mlx_gather_qmm, 77
mlx_gather_qmm(), *115*
mlx_gelu, 78
mlx_grad, 79
mlx_hadamard_transform, 80
mlx_has_gpu, 81
mlx_identity, 81
mlx_imag *(*mlx_real*)*, 124
mlx_import_function, 82
mlx_inv, 83
mlx_isclose, 84
mlx_isclose(), *10*, *34*
mlx_isfinite *(*mlx_isnan*)*, 85
mlx_isinf *(*mlx_isnan*)*, 85
mlx_isnan, 85
mlx_isneginf *(*mlx_isposinf*)*, 85
mlx_isposinf, 85
mlx_key, 86
mlx_key(), *86*
mlx_key_bits, 87
mlx_key_split *(*mlx_key*)*, 86
mlx_kron, 87
mlx_l1_loss, 88
mlx_layer_norm, 89
mlx_leaky_relu, 89
mlx_linear, 90
mlx_linspace, 91
mlx_linspace(), *35*, *133*
mlx_load, 92
mlx_load_gguf, 92
mlx_load_safetensors, 93
mlx_logcumsumexp, 93
mlx_logsumexp, 94
mlx_lu, 95
mlx_matrix, 96
mlx_maximum, 97
mlx_mean *(*mlx_sum*)*, 140
mlx_meshgrid, 97

mlx_metal_kernel, 98
mlx_minimum, 99
mlx_moveaxis, 100
mlx_mse_loss, 101
mlx_nan_to_num, 102
mlx_new_stream, 102
mlx_new_stream(), *18, 19*, *42*, *66–68*, *83*, *95*, *103*, *119*, *131*, *136*, *143*, *150*, *156*, *158*, *163*, *166*, *168*
mlx_norm, 103
mlx_ones, 104
mlx_ones_like, 105
mlx_optimizer_sgd, 106
mlx_pad, 106
mlx_pad(), *138*
mlx_param_set_values, 107
mlx_param_values, 108
mlx_parameters, 109
mlx_partition *(*mlx_topk*)*, 146
mlx_partition(), *137*
mlx_pexp *(*mlx_dexp*)*, 59
mlx_plnorm *(*mlx_dlnorm*)*, 61
mlx_plogis *(*mlx_dlogis*)*, 62
mlx_pnorm *(*mlx_dnorm*)*, 62
mlx_prod *(*mlx_sum*)*, 140
mlx_punif *(*mlx_dunif*)*, 65
mlx_put_along_axis, 109
mlx_qexp *(*mlx_dexp*)*, 59
mlx_qlnorm *(*mlx_dlnorm*)*, 61
mlx_qlogis *(*mlx_dlogis*)*, 62
mlx_qnorm *(*mlx_dnorm*)*, 62
mlx_qr_gpu, 110
mlx_quantile, 111
mlx_quantize, 113
mlx_quantize(), *57*, *58*, *77*, *113–115*
mlx_quantized_matmul, 114
mlx_quantized_matmul(), *58*, *78*, *113*
mlx_qunif *(*mlx_dunif*)*, 65
mlx_radians *(*mlx_degrees*)*, 57
mlx_rand_bernoulli, 115
mlx_rand_categorical, 116
mlx_rand_gumbel, 117
mlx_rand_laplace, 117
mlx_rand_multivariate_normal, 118
mlx_rand_normal, 119
mlx_rand_permutation, 120
mlx_rand_randint, 121
mlx_rand_truncated_normal, 122
mlx_rand_uniform, 123
mlx_real, 124
mlx_relu, 124
mlx_repeat, 125
mlx_reshape, 126
mlx_reshape(), *25*
mlx_roll, 126
mlx_save, 127
mlx_save(), *92*
mlx_save_gguf, 128
mlx_save_safetensors, 128
mlx_scalar, 129
mlx_scatter_add_axis, 129
mlx_sd *(*mlx_sum*)*, 140
mlx_sequential, 130
mlx_set_default_stream, 131
mlx_set_training, 131
mlx_shape *(*dim.mlx*)*, 24
mlx_shape(), *24*
mlx_sigmoid, 132
mlx_silu, 132
mlx_slice_update, 133
mlx_slice_update(), *35*
mlx_softmax, 134
mlx_softmax_layer, 134
mlx_solve_triangular, 135
mlx_sort, 136
mlx_split, 138
mlx_split(), *16*, *107*
mlx_squeeze, 138
mlx_squeeze(), *25*
mlx_stack, 139
mlx_stack(), *9*
mlx_std *(*mlx_sum*)*, 140
mlx_stop_gradient, 140
mlx_subset *(*[<-.mlx*)*, 7
mlx_sum, 140
mlx_swapaxes, 142
mlx_synchronize, 143
mlx_take_along_axis, 143
mlx_tanh, 145
mlx_tile, 145
mlx_topk, 146
mlx_trace, 147
mlx_train_step, 148
mlx_tri, 149
mlx_tri_inv, 150
mlx_tril *(*mlx_tri*)*, 149

mlx_triu *(mlx_tri)*, 149
mlx_unflatten, 151
mlx_value_grad *(mlx_grad)*, 79
mlx_var *(mlx_sum)*, 140
mlx_vector, 151
mlx_where, 152
mlx_zeros, 153
mlx_zeros_like, 154

names(), *31*
names.mlx *(mlx-dimnames)*, 31
names<-.mlx *(mlx-dimnames)*, 31

Ops.mlx, 155
outer, 155

pinv, 156
print.mlx, 157
print.mlx_stream, 157
prod.mlx *(Summary.mlx)*, 164

qr.mlx, 158
quantile.mlx *(mlx_quantile)*, 111

rbind.mlx, 159
Re(), *124*
Rmlx *(Rmlx-package)*, 6
Rmlx-package, 6
round.mlx *(Math.mlx)*, 29
row, 160
row(), *160*
rowMeans, 160
rownames(), *31*
rowSums, 161

S3 methods for many base generics, *6*
scale(), *162*
scale.mlx, 162
sign.mlx *(Math.mlx)*, 29
signif.mlx *(Math.mlx)*, 29
sin.mlx *(Math.mlx)*, 29
sinh.mlx *(Math.mlx)*, 29
sinpi.mlx *(Math.mlx)*, 29
solve(), *19*
solve.mlx, 163
sqrt.mlx *(Math.mlx)*, 29
stats::fft(), *26*
stats::quantile(), *111, 112*
stats::sd(), *141*
str.mlx, 164
sum.mlx *(Summary.mlx)*, 164
Summary.mlx, 164
svd, 165
svd.mlx, 166

t.mlx, 167
tan.mlx *(Math.mlx)*, 29
tanh.mlx *(Math.mlx)*, 29
tanpi.mlx *(Math.mlx)*, 29
tcrossprod.mlx, 167
trunc.mlx *(Math.mlx)*, 29

unname(), *31*

with_device, 168
with_device(), *14*, *18, 19*, *66–68*, *83*, *95*, *136*, *156*, *158*, *163*, *166*