Package: anvl (via r-universe)

November 12, 2026

Title Accelerated Array Computing and Automatic Differentiation

Version 0.6.0

Description Accelerated array computing and code transformations for R. Numerical programs operating on arrays can be just-in-time compiled to optimized executables via 'XLA' -- the same compiler that powers 'JAX' and 'TensorFlow' -- and run on CPU or NVIDIA GPU from the same source. Also provides reverse-mode automatic differentiation, returning the gradient of a function as another R function.

License MIT + file LICENSE

URL https://r-xla.github.io/anvl/, https://github.com/r-xla/anvl

BugReports https://github.com/r-xla/anvl/issues

Depends R (>= 4.4.0)

Imports checkmate, cli, fastmap, methods, pjrt (>= 0.7.0), rlang, safetensors (>= 0.2.0), stablehlo (>= 0.6.0), withr, xlamisc (>= 0.5.0)

Suggests bit64, knitr, quarto, rmarkdown, quickr (>= 0.3.0), testthat (>= 3.0.0)

Additional_repositories https://r-xla.r-universe.dev

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

Collate 'aaa.R' 'utils.R' 'array.R' 'device.R' 'default-dtypes.R' 'backend.R' 'jit.R' 'api-distributions.R' 'side-effects.R' 'primitive.R' 'promotion.R' 'rules-inference.R' 'type-converters.R' 'primitives.R' 'api.R' 'api-generics.R' 'api-like.R' 'api-rng.R' 'api-subset.R' 'asserts.R' 'backend-pjrt.R' 'backend-quickr.R' 'bibentries.R' 'box.R' 'graph-passes.R' 'graph.R' 'graph-print.R' 'rules-quickr.R' 'graph-to-quickr.R' 'quickr.R' 'rdata.R' 'reexports.R' 'reverse.R' 'rng.R' 'rng-trace.R' 'roxygen.R' 'rules-reverse.R' 'rules-stablehlo.R' 'serialize.R' 'stablehlo.R' 'zzz.R'

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake make libprotobuf-dev libuv1-dev protobuf-compiler libprotoc-dev

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

Date/Publication 2026-11-10 21:15:29 UTC

RemoteUrl https://github.com/r-xla/anvl

RemoteRef v0.6.0

RemoteSha 032f311f5f1c1b630331c70c1eaf7e5c9753448f

Contents

anvl-package . . . 9
[.AnvlArray . . . 10
[<-.AnvlArray . . . 12
active_backend . . . 13
AnvlArray . . . 14
AnvlBackendPjrt . . . 18
AnvlBackendQuickr . . . 19
AnvlPrimitiveDef . . . 20
apply_promotion . . . 20
arr . . . 21
arrayish . . . 22
as-AnvlArray . . . 23
as_anvl_array . . . 25
as_array.AnvlArray . . . 26
as_dtype . . . 27
as_raw . . . 28
await . . . 29
axes . . . 30
backend . . . 30
common_dtype . . . 31
ConcreteArray . . . 32
current_descriptor . . . 33
current_platform . . . 33
default_device . . . 34
default_dtypes . . . 35
device . . . 36
dim.AnvlArray . . . 37
dtype . . . 37
dtype_categories . . . 38
eq_type . . . 40
gradient . . . 41
graph_desc_add . . . 42
graph_gradient . . . 43
graph_to_quickr_r_function . . . 44
GraphBox . . . 45
GraphDescriptor . . . 46

GraphLiteral . . . 48
GraphNode . . . 48
GraphStatement . . . 49
GraphValue . . . 49
install_anvl . . . 50
IotaArray . . . 51
is_device . . . 52
is_dtype . . . 52
jit . . . 53
jit_cache_size . . . 55
length.AnvlArray . . . 56
LiteralArray . . . 57
local_backend . . . 58
local_default_device . . . 59
local_default_dtypes . . . 60
local_descriptor . . . 61
local_nv_rng_state . . . 62
map_tree . . . 63
naxes . . . 64
new_primitive . . . 64
nv_abs . . . 65
nv_acos . . . 66
nv_acosh . . . 67
nv_add . . . 68
nv_all . . . 69
nv_and . . . 70
nv_any . . . 71
nv_aperm . . . 72
nv_asin . . . 73
nv_asinh . . . 74
nv_atan . . . 75
nv_atan2 . . . 76
nv_atanh . . . 77
nv_aval . . . 78
nv_bind . . . 79
nv_binomial . . . 80
nv_bitcast_convert . . . 83
nv_broadcast_arrays . . . 84
nv_broadcast_scalars . . . 85
nv_broadcast_to . . . 86
nv_cbrt . . . 87
nv_ceiling . . . 87
nv_chol . . . 88
nv_clamp . . . 89
nv_concatenate . . . 90
nv_conv1d . . . 91
nv_conv2d . . . 93
nv_conv3d . . . 94

nv_convert . . . 95
nv_cos . . . 96
nv_cosh . . . 97
nv_cospi . . . 98
nv_crossprod . . . 98
nv_cummax . . . 99
nv_cummin . . . 100
nv_cumprod . . . 102
nv_cumsum . . . 103
nv_det . . . 104
nv_determinant . . . 105
nv_device . . . 106
nv_diag . . . 107
nv_digamma . . . 107
nv_div . . . 108
nv_dtype_constructors . . . 109
nv_eigh . . . 112
nv_eq . . . 113
nv_erf . . . 114
nv_erf_inv . . . 114
nv_erfc . . . 115
nv_exp . . . 116
nv_expm1 . . . 117
nv_exponential . . . 118
nv_extract_diag . . . 119
nv_eye . . . 120
nv_fill . . . 121
nv_flatten . . . 123
nv_floor . . . 123
nv_floor_div . . . 124
nv_gamma . . . 125
nv_ge . . . 126
nv_gt . . . 127
nv_if . . . 128
nv_ifelse . . . 129
nv_inv . . . 130
nv_iota . . . 131
nv_is_finite . . . 132
nv_is_infinite . . . 133
nv_is_nan . . . 134
nv_le . . . 135
nv_lgamma . . . 136
nv_linspace . . . 137
nv_log . . . 138
nv_log10 . . . 139
nv_log1p . . . 140
nv_log2 . . . 141
nv_lower_tri . . . 142

nv_lt . . . 143
nv_lu . . . 144
nv_matmul . . . 145
nv_max . . . 146
nv_mean . . . 147
nv_median . . . 148
nv_min . . . 150
nv_mod . . . 151
nv_mul . . . 152
nv_ne . . . 153
nv_negate . . . 154
nv_normal . . . 155
nv_not . . . 158
nv_or . . . 159
nv_order . . . 160
nv_outer . . . 161
nv_pad . . . 162
nv_plogis . . . 163
nv_pmax . . . 164
nv_pmin . . . 165
nv_poisson . . . 166
nv_popcnt . . . 168
nv_pow . . . 169
nv_print . . . 170
nv_prod . . . 171
nv_promote_to_common . . . 172
nv_psigamma . . . 173
nv_qr . . . 174
nv_quantile . . . 175
nv_range . . . 177
nv_rbinom . . . 178
nv_remainder . . . 179
nv_reshape . . . 180
nv_rev . . . 181
nv_rng_state . . . 182
nv_round . . . 183
nv_rsqrt . . . 184
nv_sample . . . 185
nv_sample_int . . . 186
nv_save . . . 187
nv_scan . . . 188
nv_sd . . . 189
nv_select . . . 190
nv_seq . . . 191
nv_serialize . . . 193
nv_set_seed . . . 194
nv_shift_left . . . 195
nv_shift_right_arithmetic . . . 196

nv_shift_right_logical . . . 197
nv_sign . . . 198
nv_sin . . . 199
nv_sinh . . . 200
nv_sinpi . . . 201
nv_solve . . . 201
nv_sort . . . 203
nv_sqrt . . . 204
nv_squeeze . . . 205
nv_static_slice . . . 206
nv_sub . . . 207
nv_sum . . . 208
nv_svd . . . 209
nv_tan . . . 210
nv_tanh . . . 211
nv_tanpi . . . 212
nv_tcrossprod . . . 213
nv_top_k . . . 214
nv_trace . . . 215
nv_triangular_solve . . . 216
nv_tril . . . 217
nv_triu . . . 218
nv_trunc . . . 219
nv_uniform . . . 220
nv_unsqueeze . . . 222
nv_upper_tri . . . 223
nv_var . . . 224
nv_which_max . . . 225
nv_which_min . . . 227
nv_while . . . 228
nv_xor . . . 229
peek_dtype . . . 230
platform.AnvlArray . . . 231
prim_abs . . . 232
prim_acos . . . 233
prim_acosh . . . 234
prim_add . . . 235
prim_all . . . 236
prim_and . . . 237
prim_any . . . 238
prim_asin . . . 239
prim_asinh . . . 240
prim_atan . . . 241
prim_atan2 . . . 242
prim_atanh . . . 243
prim_bitcast_convert . . . 244
prim_broadcast_in_axes . . . 245
prim_cbrt . . . 246

prim_ceiling . . . 247
prim_chol . . . 248
prim_clamp . . . 249
prim_concatenate . . . 250
prim_convert . . . 251
prim_convolution . . . 252
prim_cos . . . 254
prim_cosh . . . 255
prim_cummax . . . 256
prim_cummin . . . 257
prim_cumprod . . . 258
prim_cumsum . . . 259
prim_custom_gradient . . . 260
prim_digamma . . . 263
prim_div . . . 264
prim_dot_general . . . 265
prim_dynamic_slice . . . 266
prim_dynamic_update_slice . . . 267
prim_eigh . . . 269
prim_eq . . . 270
prim_erf . . . 271
prim_erf_inv . . . 272
prim_erfc . . . 273
prim_exp . . . 274
prim_expm1 . . . 275
prim_fill . . . 276
prim_floor . . . 277
prim_gather . . . 278
prim_ge . . . 281
prim_gt . . . 282
prim_if . . . 283
prim_ifelse . . . 284
prim_iota . . . 285
prim_is_finite . . . 286
prim_le . . . 287
prim_lgamma . . . 288
prim_log . . . 289
prim_log1p . . . 290
prim_lt . . . 291
prim_lu . . . 292
prim_max . . . 293
prim_min . . . 294
prim_mul . . . 295
prim_ne . . . 296
prim_negate . . . 297
prim_not . . . 298
prim_or . . . 299
prim_pad . . . 300

prim_plogis . . . 302
prim_pmax . . . 303
prim_pmin . . . 304
prim_popcnt . . . 305
prim_pow . . . 306
prim_print . . . 307
prim_prod . . . 308
prim_psigamma . . . 309
prim_qr . . . 310
prim_reduce . . . 311
prim_remainder . . . 312
prim_reshape . . . 313
prim_rev . . . 314
prim_round . . . 315
prim_rsqrt . . . 316
prim_sample_bits . . . 317
prim_scan . . . 319
prim_scatter . . . 320
prim_shift_left . . . 323
prim_shift_right_arithmetic . . . 324
prim_shift_right_logical . . . 325
prim_sign . . . 326
prim_sin . . . 327
prim_sinh . . . 328
prim_sort . . . 329
prim_sqrt . . . 330
prim_static_slice . . . 331
prim_sub . . . 333
prim_sum . . . 334
prim_svd . . . 335
prim_tan . . . 336
prim_tanh . . . 337
prim_top_k . . . 338
prim_transpose . . . 339
prim_triangular_solve . . . 340
prim_which_max . . . 342
prim_which_min . . . 343
prim_while . . . 344
prim_xor . . . 345
promotion_rule . . . 346
quickr_device . . . 348
RData . . . 349
rule_reverse . . . 350
shape . . . 351
Shape . . . 352
stablehlo . . . 353
subgraphs . . . 355
to_abstract . . . 355

trace_fn 356
with_dtypes 357

Index **359**

`anvl-package` *anvl: Accelerated Array Computing and Automatic Differentiation*

Description

Accelerated array computing and code transformations for R. Numerical programs operating on arrays can be just-in-time compiled to optimized executables via 'XLA' – the same compiler that powers 'JAX' and 'TensorFlow' – and run on CPU or NVIDIA GPU from the same source. Also provides reverse-mode automatic differentiation, returning the gradient of a function as another R function.

Options

- `anvl.backend` (`character(1)`): the backend every operation runs on. Must be one of `"pjrt"` and `"quickr"`. If not set, `"pjrt"` is used. Also see `active_backend()`, `local_backend()` and `with_backend()`.
- `anvl.default_dtypes` (named `character()` | named `list()`): the data types an R double and an R integer settle on when they meet no typed array. See `default_dtypes()` for more details.
- `anvl.default_device` (`character(1)` | device object): the device a call that names none allocates on. Also see `default_device()`, `local_default_device()` and `with_default_device()`.

Environment variables

- `ANVL_DEFAULT_DTYPES`: `category=dtype` pairs such as `"float=f64,int=i64"`, used for every backend when the `anvl.default_dtypes` option is not set.
- `ANVL_DEFAULT_DEVICE`: a device identifier such as `"cuda"` or `"cuda:0"`, used when the `anvl.default_device` option is not set.

Both are read once, when anvl is loaded, and kept as fallbacks for the options rather than set as them: `getOption("anvl.default_device")` stays `NULL`, and an option set later takes precedence. Changing a variable after anvl is loaded has no effect until the package is loaded again, e.g. in a new R session.

The remaining ones, all prefixed `ANVL_TEST`, affect only anvl's own test suite, not the package:

- `ANVL_TEST`: `tests/testthat.R` runs the tests only when this is `"1"`, so `R CMD check` in a shell without it runs none of them.
- `ANVL_TEST_SKIP_QUICKR`: when set to anything non-empty, the tests that need the quickr backend are skipped – they are comparatively slow.

Third-Party Licenses

The anvl package itself is MIT-licensed. The CUDA backend dynamically loads NVIDIA software which is not bundled with anvl, but downloaded from NVIDIA's official redistributable channels by the CUDA toolkit R package (e.g. pjrt.cuda) at install time. Its use is governed by the NVIDIA CUDA Toolkit EULA, with the exception of cuDNN, which is covered by the NVIDIA cuDNN SLA, and NCCL, which is covered by its own license. By installing or using the CUDA backend you accept those terms.

Author(s)

Maintainer: Sebastian Fischer <seb.fischer@tutamail.com> (ORCID)

Authors:

- Sebastian Fischer <seb.fischer@tutamail.com> (ORCID)
- Daniel Falbel <daniel@posit.co> (ORCID)
- Tomasz Kalinowski <tomasz@posit.co>
- Nikolai German <niko.german@gmail.com> (ORCID)
- Louis Aslett <louis.aslett@durham.ac.uk> (ORCID)

Other contributors:

- Hugh Graham <hugh@belian.earth> (ORCID) [contributor]

See Also

Useful links:

- https://r-xla.github.io/anvl/
- https://github.com/r-xla/anvl
- Report bugs at https://github.com/r-xla/anvl/issues

[.AnvlArray *Subset an Array*

Description

Extracts a subset from an array. You can also use the [operator. Supports R-style indexing including scalar indices (which drop axes), ranges (a:b), array(c(...)) for selecting multiple elements along an axis, and boolean masks.

Usage

```
## S3 method for class 'AnvlArray'
x[...]

nv_subset(x, ...)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
...	Subset specifications, one per axis. Omitted trailing axes select all elements. A boolean mask (an R logical array such as `arr(TRUE, FALSE)`, or an arrayish value of dtype `bool`) selects the elements at the `TRUE` positions. A mask for one axis must have as many elements as the size of that axis. A mask that is the only subscript and has the same shape as `x` selects across the whole array, yielding a 1-D result. Under `jit()`, the values of a mask must be known at compile time, because the number of selected elements determines the output shape: R logical arrays and arrays created in or closed over by the function work, a mask computed from the function's inputs does not. See the Subsetting article for details.

Value

(arrayish)
Has the input's data type, and the shape the specifications select – a scalar index drops its axis, a range, an index array or an axis mask keeps it, and a whole-array mask yields a 1-D result.

See Also

`nv_subset_assign()` for updating subsets, the Subsetting article for a comprehensive guide.

Examples

```
x <- nv_matrix(1:12, nrow = 3)
x
# select row 2
nv_subset(x, 2)
x[2, ]

# select rows 1 to 2, all columns
nv_subset(x, 1:2)
x[1:2, ]

# Select rows 1 and 3 with a mask
x[arr(TRUE, FALSE, TRUE), ]

# Select all elements greater than 6 (not in `jit()`, see above)
x[x > 6]
```

`[<-.AnvlArray` *Update Subset*

Description

Updates elements of an array at specified positions, returning a new array. You can also use the `[<-` operator.

Usage

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

nv_subset_assign(x, ..., value, update_fn = "replace", inplace = FALSE)
```

Arguments

`x`	(`arrayish`) The array to update. Can be any data type. An R object is materialized at its default data type.
`...`	Subset specifications, one per axis. Omitted trailing axes select all elements. A boolean mask (an R logical array such as `arr(TRUE, FALSE)`, or an arrayish value of dtype `bool`) selects the elements at the `TRUE` positions. A mask for one axis must have as many elements as the size of that axis. A mask that is the only subscript and has the same shape as `x` selects across the whole array, yielding a 1-D result. Under `jit()`, the values of a mask must be known at compile time, because the number of selected elements determines the output shape: R logical arrays and arrays created in or closed over by the function work, a mask computed from the function's inputs does not. See the Subsetting article for details.
`value`	(`arrayish`) Replacement values, or with another `update_fn` the values to combine the subset with. Scalars are broadcast to the subset shape and non-scalar values must match it. It is brought to `x`'s data type: an R value is built at it when its category can reach it (`0L` serves an integer and a float `x` alike, while `1.5` into an `i32` `x` is an error), and an array is converted unless that would narrow it or leave its category (an `f64` value for an `f32` `x` is an error).
`update_fn`	(`character(1)` | `function`) How `value` combines with the subset of `x`: • `"replace"` (the default): `value` replaces it, • `"add"`, `"sub"`, `"mul"`: it becomes the subset plus, minus or times `value`, • `"max"`, `"min"`: it becomes the larger or the smaller of the two, or a binary function `f(old, new)` of two scalars, see `prim_scatter()`. Unlike `x[i] <- x[i] + value` in R, a position that `i` selects more than once gets every one of its values: with `"add"`, `x[array(c(1L, 1L))]` adds both to `x[1]`.

gradient() differentiates the names, but not a function; with "mul", not with respect to value where i selects several positions of an axis by an index array or a mask, since those may repeat. With [<-, pass it among the subscripts, e.g. x[1, update_fn = "add"] <- 1.

inplace (logical(1))
Whether to write into the memory of x instead of allocating a new array. This avoids the copy of x that an eager subset assignment otherwise makes, but it consumes x: its buffer is donated, so x – and any other R variable referring to the same array – can no longer be used afterwards. With [<-, pass it among the subscripts, e.g. x[1, inplace = TRUE] <- 0, which rebinds x to the result. It is an error inside a jitted function: there, other references to x would stay valid, so the same code would behave differently in eager and in jit mode. Inside jit(), the compiler already avoids the copy anyway. Default is FALSE.

Value

(arrayish)
Has x's data type and shape, with the subset replaced (or combined with value as update_fn says).

See Also

nv_subset(), the Subsetting article for a comprehensive guide.

Examples

```
x <- nv_matrix(1:12, nrow = 3)
# set row 1 to zeros
nv_subset_assign(x, 1, value = nv_scalar(0L))
x[1, ] <- nv_scalar(0L)
x

# Zero out every element greater than 6 (not in `jit()`, see `nv_subset()`)
x[x > 6] <- 0L
x

# add to the subset instead of replacing it; a repeated index adds twice
nv_subset_assign(x, array(c(1L, 1L)), 1, value = nv_scalar(10L), update_fn = "add")

# write into the memory of `x` instead of copying it
x[2, inplace = TRUE] <- nv_scalar(1L)
x
```

---

active_backend *Get Active Backend*

Description

Retrieves the active backend (option anvl.backend), falling back to the default "pjrt" backend.

Usage

```
active_backend()
```

Value

(character(1))
The backend name (e.g. "pjrt", "quickr").

See Also

local_backend(), with_backend(), default_dtypes()

Examples

```
active_backend()
```

AnvlArray *AnvlArray*

Description

The main array object. Its type is determined by a data type and a shape and lives on a device, which can be a CPU or a GPU.

Usage

```
nv_array(data, shape = NULL, dtype = NULL, device = NULL, byrow = FALSE)

nv_scalar(data, dtype = NULL, device = NULL)

nv_matrix(
  data,
  nrow = NULL,
  ncol = NULL,
  dtype = NULL,
  device = NULL,
  byrow = FALSE
)

nv_empty(shape, dtype, device = NULL)

nv_array_like(like, data, shape = NULL, dtype = NULL, device = NULL)

nv_scalar_like(like, data, dtype = NULL, device = NULL)

nv_empty_like(like, shape = NULL, dtype = NULL, device = NULL)
```

Arguments

data	(any) integer(), double(), or logical() scalar, vector, or array. Alternatively a raw() vector holding the native little-endian byte payload of prod(shape) elements of dtype; both dtype and shape are then required (only supported on the "pjrt" backend). Raw payloads are read in column-major element order, or row-major with byrow = TRUE. An existing AnvlArray is returned unchanged if the shape, dtype and device given agree with it, and is an error otherwise.
shape	(NULL | integer()) The output shape of the array. The default (NULL) is to infer it from the data if possible. Note that nv_array interprets length 1 vectors as having shape (1). Empty data has no shape to infer – 0, c(2, 0) and c(0, 3) all hold no elements – so shape is required there. To create a "scalar" with no axes (shape ()), use nv_scalar or explicitly specify shape = integer().
dtype	(NULL | character(1) | DataType) The data type at which to create the array: a xlamisc::DataType or one of bool, i8, i16, i32, i64, ui8, ui16, ui32, ui64, f32, f64. A value it cannot hold at all is an error (nv_array(3e9, dtype = "i32") overflows). A double at an integer data type is truncated. The default (NULL) uses the default data type of data's category. nv_empty(), which has no data, requires it.
device	(NULL | character(1) | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or • a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error. The default (NULL) uses default_device().
byrow	(logical(1)) When constructing from an R object and the result has at least two axes, fill the array in row-major order rather than the default column-major order. Only allowed when data is an R object — passing an existing AnvlArray together with byrow = TRUE is an error.
nrow	(NULL | integer(1)) Number of rows. Inferred from ncol and the data length if NULL. Defaults to 1 when data is a scalar.
ncol	(NULL | integer(1)) Number of columns. Inferred from nrow and the data length if NULL. Defaults to 1 when data is a scalar.
like	(AnvlArray | GraphBox) An existing array; an R value is an error. Any of shape (which nv_scalar_like() does not take), dtype and device that are NULL (the default) are taken from like.

Value

(AnvlArray)

Terminology

An array's **axes** are the indices that identify its directions, numbered 1, 2, 3, ... The **size** of an axis (its *axis size*) is the extent along that axis, and the **shape** is the vector of all axis sizes. For example, nv_array(1:6, shape = c(2, 3)) has two axes; the size of axis 1 is 2 and the size of axis 2 is 3, so its shape is c(2, 3). Use naxes() for the number of axes, axes() for the axis indices, and shape() for the axis sizes. We speak of the *size of an axis* rather than an array's "dimensions", as the latter is generally overloaded as it is used to refer to both the axis and its size.

Extractors

The following generic functions can be used to extract information from an AnvlArray:

- dtype(): Get the data type of the array.
- shape(): Get the shape (axis sizes) of the array.
- naxes(): Get the number of axes.
- axes(): Get the axis indices.
- device(): Get the device of the array.
- platform(): Get the platform (e.g. "cpu", "cuda").

Serialization

Arrays can be serialized to and from the safetensors format:

- nv_save() / nv_read(): Save/load arrays to/from a file.
- nv_serialize() / nv_unserialize(): Serialize/deserialize arrays to/from raw vectors.

An array can also be saved like any R object, e.g. with saveRDS(). Its device is not saved: an array of the pjrt backend loads onto the default device at the time it is read, so an array saved on a GPU also loads on a machine without one. anvl arranges this when it is loaded, so load anvl before reading arrays.

Backend

An AnvlArray is backend-dependent: it belongs to exactly one backend ("pjrt" or the experimental "quickr") and lives on a device of that backend. The supported data types and devices differ between backends.

Missing values

XLA, the compiler that is used by the "pjrt" (the default) backend has no notion of a missing (NA) value. When creating a new AnvlArray, the input is therefore checked for the presence of such values. NAs are always rejected, except when:

1. Creating float arrays where we convert the NA to NaN.

2. When creating an `i32` from an R `integer()`. There, we throw a warning, but the resulting `AnvlArray` gets the bit representation of `NA_integer_`, which is `INT_MIN` (`-2147483648`). Disallowing this would prevent round-trips between the data types.

See the Gotchas article for more information.

Out of Range values

Because base R has fewer data types than anvl, creating `AnvlArrays` from R often involves type conversions. When such conversions are performed, anvl performs a scan of the inputs to ensure that the requested data type can actually hold the input data. For example, trying to create an unsigned integer from a negative R `integer()` fails. The same holds where an R value takes its data type from the array it meets rather than from an argument: `nv_scalar(1L, "ui8") + (-2L)` is refused, where converting an array with `nv_convert()` wraps around.

See Also

nv_fill, nv_iota, nv_seq, as_array, nv_serialize

Examples

```
# a 1-d array (vector) with shape (4), at the default data type for integers
nv_array(1:4)

# specify a dtype
nv_array(c(1.5, 2.5, 3.5), dtype = "f64")

# a 2x3 matrix
nv_array(1:6, shape = c(2L, 3L))

# a 2x3 matrix filled by row, like `matrix(1:6, 2, 3, byrow = TRUE)`.
nv_array(1:6, shape = c(2L, 3L), byrow = TRUE)

# a scalar array
nv_scalar(3.14)

# an uninitialized 2x3 array (contents are unspecified)
nv_empty(shape = c(2L, 3L), dtype = "f32")

# --- Extractors ---
x <- nv_array(1:6, shape = c(2L, 3L))
dtype(x)
shape(x)
naxes(x)
device(x)
platform(x)

# --- Transforming arrays with jit ---
add_one <- jit(function(x) x + 1L)
add_one(nv_array(1:4))

# --- Eager mode (calling operations directly) ---
```

```
nv_add(nv_array(1:3), nv_array(4:6))
```

`AnvlBackendPjrt` *PJRT Backend*

Description

Constructs the PJRT backend, which stores array data in PJRT buffers (via `pjrt::pjrt_buffer()`) and compiles jitted functions to XLA executables via `stablehlo()` and `pjrt::pjrt_compile()`. This is the default backend.

Usage

```
AnvlBackendPjrt()
```

Value

(`AnvlBackend`)
With subclass `"AnvlBackendPjrt"`.

Data representation

An `AnvlArray` with `backend = "pjrt"` wraps a `pjrt::pjrt_buffer()` stored in the `$data` field. The buffer owns the memory holding the array values and may live on any device supported by PJRT (CPU, CUDA, ...). Calling `as_array()` transfers the buffer contents back to an R array; calling `nv_array()` on an R object uploads it to the requested device.

Each `AnvlArray` therefore has an associated device, queryable via `device()`. A device is a `pjrt::as_pjrt_device()` object (e.g. the platform `"cpu"` or `"cuda"`, optionally with an index such as `"cuda:1"`). When `device` is `NULL` in `nv_array()` or the `jit()` wrapper, the device defaults to `default_device()`, or is inferred from the existing inputs of a jitted call. Operations require all inputs to live on the same device.

Supported data types

`bool`; the signed integers `i8`, `i16`, `i32` and `i64`; the unsigned integers `ui8`, `ui16`, `ui32` and `ui64`; and the floats `f32` and `f64`. An R double materializes at `f32` on this backend and an R integer at `i32` unless the defaults say otherwise (see `default_dtypes()`).

Floating-point behavior

Subnormal floating-point values may be preserved when stored in an array and read back into R, yet treated as zero in calculations. On CPUs, XLA enables a mode that replaces subnormal inputs and results with zero. The exact behavior depends on the platform, backend, and operation.

See the Gotchas article for an explanation and examples.

See Also

`AnvlBackend()`, `AnvlBackendQuickr()`, `local_backend()`, `jit()`.

`AnvlBackendQuickr` *Quickr Backend*

Description

Constructs the quickr backend, which stores array data as plain R arrays and compiles jitted functions to R code via the **quickr** package.

Usage

```
AnvlBackendQuickr()
```

Details

Registered automatically under the name `"quickr"` when the package is loaded; call `local_backend("quickr")` or `with_backend("quickr", ...)` to use it. Requires the quickr package to be installed.

Value

(`AnvlBackend`)
With subclass `"AnvlBackendQuickr"`.

Data representation

An `AnvlArray` with `backend = "quickr"` is, under the hood, a plain R vector or array (`numeric`, `integer`, or `logical`) stored in the `$data` field. `as_array()` returns the underlying vector/array directly without copying, and `nv_array()` simply wraps an R vector/array. Data always lives in R's memory and computation always runs on the CPU, so the only device is `quickr_device("cpu")`; every array still carries it in `$device`.

Status

This backend is **experimental** and has a number of limitations:

- Compilation (tracing + quickr lowering) is somewhat slow, so it is best suited to long-running or repeatedly-called functions where the one-time compilation cost is amortized.
- Only a subset of the primitives that the PJRT backend supports are currently lowered to quickr code. See the Primitives Reference article for an overview.
- Only CPU execution is supported.
- Only three data types are supported; see the section below.

Supported data types

`f64`, `i32` and `bool` – the three R storage types.

See Also

`AnvlBackend()`, `AnvlBackendPjrt()`, `local_backend()`, `jit()`.

AnvlPrimitiveDef *Primitive Definition*

Description

The definition of a primitive: its name, sub-graph parameters and interpretation rules. It is not callable; the function a primitive is called through is an AnvlPrimitive, which carries its AnvlPrimitiveDef as attr(<fn>, "definition"). The graph records the AnvlPrimitiveDef in each GraphStatement. Note that [[and [[<- access the interpretation rules. To access other fields, use $ and $<-.

A primitive is considered higher-order if it has subgraphs.

Usage

```
AnvlPrimitiveDef(name, subgraphs = character())
```

Arguments

name	(character(1)) The name of the primitive, without the prim_ prefix.
subgraphs	(character()) Names of parameters that are subgraphs.

Value

(AnvlPrimitiveDef)

apply_promotion *Bring a Primitive's Operands to One Data Type*

Description

Applies a promotion rule to the operands of a primitive, inside the primitive's own body.

A primitive promotes nothing on its own: an R value among its operands would materialize at its own default, so whether a call worked would depend on whether the array it met happened to be at that default. A primitive whose operands must *agree* says so with this, on the same list it goes on to hand graph_desc_add():

```
function(lhs, rhs) {
 operands <- apply_promotion(list(lhs = lhs, rhs = rhs), promotion_rdata_common())
  graph_desc_add(self, operands, infer_fn = infer_fn)[[1L]]
}
```

promotion_rdata_common() is the rule for it: an operand that has a data type keeps it, and an R value takes the one the others have, within its own category. That is what makes prim_mul(x_f64, 2) work whatever x's data type is, while keeping a primitive from widening the array it was handed – promotion across categories is the nv_* layer's job.

Usage

```
apply_promotion(operands, promote)
```

Arguments

operands	(list()) The operands, named as the primitive's graph_desc_add() call names them.
promote	(function) The rule to apply; see promotion_rule.

Details

Pass only the operands that must agree, and name them as the graph_desc_add() call names them. prim_ifelse() promotes its two branches and leaves test a bool; prim_scatter() promotes x and update and leaves the indices alone. A primitive with one arrayish operand, or with deliberately heterogeneous ones (prim_sort()'s payload, prim_while()'s loop state), calls this not at all.

Call it before the body uses the operands for anything else, so it sees settled data types throughout: prim_reduce() reads dtype(init) to trace its reducer and prim_scatter() builds its update computation's parameter slots from peek_dtype(), both before recording a call.

It is idempotent: once every operand is at the data type the rule names, materializing them again changes nothing.

Value

(list())
operands, each materialized at the data type the rule named for it.

See Also

promotion_rule, new_primitive(), the Adding a Primitive article

Examples

```
# an R value takes the data type of the operand it meets
operands <- apply_promotion(list(lhs = nv_scalar(1, "f64"), rhs = 2), promotion_rdata_common())
dtype(operands$rhs)
```

---

arr *Create an R Array*

Description

Create an R array without having to wrap data in c()

Usage

```
arr(..., shape = NULL)
```

Arguments

`...`	(atomic vectors) Values of the new array. They are combined with `c()`, so vectors are spliced in and the result takes the common R type of all values, and fill the array in column-major order. A single value is recycled to fill all of shape.
`shape`	(`NULL` | `integer()`) Shape of new array. If `NULL` (default), uses length of elements to create a 1D array.

Value

(array)

Examples

```
arr(1, 2, 3)
arr(1, 2, 3, 4, shape = c(2, 2))
# vectors are spliced in
arr(1:3, 4L)
# a single value fills the whole shape
arr(0, shape = c(2, 3))
```

---

arrayish *Array-Like Objects*

Description

An `arrayish` value is anything that represents an `AnvlArray` or can be converted to one.

Specifically, these values are `arrayish`:

- `AnvlArray`: a concrete array holding data on a device.
- `GraphBox`: the representation of an array during tracing, e.g. inside a `jit()`ted function.
- R objects:
  - `numeric(1)` and `logical(1)` which represent scalars.
  - `numeric` and `logical` R arrays.

Use `is_arrayish()` to check whether a value is arrayish.

Usage

```
is_arrayish(x, convert_ok = TRUE)
```

Arguments

x	(any) Object to check.
convert_ok	(logical(1)) Whether to accept numeric(1) and logical(1) and R arrays of type numeric and logical.

Value

(logical(1))
Whether x is arrayish.

See Also

AnvlArray, GraphBox

Examples

```
# AnvlArray objects are arrayish
is_arrayish(nv_array(1:4))

# R arrays and literals are arrayish by default
is_arrayish(array(1:4), convert_ok = TRUE)
is_arrayish(array(1:4), convert_ok = FALSE)

# length 1 vectors
is_arrayish(1.5, convert_ok = FALSE)
is_arrayish(1.5, convert_ok = TRUE)
```

---

as-AnvlArray *Coerce AnvlArray to an R Vector*

Description

Convert an AnvlArray to a flat R vector, discarding the array's shape. Each method requires a compatible dtype:

- as.double() / as.numeric(): float or (signed/unsigned) integer dtypes.
- as.integer(): signed or unsigned integer dtypes.
- bit64::as.integer64(): signed or unsigned integer dtypes. This is how to read i64, ui64 and ui32 values that an R integer cannot hold. It is lossless for i64 and ui32, but bit64::integer64 is itself signed, so a ui64 value >= 2^63 wraps to a negative one (exactly 2^63 becomes NA); that is warned about, and check = "err" makes it an error.
- as.logical(): bool.

- `as.vector()`: any dtype; the R type is chosen by the dtype. For the dtypes R has no native type for (`i64`, `ui64`, `ui32`) that is the `bit64::integer64` `as_array()` returns, since a bare double could not hold the values – and it keeps its class, so `is.vector()` is `FALSE` for it.

Use `as_array()` to obtain an R array that preserves the shape, or `nv_convert()` to change the dtype of an `AnvlArray` before coercing. `as.vector()`'s signature is fixed by the generic, so it takes no `check` argument and always reports at the default level; call `as_array()` directly to pick another one.

Usage

```
## S3 method for class 'AnvlArray'
as.double(x, check = "warn", ...)

## S3 method for class 'AnvlArray'
as.integer(x, check = "warn", ...)

## S3 method for class 'AnvlArray'
as.integer64(x, check = "warn", ...)

## S3 method for class 'AnvlArray'
as.logical(x, check = "warn", ...)

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

Arguments

`x`	(`AnvlArray`) Array to coerce.
`check`	(`character(1)` | `FALSE`) Forwarded to `as_array()`; see there for details.
`...`	Unused.
`mode`	(`character(1)`) Must be `"any"` (the default), meaning the natural R type for the array's dtype. Only present because `base::as.vector()`'s signature requires it; pick an R type with one of the other methods instead.

Value

(`double()` | `integer()` | `logical()` | `bit64::integer64`)
The elements of `x` in column-major order, without a shape.

Examples

```
x <- nv_array(c(1.5, 2.5, 3.5, 4.5), shape = c(2L, 2L))
as.numeric(x)
as.integer(nv_array(1:6, shape = c(2L, 3L)))
bit64::as.integer64(nv_array(1:6, shape = c(2L, 3L), dtype = "i64"))
```

```
as.logical(nv_array(c(TRUE, FALSE), dtype = "bool"))
as.vector(x)
```

`as_anvl_array` *Convert to AnvlArray*

Description

Use this to canonicalize inputs at the start of a function so it works both with eager executing and in combination with `jit()`. Use `as_anvl_array()` for a single input and `as_anvl_arrays()` for multiple inputs. The latter will also ensure all arrays are from the same backend and live on the same device. Both take a `.promote` rule saying which data type the input is brought to.

Usage

```
as_anvl_array(x, device = NULL, .promote = NULL)

as_anvl_arrays(..., .promote = NULL)
```

Arguments

`x`	(`arrayish`) Input to canonicalize.
`device`	(`NULL` | `device`) Target device. If `x` is an `AnvlArray` on a different device, an error is raised.
`.promote`	(`NULL` | `function`) Which data type every input is brought to. See `promotion_rule` for more information. A rule materializes an R value *at* its answer rather than converting it afterwards, so it keeps every digit.
`...`	(`arrayish`) Inputs to align.

Value

`as_anvl_array()`: (`AnvlArray` | `GraphBox`)
An `AnvlArray` in eager code and a `GraphBox` inside a trace.

`as_anvl_arrays()`: (`list`)
One such value per input, named like `...`.

See Also

`peek_dtype()`, `nv_promote_to_common()`

Examples

```
as_anvl_array(1L)
# a rule builds the R value at the data type it names
as_anvl_array(1, .promote = promotion_dtype("f64"))
as_anvl_arrays(nv_array(1:3), 1L)
as_anvl_arrays(nv_array(1L), nv_array(1.5), .promote = promotion_common())
```

---

as_array.AnvlArray *Convert to an R Array*

Description

Transfers array data to R and returns it as an R `array`. Only in the case of scalars is the result a vector of length 1, as R `arrays` cannot have 0 axes.

Usage

```
## S3 method for class 'AnvlArray'
as_array(x, check = "warn", ...)

as_array(x, ...)
```

Arguments

`x`	(`arrayish`) An array-like object.
`check`	(`character(1)` | `FALSE`) How to report a materialized value that the R type cannot hold: `"warn"` (the default) warns and returns it anyway, `"err"` aborts, and `FALSE` skips the scan. Problematic values are the bit-representations reserved for R's `NA` values, as well as out-of-range values. See the Gotchas article for more.
`...`	Passed on to methods.

Details

This is implemented via the generic `xlamisc::as_array()`.

Value

(`array` | `vector(1)`)
An R array with the input's shape, or – for a scalar, which R cannot represent as an array – a vector of length 1.

Data types

R has fewer data types than anvl, so the values are converted to the R type that can represent them:

Data type	R type
`f32`, `f64`	`double`
`i8`, `i16`, `i32`, `ui8`, `ui16`	`integer`
`i64`, `ui32`, `ui64`	`bit64::integer64`
`bool`	`logical`

This has two consequences:

- An `f32` value is widened to a `double` and keeps the rounding error of the 32-bit float, e.g. `as_array(nv_scalar(0.1))` is not exactly `0.1`.
- Some integer values cannot be represented in R: an `i32` or `i64` value equal to the smallest representable integer is read as `NA`, and a `ui64` value `>= 2^63` wraps to a negative number. The `check` argument decides whether this is reported.

To obtain a different R type, convert the array with `nv_convert()` first, or use the coercion functions described in `as.double()`.

Examples

```
x <- nv_array(1:4, dtype = "f32")
as_array(x)
y <- nv_scalar(1L)
# R arrays can't have 0 axes:
as_array(y)
```

---

as_dtype *Convert to a DataType*

Description

Coerces a value to a `DataType`. Accepts data type strings (e.g. `"f32"`, `"i64"`, `"bool"`) or existing `DataType` objects (they are returned unchanged).

Usage

```
as_dtype(x)
```

Arguments

`x`	A character string or `DataType` to convert.

Details

This is implemented via the generic `xlamisc::as_dtype()`.

Value

(`DataType`)

See Also

`is_dtype()`, `xlamisc::as_dtype()`, `xlamisc::DataType`

Examples

```
as_dtype("f32")
as_dtype("i32")
```

`as_raw` *Convert an Array to a Raw Vector*

Description

Returns the underlying bytes of an array as a raw vector.

Usage

```
as_raw(x, ...)
```

Arguments

`x`	(`AnvlArray`) An array.
`...`	Additional arguments passed to method: • `row_major` (`logical(1)`) Whether to write the elements in row-major order. The default, `FALSE`, writes them in column-major order, the order in which R stores an array.

Details

This is implemented via the generic `xlamisc::as_raw()`.

Value

(`raw`)

Examples

```
x <- nv_array(1:4, shape = c(2, 2), dtype = "f32")
# column-major, the default
as_raw(x)
as_raw(x, row_major = TRUE)
```

`await` *Block Until an Async Operation Completes*

Description

Block until the array's underlying computation has finished, and return the object invisibly. Useful for benchmarking, where the dispatch of an asynchronous operation should not be confused with its execution.

Usage

```
await(x, ...)
```

Arguments

`x`	(`AnvlArray` or other awaitable) An object with an `await()` method.
`...`	Additional arguments passed to methods (unused).

Details

Implemented via the generic `pjrt::await()`. For backends without asynchronous execution (e.g. `"quickr"`), this is a no-op.

Value

(any)
`x`, invisibly.

See Also

`pjrt::await()`, `map_tree()` (to await a tree of outputs)

Examples

```
x <- nv_array(1:4, dtype = "f32")
await(x)

# await all leaves of a (possibly nested) list of arrays
map_tree(list(x, list(y = x)), await)
```

axes *Get the Axes of an Array*

Description

Returns the axis indices of an array, i.e. `seq_len(naxes(x))`.

Usage

```
axes(x)
```

Arguments

x	(arrayish) An array-like object.

Value

`(integer())`

See Also

`naxes()`, `shape()`

Examples

```
x <- nv_array(1:6, shape = c(2, 3))
axes(x)
```

backend *Get Backend Name of an Array*

Description

Returns the name of the backend an array or device belongs to.

Usage

```
backend(x, ...)
```

Arguments

x	(AnvlArray | device object) An array or a device (see `nv_device()`).
...	Unused.

Value

(character(1))
The backend name.

Examples

```
backend(nv_array(1:3))
backend(nv_device("cpu"))
```

---

common_dtype *Common Data Type*

Description

Compute the common data type of two data types.

Two integer data types meet at one that holds every value of both: a signed and an unsigned one at the narrowest signed data type wide enough for the unsigned side (`ui8` and `i8` at `i16`, `ui32` and `i32` at `i64`). `ui64` is the exception – no signed data type holds it, and an integer does not become a float on its own – so `ui64` and a signed integer have no common data type and the pair is an error. Convert one side with `nv_convert()` to decide what they meet at.

See the Data Types and Promotion Rules article for more information.

Usage

```
common_dtype(lhs_dtype, rhs_dtype)
```

Arguments

`lhs_dtype`, `rhs_dtype`
(`xlamisc::DataType`)
The two data types.

Value

(`xlamisc::DataType`)
The narrowest common data type.

Examples

```
common_dtype("i32", "f32")
common_dtype("i32", "i64")
try(common_dtype("ui64", "i8"))
```

`ConcreteArray` *Concrete Array Class*

Description

An `AbstractArray` that also holds a reference to the actual array data. Usually represents a closed-over constant in a program. Inherits from `AbstractArray`.

Usage

```
ConcreteArray(data)
```

Arguments

`data`	(`AnvlArray`) The actual array data.

Value

(`ConcreteArray`)

Lowering

When lowering to XLA, these become inputs to the executable instead of embedding them into programs as constants. This is to avoid increasing compilation time and bloating the size of the executable.

Examples

```
y <- nv_array(c(0.5, 0.6))
x <- ConcreteArray(y)
x
shape(x)
naxes(x)
dtype(x)

# how it appears during tracing
graph <- trace_fn(function() y, list())
graph
graph$outputs[[1]]$aval
```

current_descriptor *Get the Current Graph*

Description

Get the current graph being built.

Usage

```
current_descriptor(silent = FALSE)
```

Arguments

silent	(logical(1)) Whether to return NULL if no graph is currently being built (as opposed to aborting).

Value

(GraphDescriptor | NULL)
NULL only when silent = TRUE and no graph is being built.

current_platform *Current Lowering Target Platform*

Description

Returns the target platform currently set by an enclosing stablehlo() call (e.g. "cpu", "cuda"). Platform-aware lowering rules call this to branch on the target — e.g. SVD switches to a layout-flip variant when targeting CUDA with m < n because cuSOLVER's gesvd requires m >= n. Returns NULL outside of a lowering call.

local_platform() sets the current platform for the duration of the calling scope, restoring the previous value via withr::defer() when the scope exits. Useful in tests and for manually exercising platform-aware lowering rules outside of a stablehlo() call.

Usage

```
current_platform()

local_platform(platform, envir = parent.frame())
```

Arguments

platform	(character(1)	NULL) Target platform name (e.g. "cpu", "cuda"), or NULL to clear it.
envir	(environment) Environment whose exit triggers restoration of the previous platform.

Value

(NULL | character(1))
The current platform. local_platform() invisibly returns the previous platform.

See Also

stablehlo()

Examples

```
current_platform()
f <- function() {
  local_platform("cuda")
  current_platform()
}
f()
```

---

default_device *Get the Default Device*

Description

Returns the default device of the active backend: the device the anvl.default_device option names (see local_default_device()), else the one the ANVL_DEFAULT_DEVICE environment variable names (e.g. ANVL_DEFAULT_DEVICE=cuda, read once when anvl is loaded), else the first CPU device.

Usage

```
default_device(backend = NULL)
```

Arguments

backend	(NULL | character(1)) Backend. Defaults to active_backend() when NULL.

Value

(device object)
Backend-specific.

See Also

nv_device(), active_backend(), local_default_device()

Examples

```
default_device()
```

default_dtypes *Default Data Types*

Description

The default data types for the active backend. They are the data types an R value settles on when it meets no typed array, e.g. `nv_array(1)` or `prim_exp(1)`.

`default_dtypes()` reports both categories at once; `default_float()` and `default_int()` report one each.

Each backend registers its own – `f32` / `i32` for `"pjrt"`, `f64` / `i32` for `"quickr"` – and the `anvl.default_dtypes` option overrides them. `local_default_dtypes()` and `with_default_dtypes()` set the option for a scope.

The option maps the categories `float` and `int` to data types, e.g. `c(float = "f64", int = "i64")`, and applies to every backend. It may name only one of them, like `c(float = "f64")`, in which case the other category keeps the backend's default. To set the defaults of a single backend, give a list with an entry named after that backend instead: `list(float = "f64", pjrt = list(int = "i64"))` sets `f64` for every backend and additionally `i64` for `"pjrt"`. An entry that names a backend wins over the categories beside it.

When the option is not set, the `ANVL_DEFAULT_DTYPES` environment variable (read once, when anvl is loaded) is used instead, written as `category=dtype` pairs that apply to every backend, e.g. `ANVL_DEFAULT_DTYPES="float=f64,int=i64"`.

Usage

```
default_dtypes()

default_float()

default_int()
```

Value

`default_dtypes()`: (named `list`)
Elements `float` and `int`, each a `DataType`.

`default_float()`, `default_int()`: (`DataType`)
The default of one category.

See Also

`local_default_dtypes()`, `with_default_dtypes()`, `with_dtypes()`

Examples

```
with_backend("quickr", default_dtypes())
with_backend("pjrt", default_dtypes())
default_dtypes()
```

```
default_float()
default_int()
dtype(nv_array(1.5))
```

`device` *Get the Device of an Array*

Description

Returns the device on which an array is allocated.

Usage

```
device(x, ...)
```

Arguments

`x`	(`arrayish`) An array-like object.
`...`	Additional arguments passed to methods (unused).

Details

This is implemented via the generic `xlamisc::device()`.

Value

(device object)
Backend-dependent device object. One of:

- `PJRTDevice`
- `quickr_device`

Examples

```
x <- nv_array(1:4, dtype = "f32")
device(x)
```

dim.AnvlArray *Shape of an Array*

Description

The shape of an array, i.e. its axis sizes – the generic `base::dim()` on an anvl array, and the same thing as shape().

Unlike base R, it also has a value for an array with a single axis, where `dim()` on an R vector is `NULL`.

Usage

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

Arguments

`x`	(`arrayish`) Input array.

Value

`(integer())`

See Also

length(), shape()

Examples

```
dim(nv_matrix(1:6, nrow = 2))
dim(nv_array(1:3))
```

dtype *Get the Data Type of an Array*

Description

Returns the data type of an array (e.g. `f32`, `i64`).

Usage

```
dtype(x, ...)
```

Arguments

x	(AnvlArray | GraphBox | AbstractArray) An array. See Details for the values that have no data type.
...	Additional arguments passed to methods (unused).

Details

An R value has no data type of its own: it only takes one when it meets a typed array, or when it materializes at the default. So `dtype()` is an error for a plain R value, and also for an RData, the AbstractArray of an R value passed to a jit-compiled function, and so for the GraphBox that carries one during tracing. Use `peek_dtype()` for the data type such a value would take.

This is implemented via the generic `xlamisc::dtype()`.

Value

(DataType)

See Also

`xlamisc::dtype()`, `peek_dtype()`, RData

Examples

```
x <- nv_array(1:4, dtype = "f32")
dtype(x)
# an R value passed to a jit-compiled function has no data type
try(jit(dtype)(1))
# an array passed to it does
jit(function(x) {
  print(dtype(x))
  x
})(nv_scalar(1))
```

---

dtype_categories *Data Type Categories*

Description

For promotion, every data type belongs to one of three categories, ordered boolean < integer < float:

- **boolean** – `bool`
- **integer** – `i8`, `i16`, `i32`, `i64` and their unsigned counterparts `ui8`, `ui16`, `ui32`, `ui64`
- **float** – `f32` and `f64`.

These are the categories promotion works in, where signed and unsigned integers count as one. `xlamisc::dtype_category()` reports a finer split that names `int` and `uint` separately.

Data Type Vocabulary

Where a page says which data types an argument takes, it names a group of them with a single word:

- *any data type* – all of them: `bool`, the signed and unsigned integers, and the floats.
- *numeric* – signed integer, unsigned integer and float.
- *integer* – signed and unsigned integer.
- *integerish* – boolean and integer, signed or unsigned.
- *signed numeric* – signed integer and float.

Data Types for R Values

An R value has no data type of its own. Within its own category, it takes the data type of the array it meets, and is built at it directly rather than converted to it, which is what keeps `nv_scalar(1, "f64") / sqrt(2)` exact. Where it meets nothing, it settles on the default of its category, which `default_dtypes()` reports and the `anvl.default_dtypes` option configures. `peek_dtype()` reports the data type a given R value would take.

The same defaults are used wherever a function needs a data type that its inputs do not give it. `nv_exp()` computes an integer input at the default float, and `nv_add()` of an integer array and a double R value promotes to the default float.

The primitives require operands that have a data type to agree on it; the `nv_*` functions promote them to a common one.

See Also

`default_dtypes()`, `common_dtype()`, `nv_promote_to_common()`, `nv_convert()`, the Data Types and Promotion Rules article

Examples

```
x <- nv_array(1:3, dtype = "i16")
# an R value takes the data type of the array it meets
dtype(x + 1L)
# and settles on the default of its category when it meets nothing
dtype(nv_add(1L, 2L))
peek_dtype(1)
# a double meeting an integer array promotes to the default float
dtype(x + 0.5)
# an integer input to a float function computes at the default float
dtype(nv_exp(x))
with_default_dtypes(c(float = "f64"), dtype(nv_exp(x)))
```

eq_type *Compare AbstractArray Types*

Description

Compare two abstract arrays for type equality.

An RData has no data type to compare, so it is an error here, just as dtype() is. Give it a data type first, e.g. with nv_convert().

Usage

```
eq_type(e1, e2)

neq_type(e1, e2)
```

Arguments

e1	(AbstractArray) First array to compare. Must not be an RData.
e2	(AbstractArray) Second array to compare. Must not be an RData.

Value

(logical(1))

Examples

```
a <- nv_aval("f32", c(2L, 3L))
b <- nv_aval("f32", c(2L, 3L))

# same dtype and shape
eq_type(a, b)

# different dtype
eq_type(a, nv_aval("i32", c(2L, 3L)))

# different shape
eq_type(a, nv_aval("f32", c(3L, 2L)))

# neq_type is the negation of eq_type
neq_type(a, b)

# an RData has no data type, so it cannot be compared
r <- RData(c(2L, 3L), "double")
try(eq_type(a, r))
try(neq_type(r, r))
```

gradient *Gradients*

Description

Return a new function that computes the gradient of `f` via reverse-mode automatic differentiation. `f` must return a single float scalar. The returned function has the same signature as `f`.

- `gradient()` returns only the gradients, structured like the inputs (or the subset selected by `wrt`).
- `value_and_gradient()` returns both the output of `f` and its gradients, computed in a single forward and reverse pass.

Usage

```
gradient(f, wrt = NULL)

value_and_gradient(f, wrt = NULL)
```

Arguments

`f`	(`function`) Function to differentiate. Must return a single scalar float array.
`wrt`	(`character` | `integer` | `NULL`) Names or positions of the arguments to compute the gradient with respect to. Only float arrays can be included; static arguments must not appear in `wrt`. At call time, an argument in `wrt` must be an array with a data type, not an R value such as `3`, since the data type of its gradient would otherwise be undetermined. If `NULL` (the default), the gradient is computed with respect to all arguments (which must all be arrayish in that case).

Value

(`function`)
Has the same formals as `f` and must be called inside `jit()`. For `gradient()`, it returns a named `list` of gradients, one per argument of `f` (or per `wrt` entry), each structured like that argument. For `value_and_gradient()`, it returns `list(value = , grad = )`: the return value of `f`, and that same `list` of gradients.

See Also

the Automatic Differentiation article, `graph_gradient()` for the low-level graph transformation.

Examples

```
f <- function(x, y) sum(x * y)
g <- jit(gradient(f))
g(nv_array(c(1, 2), dtype = "f32"), nv_array(c(3, 4), dtype = "f32"))

# differentiate with respect to a single argument
g_x <- jit(gradient(f, wrt = "x"))
g_x(nv_array(c(1, 2), dtype = "f32"), nv_array(c(3, 4), dtype = "f32"))

# static (non-array) arguments are passed through but cannot be in wrt
f2 <- function(x, power) sum(x^power)
g2 <- jit(gradient(f2, wrt = "x"), static = "power")
g2(nv_array(c(1, 2, 3), dtype = "f32"), power = 2L)

# an argument in `wrt` must be passed as an array, not as an R value
g3 <- jit(gradient(function(x) x^2L))
g3(nv_scalar(3))
try(g3(3))

# the value of `f` together with its gradient
loss_fn <- function(x) sum(x^2L)
vg <- jit(value_and_gradient(loss_fn))
result <- vg(nv_array(c(3, 4), dtype = "f32"))
result$value
result$grad
```

---

graph_desc_add *Add a Statement to a Graph Descriptor*

Description

Record a call of a primitive as a `GraphStatement` in a graph descriptor. Inside a primitive body created with `new_primitive()`, pass the lexically-bound `self` as the primitive argument.

Usage

```
graph_desc_add(
  primitive,
  args,
  params = list(),
  infer_fn,
  desc = NULL,
  device = NULL
)
```

Arguments

primitive	(AnvlPrimitiveDef | AnvlPrimitive) The primitive the statement applies. An AnvlPrimitive is accepted and unwrapped to its underlying AnvlPrimitiveDef.
args	(list of arrayish) The arguments to the primitive: GraphBoxes, AnvlArrays (registered as constants of the graph) or R values (materialized at their default data type).
params	(list) The parameters to the primitive.
infer_fn	(function) The inference function to use. Must output a list of AbstractArrays.
desc	(GraphDescriptor | NULL) The graph descriptor to add the statement to. Uses the current descriptor if NULL.
device	(NULL | character(1) | device object) The device the call places its result on, for a primitive that constructs an array out of nothing (e.g. prim_fill(), prim_iota()) and so has no operand to carry one. It is declared to desc, where it counts like the device of an array input to the same trace: it decides what that program is compiled for, and disagreeing with another device in it is an error. Every other primitive takes its device from its operands and leaves this NULL.

Value

(list of GraphBox)

---

graph_gradient *Transform a Graph to Its Gradient*

Description

Low-level graph transformation that transforms a graph into its gradient. The function f represented by graph must return a single float scalar. The resulting graph computes the gradients of that scalar with respect to the inputs specified by wrt.

Usage

```
graph_gradient(graph, wrt)
```

Arguments

graph	(AnvlGraph) The graph to transform. Must produce a single scalar float output.
wrt	(NULL | character()) Names of the graph inputs to differentiate with respect to. NULL differentiates with respect to all inputs.

Details

To support alternative forward passes for more efficient backward passes, we replay and possibly rewrite the graph into a new descriptor. Afterwards, we traverse it backwards and call the gradient rules where necessary.

See `rule_reverse()` for more information.

`gradient()` and `value_and_gradient()` differentiate the same way, but into the trace they are called in rather than into a graph of its own; prefer them unless you need to operate on graphs directly.

Value

(`AnvlGraph`)
Its outputs are the requested gradients.

See Also

`gradient()`, `value_and_gradient()`, `rule_reverse()`

Examples

```
graph <- trace_fn(prim_mul, list(nv_aval("f32", integer()), nv_aval("f32", integer())))
graph
graph_gradient(graph, "lhs")
```

`graph_to_quickr_r_function`
Convert an AnvlGraph to a Plain R Function

Description

Lowers a supported subset of `AnvlGraph` objects to a plain R function (no compilation) suitable for `quickr::quick()`. The returned function expects plain R scalars/vectors/arrays and returns plain R values/arrays.

Usage

```
graph_to_quickr_r_function(graph)
```

Arguments

`graph`	(`AnvlGraph`) Graph to convert.

Details

Most users will prefer `jit()` under `with_backend("quickr", ...)`. This function is the lower-level graph API.

Value

(function)
Takes one plain R value per graph input and returns plain R values.

See Also

jit() under with_backend("quickr", ...) for tracing and compiling a regular R function in one step.

GraphBox *Graph Box*

Description

Wraps a GraphNode during graph construction (tracing). When a function is traced via trace_fn(), each intermediate array value is represented as a GraphBox. It also contains an associated GraphDescriptor in which the node "lives".

Usage

```
GraphBox(gnode, desc)
```

Arguments

gnode	(GraphNode) The graph node – either a GraphValue or a GraphLiteral.
desc	(GraphDescriptor) The descriptor of the graph being built.

Value

(GraphBox)

See Also

trace_fn(), jit()

`GraphDescriptor` *Graph Descriptor*

Description

The in-progress representation of an `AnvlGraph` during tracing. This is a mutable class.

While `trace_fn()` runs a function, every primitive call is recorded as a `GraphStatement` into the current descriptor (see `graph_desc_add()` and `current_descriptor()`). The descriptor also does the book-keeping the trace needs: which `GraphValue` an `AnvlArray` or `GraphBox` stands for, the constants and devices encountered, and the default data types the trace is pinned to. Once tracing finishes, it is converted to an `AnvlGraph`, which only keeps what is needed to lower and run the program.

Usage

```
GraphDescriptor(
  statements = list(),
  array_to_gval = NULL,
  gval_to_box = NULL,
  constants = list(),
  in_tree = NULL,
  out_tree = NULL,
  inputs = list(),
  outputs = list(),
  is_static_flat = NULL,
  static_args_flat = NULL,
  devices = character(),
  default_dtypes = NULL,
  backend,
  side_effects_mode = NULL,
  parent = NULL
)
```

Arguments

`statements`	(`list(GraphStatement)`) The statements that make up the graph.
`array_to_gval`	(`hashtab`) Mapping: `AnvlArray` -> `GraphValue`
`gval_to_box`	(`hashtab`) Mapping: `GraphValue` -> `GraphBox`
`constants`	(`list(GraphValue)`) The constants of the graph.

`in_tree` (`NULL` | `RTree`)
The tree of inputs. May contain leaves for both array inputs and static (non-array) arguments. Only the array leaves correspond to entries in `inputs`; use `is_static_flat` to distinguish them.

`out_tree` (`NULL` | `RTree`)
The tree of outputs.

`inputs` (`list(GraphValue)`)
The inputs to the graph (array arguments only).

`outputs` (`list(GraphValue)`)
The outputs of the graph.

`is_static_flat` (`NULL | logical()`)
Boolean mask indicating which flat positions in `in_tree` are static (non-array) args. `NULL` when all args are array inputs.

`static_args_flat`
(`NULL | list()`)
Flattened traced values for the static arguments indicated by `is_static_flat`.

`devices` (`list()`)
Devices encountered during tracing: the device of every concrete array registered in the graph, plus the ones declared by `graph_desc_add()`.

`default_dtypes` (`NULL` | `list(float, int)`)
The data types every R value in this trace materializes at when nothing else decides one (see `default_dtypes()`).

`backend` (`character(1)`)
The backend this trace is compiled for. Required: it decides which entry of the `anvl.default_dtypes` option applies to the trace, so switching the active backend inside a traced body changes nothing. `local_descriptor()` fills it in from `active_backend()`, so only a direct call has to name it.

`side_effects_mode`
(`NULL` | `character(1)`)
Where the trace's view of a side effect starts on its first use:

- `"program"`: where the side effect says: an implicit input of the program, which the graph records in its `side_effects` rather than among its inputs, an argument, or a value the program creates. The top level of `jit()` and `trace_fn()` takes side effects this way.
- `"loop"`: as fresh inputs of the graph, which the graph also returns there. The bodies of `nv_while()` and `nv_scan()` take side effects this way, and hand them on to the next iteration.
- `"capture"`: as the value of `parent`'s view, which the graph closes over and continues. The branches of `nv_if()` and the function `gradient()` differentiates take side effects this way.
- `NULL`: not at all. Such a trace cannot hand back what it did, so using a side effect there is an error.

`parent` (`NULL` | `GraphDescriptor`)
The trace this one is opened in, or `NULL` for a trace at the top level. `local_descriptor()` fills it in.

Value

(GraphDescriptor)

GraphLiteral *Graph Literal*

Description

Literal in an AnvlGraph. This is a mutable class.

Usage

```
GraphLiteral(aval)
```

Arguments

aval	(LiteralArray) The value of the literal.

Value

(GraphLiteral)

GraphNode *Graph Node*

Description

Virtual base class for nodes in an AnvlGraph. Is either a GraphValue or a GraphLiteral. Cannot be instantiated directly - use GraphValue() or GraphLiteral() instead.

GraphStatement *Graph Statement*

Description

One statement of an AnvlGraph: a primitive applied to inputs, with its results assigned to outputs.

Usage

```
GraphStatement(primitive, inputs, params, outputs)
```

Arguments

primitive	(AnvlPrimitiveDef) The function.
inputs	(list(GraphValue)) The (array) inputs to the primitive.
params	(list(<any>)) The (static) parameters of the function call.
outputs	(list(GraphValue)) The (array) outputs of the primitive.

Value

(GraphStatement)

GraphValue *Graph Value*

Description

Value in an AnvlGraph. This is a mutable class.

Usage

```
GraphValue(aval, label = NULL)
```

Arguments

aval	(AbstractArray) The abstract value of the variable.
label	(NULL | character(1)) What the value is, for a printed graph to name it after, e.g. "key" for the key of the random state, which is printed as %key1.

Value

(GraphValue)

`install_anvl` *Install What a Backend Needs to Run*

Description

A backend needs more than the packages anvl declares as dependencies: the "pjrt" backend runs on PJRT plugins that are downloaded rather than shipped with the package, and the "quickr" backend needs **quickr**, which is only suggested. This installs whichever of the two the given backend is missing.

Usage

```
install_anvl(backend = active_backend(), ...)
```

Arguments

backend	(character(1)) Backend to install for. Defaults to active_backend().
...	Passed to the underlying installer: pjrt::install_pjrt() for "pjrt", utils::install.packages() for "quickr".

Details

The PJRT plugins are downloaded on demand, but not silently: the first time a plugin is needed, an interactive session asks for confirmation, while a non-interactive session does not download at all. Call this to make the download an explicit step instead, for instance in a Dockerfile layer of its own or at the start of a script that later runs unattended. The PJRT_INSTALL environment variable overrides the prompt: "1" always downloads without asking, "0" never downloads.

For "pjrt", the CPU plugin is always installed, and the CUDA plugin too when an NVIDIA GPU is detected on Linux (or cuda = TRUE is passed). The CUDA plugin additionally needs the CUDA libraries, which come in the pjrt.cuda R package from the r-xla r-universe; install_anvl() installs it along with the CUDA plugin. See pjrt::install_pjrt() for details.

Value

(NULL)
Invisibly. Called for its side effect.

IotaArray *Iota Array Class*

Description

An AbstractArray representing an arithmetic sequence with step 1 along one axis. Usually created by nv_iota() / nv_seq(), which both call prim_iota() internally. Inherits from AbstractArray.

Usage

```
IotaArray(shape, dtype, axis, start = 1L)
```

Arguments

shape	(stablehlo::Shape | integer()) The shape of the array.
dtype	(xlamisc::DataType) The data type.
axis	(integer(1)) The axis along which values increase.
start	(integer(1)) The starting value.

Value

(IotaArray)

Lowering

When lowering to StableHLO, these become `iota` operations that generate the integer sequence so they do not need to actually hold the data in the executable, similar to ALTREPs in R. It lowers to hlo_iota(), optionally shifting the starting value via hlo_add().

Examples

```
x <- IotaArray(shape = 4L, dtype = "i32", axis = 1L)
x
shape(x)
naxes(x)
dtype(x)
# how it appears during tracing:
graph <- trace_fn(function() nv_iota(axis = 1L, shape = 4L, dtype = "i32"), list())
graph
graph$outputs[[1]]$aval
```

is_device *Test Whether an Object Is a Device*

Description

Test Whether an Object Is a Device

Usage

```
is_device(x)
```

Arguments

x	An object to test.

Value

(logical(1))
Whether x is a device of one of the backends.

is_dtype *Check If an Object Is a DataType*

Description

Tests whether x is a DataType object.

Usage

```
is_dtype(x)
```

Arguments

x	An object to test.

Value

(logical(1))

See Also

as_dtype(), xlamisc::is_dtype()

Examples

```
is_dtype("f32")
is_dtype(as_dtype("f32"))
```

`jit` *JIT Compile a Function*

Description

Wraps a function so that it is traced and compiled on first call. Subsequent calls with the same input structure, shapes, and dtypes hit an LRU cache and skip recompilation.

Usage

```
jit(f, static = character(), cache_size = 100L, device = NULL, ...)
```

Arguments

`f`	(`function`) Function to compile.
`static`	(`character()` | `integer()`) Names or positions of parameters of `f` that are *not* arrays. Static values are embedded as constants in the compiled program; a new compilation is triggered whenever a static value changes. For example useful when you want R control flow in your function. A static value must not have reference semantics: an environment or an external pointer, also inside a `list`, is an error, since it could be mutated in place while the cache key stays equal. Closures are allowed, but their enclosing environment must not change in a way that modifies their behavior.
`cache_size`	(`integer(1)`) Maximum number of compiled executables to keep in the LRU cache.
`device`	(`NULL` | `character(1)` | device) Target device, of the active backend. When a device is specified, all arrays are moved to it. The default (`NULL`) infers the device at call time from the array inputs, falling back to `default_device()`.
`...`	Backend-specific options: • `donate` (`character()`, default `character()`), `"pjrt"` only: names of arguments whose buffers the compiled program may reuse for its outputs. A donated array must not be used again after the call; this can save memory and copies for large inputs. Must not overlap with `static`. • `xla_options` (named `list()`, default `list()`), `"pjrt"` only: XLA options for compiling this function, passed to `pjrt::pjrt_compile()`, which describes the names and values it takes. They apply to this function's compiles only, unlike `XLA_FLAGS`, which applies to every compile of the R session. A jitted function called inside another one's trace is traced into the outer program, so only the options of the outermost `jit()` take effect: pass them as `jit(value_and_gradient(f), xla_options = ...)`, not to a jitted `f`.

- unwrap (logical(1), default FALSE), "quickr" only: if TRUE, the compiled function returns plain R arrays instead of AnvlArrays, which is useful when the output is consumed by non-anvl R code.

An option no backend takes is rejected here; one that only another backend takes is rejected when the function is called on a backend that does not, since the backend is not known until then.

Value

(JitFunction)
A function with the same formals as f.

Stochastic Functions

A function that draws random numbers also reads and advances the global random state. See the the Random Number Generation article for more details.

Backend and device

A jitted function runs on the active backend *when it is called* (active_backend(), set with with_backend() / local_backend()), so one JitFunction serves every backend, and a function created under one backend and called under another runs on the latter. Array inputs must belong to that backend; an array of another backend is rejected. Each backend keeps its own compilation cache.

The device is a choice within that backend. Setting device explicitly enforces that the function always uses it, e.g. "cuda:0", and copies every array input to it. With device = NULL (default) the device is inferred from the input arrays and the constants within the program; conflicting devices are an error, and with no array to read a device from the default device is used.

Default Data Types

It is possible to configure the default data types for floats and ints via the anvl.default_dtypes option, see default_dtypes(). The defaults are read at *call time*, not when jit() is called.

To pin a jitted function to a pair of data types instead of letting it follow the configured defaults, wrap it in with_dtypes(): the wrapper converts the array arguments and results of a category it names, and sets the defaults for the duration of the call, so f_f64 <- with_dtypes(f, c(float = "f64")) runs f at f64, unless f itself changes the default data types.

See Also

jit_cache_size() for how many programs a jitted function has cached.

Examples

```
f <- jit(function(x, y) x + y)
f(nv_array(1), nv_array(2))

# A non-static R value is an input of the program, just like an array: a
# new value reuses the compiled program
scale <- jit(function(x, n) x * n)
```

```
scale(nv_array(1:3), 2)
scale(nv_array(1:3), 3)
jit_cache_size(scale)

# A static R value is embedded as a constant, so each new value compiles a
# new program
scale_static <- jit(function(x, n) x * n, static = "n")
scale_static(nv_array(1:3), 2)
scale_static(nv_array(1:3), 3)
jit_cache_size(scale_static)

# static arguments enable data-dependent control flow
g <- jit(function(x, flag) {
  if (flag) x + 1 else x * 2
}, static = "flag")
g(nv_array(3), TRUE)
g(nv_array(3), FALSE)

# R values that meet no typed array take the default data types, which are
# read when the function is called
half <- jit(function(x) x / 2)
half(1)
with_default_dtypes(c(float = "f64"), half(1))
# or pin the data types of a function with `with_dtypes()`
half_f64 <- with_dtypes(half, c(float = "f64"))
half_f64(nv_array(1, dtype = "f32"))

# a donated input's buffer is reused for the output, so the input must not
# be used afterwards
step <- jit(function(x) x + 1L, donate = "x")
x <- nv_array(1:3)
step(x)
try(as_array(x))


# the same function runs on whichever backend is active when it is called
f <- jit(function(x, y) x + y)
with_backend("quickr", f(nv_array(1), nv_array(2)))
```

`jit_cache_size` *Number of Cached Programs of a Jitted Function*

Description

The number of compiled programs a function returned by `jit()` currently holds for one backend, i.e. how many entries of its compilation cache are filled. A call whose inputs hit an existing entry leaves this unchanged; a call that misses adds one, up to the `cache_size` cap `jit()` was given, beyond which the least recently used entry is evicted.

Each backend a jitted function has run on keeps its own cache, so this is reported for one backend at a time.

Usage

```
jit_cache_size(f, backend = active_backend())
```

Arguments

f	(function) A function returned by jit().
backend	(character(1)) The backend whose cache to report on. Defaults to the active one (active_backend()).

Value

integer(1). A backend that f has not run on yet reports 0, since the caches are created on first use.

See Also

jit()

Examples

```
f <- jit(function(x, y) x + y)
jit_cache_size(f)

f(nv_scalar(1), nv_scalar(2))
jit_cache_size(f)

# same dtypes and shapes -- a cache hit, no new entry
f(nv_scalar(3), nv_scalar(4))
jit_cache_size(f)

# a different shape -- a second program is compiled
f(nv_array(c(1, 2)), nv_array(c(3, 4)))
jit_cache_size(f)
```

---

length.AnvlArray *Number of Elements*

Description

The number of elements of an array – the generic base::length() on an anvl array, i.e. the product of its axis sizes.

Usage

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

Arguments

x	(arrayish) Input array.

Value

(integer(1))

See Also

dim(), nelts()

Examples

```
length(nv_matrix(1:6, nrow = 2))
```

LiteralArray *Literal Array Class*

Description

An AbstractArray where all elements have the same constant value. This arises from a literal in traced code (x + 1, say). A nv_fill() is a recorded operation rather than a constant, so its output is an ordinary AbstractArray.

Usage

```
LiteralArray(data, shape, dtype = default_dtype(data))
```

Arguments

data	(double(1) | integer(1) | logical(1) | AnvlArray) The scalar value, or a one-element AnvlArray – for which dtype has to be named, since the default takes it from an R value's storage type.
shape	(stablehlo::Shape | integer()) The shape of the array.
dtype	(xlamisc::DataType) The data type. For the default, see default_dtypes().

Value

(LiteralArray)

Lowering

LiteralArrays become constants inlined into the StableHLO program. I.e., they lower to hlo_tensor().

Examples

```
x <- LiteralArray(1L, shape = integer())
x
shape(x)
naxes(x)
dtype(x)
# how it appears during tracing: an R literal that meets nothing
graph <- trace_fn(function() 1, list())
graph
graph$outputs[[1]]$aval
# a `nv_fill()`, by contrast, is a recorded operation
graph <- trace_fn(function() nv_fill(2L, shape = c(2, 2)), list())
graph
graph$outputs[[1]]$aval
```

---

local_backend *Temporarily Set the Backend*

Description

Set the `anvl.backend` option for a scope: `local_backend()` until the calling frame exits, `with_backend()` for the duration of `code`. Every array built and every operation run in that scope uses the backend, and R values materialize at its default data types (see `default_dtypes()`).

Usage

```
local_backend(backend, envir = parent.frame())

with_backend(backend, code)
```

Arguments

backend	(`character(1)`) Backend to use (`"pjrt"` or `"quickr"`).
envir	(`environment`) The environment to scope the change to.
code	(any) An expression to evaluate with the given backend.

Value

`local_backend()` returns the previous value of the option, as `list(anvl.backend = )`, invisibly. `with_backend()` returns the result of evaluating `code`.

See Also

`active_backend()`

Examples

```
f <- function() {
  local_backend("quickr")
  active_backend()
}
f()
active_backend()
with_backend("quickr", active_backend())
```

`local_default_device` *Temporarily Set the Default Device*

Description

Sets the `anvl.default_device` option, which `default_device()` returns in place of the first CPU device: `local_default_device()` for the calling scope, `with_default_device()` for one expression.

This is what a call that names no device allocates on, and what a jitted function whose graph pins no device of its own compiles for.

Usage

```
local_default_device(device, envir = parent.frame())

with_default_device(device, code)
```

Arguments

`device`	(`NULL` | `character(1)` | device object) The device to make the default, e.g. `"cuda:0"`. A string is looked up on whichever backend asks for the default, so an identifier only one backend knows (quickr has no `"cuda"`) makes the default an error on the others. `NULL` clears the option.
`envir`	(`environment`) Scope the option is reset at the end of. Defaults to the caller.
`code`	(`any`) Expression to evaluate with the default device set.

Value

`local_default_device()`: (named `list`)
The previous value of the option, as `list(anvl.default_device = )`, invisibly.

`with_default_device()`: (any)
The value of `code`.

See Also

default_device(), local_backend()

Examples

```
getOption("anvl.default_device")
# the option holds the identifier; it is resolved to a device only when
# `default_device()` is called
with_default_device("cuda:0", getOption("anvl.default_device"))
```

---

local_default_dtypes *Set the Default Data Types*

Description

Set the default data types (see default_dtypes()) for a scope: local_default_dtypes() until the calling frame exits, with_default_dtypes() for the duration of code. Both write the anvl.default_dtypes option for one backend, and change only the categories they name.

Usage

```
local_default_dtypes(dtypes, backend = NULL, envir = parent.frame())

with_default_dtypes(dtypes, code, backend = NULL)
```

Arguments

dtypes	(named character() | named list()) A mapping of the data type categories (float and int) to data types, e.g. c(float = "f64", int = "i32"). Each may be a string or a DataType. Can also be a partial override, such as c(float = "f64"), in which case the category it does not name is left as it is.
backend	(NULL | character(1)) The backend whose defaults to set. Uses active_backend() by default.
envir	(environment) The environment to scope the change to.
code	An expression to evaluate with the given defaults.

Details

Inside a jit()ted body the defaults the program was keyed on are the *baseline* and an override applies to its scope, so one program can use different precisions in different parts of itself.

Value

local_default_dtypes() returns the previous values of the options it set, invisibly. with_default_dtypes() returns the result of evaluating code.

See Also

default_dtypes(), local_backend(), with_backend()

Examples

```
with_default_dtypes(c(float = "f64"), dtype(nv_array(1.5)))
# A value that meets a typed array still takes that array's data type
with_default_dtypes(c(float = "f64"), dtype(nv_array(1, dtype = "f32") + 1.5))
# untyped values in one program can materialize at different precisions: one at
# whatever the default is, one at the `f64` the override asks for
jit(function() {
  list(
    at_default = nv_fill(0, 2),
    forced_f64 = with_default_dtypes(c(float = "f64"), nv_fill(0, 2))
  )
})()
```

---

local_descriptor *Create a Graph*

Description

Creates a new GraphDescriptor which is afterwards accessible via current_descriptor(). The graph is automatically removed when exiting the current scope. After the graph is either cleaned up automatically (by exiting the scope) or finalized, the previously built graph is restored, i.e., accessible via current_descriptor(). That previously built graph, NULL if there is none, is the parent of the new one.

Usage

```
local_descriptor(..., envir = parent.frame())
```

Arguments

...	(any) Additional arguments to pass to the GraphDescriptor constructor, except parent.
envir	(environment) Environment where exit handler will be registered for cleaning up the GraphDescriptor if it was not returned yet.

Value

(GraphDescriptor)

`local_nv_rng_state` *Draw From a Random State for a Scope*

Description

`local_nv_rng_state()` and `with_nv_rng_state()` make `rng_state` the random state the samplers draw from, for the calling scope or for `code` only, and restore the one from before afterwards. The draws advance `rng_state` in place. In a function traced by `jit()`, they cannot be called in the body of `nv_while()` or `nv_scan()`: called before the loop, they reach its body.

See the Random Number Generation article for more details.

Usage

```
local_nv_rng_state(rng_state, envir = parent.frame())

with_nv_rng_state(rng_state, code)
```

Arguments

`rng_state`	(`AnvlRngState`) The random state to draw from, as `nv_rng_state()` returns.
`envir`	(`environment`) The scope whose exit restores the default from before.
`code`	(any) An expression to evaluate drawing from `rng_state`.

Value

`local_nv_rng_state()` returns `NULL`, invisibly; `with_nv_rng_state()` returns the result of evaluating `code`.

See Also

Other rng: `nv_normal`, `nv_rbinom()`, `nv_rng_state()`, `nv_sample()`, `nv_sample_int()`, `nv_set_seed()`, `nv_uniform`

Examples

```
rng <- nv_rng_state(1L)
with_nv_rng_state(rng, list(nv_runif(2L), nv_rnorm(2L)))
# both samplers drew from `rng`, three sampling calls in all
rng

# in a jitted function, from a random state it takes as an argument
step <- jit(function(x, rng) {
  local_nv_rng_state(rng)
  x + nv_rnorm(2L) + nv_runif(2L)
```

```
})
step(nv_array(c(0, 0)), rng)
rng
```

`map_tree` *Map a Function over Trees*

Description

Apply a function to each leaf of a (possibly nested) list, keeping its structure, e.g. to a list of arrays a jitted function returns.

- `map_tree()` maps over one tree.
- `pmap_tree()` maps over several trees of the same structure in parallel, calling `.f` with one leaf from each.

Usage

```
map_tree(.x, .f, ...)

pmap_tree(.l, .f, ...)
```

Arguments

`.x`	(any) A leaf or a (nested) list of leaves.
`.f`	(`function`) Function to apply. `map_tree()` calls it with each leaf of `.x`, `pmap_tree()` with one leaf from each tree in `.l`, in order.
`...`	Additional arguments passed to `.f` after the leaves.
`.l`	(`list`) A non-empty list of trees, all with the same structure.

Details

These are implemented in `pjrt::map_tree()` and `pjrt::pmap_tree()`.

Value

A tree with the same structure as `.x` (or `.l[[1]]`), where each leaf is the result of `.f`.

Examples

```
out <- list(a = nv_array(1:2), b = list(c = nv_scalar(3)))
map_tree(out, dtype)
map_tree(out, as_array)
pmap_tree(list(list(a = 1, b = 2), list(a = 10, b = 20)), `+`)
```

naxes *Get the Number of Axes of an Array*

Description

Returns the number of axes (sometimes also referred to as rank) of an array. Equivalent to `length(shape(x))`.

Usage

```
naxes(x)
```

Arguments

x	(arrayish) An array-like object.

Value

`(integer(1))`

See Also

`xlamisc::naxes()`

Examples

```
x <- nv_array(1:4, dtype = "f32")
naxes(x)
```

new_primitive *Create a Primitive*

Description

`new_primitive()` creates a new primitive: an `AnvlPrimitive`, the function the primitive is called through (e.g. `prim_add()`). For details on how to do this, see the Adding a Primitive article. Like every jitted function it runs on the active backend when called.

Usage

```
new_primitive(
  name,
  fn,
  subgraphs = character(),
  static = character(),
  register = TRUE
)
```

Arguments

name	(character(1)) Primitive name, without the prim_ prefix ("add" for prim_add()).
fn	(function) Body of the primitive. Its formals become the formals of the returned JIT-compiled callable. Inside fn, the primitive is accessible via the lexically-bound symbol self (an AnvlPrimitiveDef); pass it as the first argument to graph_desc_add().
subgraphs	(character()) Names of parameters that are subgraphs (for higher-order primitives).
static	(character() | integer()) Passed to jit().
register	(logical(1)) Whether to add the primitive to anvl's internal registry of primitives, under name, replacing one registered under the same name. The quickr backend reads that registry to know which primitives it can lower, so a primitive created with register = FALSE is rejected on quickr even if it has a quickr rule. The other backends only read the rules of the primitive itself. This does not bind the result to a prim_<name> variable; assign it yourself.

Value

(AnvlPrimitive)
The function the primitive is called through, of class c("AnvlPrimitive", "JitFunction"). Its AnvlPrimitiveDef is attr(<fn>, "definition"), and [[/ [[<- on it access the rules of that definition.

---

nv_abs *Absolute Value*

Description

Element-wise absolute value. You can also use abs().

Usage

```
nv_abs(x)

## S3 method for class 'AnvlArray'
abs(x)
```

Arguments

x	(arrayish) One input. Can be any signed numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_abs() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 2, -3))
abs(x)

# an R value materializes at its default data type
nv_abs(-1)
```

---

nv_acos *Arc Cosine*

Description

Element-wise inverse cosine. You can also use acos().

Usage

```
nv_acos(x)

## S3 method for class 'AnvlArray'
acos(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_acos() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
acos(x)

# an R value materializes at its default data type
nv_acos(0.5)
```

---

nv_acosh *Inverse Hyperbolic Cosine*

Description

Element-wise inverse hyperbolic cosine. You can also use `acosh()`.

Usage

```
nv_acosh(x)

## S3 method for class 'AnvlArray'
acosh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_acosh()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 2, 10))
acosh(x)

# an R value materializes at its default data type
nv_acosh(2)
```

nv_add *Addition*

Description

Adds two arrays element-wise. You can also use the + operator.

Usage

```
nv_add(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 + e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_add() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
nv_add(x, y)
x + y

# different data types are promoted to their common one
nv_add(nv_scalar(1, "f32"), nv_scalar(2, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x + 1L
```

nv_all *All Reduction*

Description

Performs logical AND along the specified axes. Returns TRUE only if all elements are TRUE.

Usage

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

## S3 method for class 'AnvlArray'
all(..., na.rm = FALSE)
```

Arguments

x	(arrayish) One input. Must be a boolean or an R logical.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
...	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of nv_all(), such as axes.
na.rm	(logical(1)) Only used for R values among ..., which are reduced by base R, as a boolean array has no missing values.

Value

(arrayish)
Has the boolean data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

The all() generic

all() reduces over all axes and, like base::all(), takes several data arguments: all(x, y) asks about both arrays. It is *logical*, so – unlike base R – a non-boolean argument is an error rather than a comparison against zero. Beyond base R, named arguments are passed on, so all(x, axes = 1L) reduces a single axis – but only when x is the only data argument.

See Also

prim_all() for the underlying primitive.

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
# no axes given: reduce over all of them
nv_all(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_all(x, axes = 1L)
nv_all(x, axes = 1L, drop = FALSE)
```

---

nv_and *Bitwise AND*

Description

Element-wise bitwise AND – a logical AND on a boolean input, and a bit-by-bit one on an integer. You can also use the `&` operator, which expects `bool` inputs, however.

Usage

```
nv_and(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 & e2
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs with a common data type. Can be any integerish data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
`e1`, `e2`	(`arrayish`) The operands of the operator, which it passes on as `lhs` and `rhs`.

Value

(`arrayish`)
Has the inputs' broadcast shape and common data type.

The `&` operator

`&` is *logical*, like in base R. Unlike base R it only accepts booleans and does not auto-convert non-booleans by comparing them with 0.

See Also

`prim_and()` for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
x & y

# different data types are promoted to their common one
nv_and(nv_scalar(12L, "i32"), nv_scalar(10L, "i64"))

# a scalar is broadcast
x & TRUE
```

---

nv_any *Any Reduction*

Description

Performs logical OR along the specified axes. Returns TRUE if any element is TRUE.

Usage

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

## S3 method for class 'AnvlArray'
any(..., na.rm = FALSE)
```

Arguments

x	(arrayish) One input. Must be a boolean or an R logical.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
...	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of nv_any(), such as axes.
na.rm	(logical(1)) Only used for R values among ..., which are reduced by base R, as a boolean array has no missing values.

Value

(arrayish)
Has the boolean data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

The `any()` generic

`any()` reduces over all axes and, like `base::any()`, takes several data arguments: `any(x, y)` asks about both arrays. It is *logical*, so – unlike base R – a non-boolean argument is an error rather than a comparison against zero. Beyond base R, named arguments are passed on, so `any(x, axes = 1L)` reduces a single axis – but only when `x` is the only data argument.

See Also

`prim_any()` for the underlying primitive.

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
# no axes given: reduce over all of them
nv_any(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_any(x, axes = 1L)
nv_any(x, axes = 1L, drop = FALSE)
```

`nv_aperm` *Transpose*

Description

Permutes the axes of an array, like `base::aperm()`. `nv_transpose()` is another spelling of the same function. You can also use `aperm()`, or `t()` for matrices.

Usage

```
nv_aperm(x, perm = NULL)

nv_transpose(x, perm = NULL)

## S3 method for class 'AnvlArray'
t(x)

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

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

perm	(integer() | NULL) New ordering of axes. If NULL (default), reverses the axes. Negative values count from the end, i.e. -1 refers to the last axis.
a	(arrayish) The array whose axes to permute.
...	No additional arguments.

Value

(arrayish)
Has x's data type and shape shape(x)[perm], or rev(shape(x)) when perm is NULL.

The t() generic

t() requires a matrix, whereas base::t() also transposes a vector (into a one-row matrix). Like base::t(), it rejects a higher-rank array; use aperm() to reverse its axes.

The aperm() generic

aperm() permutes the axes exactly as nv_aperm() does; like base::aperm(), perm = NULL reverses them.

See Also

prim_transpose() for the underlying primitive.

Examples

```
# the 2x3 becomes a 3x2, keeping its data type
x <- nv_matrix(1:6, nrow = 2)
nv_aperm(x)
t(x)
nv_aperm(nv_array(1:24, shape = c(2, 3, 4)), c(3, 1, 2))
```

---

nv_asin *Arc Sine*

Description

Element-wise inverse sine. You can also use asin().

Usage

```
nv_asin(x)

## S3 method for class 'AnvlArray'
asin(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_asin() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
asin(x)

# an R value materializes at its default data type
nv_asin(0.5)
```

---

nv_asinh *Inverse Hyperbolic Sine*

Description

Element-wise inverse hyperbolic sine. You can also use asinh().

Usage

```
nv_asinh(x)

## S3 method for class 'AnvlArray'
asinh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_asinh() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
asinh(x)

# an R value materializes at its default data type
nv_asinh(1)
```

---

nv_atan *Arc Tangent*

Description

Element-wise inverse tangent. You can also use atan().

Usage

```
nv_atan(x)

## S3 method for class 'AnvlArray'
atan(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_atan() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
atan(x)

# an R value materializes at its default data type
nv_atan(1)
```

`nv_atan2` *Arctangent 2*

Description

Element-wise two-argument arctangent, i.e. the angle (in radians) between the positive x-axis and the point `(x, y)`.

Usage

```
nv_atan2(y, x)
```

Arguments

`y`, `x`	(`arrayish`) Ordinate and abscissa of the point. Can be any numeric data type: the two are first brought to a common data type and that is then converted to the default float data type (see `default_dtypes()`) where it is not a float already, so the result is always a float. Scalars are broadcast. An R value assumes the other operand's data type within its data type category, and settles on the default float when neither operand has one.

Details

The operands are named `y` and `x`, in that order, after `base::atan2()`, rather than `lhs` / `rhs`.

Value

(`arrayish`)
Has the inputs' broadcast shape, and their common data type – or the default float data type (see `default_dtypes()`) where that was an integer one.

See Also

`prim_atan2()` for the underlying primitive.

Examples

```
y <- nv_array(c(1, 0, -1))
x <- nv_array(c(0, 1, 0))
nv_atan2(y, x)

# different data types are promoted to their common one
nv_atan2(nv_scalar(1, "f32"), nv_scalar(1, "f64"))

# a scalar is broadcast and an R integer is converted to a float
nv_atan2(y, 1L)
```

---

nv_atanh *Inverse Hyperbolic Tangent*

Description

Element-wise inverse hyperbolic tangent. You can also use `atanh()`.

Usage

```
nv_atanh(x)

## S3 method for class 'AnvlArray'
atanh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_atanh()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-0.5, 0, 0.5))
atanh(x)

# an R value materializes at its default data type
nv_atanh(0.5)
```

`nv_aval` *Abstract Array Class*

Description

Representation of an abstract array type.

The base class represents an *unknown* value, but child classes exist for:

- closed-over constants: `ConcreteArray`
- scalar arrays arising from R literals: `LiteralArray`
- sequence patterns: `IotaArray`
- R values `RData`. They are special because they do not have a data type.

To convert an `arrayish` value to an abstract array, use `to_abstract()`.

Usage

```
nv_aval(dtype, shape)

AbstractArray(dtype, shape)
```

Arguments

dtype	(`xlamisc::DataType` | `character(1)`) The data type of the array. For `nv_aval()` only, `"double"`, `"integer"` or `"logical"` create an `RData` instead.
shape	(`stablehlo::Shape` | `integer()`) The shape of the array. Can be provided as an integer vector.

Value

`AbstractArray()`: (`AbstractArray`)

`nv_aval()`: (`AbstractArray` | `RData`)
An `RData` when `dtype` names an R storage type.

See Also

LiteralArray, ConcreteArray, IotaArray, RData, GraphValue, `to_abstract()`, GraphBox

Examples

```
# -- Creating abstract arrays --
a <- AbstractArray("f32", c(2L, 3L))
a
dtype(a)
shape(a)

# shorthand
nv_aval("f32", c(2L, 3L))

# an R value, which has no dtype until it is used
nv_aval("double", c(2L, 3L))

# how AbstractArrays appear in an AnvlGraph
graph <- trace_fn(function(x) x + 1L, list(x = nv_aval("i32", 4L)))
graph
graph$inputs[[1]]$aval
```

`nv_bind` *Combine Arrays by Rows or Columns*

Description

Combine arrays along the row (`nv_rbind`) or column (`nv_cbind`) axis.

Each input is handled according to its rank:

- a scalar: broadcast to match the non-stacked axes of the other inputs.
- 1-D: treated as a single row/column.
- Other: used as-is.

Usage

```
nv_rbind(...)

nv_cbind(...)

## S3 method for class 'AnvlArray'
rbind(..., deparse.level = 1L)

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

Arguments

`...`	(`arrayish`) Arrays to combine. Can be of any data type; they are promoted to a common data type, and a scalar is broadcast to match the non-stacked axes.
`deparse.level`	Ignored. Kept for compatibility with `base::rbind()` and `base::cbind()`.

Value

(arrayish)
Has the inputs' common data type. The stacked axis is the sum of their sizes along it – rows for nv_rbind(), columns for nv_cbind() – and every other axis is theirs unchanged.

Differences from base R

base::rbind() and base::cbind() applied to an array() of rank > 2 flatten the trailing axes into the column axis (so a c(2, 3, 4) array becomes a 2 x 12 matrix). nv_rbind and nv_cbind instead preserve all non-stacked axes: combining two c(2, 3, 4) arrays with nv_rbind produces a c(4, 3, 4) array, and with nv_cbind a c(2, 6, 4) array.

See Also

nv_concatenate()

Examples

```
# vectors as rows / columns
nv_rbind(nv_array(1:3), nv_array(4:6))
nv_cbind(nv_array(1:3), nv_array(4:6))

# scalar broadcasting
nv_rbind(nv_matrix(1:6, nrow = 2), nv_scalar(0))

# rank-3 arrays preserve trailing axes
a <- nv_array(1:24, shape = c(2, 3, 4))
shape(nv_rbind(a, a)) # c(4, 3, 4)
```

---

nv_binomial *The Binomial Distribution*

Description

Density (nv_dbinom), distribution function (nv_pbinom), and quantile function (nv_qbinom) for the Binomial distribution with parameters size and prob.

Usage

```
nv_dbinom(x, size, prob, log = FALSE)

nv_pbinom(q, size, prob, lower_tail = TRUE, log_p = FALSE)

nv_qbinom(p, size, prob, lower_tail = TRUE, log_p = FALSE)
```

Arguments

x, q	(arrayish) Quantiles at which to evaluate the density (x) or the distribution function (q). These are counts, given as integers or floats; see Non-Integer Values for values that are not whole numbers.
size	(arrayish) Number of trials, zero or more. Either a scalar, or an array of exactly the shape of x/q/p.
prob	(arrayish) Probability of success on each trial, in $[0, 1]$, shaped like size.
log, log_p	(logical(1)) If TRUE, the densities/probabilities are given as logarithms. For nv_qbinom this describes the input p.
lower_tail	(logical(1)) If TRUE (default), probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
p	(arrayish) Probabilities at which to evaluate the quantile function. Values outside $[0, 1]$ give NaN. With log_p = TRUE, supply log-probabilities in $[-\infty, 0]$ instead.

Details

The Binomial distribution with size $= n$ and prob $= p$ has probability mass function:

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, \ldots, n$$

and zero elsewhere.

Invalid arguments give NaN, as in base R, but without a warning (that is prob outside $[0, 1]$, or a size that is negative or not a whole number). size is taken as a whole number with the same tolerance as x (see Non-Integer Values).

nv_dbinom uses the saddle point expansion of Loader (2000), as base R does. nv_pbinom uses the incomplete beta function methods of DiDonato & Morris (1992) that base R's stats::pbeta() uses: a continued fraction, and an asymptotic expansion for large size near the mean. If the continued fraction has not converged after 1000 iterations, the result is NaN. nv_qbinom finds the smallest x with $P(X \leq x) \geq p$, starting from a Cornish-Fisher approximation as base R does. Above 2^24 in f32 and 2^53 in f64 not every whole number is a float, and nv_qbinom then returns the smallest float x with $P(X \leq x) \geq p$.

Value

(arrayish)
nv_dbinom(), nv_pbinom(), and nv_qbinom() return an arrayish with the shape of x/q/p and the float data type described under Data Types.

Non-Integer Values

x and q are counts, but any value is accepted, and one that is not a whole number is treated as in base R, without its warnings:

- nv_dbinom() takes an x within a relative 1e-9 of a whole number, that is with $|x - \mathrm{round}(x)| \leq 10^{-9} \max(|x|, 1)$, as that whole number. Any other x has probability zero.
- nv_pbinom() rounds q down, except that a q within 1e-7 below a whole number counts as that number.
- A negative x or q is below the support even where it is within the tolerance of zero: such an x has probability zero, and such a q has $P(X \leq q) = 0$.

Finite quantiles from nv_qbinom() are always whole numbers. Since x and q are used only as whole numbers, their gradient is zero wherever a gradient can be taken.

Gradients

nv_dbinom can be differentiated with respect to prob, including at prob of 0 and 1. Where the density is zero there, as at prob = 0 for x > 0, the log-density is -Inf and has no derivative: it diverges. With log = TRUE the gradient returned there is zero, which is a convention of this implementation, not the mathematical derivative. nv_pbinom iterates until convergence in a nv_while() loop, so gradient() cannot differentiate it. nv_qbinom is a step function, and autodiff returns a zero gradient for all of its arguments.

Data Types

nv_dbinom(), nv_pbinom() and nv_qbinom() compute at a float data type and return it. If x/q/p is a float, they compute at its data type; an R double settles on the default float. An integer x/q/p (an array or an R integer) is converted to the widest float among the arrays given for size or prob, or to the default float if there is none. Integers beyond 2^{24} are not all exact at f32, so a large integer may change in that conversion.

An R value for size or prob takes the computing data type. An array is promoted to it, so a wider one (e.g. f64 for an f32 x) is an error.

References

Loader C (2000). "Fast and Accurate Computation of Binomial Probabilities." https://www.r-project.org/doc/reports/CLoader-dbinom-2002.pdf.

DiDonato A, Morris A (1992). "Algorithm 708: Significant Digit Computation of the Incomplete Beta Function Ratios." *ACM Transactions on Mathematical Software*, **18**(3), 360–373. doi:10.1145/131766.131776.

Dekker T (1971). "A floating-point technique for extending the available precision." *Numerische Mathematik*, **18**, 224–242. doi:10.1007/BF01397083.

See Also

nv_rbinom() for sampling from a Binomial distribution.

Examples

```
x <- nv_array(c(0, 1, 2, 5, 10))
nv_dbinom(x, size = 10, prob = 0.3)
nv_dbinom(x, size = 10, prob = 0.3, log = TRUE)

nv_pbinom(x, size = 10, prob = 0.3)
nv_pbinom(x, size = 10, prob = 0.3, lower_tail = FALSE)
nv_pbinom(x, size = 10, prob = 0.3, log_p = TRUE)

p <- nv_array(c(0.025, 0.5, 0.975))
nv_qbinom(p, size = 10, prob = 0.3)
nv_qbinom(p, size = 10, prob = 0.3, lower_tail = FALSE)
nv_qbinom(nv_array(c(-700, -2, -0.1), dtype = "f64"), size = 1000, prob = 0.3, log_p = TRUE)
```

---

nv_bitcast_convert *Bitcast Conversion*

Description

Reinterprets the bits of an array as a different data type without modifying the underlying data. If the target type is narrower, a new leading axis is added; if wider, the first axis is consumed.

Usage

```
nv_bitcast_convert(x, dtype)
```

Arguments

x	(arrayish) One input. Can be any data type except bool. An R value materializes at its default data type.
dtype	(character(1) | DataType) Any target data type except bool. One of the same bit width as the input's leaves the shape unchanged; a narrower one adds a *leading* axis holding the pieces; a wider one consumes the first axis, whose size must equal the ratio of the two widths. The pieces of one element are therefore adjacent in the column-major element order nv_flatten() reads, and a narrowing conversion lays the bytes out the way as_raw() writes them.

Value

(arrayish)
Has the given dtype, and the shape described under dtype.

See Also

prim_bitcast_convert(), which this is an alias of, and nv_convert() for value-preserving type conversion.

Examples

```
# the bits of one i32 reread as four i8, in a new leading axis
x <- nv_array(1L, dtype = "i32")
nv_bitcast_convert(x, dtype = "i8")
```

`nv_broadcast_arrays` *Broadcast Arrays to a Common Shape*

Description

Broadcasts arrays to a common shape, aligning their axes from the first one, so that a vector meets a matrix as a column.

Usage

```
nv_broadcast_arrays(...)
```

Arguments

`...`	(arrayish) Arrays to broadcast.

Value

(`list()` of arrayish)
The inputs, each with its own data type and the common shape.

Broadcasting Rules

1. If the arrays have different numbers of axes, append size-1 axes to the shorter shape, so axis 1 meets axis 1. A length-`n` vector therefore lines up with the rows of an `n` by `m` matrix and is replicated across its columns. NumPy prepends instead.
2. For each axis: if the sizes match, keep them; if one is 1, expand it to the other's size; otherwise raise an error.

Relation to base R

Base R has no broadcasting between arrays – `matrix(1, 3, 3) + matrix(1, 1, 3)` is a "non-conformable arrays" error. It does recycle a *vector* over a matrix, though, and for a vector as long as the first axis that lands on exactly this broadcast, which is why a vector meets a matrix as a column in both. The two part ways once the lengths stop lining up: `matrix(1, 2, 3) + c(1, 2, 3)` recycles on regardless, where the matching broadcast is an error.

The deviation from NumPy's broadcasting rules is still motivated by keeping anvl's behaviour similar to base R in spirit, see the examples for more.

See Also

nv_broadcast_scalars(), nv_broadcast_to()

Examples

```
# interpreting vectors as columns
# base R:
x1 <- c(1, 2)
m1 <- array(1, dim = c(2, 2))
x1 + m1
# anvl:
args <- nv_broadcast_arrays(nv_array(x1), nv_array(m1))
print(args)
args[[1]] + args[[2]]


# axes of size 1 are expanded to the other operand's size
y1 <- nv_array(1:3, shape = c(1, 3))
y2 <- nv_array(1:3, shape = c(3, 1))
nv_broadcast_arrays(y1, y2)
```

---

nv_broadcast_scalars *Broadcast Scalars to Common Shape*

Description

Broadcast scalar arrays to match the shape of non-scalar arrays. All non-scalar arrays must have the same shape.

Usage

```
nv_broadcast_scalars(...)
```

Arguments

...	(arrayish) Arrays to broadcast.

Value

(list() of arrayish)
The inputs, each with its own data type and the common shape.

Examples

```
x <- nv_array(c(1, 2, 3))
# scalar 1 is broadcast to shape [3]
nv_broadcast_scalars(x, nv_scalar(1))
```

`nv_broadcast_to` *Broadcast to Shape*

Description

Broadcasts an array to a target shape, aligning the array's axes with the leading axes of `shape`, so that a vector fills a column. See `nv_broadcast_arrays` for more information.

Usage

```
nv_broadcast_to(x, shape)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`shape`	(`integer()`) Target shape. The input's axes are matched against its leading axes, and each must either match or be 1; trailing axes are added.

Value

(`arrayish`)
Has the given `shape` and the same data type as `x`.

See Also

`nv_broadcast_arrays()`, `nv_broadcast_scalars()`, `prim_broadcast_in_axes()` for the underlying primitive.

Examples

```
# the vector fills a column and is repeated along the new trailing axis
x <- nv_array(c(1, 2, 3))
nv_broadcast_to(x, shape = c(3, 2))
```

nv_cbrt *Cube Root*

Description

Element-wise cube root.

Usage

```
nv_cbrt(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_cbrt() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 8, 27))
nv_cbrt(x)

# an R value materializes at its default data type
nv_cbrt(8)
```

nv_ceiling *Ceiling*

Description

Element-wise ceiling (round toward positive infinity). You can also use ceiling().

Usage

```
nv_ceiling(x)

## S3 method for class 'AnvlArray'
ceiling(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float is rounded and keeps its own, and an integer one is already whole and is returned unchanged. An R value materializes at its default data type and is treated in the same way.

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_ceiling() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1.2, 2.7, -1.5))
ceiling(x)
ceiling(nv_array(1:3)) # an integer array is already whole
```

---

nv_chol *Cholesky Decomposition*

Description

Computes the Cholesky decomposition of a symmetric positive-definite matrix. Supports batched inputs: axes before the last two are batch axes.

Usage

```
nv_chol(x, lower = FALSE)

## S3 method for class 'AnvlArray'
chol(x, ..., lower = FALSE)
```

Arguments

x	(arrayish) One input, a symmetric positive-definite matrix with at least 2 axes, the last two forming the square matrix and any leading ones batch axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.
lower	(logical(1)) If TRUE, return the lower-triangular factor.
...	No additional arguments.

Details

Differentiation is only implemented for a single matrix: a gradient() of a batched decomposition errors.

Value

(arrayish)
Triangular matrix with the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one. The values in the triangle not selected by lower are implementation-defined. A matrix that is not positive definite factors to NaN.

See Also

nv_solve(), prim_chol()

Examples

```
# the factor has the matrix's shape and data type
a <- nv_matrix(c(4, 2, 2, 3), nrow = 2, dtype = "f32")
nv_chol(a)

# an integer matrix is factored at the default float data type
nv_chol(nv_matrix(c(4L, 2L, 2L, 3L), nrow = 2))
```

---

nv_clamp *Clamp*

Description

Element-wise clamp: min(max(min, x), max).

Usage

```
nv_clamp(x, min, max)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
min, max	(arrayish) Lower and upper bound, each scalar or the same shape as x. They are brought to x's data type: an R value is built at it when its category can reach it (0L serves an integer and a float x alike, 0 only a float one), and a value that already has a data type is converted unless that would narrow it – an f64 bound for an f32 x is an error rather than a silent narrowing.

Value

(arrayish)
Has x's shape and data type.

See Also

prim_clamp() for the underlying primitive.

Examples

```
# the bounds are brought to `x`'s data type
x <- nv_array(c(-1, 0.5, 2))
nv_clamp(x, nv_scalar(0), nv_scalar(1))

# an R integer serves a float `x` too, since a float can hold it
nv_clamp(x, 0L, 1L)
```

---

nv_concatenate *Concatenate*

Description

Concatenates arrays along an axis. Operands are promoted to a common data type and scalars are broadcast before concatenation.

You can also use c() on scalars and 1-D arrays, like base R.

Usage

```
nv_concatenate(..., axis = NULL)

## S3 method for class 'AnvlArray'
c(...)
```

Arguments

`...`	(`arrayish`) Arrays to concatenate. Can be of any data type; they are promoted to a common data type and scalars are broadcast. Must have the same shape except along `axis`.
`axis`	(`integer(1)` | `NULL`) Axis along which to concatenate. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), concatenates along axis 1, which requires every input to have at most one axis; for anything else `axis` must be given, since there is no neutral axis to join two matrices along.

Value

(`arrayish`)
Has the common data type and a shape matching the inputs in all axes except `axis`, which is the sum of input sizes.

The `c()` generic

`c()` concatenates scalars and 1-D arrays into a 1-D array, like `base::c()` does for vectors. An array with more than one axis is an error rather than being flattened the way base R does; concatenate those along an explicit `axis`, or `nv_flatten()` them first.

See Also

`prim_concatenate()` for the underlying primitive.

Examples

```
# the operands are promoted to a common data type; axis 1 grows to 6
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
nv_concatenate(x, y)

m <- nv_matrix(1:4, nrow = 2)
nv_concatenate(m, m, axis = 1L) # required: `m` has two axes
```

---

nv_conv1d *1D Convolution*

Description

Torch-style 1D convolution in NCW layout: `x` is `[batch, in_channels, width]`, `kernel` is `[out_channels, in_channels / groups, kW]`, output is `[batch, out_channels, out_w]`. Symmetric zero padding.

Usage

```
nv_conv1d(
  x,
  kernel,
  stride = 1L,
  padding = 0L,
  dilation = 1L,
  groups = 1L,
  precision = "highest"
)
```

Arguments

x	(arrayish) `[N, C_in, W]`. Can be any data type; `x` and `kernel` are promoted to a common data type. An R value assumes the other operand's data type, and materializes at its default data type when that has none either.
kernel	(arrayish) `[C_out, C_in / groups, kW]`. Promoted together with `x` – see `x`.
stride, padding, dilation	(integer()) Length 1.
groups	(integer(1)) Grouped/depthwise convolution.
precision	(character(1)) `"highest"`, `"high"` or `"default"`.

Value

(arrayish)
Has the operands' common data type, and shape `[N, C_out, out_W]`.

See Also

nv_conv2d(), nv_conv3d(), prim_convolution().

Examples

```
# one batch, one channel, width 5, convolved with a width-3 kernel
x <- nv_array(1:5, shape = c(1, 1, 5), dtype = "f32")
kernel <- nv_array(c(1, 0, -1), shape = c(1, 1, 3), dtype = "f32")
nv_conv1d(x, kernel)

# `padding = 1` keeps the input width, `stride = 2` visits every other
# window position
nv_conv1d(x, kernel, padding = 1L)
nv_conv1d(x, kernel, stride = 2L)
```

nv_conv2d *2D Convolution*

Description

Torch-style 2D convolution in NCHW layout: `x` is `[batch, in_channels, height, width]`, `kernel` is `[out_channels, in_channels / groups, kh, kw]`, output is `[batch, out_channels, out_h, out_w]`. Symmetric zero padding.

Usage

```
nv_conv2d(
  x,
  kernel,
  stride = 1L,
  padding = 0L,
  dilation = 1L,
  groups = 1L,
  precision = "highest"
)
```

Arguments

`x`	(arrayish) `[N, C_in, H, W]`. Can be any data type; `x` and `kernel` are promoted to a common data type. An R value assumes the other operand's data type, and materializes at its default data type when that has none either.
`kernel`	(arrayish) `[C_out, C_in / groups, kH, kW]`. Promoted together with `x` – see `x`.
`stride`	(`integer()`) Length 1 or 2.
`padding`	(`integer()`) Symmetric padding, length 1 or 2.
`dilation`	(`integer()`) Kernel dilation, length 1 or 2.
`groups`	(`integer(1)`) Grouped/depthwise convolution.
`precision`	(`character(1)`) `"highest"`, `"high"` or `"default"`.

Value

(arrayish)
Has the operands' common data type, and shape `[N, C_out, out_H, out_W]`.

See Also

nv_conv1d(), nv_conv3d(), prim_convolution().

Examples

```
# one batch, one channel, 4x4, convolved with a 3x3 kernel
x <- nv_array(1:16, shape = c(1, 1, 4, 4), dtype = "f32")
kernel <- nv_fill(1, shape = c(1, 1, 3, 3), dtype = "f32")
nv_conv2d(x, kernel)

# two output channels give a result with two channels
kernel2 <- nv_fill(1, shape = c(2, 1, 3, 3), dtype = "f32")
shape(nv_conv2d(x, kernel2))
```

---

nv_conv3d *3D Convolution*

Description

Torch-style 3D convolution in NCDHW layout. x is [batch, in_channels, depth, height, width], kernel is [out_channels, in_channels / groups, kD, kH, kW]. Asymmetric padding (e.g. causal temporal padding) is available via prim_convolution().

Usage

```
nv_conv3d(
  x,
  kernel,
  stride = 1L,
  padding = 0L,
  dilation = 1L,
  groups = 1L,
  precision = "highest"
)
```

Arguments

x	(arrayish) [N, C_in, D, H, W]. Can be any data type; x and kernel are promoted to a common data type. An R value assumes the other operand's data type, and materializes at its default data type when that has none either.
kernel	(arrayish) [C_out, C_in / groups, kD, kH, kW]. Promoted together with x – see x.
stride, padding, dilation	(integer()) Length 1 or 3.

groups (integer(1))
Grouped/depthwise convolution.

precision (character(1))
"highest", "high" or "default".

Value

(arrayish)
Has the operands' common data type, and shape [N, C_out, out_D, out_H, out_W].

See Also

nv_conv1d(), nv_conv2d(), prim_convolution().

Examples

```
# one batch, one channel, 2x3x3, convolved with a 1x2x2 kernel
x <- nv_array(1:18, shape = c(1, 1, 2, 3, 3), dtype = "f32")
kernel <- nv_fill(1, shape = c(1, 1, 1, 2, 2), dtype = "f32")
shape(nv_conv3d(x, kernel))
```

---

nv_convert *Convert Data Type*

Description

Converts the elements of an array to a different data type. Note that R objects are handled differently than AnvlArray inputs. For R objects, we check whether the requested data type can meaningfully hold the provided data and otherwise err. For AnvlArray inputs such a check is *not* performed.

Usage

```
nv_convert(x, dtype)
```

Arguments

x (arrayish)
The input to convert.

dtype (character(1) | DataType)
Target data type.

Value

(arrayish)
Has the given dtype and the input's shape.

See Also

prim_convert() for the underlying primitive.

Examples

```
# the values are preserved, the data type changes
x <- nv_array(c(1L, 2L, 3L))
# For R inputs, we check compatability
nv_convert(x, dtype = "f32")
try(nv_convert(257L, dtype = "i8"))
# For AnvlArrays, we wrap around
nv_convert(nv_scalar(257L), dtype = "i8")
```

---

nv_cos *Cosine*

Description

Element-wise cosine. You can also use cos().

Usage

```
nv_cos(x)

## S3 method for class 'AnvlArray'
cos(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_cos() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, pi / 2, pi))
cos(x)

# an R value materializes at its default data type
nv_cos(0)
```

---

nv_cosh *Hyperbolic Cosine*

Description

Element-wise hyperbolic cosine. You can also use `cosh()`.

Usage

```
nv_cosh(x)

## S3 method for class 'AnvlArray'
cosh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_cosh()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
cosh(x)

# an R value materializes at its default data type
nv_cosh(1)
```

nv_cospi *Cosine of a Multiple of Pi*

Description

Element-wise `cos(pi * x)`. You can also use `cospi()`.

Usage

```
nv_cospi(x)

## S3 method for class 'AnvlArray'
cospi(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`nv_sinpi()`, `nv_tanpi()`, `nv_cos()`

Examples

```
cospi(nv_array(c(0, 0.5, 1, 1.5)))
```

nv_crossprod *Cross Product (Matrix)*

Description

Computes `t(x) %*% y`. If `y` is missing, computes `t(x) %*% x`. Above rank 2 the last two axes are the matrix and the leading ones are batch axes, as in `nv_matmul()`: only the matrix is transposed.

Usage

```
nv_crossprod(x, y = NULL)

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

Arguments

x	(arrayish) An array with at least 2 axes, as for base::crossprod(). Can be any numeric data type; x and y are promoted to a common data type.
y	(arrayish | NULL) Optional second array. If NULL, uses x.
...	No additional arguments.

Value

(arrayish)
Has the operands' common data type, and the shape of `t(x) %*% y`.

See Also

nv_tcrossprod(), nv_matmul()

Examples

```
# `t(x) %*% x`, so a 3x2 gives a 2x2
x <- nv_matrix(1:6, nrow = 3, dtype = "f32")
nv_crossprod(x)
```

---

nv_cummax *Cumulative Maximum*

Description

Running maximum, optionally along a single axis.

Usage

```
nv_cummax(x, axis = NULL, indices = FALSE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
cummax(x)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1) | NULL) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), the input is first flattened to a 1-D array, like base::cummax().
indices	(logical(1)) If FALSE (default), returns the running-maximum array. If TRUE, returns list(values = ..., indices = ...) where indices is the index of the last occurrence of the running maximum at each position, of the default integer data type (see default_dtypes()). When axis = NULL, indices refer to the flattened input.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates forward from its first occurrence. If TRUE, NaN is treated as the identity element of the cumulative op (0 for sum, 1 for prod, -Inf / +Inf for max / min) and contributes nothing to the running value.

Value

(arrayish | named list of two arrayish)
One array when indices = FALSE, a named list of values and indices when indices = TRUE. The values have the input's data type and the indices the default integer data type; both have the input's shape when axis is given, and are 1-D of length prod(shape(x)) when axis is NULL.

See Also

prim_cummax() for the underlying primitive.

Examples

```
# the running maximum keeps the data type; the indices are the default integer
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
nv_cummax(x)
nv_cummax(x, axis = 1L)
nv_cummax(x, axis = 1L, indices = TRUE)
nv_cummax(nv_array(c(1, NaN, 3)))                # NaN propagates
nv_cummax(nv_array(c(1, NaN, 3)), nan_rm = TRUE) # NaN skipped
```

---

nv_cummin *Cumulative Minimum*

Description

Running minimum, optionally along a single axis.

Usage

```
nv_cummin(x, axis = NULL, indices = FALSE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
cummin(x)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1) | NULL) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), the input is first flattened to a 1-D array, like base::cummin().
indices	(logical(1)) If FALSE (default), returns the running-minimum array. If TRUE, returns list(values = ..., indices = ...) where indices is the index of the last occurrence of the running minimum at each position, of the default integer data type (see default_dtypes()). When axis = NULL, indices refer to the flattened input.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates forward from its first occurrence. If TRUE, NaN is treated as the identity element of the cumulative op (0 for sum, 1 for prod, -Inf / +Inf for max / min) and contributes nothing to the running value.

Value

(arrayish | named list of two arrayish)
One array when indices = FALSE, a named list of values and indices when indices = TRUE. The values have the input's data type and the indices the default integer data type; both have the input's shape when axis is given, and are 1-D of length prod(shape(x)) when axis is NULL.

See Also

prim_cummin() for the underlying primitive.

Examples

```
# the running minimum keeps the data type; the indices are the default integer
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
nv_cummin(x)
nv_cummin(x, axis = 1L)
nv_cummin(x, axis = 1L, indices = TRUE)
nv_cummin(nv_array(c(3, NaN, 1)))               # NaN propagates
nv_cummin(nv_array(c(3, NaN, 1)), nan_rm = TRUE) # NaN skipped
```

nv_cumprod *Cumulative Product*

Description

Cumulative product, optionally along a single axis. A boolean array is multiplied as zeroes and ones, like `base::cumprod()` does.

Usage

```
nv_cumprod(x, axis = NULL, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
cumprod(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axis`	(`integer(1)` | `NULL`) Axis along which to accumulate. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), the input is first flattened to a 1-D array, like `base::cumprod()`.
`nan_rm`	(`logical(1)`) How to handle `NaN` values in float inputs. If `FALSE` (default), `NaN` propagates forward from its first occurrence. If `TRUE`, `NaN` is treated as the identity element of the cumulative op (`0` for sum, `1` for prod, `-Inf` / `+Inf` for max / min) and contributes nothing to the running value.

Value

(`arrayish`)
Has the input's shape when `axis` is given, and is 1-D of length `prod(shape(x))` when `axis` is `NULL`, which flattens first. Has the input's data type, except for a boolean input, which is accumulated at the default integer data type (see `default_dtypes()`).

See Also

`prim_cumprod()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_cumprod(x)              # flatten, then accumulate
nv_cumprod(x, axis = 1L)    # accumulate along rows
nv_cumprod(nv_array(c(2, NaN, 3)))               # NaN propagates
```

```
nv_cumprod(nv_array(c(2, NaN, 3)), nan_rm = TRUE) # NaN treated as 1
```

`nv_cumsum` *Cumulative Sum*

Description

Cumulative sum, optionally along a single axis. A boolean array is counted, like `base::cumsum()` does.

Usage

```
nv_cumsum(x, axis = NULL, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
cumsum(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axis`	(`integer(1)` | `NULL`) Axis along which to accumulate. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), the input is first flattened to a 1-D array, like `base::cumsum()`.
`nan_rm`	(`logical(1)`) How to handle `NaN` values in float inputs. If `FALSE` (default), `NaN` propagates forward from its first occurrence. If `TRUE`, `NaN` is treated as the identity element of the cumulative op (`0` for sum, `1` for prod, `-Inf` / `+Inf` for max / min) and contributes nothing to the running value.

Value

(`arrayish`)
Has the input's shape when `axis` is given, and is 1-D of length `prod(shape(x))` when `axis` is `NULL`, which flattens first. Has the input's data type, except for a boolean input, which is accumulated at the default integer data type (see `default_dtypes()`).

See Also

`prim_cumsum()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_cumsum(x)              # flatten, then accumulate
nv_cumsum(x, axis = 1L)    # accumulate along rows
nv_cumsum(nv_array(c(1, NaN, 3)))               # NaN propagates
nv_cumsum(nv_array(c(1, NaN, 3)), nan_rm = TRUE) # NaN treated as 0
```

---

nv_det *Determinant*

Description

Computes the determinant of a square matrix via nv_determinant(). A singular matrix has determinant 0.

Usage

```
nv_det(x)
```

Arguments

x	(arrayish) One input, a square matrix with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
A scalar with the input's data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

nv_determinant(), nv_solve(), prim_lu()

Examples

```
# a scalar with the matrix's data type
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_det(a)
```

nv_determinant *Determinant in Modulus/Sign Form*

Description

Computes the determinant of a square matrix in the modulus / sign decomposition matching base R's `base::determinant()`. For the plain scalar determinant, use `nv_det()`.

Usage

```
nv_determinant(x, logarithm = TRUE)

## S3 method for class 'AnvlArray'
determinant(x, logarithm = TRUE, ...)
```

Arguments

`x`	(`arrayish`) One input, a square matrix with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.
`logarithm`	(`logical(1)`) If `TRUE` (default), return the log of the absolute determinant.
`...`	No additional arguments.

Details

For computing the determinant, we use:

$$PA = LU$$

$$\det(L) = 1$$

$$\det(A) = \det(U)/\det(P) = \text{sign}(P^{-1}) \prod_i U_{ii} = \text{sign}(P) \prod_i U_{ii}$$

Matching base R's `det_ge_real`, the magnitude is computed in log space when `logarithm = TRUE` ($\sum_i \log |U_{ii}|$) and as a direct product when `logarithm = FALSE` ($\prod_i |U_{ii}|$).

A singular matrix has determinant `0`: `modulus` is `-Inf` (with `logarithm = TRUE`) or `0` (with `logarithm = FALSE`), and `sign` is `0`. This deviates from `base::determinant()`, which reports `sign 1`.

Value

(named `list` of two `arrayish`)
Elements `modulus` and `sign`, both scalar `arrayish` with `x`'s data type – or the default float data type (see `default_dtypes()`) where `x` was an integer one. The full determinant is `sign * exp(modulus)` (with `logarithm = TRUE`) or `sign * modulus` (with `logarithm = FALSE`).

See Also

nv_det(), nv_solve(), prim_lu()

Examples

```
# `modulus` and `sign` are scalars with the matrix's data type
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_determinant(a)
nv_determinant(a, logarithm = FALSE)
```

---

nv_device *Create a Device*

Description

Constructs a backend-specific device object for the active backend (active_backend()).

A device identifies a compute resource, such as a CPU, or a specific GPU. It is relevant for data allocation (e.g. via nv_array()) but also compilation (jit()). A device is constructed for the active backend.

Usage

```
nv_device(x)
```

Arguments

x	(character(1) | device object) Identifier for the device (e.g. "cpu", "cuda", "cuda:<n>"), or an existing device object of the active backend (returned as-is).

Value

(device object)
Backend-specific (e.g. PJRTDevice for "pjrt", quickr_device for "quickr").

See Also

backend(), AnvlBackend(), active_backend().

Examples

```
# create CPU device for the active backend
nv_device("cpu")
# create CPU device for the quickr backend:
with_backend("quickr", nv_device("cpu"))
# pass through an existing device:
dev <- nv_device("cpu")
identical(nv_device(dev), dev)
```

nv_diag *Diagonal Matrix*

Description

Creates a diagonal matrix from a 1-D array.

Usage

```
nv_diag(x)
```

Arguments

x	(arrayish) One input, a 1-D array of length n whose elements become the diagonal entries. Can be any data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's data type and shape `(n, n)`, with the input on the diagonal and zeros elsewhere.

Examples

```
# the vector becomes the diagonal of a 3x3 matrix
nv_diag(nv_array(c(1, 2, 3)))
```

nv_digamma *Digamma*

Description

Element-wise digamma function (logarithmic derivative of the gamma function). You can also use `digamma()`.

Usage

```
nv_digamma(x)

## S3 method for class 'AnvlArray'
digamma(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_digamma() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0.5, 1, 2, 5))
digamma(x)

# an R value materializes at its default data type
nv_digamma(2)
```

---

nv_div *Division*

Description

Divides two arrays element-wise. You can also use the / operator.

Like base R's /, this is a true division: integer and boolean operands are converted to the default float before dividing, so 7L / 2L is 3.5. Use nv_floor_div() (%/%) for integer division.

Usage

```
nv_div(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 / e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and their common data type, or the default float when that is not a float type.

See Also

prim_div() for the underlying primitive, which divides integers with truncation.

Examples

```
x <- nv_array(c(10, 20, 30))
y <- nv_array(c(2, 5, 10))
nv_div(x, y)
x / y

# different data types are promoted to their common one
nv_div(nv_scalar(10, "f32"), nv_scalar(4, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x / 2L

# integers are divided as floats
nv_array(c(7L, -7L)) / 2L
```

---

nv_dtype_constructors *Create an Array of a Given Data Type*

Description

Shorthands for nv_array() with the data type fixed by the function name: nv_float32(data) is nv_array(data, dtype = "f32").

Function	Data type
nv_bool()	bool
nv_int8()	i8

```
nv_int16()    i16
nv_int32()    i32
nv_int64()    i64
nv_uint8()    ui8
nv_uint16()   ui16
nv_uint32()   ui32
nv_uint64()   ui64
nv_float32()  f32
nv_float64()  f64
```

Usage

```
nv_bool(data, shape = NULL, device = NULL, byrow = FALSE)

nv_int8(data, shape = NULL, device = NULL, byrow = FALSE)

nv_int16(data, shape = NULL, device = NULL, byrow = FALSE)

nv_int32(data, shape = NULL, device = NULL, byrow = FALSE)

nv_int64(data, shape = NULL, device = NULL, byrow = FALSE)

nv_uint8(data, shape = NULL, device = NULL, byrow = FALSE)

nv_uint16(data, shape = NULL, device = NULL, byrow = FALSE)

nv_uint32(data, shape = NULL, device = NULL, byrow = FALSE)

nv_uint64(data, shape = NULL, device = NULL, byrow = FALSE)

nv_float32(data, shape = NULL, device = NULL, byrow = FALSE)

nv_float64(data, shape = NULL, device = NULL, byrow = FALSE)
```

Arguments

data	(any) `integer()`, `double()`, or `logical()` scalar, vector, or array. Alternatively a `raw()` vector holding the native little-endian byte payload of `prod(shape)` elements of `dtype`; both `dtype` and `shape` are then required (only supported on the `"pjrt"` backend). Raw payloads are read in column-major element order, or row-major with `byrow = TRUE`. An existing `AnvlArray` is returned unchanged if the `shape`, `dtype` and `device` given agree with it, and is an error otherwise.
shape	(`NULL` | `integer()`) The output shape of the array. The default (`NULL`) is to infer it from the data if possible. Note that `nv_array` interprets length 1 vectors as having shape `(1)`. Empty data has no shape to infer – `0`, `c(2, 0)` and `c(0, 3)` all hold no elements

– so `shape` is required there. To create a "scalar" with no axes (shape `()`), use `nv_scalar` or explicitly specify `shape = integer()`.

`device` (`NULL` | `character(1)` | device)
The device the data lives on, given either as:

- a *device string* naming the platform (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), which is resolved against the backend in use, or
- a *device object* as returned by `nv_device()`: a `PJRTDevice` for the `"pjrt"` backend or a `quickr_device` for the `"quickr"` backend. It must belong to the active backend (`active_backend()`); a device of another backend is an error.

The default (`NULL`) uses `default_device()`.

`byrow` (`logical(1)`)
When constructing from an R object and the result has at least two axes, fill the array in row-major order rather than the default column-major order. Only allowed when `data` is an R object — passing an existing `AnvlArray` together with `byrow = TRUE` is an error.

Value

(`AnvlArray`)

Out of Range values

Because base R has fewer data types than anvl, creating `AnvlArrays` from R often involves type conversions. When such conversions are performed, anvl performs a scan of the inputs to ensure that the requested data type can actually hold the input data. For example, trying to create an unsigned integer from a negative R `integer()` fails. The same holds where an R value takes its data type from the array it meets rather than from an argument: `nv_scalar(1L, "ui8") + (-2L)` is refused, where converting an array with `nv_convert()` wraps around.

See Also

`nv_array()`, `nv_convert()` to change the data type of an existing array.

Examples

```
nv_float32(1:4)
nv_uint8(c(0, 255))
nv_int64(1:6, shape = c(2, 3))
nv_bool(c(TRUE, FALSE))
```

nv_eigh *Symmetric Eigendecomposition*

Description

Computes the eigendecomposition of a symmetric matrix `x` of shape `(n, n)`:

$$A = \text{vectors}\,\text{diag}(\text{values})\,\text{vectors}^\top.$$

Only the lower triangle of `x` is read. The columns of `vectors` are the (orthonormal) eigenvectors and `values` is the length-`n` vector of (real) eigenvalues in ascending order. Output names and order match `base::eigen()`, which unlike this primitive also handles non-symmetric matrices.

Usage

```
nv_eigh(x)
```

Arguments

`x`	(`arrayish`) One input, a symmetric square matrix with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(named `list` of two `arrayish`)
Elements `values` (length `n`) and `vectors` (shape `(n, n)`). Both have the input's data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_eigh()`, `base::eigen()`

Examples

```
# values and vectors both have the input's data type
x <- nv_matrix(c(2, 1, 1, 2), nrow = 2, dtype = "f64")
nv_eigh(x)

# an integer matrix is decomposed at the default float data type
nv_eigh(nv_matrix(c(2L, 1L, 1L, 2L), nrow = 2))
```

nv_eq *Equal*

Description

Element-wise equality comparison. You can also use the == operator.

Usage

```
nv_eq(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 == e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and boolean data type.

See Also

prim_eq() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
nv_eq(x, y)
x == y

# different data types are promoted to their common one
nv_eq(nv_scalar(1, "f32"), nv_scalar(1, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x == 2L
```

nv_erf *Error Function*

Description

Element-wise error function `erf(x) = (2 / sqrt(pi)) * integral_0^x exp(-t^2) dt`.

Usage

```
nv_erf(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_erf() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
nv_erf(x)

# an R value materializes at its default data type
nv_erf(1)
```

nv_erf_inv *Inverse Error Function*

Description

Element-wise inverse error function (the inverse of `erf` on `(-1, 1)`).

Usage

```
nv_erf_inv(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_erf_inv() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-0.5, 0, 0.5))
nv_erf_inv(x)

# an R value materializes at its default data type
nv_erf_inv(0.5)
```

---

nv_erfc *Complementary Error Function*

Description

Element-wise complementary error function `erfc(x) = 1 - erf(x)`.

Usage

```
nv_erfc(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_erfc() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
nv_erfc(x)

# an R value materializes at its default data type
nv_erfc(1)
```

---

nv_exp *Exponential*

Description

Element-wise exponential. You can also use exp().

Usage

```
nv_exp(x)

## S3 method for class 'AnvlArray'
exp(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_exp() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 1, 2))
exp(x)

# an R value materializes at its default data type
nv_exp(1)
```

---

nv_expm1 *Exponential Minus One*

Description

Element-wise `exp(x) - 1`, more accurate for small `x`. You can also use `expm1()`.

Usage

```
nv_expm1(x)

## S3 method for class 'AnvlArray'
expm1(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_expm1()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.001, 1))
nv_expm1(x)

# an R value materializes at its default data type
nv_expm1(0.001)
```

nv_exponential *The Exponential Distribution*

Description

Density (nv_dexp), distribution function (nv_pexp), and quantile function (nv_qexp) for the Exponential distribution with rate rate (i.e., mean 1 / rate).

Usage

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

nv_pexp(q, rate = 1, lower_tail = TRUE, log_p = FALSE)

nv_qexp(p, rate = 1, lower_tail = TRUE, log_p = FALSE)
```

Arguments

x, q	(arrayish) Quantiles at which to evaluate the density (x) or the distribution function (q).
rate	(arrayish) Rate of the distribution. Scalars broadcast; otherwise rate and x/q/p must have the same shape. Negative rates (including -Inf and negative zero) give NaN.
log, log_p	(logical(1)) If TRUE, the densities/probabilities are given as logarithms. For nv_qexp this describes the input p.
lower_tail	(logical(1)) If TRUE (default), probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
p	(arrayish) Probabilities at which to evaluate the quantile function. Values outside $[0, 1]$ give NaN. With log_p = TRUE, supply log-probabilities in $[-\infty, 0]$ instead.

Details

The Exponential distribution has probability density function:

$$f(x) = \lambda e^{-\lambda x}, \quad x \geq 0$$

and zero elsewhere, where λ is rate.

For positive finite rates, the mean is 1 / rate. Zero and infinite rates follow base R's endpoint conventions:

- rate = 0: the density and lower-tail probability are zero at finite quantiles, but NaN at infinity. Quantiles are infinite except at lower-tail probability zero, where the result is zero.
- rate = Inf: densities are NaN. The lower-tail probability is zero at or below zero and one above zero. Quantiles are zero except at lower-tail probability one, where the result is NaN.

Value

(arrayish)
nv_dexp(), nv_pexp(), and nv_qexp() return an arrayish with the float data type described under Data Types and the shape after scalar broadcasting with rate.

Data Types

nv_dexp(), nv_pexp() and nv_qexp() compute at a float data type and return it. If x/q/p is a float, they compute at its data type; an R double settles on the default float. An integer x/q/p (an array or an R integer) is converted to the widest float among the arrays given for rate, or to the default float if there is none. Integers beyond 2^{24} are not all exact at f32, so a large integer may change in that conversion.

An R value for rate takes the computing data type. An array is promoted to it, so a wider one (e.g. f64 for an f32 x) is an error.

Examples

```
x <- nv_array(c(-1, 0, 0.5, 1, 2))
nv_dexp(x)
nv_dexp(x, rate = 2)
nv_dexp(x, log = TRUE)

nv_pexp(x)
nv_pexp(x, rate = 2)
nv_pexp(x, lower_tail = FALSE)
nv_pexp(x, log_p = TRUE)

p <- nv_array(c(0.025, 0.5, 0.975))
nv_qexp(p)
nv_qexp(p, rate = 2)
nv_qexp(p, lower_tail = FALSE)
nv_qexp(nv_array(c(-700, -2, -0.1), dtype = "f64"), log_p = TRUE)
```

---

nv_extract_diag *Extract Diagonal*

Description

Extracts the diagonal elements from a 2-D array.

Usage

```
nv_extract_diag(x)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's data type, and one axis of size `min(nrow, ncol)` holding the diagonal elements.

See Also

`nv_diag()` for creating a diagonal matrix, `nv_trace()`

Examples

```
# the diagonal of a 3x3 matrix, keeping its data type
x <- nv_array(1:9, shape = c(3, 3))
nv_extract_diag(x)
```

---

nv_eye *Identity Matrix*

Description

Creates an `n x n` identity matrix.

`nv_eye_like()` is a variant where `dtype` and `device` default to those of `like`.

Usage

```
nv_eye(n, dtype = NULL, device = NULL)

nv_eye_like(like, n, dtype = NULL, device = NULL)
```

Arguments

n	(integer(1)) Size of the identity matrix.
dtype	(NULL | character(1) | DataType) Data type of the result. Can be any data type; NULL (default) uses the default float data type (see `default_dtypes()`). For `nv_eye_like()`, NULL uses `dtype(like)`.
device	(NULL | character(1) | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or

- a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error.

The default (NULL) uses default_device().

like (AnvlArray)
Existing array whose attributes are used as defaults (only for nv_eye_like()).

Value

(arrayish)
Has the given dtype and shape (n, n): ones on the diagonal, zeros elsewhere.

See Also

nv_diag() for general diagonal matrices.

Examples

```
# a 3x3 identity matrix
nv_eye(3L)

# `_like` takes the data type and device from an existing array
x <- nv_fill(0, shape = c(3, 3), dtype = "f64")
nv_eye_like(x, 3L)
```

---

nv_fill *Fill Constant*

Description

Creates an array filled with a scalar value. More memory-efficient than nv_array(value, shape = shape) for large arrays.

nv_fill_like() is a variant where dtype, shape, and device default to those of like.

Usage

```
nv_fill(value, shape, dtype = NULL, device = NULL)

nv_fill_like(like, value, shape = NULL, dtype = NULL, device = NULL)
```

Arguments

value (numeric(1) | logical(1))
Scalar value to fill the array with. It has to be something dtype can hold: a whole number in its range for an integer data type, a non-negative one for an unsigned integer, and a logical or 0 / 1 for bool.

shape (integer())
Shape of the output array.

dtype (NULL | character(1) | DataType)
Data type of the result. The default (NULL) uses the default data type for nv_fill and dtype(like) for nv_fill_like.

device (NULL | character(1) | device)
The device the data lives on, given either as:

- a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or
- a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error.

The default (NULL) uses default_device().

like (AnvlArray)
Existing array whose attributes are used as defaults (only for nv_fill_like()).

Value

(arrayish)
Has the given shape and dtype.

See Also

prim_fill() for the underlying primitive.

Examples

```
# the R double settles the data type, the shape is given
nv_fill(0, shape = c(2, 3))

# `_like` takes shape, data type and device from an existing array
x <- nv_matrix(1:6, nrow = 2)
nv_fill_like(x, 0)
```

nv_flatten *Flatten*

Description

Flattens an array with one or more axes into a 1-D array, using col-major semantics.

Usage

```
nv_flatten(x)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's data type, and one axis holding all of its elements – so a scalar, which has none, becomes a length-1 vector.

Examples

```
# the 2x2 matrix becomes a length-4 vector
x <- matrix(1:4, nrow = 2)
nv_flatten(x)
# the same column-major order base R uses
c(x)
```

nv_floor *Floor*

Description

Element-wise floor (round toward negative infinity). You can also use `floor()`.

Usage

```
nv_floor(x)

## S3 method for class 'AnvlArray'
floor(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float is rounded and keeps its own, and an integer one is already whole and is returned unchanged. An R value materializes at its default data type and is treated in the same way.

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_floor() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1.2, 2.7, -1.5))
floor(x)
floor(nv_array(1:3)) # an integer array is already whole
```

---

nv_floor_div *Flooring Division*

Description

Element-wise flooring division. You can also call this via the %/% operator. The result is the largest whole number that does not exceed lhs / rhs.

Usage

```
nv_floor_div(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 %/% e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

nv_mod() for the matching remainder, nv_div() for the division itself.

Examples

```
x <- nv_array(c(7L, -7L))
y <- nv_array(c(2L, 2L))
nv_floor_div(x, y)
x %/% y
```

---

nv_gamma *Gamma Function*

Description

Element-wise gamma function. You can also use gamma().

Usage

```
nv_gamma(x)

## S3 method for class 'AnvlArray'
gamma(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

nv_lgamma(), which is what the hardware computes.

Examples

```
gamma(nv_array(c(0.5, 1, 5, -1.5)))
```

---

nv_ge *Greater Than or Equal*

Description

Element-wise greater than or equal comparison. You can also use the >= operator.

Usage

```
nv_ge(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 >= e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and boolean data type.

See Also

prim_ge() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
nv_ge(x, y)
x >= y

# different data types are promoted to their common one
nv_ge(nv_scalar(2, "f32"), nv_scalar(1, "f64"))

# a scalar is broadcast and an R integer is converted to a float
```

```
x >= 2L
```

---

nv_gt *Greater Than*

Description

Element-wise greater than comparison. You can also use the > operator.

Usage

```
nv_gt(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 > e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and boolean data type.

See Also

prim_gt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
nv_gt(x, y)
x > y

# different data types are promoted to their common one
nv_gt(nv_scalar(2, "f32"), nv_scalar(1, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x > 2L
```

nv_if *Conditional Branching*

Description

Conditional execution of two branches, mirroring R's `if` construct: it branches between two *functions* and evaluates only the selected one. Its arguments are named after that construct, where `nv_ifelse()` – which selects element-wise between two *arrays* – is named after `ifelse()`.

Usage

```
nv_if(pred, true, false)
```

Arguments

`pred`	(`arrayish`) Predicate. Must be a scalar of the boolean data type, or an R logical.
`true`	(`function()`) Zero-argument function for the true branch.
`false`	(`function()`) Zero-argument function for the false branch. Must return the same structure, data types and shapes as the true branch; nothing is promoted.

Value

(`arrayish | list`)
Result of the executed branch: an array, or a tree of them in the sense of pjrt's `RTree` – a `list`, nested arbitrarily – with the structure, data types and shapes both branches share.

Gradients

`gradient()` differentiates through the branch the predicate selects, including the values the branches close over; a value only the other branch uses gets a zero. See `prim_if()`.

See Also

`prim_if()` for the underlying primitive, `nv_ifelse()` for element-wise selection.

Examples

```
# both branches must return the same structure, data types and shapes
nv_if(nv_scalar(TRUE), \() nv_scalar(1), \() nv_scalar(2))

# the gradient flows through the branch that is taken
f <- function(p, x) nv_if(p, \() sum(x * x), \() sum(x))
jit(gradient(f, wrt = "x"))(nv_scalar(TRUE), nv_array(c(1, 2, 3)))
```

`nv_ifelse` *Conditional Element Selection*

Description

Selects elements from `yes` or `no` based on `test`, mirroring R's `ifelse()`.

`nv_if()` is the other conditional: it mirrors R's `if` construct and branches between two *functions*, evaluating only the selected one.

Usage

```
nv_ifelse(test, yes, no)
```

Arguments

`test`	(`arrayish`) Predicate array. Must be a boolean or an R logical, and scalar or the same shape as the non-scalar arguments.
`yes`, `no`	(`arrayish`) Values to return where `test` is `TRUE` / `FALSE`. Can be of any data type; the two are promoted to a common data type. Scalars (including `test`) are broadcast to the shape of the non-scalar arguments.

Value

(`arrayish`)
Has the common data type of `yes` and `no`, and their broadcast shape.

See Also

`prim_ifelse()` for the underlying primitive, `nv_if()`.

Examples

```
test <- nv_array(c(TRUE, FALSE, TRUE))
nv_ifelse(test, nv_array(c(1, 2, 3)), nv_array(c(4, 5, 6)))
# scalar branches are broadcast and promoted to a common data type
nv_ifelse(test, nv_scalar(1L), nv_scalar(0.5))
```

nv_inv *Matrix Inverse*

Description

Computes `x^-1`, the inverse of a square non-singular matrix `x`, by solving `x %*% y = I` for `y`. Unlike `base::solve()`, a singular `x` is not an error: the result then holds non-finite values.

For most use cases prefer `nv_solve()` directly: forming the explicit inverse is both slower and less numerically stable than solving against a right-hand side.

Usage

```
nv_inv(x)
```

Arguments

`x`	(`arrayish`) One input, a square non-singular matrix with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
The inverse, with the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`nv_solve()`

Examples

```
# the inverse has the matrix's shape and data type
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_inv(a)
```

nv_iota *Iota*

Description

Creates an array with values increasing along the specified axis, starting from `start`.

`nv_iota_like()` is a variant where `shape`, `dtype`, and `device` default to those of `like`.

Usage

```
nv_iota(axis, shape, dtype, start = 1L, device = NULL)

nv_iota_like(like, axis, shape = NULL, dtype = NULL, start = 1L, device = NULL)
```

Arguments

`axis`	(`integer(1)`) Axis along which values increase. Negative values count from the end of `shape`, i.e. `-1L` refers to the last axis.
`shape`	(`integer()`) Shape of the result.
`dtype`	(`character(1)` | `DataType`) Data type of the result, required here. Can be any numeric data type. For `nv_iota_like()` it may be `NULL`, which uses `dtype(like)`.
`start`	(`integer(1)`) Starting value (default 1).
`device`	(`NULL` | `character(1)` | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), which is resolved against the backend in use, or • a *device object* as returned by `nv_device()`: a `PJRTDevice` for the `"pjrt"` backend or a `quickr_device` for the `"quickr"` backend. It must belong to the active backend (`active_backend()`); a device of another backend is an error. The default (`NULL`) uses `default_device()`.
`like`	(`AnvlArray`) Existing array whose attributes are used as defaults (only for `nv_iota_like()`).

Value

(`arrayish`)
Has the given `dtype` and `shape`.

See Also

nv_seq() for a simpler 1-D sequence, nv_linspace() for evenly spaced values, prim_iota() for the underlying primitive.

Examples

```
# the sequence is built at the requested data type
nv_iota(axis = 1L, shape = 5L, dtype = "i32")

# `_like` takes shape, data type and device from an existing array
x <- nv_fill(0L, shape = c(2, 3))
nv_iota_like(x, axis = 1L)
```

---

nv_is_finite *Is Finite*

Description

Element-wise check if values are finite (not Inf, -Inf, or NaN). Only a float holds a non-finite value, so the answer for any other data type is all TRUE and is built as a constant rather than computed, the way base R's is.finite() answers TRUE for an integer.

Usage

```
nv_is_finite(x)

## S3 method for class 'AnvlArray'
is.finite(x)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and boolean data type.

See Also

prim_is_finite() for the underlying primitive, which takes a float only.

Examples

```
# the result is boolean, whatever float data type the input has
x <- nv_array(c(1, Inf, NaN, -Inf, 0))
nv_is_finite(x)

# all TRUE for an integer input, which has no non-finite value
nv_is_finite(nv_array(1:3))

# an R value materializes at its default data type before the test
nv_is_finite(1)
```

---

nv_is_infinite *Is Infinite*

Description

Element-wise check if values are infinite (`Inf` or `-Inf`). You can also use `is.infinite()`. Only a float holds an infinity, so the answer for any other data type is all `FALSE` and is built as a constant rather than computed.

Usage

```
nv_is_infinite(x)

## S3 method for class 'AnvlArray'
is.infinite(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and boolean data type.

See Also

`nv_is_finite()`, `nv_is_nan()`

Examples

```
# the result is boolean, whatever float data type the input has
x <- nv_array(c(1, NaN, Inf, -Inf, 0))
nv_is_infinite(x)

# all FALSE for an integer input, which has no infinity
nv_is_infinite(nv_array(1:3))

# an R value materializes at its default data type before the test
nv_is_infinite(1)
```

`nv_is_nan` *Is NaN*

Description

Element-wise check if values are NaN. You can also use `is.nan()`. Only a float holds a NaN, so the answer for any other data type is all `FALSE` and is built as a constant rather than computed.

Usage

```
nv_is_nan(x)

## S3 method for class 'AnvlArray'
is.nan(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and boolean data type.

See Also

`nv_is_finite()`, `nv_is_infinite()`

Examples

```
# a boolean result, whatever the input's data type
x <- nv_array(c(1, NaN, Inf, -Inf, 0))
nv_is_nan(x)

# all FALSE for an integer input, which has no NaN to find
```

```
nv_is_nan(nv_array(1:3))
```

`nv_le` *Less Than or Equal*

Description

Element-wise less than or equal comparison. You can also use the <= operator.

Usage

```
nv_le(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 <= e2
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
`e1`, `e2`	(`arrayish`) The operands of the operator, which it passes on as `lhs` and `rhs`.

Value

(`arrayish`)
Has the inputs' broadcast shape and boolean data type.

See Also

`prim_le()` for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
nv_le(x, y)
x <= y

# different data types are promoted to their common one
nv_le(nv_scalar(1, "f32"), nv_scalar(2, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x <= 2L
```

nv_lgamma *Log-Gamma*

Description

Element-wise natural logarithm of the absolute value of the gamma function. You can also use `lgamma()`.

Usage

```
nv_lgamma(x)

## S3 method for class 'AnvlArray'
lgamma(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_lgamma()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0.5, 1, 2, 5))
lgamma(x)

# an R value materializes at its default data type
nv_lgamma(2)
```

nv_linspace *Evenly Spaced Sequence*

Description

Creates a 1-D array with `length_out` evenly spaced values from `from` to `to` (both inclusive), like R's `seq(from, to, length.out = length_out)`.

The spacing `(to - from) / (length_out - 1)` is generally not a whole number, so the result is a float.

`nv_linspace_like()` is a variant where `dtype` and `device` default to those of `like`.

Usage

```
nv_linspace(from, to, length_out, dtype = NULL, device = NULL)

nv_linspace_like(like, from, to, length_out, dtype = NULL, device = NULL)
```

Arguments

`from`, `to`	(`numeric(1)`) First and last value of the sequence. `to` may lie below `from`, in which case the values decrease.
`length_out`	(`integer(1)`) Number of values to generate. Must be at least 1; for `length_out = 1` the result is `from`.
`dtype`	(`NULL` | `character(1)` | `DataType`) Data type of the result. Must be a float data type; `NULL` (default) uses the default float data type (see `default_dtypes()`), since the spacing is fractional. For `nv_linspace_like()`, `NULL` uses `dtype(like)`, which must then be a float too.
`device`	(`NULL` | `character(1)` | `device`) The device the data lives on, given either as: • a *device string* naming the platform (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), which is resolved against the backend in use, or • a *device object* as returned by `nv_device()`: a `PJRTDevice` for the `"pjrt"` backend or a `quickr_device` for the `"quickr"` backend. It must belong to the active backend (`active_backend()`); a device of another backend is an error. The default (`NULL`) uses `default_device()`.
`like`	(`AnvlArray`) Existing array whose attributes are used as defaults (only for `nv_linspace_like()`).

Value

(`arrayish`)
Has `dtype` and shape `length_out`.

See Also

nv_seq() for consecutive integers, nv_iota() for values increasing along an axis of any shape, dtype_categories for the data type categories.

Examples

```
nv_linspace(0, 1, length_out = 5L)

# to below from counts down
nv_linspace(1, 0, length_out = 3L)

# length_out = 1 gives from alone
nv_linspace(2.5, 10, length_out = 1L)

# the data type must be a float; convert afterwards for integers
nv_convert(nv_linspace(0, 10, length_out = 5L), "i32")

# nv_linspace_like() takes the data type and device from an existing array
x <- nv_array(c(1, 2, 3), dtype = "f64")
nv_linspace_like(x, 0, 1, length_out = 3L)
```

---

nv_log *Natural Logarithm*

Description

Element-wise natural logarithm. You can also use log().

Usage

```
nv_log(x)

## S3 method for class 'AnvlArray'
log(x, base = exp(1))
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.
base	(numeric(1) | arrayish) Base of the logarithm; the natural logarithm by default.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_log() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 2.718, 7.389))
log(x)

# an R value materializes at its default data type
nv_log(2)
```

---

nv_log10 *Base-10 Logarithm*

Description

Element-wise base-10 logarithm. You can also use `log10()`.

Usage

```
nv_log10(x)

## S3 method for class 'AnvlArray'
log10(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

nv_log(), nv_log2()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 10, 100, 1000))
nv_log10(x)

# an R value materializes at its default data type
nv_log10(100)
```

---

nv_log1p *Log Plus One*

Description

Element-wise `log(1 + x)`, more accurate for small `x`. You can also use `log1p()`.

Usage

```
nv_log1p(x)

## S3 method for class 'AnvlArray'
log1p(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_log1p()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.001, 1))
nv_log1p(x)

# an R value materializes at its default data type
nv_log1p(0.001)
```

nv_log2 *Base-2 Logarithm*

Description

Element-wise base-2 logarithm. You can also use `log2()`.

Usage

```
nv_log2(x)

## S3 method for class 'AnvlArray'
log2(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`nv_log()`, `nv_log10()`

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 2, 4, 8))
nv_log2(x)

# an R value materializes at its default data type
nv_log2(8)
```

nv_lower_tri *Lower Triangular Mask*

Description

Returns a boolean matrix that is TRUE on and below the given diagonal, mirroring base R's `lower.tri()`. Use `nv_tril()` to zero out the other triangle of an existing array instead.

Usage

```
nv_lower_tri(shape, diagonal = -1L, device = NULL)

nv_lower_tri_like(like, shape = NULL, diagonal = -1L, device = NULL)
```

Arguments

shape	(`integer()`) Shape of the result: exactly two axis sizes, since the result is a matrix.
diagonal	(`integer(1)`) Diagonal offset, with the same meaning as in `nv_tril()`. The default `-1` excludes the main diagonal, matching `lower.tri()`; use `0` to include it, matching `lower.tri(diag = TRUE)`.
device	(`NULL` | `character(1)` | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), which is resolved against the backend in use, or • a *device object* as returned by `nv_device()`: a `PJRTDevice` for the `"pjrt"` backend or a `quickr_device` for the `"quickr"` backend. It must belong to the active backend (`active_backend()`); a device of another backend is an error. The default (`NULL`) uses `default_device()`.
like	(`AnvlArray`) Existing array whose attributes are used as defaults (only for `nv_lower_tri_like()`).

Value

(`arrayish`)
Has the given shape and boolean data type. It is a mask over positions, so no array data enters it – pass it to `nv_ifelse()` or multiply by it to use it.

See Also

`nv_upper_tri()`, `nv_tril()`, `prim_iota()` for the underlying primitive.

Examples

```
# a boolean mask, whatever the array it is later used with
nv_lower_tri(c(3, 3))
nv_lower_tri(c(3, 3), diagonal = 0L)
x <- nv_fill(0, shape = c(3, 3))
nv_lower_tri_like(x)
```

---

nv_lt *Less Than*

Description

Element-wise less than comparison. You can also use the < operator.

Usage

```
nv_lt(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 < e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and boolean data type.

See Also

prim_lt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
nv_lt(x, y)
x < y
```

```
# different data types are promoted to their common one
nv_lt(nv_scalar(1, "f32"), nv_scalar(2, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x < 2L
```

`nv_lu` *LU Decomposition*

Description

Computes the partial-pivoted LU decomposition of a matrix `x`:

$$PA = LU,$$

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular.

This function returns `L` and `U` as separate matrices. Use `prim_lu()` to get them in packed `LU` form.

Usage

```
nv_lu(x)
```

Arguments

`x`	(`arrayish`) One input, with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(named `list` of `arrayish`)
`L` and `U` have the input's data type – or the default float data type (see `default_dtypes()`) where the input was an integer one; `pivots` and `permutation` are indices at the default integer data type (see `default_dtypes()`).

- `L` – unit lower-triangular factor of shape `(m, k)`, where `(m, n) = shape(x)` and `k = min(m, n)`.
- `U` – upper-triangular factor of shape `(k, n)`.
- `pivots` – length `k`, at the default integer data type (see `default_dtypes()`). LAPACK-style sequential row swaps as returned by `getrf`.
- `permutation` – length `m`, at that same data type. A permutation vector representing P.

See Also

`prim_lu()`

Examples

```
# `L` and `U` keep the input's data type; the pivots are the default integer
x <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_lu(x)
```

`nv_matmul` *Matrix Multiplication*

Description

Matrix multiplication of two arrays. You can also use the `%*%` operator. Supports batched matrix multiplication when inputs have more than 2 axes.

Usage

```
nv_matmul(x, y, precision = "highest")

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

Arguments

`x`, `y`	(`arrayish`) Numeric arrays with at least 2 axes. Can be any numeric data type; the two are promoted to a common data type. An R value assumes the data type of the other operand, and materializes at its default data type when that has none either.
`precision`	(`character(1)`) Controls the trade-off between speed and numerical accuracy of the operation. One of `"highest"` (default), `"high"` or `"default"`. See `prim_dot_general()` for details.

Value

(`arrayish`)
Has the operands' common data type and the shape given under Shapes.

Shapes

- `x`: `(b1, ..., bk, m, n)`
- `y`: `(b1, ..., bk, n, p)`
- output: `(b1, ..., bk, m, p)`

See Also

`prim_dot_general()` for the underlying primitive.

Examples

```
# a 2x3 times a 3x2 gives a 2x2 at the operands' common data type
x <- nv_matrix(1:6, nrow = 2)
y <- nv_matrix(1:6, nrow = 3)
nv_matmul(x, y)
x %*% y
```

---

nv_max *Max Reduction*

Description

Finds the maximum of array elements along the specified axes.

Usage

```
nv_max(x, axes = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
max(..., na.rm = FALSE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.
...	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of nv_max(), such as axes.
na.rm	(logical(1)) Forwarded to nv_max()'s nan_rm argument.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

The `max()` generic

`max()` reduces over all axes and, like `base::max()`, takes several data arguments: `max(x, y)` is the largest element of both arrays. `na.rm` becomes `nan_rm`. Beyond base R, named arguments are passed on, so `max(x, axes = 1L)` reduces a single axis – but only when `x` is the only data argument.

See Also

`prim_max()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# no axes given: reduce over all of them
nv_max(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_max(x, axes = 1L)
nv_max(x, axes = 1L, drop = FALSE)

# negative axes count from the end
nv_max(x, axes = -1L)

# NaN propagates unless nan_rm = TRUE
nv_max(nv_array(c(1, NaN, 3)))
nv_max(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

`nv_mean` *Mean*

Description

Computes the arithmetic mean along the specified axes. You can also use `mean()`.

Usage

```
nv_mean(x, axes = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
mean(x, trim = 0, na.rm = FALSE, ..., axes = NULL, drop = TRUE)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axes`	(`integer()` | `NULL`) Axes to reduce over. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), reduces over all axes.

drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.
trim	Currently not supported.
na.rm	Forwarded to nv_mean()'s nan_rm argument.
...	No additional arguments.

Value

(arrayish)
Has the input's data type where that is a float, and the default float data type (see default_dtypes()) otherwise. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

See Also

nv_sum()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# an integer input is averaged at the default float data type
nv_mean(x)

# a float input keeps its own, whatever the default float is
nv_mean(nv_array(c(1, 2), dtype = "f64"))

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_mean(x, axes = 1L)
nv_mean(x, axes = 1L, drop = FALSE)

# NaN propagates unless nan_rm = TRUE
nv_mean(nv_array(c(1, NaN, 3)))
nv_mean(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

---

nv_median *Median*

Description

Computes the median over one or more axes. Equivalent to nv_quantile(x, 0.5, axes, drop, method); for an even number of reduced elements with the default "linear" interpolation, the average of the two middle values is returned, matching base R's median().

You can also use median() directly on an AnvlArray or GraphBox; extra arguments (e.g. method) are forwarded via ....

Usage

```
nv_median(x, axes = NULL, drop = TRUE, method = "linear", nan_rm = FALSE)

## S3 method for class 'AnvlArray'
median(x, na.rm = FALSE, ..., axes = NULL, drop = TRUE, method = "linear")
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
method	(character(1)) Forwarded to nv_quantile(). One of "linear" (default), "lower", "higher", "nearest", "midpoint".
nan_rm	(logical(1)) Forwarded to nv_quantile(). See its documentation for details.
na.rm	Forwarded to nv_median()'s nan_rm argument.
...	No additional arguments.

Value

(arrayish)
Same shape as x with axes removed (or set to 1 if drop = FALSE). The data type is that of x, or the default float for a non-float x.

The median() generic

stats::median() reduces every axis of a multi-axis array, and so does nv_median() by default, so the two agree. Pass axes to reduce a subset instead. A non-float x is computed at the default float, like base R returns a double.

See Also

nv_quantile(), nv_sort(), prim_sort().

Examples

```
nv_median(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
median(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_median(m) # over every element
nv_median(m, axes = 2L) # one median per row
```

```
# forwards through the S3 generic via `...`
median(nv_array(c(1, 2, 3, 4)), method = "lower")
nv_median(nv_array(c(1, NaN, 3, 5)))
nv_median(nv_array(c(1, NaN, 3, 5)), nan_rm = TRUE)
```

`nv_min` *Min Reduction*

Description

Finds the minimum of array elements along the specified axes.

Usage

```
nv_min(x, axes = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
min(..., na.rm = FALSE)
```

Arguments

`x`	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
`axes`	(`integer()` | `NULL`) Axes to reduce over. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), reduces over all axes.
`drop`	(`logical(1)`) Whether to drop the reduced axes: removed from the output shape if `TRUE`, set to 1 if `FALSE`.
`nan_rm`	(`logical(1)`) How to handle `NaN` values in float inputs. If `FALSE` (default), `NaN` propagates. If `TRUE`, `NaN` values are skipped.
`...`	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of `nv_min()`, such as `axes`.
`na.rm`	(`logical(1)`) Forwarded to `nv_min()`'s `nan_rm` argument.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (`drop = TRUE`) or set to 1 (`drop = FALSE`).

The `min()` generic

`min()` reduces over all axes and, like `base::min()`, takes several data arguments: `min(x, y)` is the smallest element of both arrays. `na.rm` becomes `nan_rm`. Beyond base R, named arguments are passed on, so `min(x, axes = 1L)` reduces a single axis – but only when `x` is the only data argument.

See Also

`prim_min()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# no axes given: reduce over all of them
nv_min(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_min(x, axes = 1L)
nv_min(x, axes = 1L, drop = FALSE)

# negative axes count from the end
nv_min(x, axes = -1L)

# NaN propagates unless nan_rm = TRUE
nv_min(nv_array(c(1, NaN, 3)))
nv_min(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

---

nv_mod *Modulo (Flooring Remainder)*

Description

Element-wise flooring remainder of division. The sign of the result equals the sign of `rhs`, matching base R's `%%` operator.

Usage

```
nv_mod(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 %% e2
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

e1, e2 (arrayish)
The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

nv_remainder() for truncating remainder, nv_floor_div() for the matching division, prim_remainder() for the underlying primitive.

Examples

```
x <- nv_array(c(1L, -1L))
y <- nv_array(c(-3L, 3L))
nv_mod(x, y)
as.vector(x) %% as.vector(y)

# different data types are promoted to their common one
nv_mod(nv_scalar(1L, "i32"), nv_scalar(-3L, "i64"))

# a scalar is broadcast
x %% 2L
```

---

nv_mul *Multiplication*

Description

Multiplies two arrays element-wise. You can also use the * operator.

Usage

```
nv_mul(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 * e2
```

Arguments

lhs, rhs (arrayish)
Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

e1, e2 (arrayish)
The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_mul() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
nv_mul(x, y)
x * y

# different data types are promoted to their common one
nv_mul(nv_scalar(2, "f32"), nv_scalar(3, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x * 2L
```

---

nv_ne *Not Equal*

Description

Element-wise inequality comparison. You can also use the != operator.

Usage

```
nv_ne(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 != e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and boolean data type.

See Also

prim_ne() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
nv_ne(x, y)
x != y

# different data types are promoted to their common one
nv_ne(nv_scalar(1, "f32"), nv_scalar(2, "f64"))

# a scalar is broadcast and an R integer is converted to a float
x != 2L
```

---

nv_negate *Negation*

Description

Negates an array element-wise. You can also use the unary - operator.

Usage

```
nv_negate(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_negate() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, -2, 3))
-x

# an R value materializes at its default data type
nv_negate(1)
```

---

nv_normal *The Normal Distribution*

Description

Density (nv_dnorm), distribution function (nv_pnorm), quantile function (nv_qnorm), and random generation (nv_rnorm) for the Normal distribution with mean mean and standard deviation sd.

Usage

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

nv_pnorm(q, mean = 0, sd = 1, lower_tail = TRUE, log_p = FALSE)

nv_qnorm(p, mean = 0, sd = 1, lower_tail = TRUE, log_p = FALSE)

nv_rnorm(shape, mean = 0, sd = 1, dtype = NULL)
```

Arguments

x, q	(arrayish) Quantiles at which to evaluate the density (x) or the distribution function (q).
mean	(arrayish) Mean of the distribution. Either a scalar, or an array of exactly the shape of x/q/p (or the sample, for nv_rnorm).
sd	(arrayish) Standard deviation of the distribution, shaped like mean. Must be positive, otherwise results are invalid.
log, log_p	(logical(1)) If TRUE, the densities/probabilities are given as logarithms. For nv_qnorm this describes the input p.
lower_tail	(logical(1)) If TRUE (default), probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
p	(arrayish) Probabilities at which to evaluate the quantile function. Values outside $[0, 1]$ give NaN.

shape (integer())
Shape of the result.

dtype (NULL | character(1) | DataType)
Floating point data type of the sample. The default (NULL) uses the common data type of mean and sd, and the default float type when both are R values.

Details

The Normal distribution has probability density function:

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

where μ is the mean and σ is the standard deviation.

nv_pnorm uses the asymptotic expansion from Abramowitz & Stegun (1964), equation 26.2.12, in the left tail when log_p = TRUE to maintain accuracy.

nv_qnorm uses the same minimax rational approximation as Moshier (1989) (this is ndtri in the Cephes library as used by JAX) for f64, and uses a new lower degree Remez minimax rational approximation on the same intervals for f32.

Value

(arrayish)
nv_dnorm(), nv_pnorm() and nv_qnorm() return an arrayish with the shape of x/q/p and the float data type described under Data Types.

nv_rnorm() returns the sample, an arrayish of shape shape and the data type described under dtype.

Data Types

nv_dnorm(), nv_pnorm() and nv_qnorm() compute at a float data type and return it. If x/q/p is a float, they compute at its data type; an R double settles on the default float. An integer x/q/p (an array or an R integer) is converted to the widest float among the arrays given for mean or sd, or to the default float if there is none. Integers beyond 2^{24} are not all exact at f32, so a large integer may change in that conversion.

An R value for mean or sd takes the computing data type. An array is promoted to it, so a wider one (e.g. f64 for an f32 x) is an error.

Random State

The draws come from the active random state: the global random state (see nv_set_seed()), or the random state that with_nv_rng_state() or local_nv_rng_state() make the default for a scope. They advance it in place.

Random generation

nv_rnorm samples via the Box-Muller transform. To sample with a covariance structure, use a Cholesky decomposition.

`mean` and `sd` are `arrayish`, so they may vary across the sample: they are applied to the draws after they have been reshaped to `shape`, and so may either be scalars or have exactly that shape. As in base R's `rnorm()`, an element whose `mean` is `NaN`, or whose `sd` is not finite or negative, is `NaN`. The random state is advanced regardless.

References

Abramowitz M, Stegun I (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, number 55 series Applied Mathematics Series. Dover Publications, New York. ISBN 0-486-61272-4.

Moshier S (1989). *Methods and Programs for Mathematical Functions*. Ellis Horwood. ISBN 0-7458-0289-3.

See Also

`nv_rnorm()` for sampling from a normal distribution.

Other rng: `local_nv_rng_state()`, `nv_rbinom()`, `nv_rng_state()`, `nv_sample()`, `nv_sample_int()`, `nv_set_seed()`, `nv_uniform`

Examples

```
x <- nv_array(c(-1, 0, 1))
nv_dnorm(x)
nv_dnorm(x, mean = 1, sd = 2)
nv_dnorm(x, log = TRUE)

nv_pnorm(x)
nv_pnorm(x, mean = 1, sd = 2)
nv_pnorm(x, lower_tail = FALSE)
nv_pnorm(x, log_p = TRUE)

p <- nv_array(c(0.025, 0.5, 0.975))
nv_qnorm(p)
nv_qnorm(p, mean = 1, sd = 2)
nv_qnorm(p, lower_tail = FALSE)
nv_qnorm(nv_array(c(-700, -2, -0.1), dtype = "f64"), log_p = TRUE)


with_nv_seed(42L, nv_rnorm(c(2, 3)))

# `sd` may also be an array of the same shape as the sample
sds <- nv_array(matrix(c(0.01, 0.1, 1, 10, 100, 1000), nrow = 2))
nv_rnorm(c(2, 3), sd = sds)
```

nv_not *Bitwise Not*

Description

Element-wise bitwise NOT – a logical negation on a boolean input, and a bit-by-bit complement on an integer, so `nv_not(12L)` is `-13`. You can also use the `!` operator, which expects `bool` inputs.

Usage

```
nv_not(x)

## S3 method for class 'AnvlArray'
!x
```

Arguments

`x`	(`arrayish`) One input. Can be any integerish data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

The `!` operator

`!` is *logical*, like in base R. Unlike base R it only accepts booleans and does not auto-convert non-booleans by comparing them with 0.

See Also

`prim_not()` for the underlying primitive.

Examples

```
# on a boolean this is a logical negation
x <- nv_array(c(TRUE, FALSE, TRUE))
!x

# on an integer it complements every bit, so `12L` becomes `-13`
nv_not(nv_array(12L))
```

nv_or *Bitwise OR*

Description

Element-wise bitwise OR – a logical OR on a boolean input, and a bit-by-bit one on an integer. You can also use the | operator, which expects bool inputs, however.

Usage

```
nv_or(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 | e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any integerish data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

The | operator

| is *logical*, like in base R. Unlike base R it only accepts booleans and does not auto-convert non-booleans by comparing them with 0.

See Also

prim_or() for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
x | y

# different data types are promoted to their common one
nv_or(nv_scalar(12L, "i32"), nv_scalar(10L, "i64"))
```

```
# a scalar is broadcast
x | TRUE
```

---

nv_order *Sort Order*

Description

Returns the indices that would sort the array, like `base::order()`: over every element by default, or along one axis. It is the index twin of `nv_sort()` and takes the same `axis`, so the two always describe the same ordering.

Usage

```
nv_order(x, axis = NULL, decreasing = FALSE, stable = FALSE)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axis`	(`integer(1)` | `NULL`) Axis along which to compute the sort permutation. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), the input is first flattened to a 1-D array, like `nv_sort()`, and the indices refer to that flattening.
`decreasing`	(`logical(1)`) If `TRUE`, returns indices that produce a decreasing sort. Default `FALSE`.
`stable`	(`logical(1)`) If `TRUE`, the sort is stable: indices for equal values keep their original relative order. Default `FALSE`.

Value

(`arrayish`)
Has the default integer data type (see `default_dtypes()`) regardless of the input's, and the input's shape – or 1-D holding every element's index when `axis` is `NULL`. For a size-0 axis, the output is an empty array of the same shape (a valid empty permutation). Indexing the flattened input by the result reproduces `nv_sort()`'s output.

NaN handling

`NaN` values sort to the **end** (ascending) or **beginning** (descending), regardless of sign. `+0` and `-0` compare equal.

See Also

`nv_sort()`, `prim_sort()`.

Examples

```
# the indices come out at the default integer data type
x <- nv_array(c(3, 1, 4, 1, 5))
nv_order(x)

m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_order(m) # indexes the flattened matrix
nv_order(m, axis = 2L) # a permutation per row
```

---

nv_outer *Outer Product*

Description

Computes the outer product of two 1-D arrays.

Usage

```
nv_outer(x, y)
```

Arguments

x, y	(arrayish) Two 1-D arrays. Can be of any data type; the two are promoted to a common data type.

Value

(arrayish)
Has the operands' common data type and shape `(length(x), length(y))`.

Examples

```
# a length-3 and a length-2 vector give a 3x2 at their common data type
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5))
nv_outer(x, y)
```

nv_pad *Pad*

Description

Pads an array with a given value at the edges and optionally between elements.

Usage

```
nv_pad(x, value, low, high, interior = NULL)
```

Arguments

x	(arrayish) The array to pad. Can be any data type; value is brought to it.
value	(arrayish) Scalar value to use for padding. It is brought to x's data type: an R value is built at it when its category can reach it (0L serves an integer and a float x alike, 0 only a float one), and a value that already has a data type is converted unless that would narrow it – an f64 padding value for an f32 x is an error rather than a silent narrowing.
low	(integer()) Amount of padding to add at the start of each axis.
high	(integer()) Amount of padding to add at the end of each axis.
interior	(integer() | NULL) Amount of padding to add between elements in each axis. If NULL (default), no interior padding is applied.

Value

(arrayish)
Has x's data type. Each axis grows by low + high, plus interior between every pair of elements; negative edge padding trims.

See Also

prim_pad() for the underlying primitive.

Examples

```
# two zeros in front, one behind
x <- nv_array(c(1, 2, 3))
nv_pad(x, nv_scalar(0), low = 2L, high = 1L)
```

`nv_plogis` *Logistic (Sigmoid)*

Description

Element-wise logistic sigmoid: `1 / (1 + exp(-x))`. This is like `stats::plogis()` with the default location and scale.

Usage

```
nv_plogis(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_plogis()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-2, 0, 2))
nv_plogis(x)

# an R value materializes at its default data type
nv_plogis(2)
```

nv_pmax *Parallel Maximum*

Description

Element-wise maximum of two arrays, like base::pmax().

Usage

```
nv_pmax(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_pmax() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
nv_pmax(x, y)

# different data types are promoted to their common one
nv_pmax(nv_scalar(1, "f32"), nv_scalar(5, "f64"))

# a scalar is broadcast and an R integer is converted to a float
nv_pmax(x, 2L)
```

nv_pmin *Parallel Minimum*

Description

Element-wise minimum of two arrays, like base::pmin().

Usage

```
nv_pmin(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_pmin() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
nv_pmin(x, y)

# different data types are promoted to their common one
nv_pmin(nv_scalar(1, "f32"), nv_scalar(5, "f64"))

# a scalar is broadcast and an R integer is converted to a float
nv_pmin(x, 2L)
```

nv_poisson *The Poisson Distribution*

Description

Density (nv_dpois), distribution function (nv_ppois), and quantile function (nv_qpois) for the Poisson distribution with mean lambda.

Usage

```
nv_dpois(x, lambda, log = FALSE)

nv_ppois(q, lambda, lower_tail = TRUE, log_p = FALSE)

nv_qpois(p, lambda, lower_tail = TRUE, log_p = FALSE)
```

Arguments

x, q	(arrayish) Quantiles at which to evaluate the density (x) or the distribution function (q). These are counts, given as integers or floats; see Non-Integer Values for values that are not whole numbers.
lambda	(arrayish) Mean of the distribution. Scalars broadcast; otherwise lambda and x/q/p must have the same shape.
log, log_p	(logical(1)) If TRUE, the densities/probabilities are given as logarithms. For nv_qpois this describes the input p.
lower_tail	(logical(1)) If TRUE (default), probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
p	(arrayish) Probabilities at which to evaluate the quantile function. Values outside $[0, 1]$ give NaN. With log_p = TRUE, supply log-probabilities in $[-\infty, 0]$ instead.

Details

The Poisson distribution has probability mass function:

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad x = 0, 1, 2, \ldots$$

where λ is lambda.

Invalid arguments give NaN, as in base R, but without a warning (that is a negative lambda). Zero and infinite lambda follow base R's endpoint conventions:

- lambda = 0: the point mass at zero.

- `lambda = Inf`: densities are zero. The lower-tail probability is zero at finite `q` and one at `q = Inf`. Quantiles are `NaN`.

`nv_dpois` uses the saddle point expansion of Loader (2000), as base R does. `nv_ppois` uses the incomplete gamma function methods of base R's `stats::pgamma()`: a series in the tails, and an asymptotic expansion where `q` and `lambda` are large and close. `nv_qpois` finds the smallest x with $P(X \leq x) \geq p$, starting from a Cornish-Fisher approximation as base R does. Above `2^24` in `f32` and `2^53` in `f64` not every whole number is a float, and `nv_qpois` then returns the smallest float x with $P(X \leq x) \geq p$.

Value

(`arrayish`)
`nv_dpois()`, `nv_ppois()`, and `nv_qpois()` return an `arrayish` with the float data type described under Data Types and the shape after scalar broadcasting with `lambda`.

Non-Integer Values

`x` and `q` are counts, but any value is accepted, and one that is not a whole number is treated as in base R, without its warnings:

- `nv_dpois()` takes an `x` within a relative `1e-9` of a whole number, that is with $|x - \mathrm{round}(x)| \leq 10^{-9} \max(|x|, 1)$, as that whole number. Any other `x` has probability zero.
- `nv_ppois()` rounds `q` down, except that a `q` within `1e-7` below a whole number counts as that number.
- A negative `x` or `q` is below the support even where it is within the tolerance of zero: such an `x` has probability zero, and such a `q` has $P(X \leq q) = 0$.

Finite quantiles from `nv_qpois()` are always whole numbers. Since `x` and `q` are used only as whole numbers, their gradient is zero wherever a gradient can be taken.

Gradients

`nv_dpois` and `nv_ppois` can be differentiated with respect to `lambda`, to any order. At `lambda = 0`, where only a right derivative exists, the implementation returns it for `nv_dpois(0, lambda)` and, by convention, zero for the other probabilities at `lambda = 0`. `nv_qpois` is a step function, and autodiff returns a zero gradient for all of its arguments.

Data Types

`nv_dpois()`, `nv_ppois()` and `nv_qpois()` compute at a float data type and return it. If `x`/`q`/`p` is a float, they compute at its data type; an R double settles on the default float. An integer `x`/`q`/`p` (an array or an R integer) is converted to the widest float among the arrays given for `lambda`, or to the default float if there is none. Integers beyond 2^{24} are not all exact at `f32`, so a large integer may change in that conversion.

An R value for `lambda` takes the computing data type. An array is promoted to it, so a wider one (e.g. `f64` for an `f32` `x`) is an error.

References

Loader C (2000). "Fast and Accurate Computation of Binomial Probabilities." https://www.r-project.org/doc/reports/CLoader-dbinom-2002.pdf.

Examples

```
x <- nv_array(c(0, 1, 2, 5, 10))
nv_dpois(x, lambda = 4)
nv_dpois(x, lambda = 4, log = TRUE)

nv_ppois(x, lambda = 4)
nv_ppois(x, lambda = 4, lower_tail = FALSE)
nv_ppois(x, lambda = 4, log_p = TRUE)

p <- nv_array(c(0.025, 0.5, 0.975))
nv_qpois(p, lambda = 4)
nv_qpois(p, lambda = 4, lower_tail = FALSE)
nv_qpois(
  nv_array(c(-700, -2, -0.1), dtype = "f64"),
  lambda = 1000,
  log_p = TRUE
)
```

---

nv_popcnt *Population Count*

Description

Element-wise population count (number of set bits).

Usage

```
nv_popcnt(x)
```

Arguments

x	(arrayish) One input. Can be any integer data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_popcnt() for the underlying primitive.

Examples

```
# the set bits are counted, at the input's own integer data type
x <- nv_array(c(7L, 3L, 15L))
nv_popcnt(x)
```

---

nv_pow *Power*

Description

Raises x to the power of y element-wise. You can also use the ^ operator.

Usage

```
nv_pow(x, y)

## S3 method for class 'AnvlArray'
e1 ^ e2
```

Arguments

x, y	(arrayish) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_pow() for the underlying primitive.

Examples

```
x <- nv_array(c(2, 3, 4))
y <- nv_array(c(3, 2, 1))
x ^ y

# different data types are promoted to their common one
nv_pow(nv_scalar(2, "f32"), nv_scalar(3, "f64"))
```

```
# a scalar is broadcast and an R integer is converted to a float
x^2L
```

`nv_print` *Print Array*

Description

Prints an array value to the console during JIT execution and returns the input unchanged. Useful for debugging. For `RData` inputs, that do not have an actual data type, the default data type is used for printing.

Usage

```
nv_print(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

Value

(`arrayish`)
Returns the input unchanged, data type and shape included.

Order of prints

A print is currently not ordered with respect to other prints. Due to compiler reordering it is possible that the order of the prints differs from their location in the code.

See Also

`prim_print()` for the underlying primitive.

Examples

```
# the value is printed and handed back unchanged
x <- nv_array(c(1, 2, 3))
nv_print(x)
# RData is printed at the default dtype
nv_print(1)
```

nv_prod *Product Reduction*

Description

Multiplies array elements along the specified axes. A boolean array is multiplied as zeroes and ones, like base::prod() does.

Usage

```
nv_prod(x, axes = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
prod(..., na.rm = FALSE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.
...	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of nv_prod(), such as axes.
na.rm	(logical(1)) Forwarded to nv_prod()'s nan_rm argument.

Value

(arrayish)
Has the input's data type, except a boolean input, which is accumulated at the default integer data type (see default_dtypes()). The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

The prod() generic

prod() reduces over all axes and, like base::prod(), takes several data arguments: prod(x, y) is the product of both arrays. na.rm becomes nan_rm. Beyond base R, named arguments are passed on, so prod(x, axes = 1L) reduces a single axis – but only when x is the only data argument.

See Also

prim_prod() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# no axes given: reduce over all of them
nv_prod(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_prod(x, axes = 1L)
nv_prod(x, axes = 1L, drop = FALSE)

# negative axes count from the end
nv_prod(x, axes = -1L)

# NaN propagates unless nan_rm = TRUE
nv_prod(nv_array(c(2, NaN, 3)))
nv_prod(nv_array(c(2, NaN, 3)), nan_rm = TRUE)
```

---

nv_promote_to_common *Promote Arrays to a Common Data Type*

Description

Promote arrays to a common data type, see promotion_common() for more details.

Usage

```
nv_promote_to_common(...)
```

Arguments

...	(arrayish) Values to promote.

Value

(list() of arrayish)
The inputs, each at their common data type and with its own shape.

Examples

```
# An integer is promoted to float
x <- nv_scalar(1, dtype = "f32")
y <- nv_scalar(1L, dtype = "i32")
nv_promote_to_common(x, y)
# an R value yields to the data type it meets within its category
```

```
with_default_dtypes(c(float = "f64"), nv_promote_to_common(1, x))
# and settles on its default otherwise
with_default_dtypes(c(float = "f64"), nv_promote_to_common(1, y))
```

`nv_psigamma` *Psigamma*

Description

Element-wise psigamma function, like `base::psigamma()`: the `deriv`-th derivative of the digamma function, i.e. the `(deriv + 1)`-th derivative of the log-gamma function. For `deriv = 0` this is the digamma function; for `deriv = 1`, `trigamma()` dispatches here.

Usage

```
nv_psigamma(x, deriv = 0L)

## S3 method for class 'AnvlArray'
trigamma(x)
```

Arguments

`x`, `deriv`	(`arrayish`) The value to evaluate at and the order of the derivative. `deriv` typically holds non-negative whole numbers. Can be any numeric data type: the two are promoted to a common data type and that is then converted to the default float data type (see `default_dtypes()`) where it is not a float already, since a float is all `prim_psigamma()` takes. An R value assumes the other operand's data type within its data type category, and settles on the default float when neither has one. Scalars are broadcast to the shape of the other, so `nv_psigamma(x, 1)` works for any float `x`.

Value

(`arrayish`)
Has the inputs' broadcast shape, and their common data type – or the default float data type (see `default_dtypes()`) where that was an integer one.

The `trigamma()` generic

`trigamma(x)` is `nv_psigamma(x, 1L)`.

See Also

`prim_psigamma()` for the underlying primitive.

Examples

```
# the R `1` is built at `x`'s float data type and broadcast
x <- nv_array(c(0.5, 1, 2, 5))
nv_psigamma(x) # digamma
nv_psigamma(x, 1) # trigamma
```

---

nv_qr *QR Decomposition*

Description

Computes the reduced QR decomposition of a matrix x:

$$A = QR,$$

where Q has orthonormal columns ($Q^\top Q = I$) and R is upper triangular. For an $m \times n$ input with $k = \min(m, n)$, Q has shape $m \times k$ and R has shape $k \times n$.

Usage

```
nv_qr(x)

## S3 method for class 'AnvlArray'
qr(x, ...)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.
...	No additional arguments.

Value

(named list of two arrayish)
Elements Q (shape (m, k)) and R (shape (k, n)), where (m, n) = shape(x) and k = min(m, n). Both have the input's data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_qr()

Examples

```
# `Q` is 3x2 and `R` 2x2, both at the input's data type
x <- nv_matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, dtype = "f32")
nv_qr(x)

# an integer matrix is decomposed at the default float data type
nv_qr(nv_matrix(1:6, nrow = 3))
```

`nv_quantile` *Quantile*

Description

Computes the `probs` quantile(s) of an array over one or more axes.

`probs` follows the same scalar-vs-array convention as `nv_select()`'s `index`:

- a length-1 numeric (e.g. `0.5`) treats `probs` as scalar — the result is the reduction alone;
- a 1-D R array (e.g. `array(c(0.25, 0.5, 0.75))`) prepends a leading axis of size `length(probs)`.

Plain length-K (K > 1) vectors are rejected; wrap with `array()` to make the array intent explicit.

A quantile generally falls between two elements, so a non-float `x` is computed (and returned) at the default float data type.

Reducing over an axis of size 0 is an error: there is no element to take a quantile of.

Usage

```
nv_quantile(
  x,
  probs,
  axes = NULL,
  drop = TRUE,
  method = "linear",
  nan_rm = FALSE
)

## S3 method for class 'AnvlArray'
quantile(
  x,
  probs = array(seq(0, 1, 0.25)),
  na.rm = FALSE,
  ...,
  axes = NULL,
  drop = TRUE,
  method = "linear"
)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
probs	(numeric(1) | 1-D array) One or more probabilities in [0, 1]. Either a length-1 numeric (scalar) or a 1-D array (a leading axis of size length(probs) is prepended). Plain length-K (K > 1) vectors are rejected — wrap with array().
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
method	(character(1)) One of "linear" (default), "lower", "higher", "nearest", "midpoint". See "Interpolation modes".
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.
na.rm	Forwarded to nv_quantile()'s nan_rm argument.
...	No additional arguments.

Value

(arrayish)
Same shape as x with axes removed (or set to 1 if drop = FALSE). For array probs, a **leading** axis of size length(probs) is prepended. The data type is that of x, or the default float for a non-float x.

Interpolation modes

For n reduced elements and a probability q, let h = 1 + (n - 1) * q be the position q falls at in the sorted values, with lo = floor(h), hi = ceiling(h) and frac = h - lo. Then, writing sorted for the reduced values in sorted order:

- "linear" (default): (1 - frac) * sorted[lo] + frac * sorted[hi].
- "lower": sorted[lo] — the lower bracket of linear.
- "higher": sorted[hi] — the upper bracket of linear.
- "nearest": sorted[lo] if frac < 0.5 else sorted[hi].
- "midpoint": (sorted[lo] + sorted[hi]) / 2.

Reducing several axes at once ranks all of their elements together, so nv_quantile(x, q, axes = c(1, 2)) equals nv_quantile(nv_flatten(x), q) for a matrix x.

The `quantile()` generic

`stats::quantile()` reduces every axis of a multi-axis array, and so does `nv_quantile()` by default, so the two agree. Pass `axes` to reduce a subset instead. The result is unnamed, and `probs` follows `nv_quantile()`'s rules: several probabilities have to be wrapped in `array()`, which the default already is.

See Also

`nv_median()`, `nv_sort()`.

Examples

```
# a float result even for an integer input, since it interpolates
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_quantile(x, 0.5) # = nv_median(x)
nv_quantile(x, array(c(0.25, 0.5, 0.75)))
nv_quantile(x, 0.5, method = "lower")
m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_quantile(m, 0.5) # over every element
nv_quantile(m, 0.5, axes = 2L) # one quantile per row
nv_quantile(nv_array(c(1, NaN, 3, 5)), 0.5)
nv_quantile(nv_array(c(1, NaN, 3, 5)), 0.5, nan_rm = TRUE)
```

`nv_range` *Range Reduction*

Description

The smallest and the largest element along the specified axes, stacked along a new first axis. You can also use the `range()` generic.

Usage

```
nv_range(x, axes = NULL, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
range(..., na.rm = FALSE)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axes`	(`integer()` | `NULL`) Axes to reduce over. `NULL` (default) reduces over all of them, which makes the result a length-2 array like `base::range()`. Negative values count from the end.

`nan_rm` (`logical(1)`)
How to handle `NaN` values in float inputs. If `FALSE` (default), `NaN` propagates. If `TRUE`, `NaN` values are skipped.

`...` (`arrayish`)
Arrays to reduce, plus named arguments for `nv_min()` and `nv_max()` (e.g. `axes`), which are only accepted when there is a single array to reduce.

`na.rm` (`logical(1)`)
Forwarded to the `nan_rm` argument of `nv_min()` and `nv_max()`.

Value

(`arrayish`)
Has the same data type as `x` and the shape of the reduced array with a leading axis of size 2 added: element 1 is the minimum, element 2 the maximum.

The `range()` generic

`range()` reduces over all axes and, like `base::range()`, takes several data arguments: `range(x, y)` is the range of both arrays. Beyond base R, named arguments are passed on, so `range(x, axes = 1L)` reduces a single axis – but only when `x` is the only data argument.

See Also

`nv_min()`, `nv_max()`

Examples

```
nv_range(nv_array(c(3, 1, 4)))
nv_range(nv_matrix(1:6, nrow = 2), axes = 1L)
```

---

nv_rbinom *Sample from a Binomial Distribution*

Description

Samples from a binomial distribution with n trials and success probability p. When `size = 1` (the default), this is a Bernoulli distribution.

Usage

```
nv_rbinom(shape, size = 1L, prob = 0.5, dtype = NULL)
```

Arguments

shape	(integer()) Shape of the result.
size	(integer(1)) Number of trials. 0 gives a sample of zeros, like rbinom().
prob	(numeric(1)) Probability of success on each trial.
dtype	(NULL | character(1) | DataType) Numeric type of the sample. NULL (default) uses the default integer type. The number of successes are converted to it, so it must hold size.

Value

(arrayish)
The sample, of shape shape and data type dtype.

Random State

The draws come from the active random state: the global random state (see nv_set_seed()), or the random state that with_nv_rng_state() or local_nv_rng_state() make the default for a scope. They advance it in place.

See Also

Other rng: local_nv_rng_state(), nv_normal, nv_rng_state(), nv_sample(), nv_sample_int(), nv_set_seed(), nv_uniform

Examples

```
# Bernoulli samples
with_nv_seed(42L, nv_rbinom(c(2, 3)))
```

---

nv_remainder *Remainder (Truncating)*

Description

Element-wise remainder. This differs from base R's %%, use nv_mod()/%% instead.

Usage

```
nv_remainder(x, y)
```

Arguments

x, y	(arrayish) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

nv_mod() for the flooring remainder, prim_remainder() for the underlying primitive.

Examples

```
x <- nv_array(c(7, 8, 9))
y <- nv_array(c(3, 3, 4))
nv_remainder(x, y)

# different data types are promoted to their common one
nv_remainder(nv_scalar(7, "f32"), nv_scalar(3, "f64"))

# a scalar is broadcast and an R integer is converted to a float
nv_remainder(x, 3L)
```

---

nv_reshape *Reshape*

Description

Reshapes an array to a new shape using col-major semantics.

Usage

```
nv_reshape(x, shape)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
shape	(integer()) Target shape. Must have the same number of elements as x. At most one entry may be -1, in which case its extent is inferred from the remaining entries and the number of elements of x.

Value

(arrayish)
Has the given shape and the same data type as x.

See Also

prim_reshape() for the underlying primitive.

Examples

```
# the elements keep their column-major order; the data type is untouched
x <- array(1:6, dim = c(3, 2))
nv_reshape(x, 6L)
# the order base R reads them in, too
c(x)

# infer the size of the second axis
nv_reshape(x, c(2, -1))
# flatten
nv_reshape(x, -1)
```

---

nv_rev *Reverse*

Description

Reverses the order of elements along the given axes, every axis by default. You can also use rev(), which always reverses along every axis.

Usage

```
nv_rev(x, axes = NULL)

## S3 method for class 'AnvlArray'
rev(x)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reverse. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reverses along every axis.

Value

(arrayish)
Has the same shape and data type as x.

The rev() generic

rev() reverses along every axis, which puts the elements in the same order as base::rev() does (base R flattens the array to a vector first, whereas rev() on an anvl array keeps the shape).

See Also

prim_rev() for the underlying primitive.

Examples

```
# the order along axis 1 is flipped
x <- nv_array(c(1, 2, 3, 4, 5))
nv_rev(x)

m <- nv_matrix(1:6, nrow = 2)
nv_rev(m) # every axis
nv_rev(m, axes = 2L) # columns only
```

---

nv_rng_state *Create a Random State*

Description

Creates a random state of its own, which the samplers draw from in place of the global random state within with_nv_rng_state() or after local_nv_rng_state(). Draws advance it in place. A function compiled with jit() can draw from a random state it takes as an argument or closes over, but cannot create or return one.

See the Random Number Generation article for more details.

Usage

```
nv_rng_state(seed)
```

Arguments

seed	(integer(1)) The seed, a 32-bit integer, which gives the same random state as nv_set_seed() gives the global random state.

Value

(AnvlRngState)
A random state of the key that seed gives and the counter 0.

See Also

Other rng: local_nv_rng_state(), nv_normal, nv_rbinom(), nv_sample(), nv_sample_int(), nv_set_seed(), nv_uniform

Examples

```
rng <- nv_rng_state(42L)
with_nv_rng_state(rng, nv_runif(3L))
# the random state was advanced, so the next draw differs
with_nv_rng_state(rng, nv_runif(3L))
rng

# a jitted function takes a random state as an argument
f <- jit(function(x, rng) with_nv_rng_state(rng, x + nv_rnorm(2L)))
f(nv_array(c(1, 2)), nv_rng_state(1L))
```

---

nv_round *Round*

Description

Element-wise rounding to a whole number.

Usage

```
nv_round(x, method = "nearest_even")
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float is rounded and keeps its own, and an integer one is already whole and is returned unchanged. An R value materializes at its default data type and is treated in the same way.
method	(character(1)) Rounding method. Either "nearest_even" (default) or "afz" (away from zero).

Value

(arrayish)
Has the input's shape and data type.

See Also

prim_round() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1.4, 2.5, 3.6))
nv_round(x)
nv_round(nv_array(1:3)) # an integer array is already whole
```

`nv_rsqrt` *Reciprocal Square Root*

Description

Element-wise reciprocal square root, i.e. `1 / sqrt(x)`.

Usage

```
nv_rsqrt(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_rsqrt()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 4, 9))
nv_rsqrt(x)

# an R value materializes at its default data type
nv_rsqrt(4)
```

nv_sample *Sample from a Population*

Description

Samples elements of a 1-D array with equal probability and with replacement, analogous to R's `sample()`.

Unlike R's `sample()`, `x` is always the population itself: sampling the integers 1 to `n` is `nv_sample_int()` and never an overload of `x`.

Usage

```
nv_sample(shape, x)
```

Arguments

shape	(integer()) Shape of the result.
x	(arrayish) The population vector to sample from. Must not be empty. An R value materializes at its default data type.

Value

(arrayish)
The sample, of shape `shape` and `x`'s data type.

Random State

The draws come from the active random state: the global random state (see `nv_set_seed()`), or the random state that `with_nv_rng_state()` or `local_nv_rng_state()` make the default for a scope. They advance it in place.

See Also

`nv_sample_int()` to sample the integers 1 to `n`.

Other rng: `local_nv_rng_state()`, `nv_normal`, `nv_rbinom()`, `nv_rng_state()`, `nv_sample_int()`, `nv_set_seed()`, `nv_uniform`

Examples

```
# the sample takes the population's data type
with_nv_seed(42L, nv_sample(5, nv_array(c(10, 20, 30))))
```

nv_sample_int *Sample Integers*

Description

Samples integers from 1 to n with equal probability and with replacement, analogous to R's `sample.int()`.

To sample from a population other than `1:n`, use `nv_sample()`.

Usage

```
nv_sample_int(shape, n, dtype = NULL)
```

Arguments

shape	(integer()) Shape of the result.
n	(integer(1)) Size of the population, i.e. the integers 1 to n are sampled.
dtype	(NULL | character(1) | DataType) Numeric type of the sampled integers. The sampled values are converted to it, so it must hold n. NULL (default) uses the default integer type.

Value

(arrayish)
The sampled integers, of shape shape and data type dtype.

Random State

The draws come from the active random state: the global random state (see `nv_set_seed()`), or the random state that `with_nv_rng_state()` or `local_nv_rng_state()` make the default for a scope. They advance it in place.

See Also

`nv_sample()` to sample from an arbitrary population.

Other rng: `local_nv_rng_state()`, `nv_normal`, `nv_rbinom()`, `nv_rng_state()`, `nv_sample()`, `nv_set_seed()`, `nv_uniform`

Examples

```
# roll six dice
with_nv_seed(42L, nv_sample_int(6, 6L))
```

nv_save *Save and Read Arrays in a File*

Description

nv_save() saves a named list of arrays to a file in the safetensors format, and nv_read() loads them back. The data type and shape of each array are restored.

Usage

```
nv_save(arrays, path)

nv_read(path, device = NULL)
```

Arguments

arrays	(named list of AnvlArray) Named list of arrays. Names must be unique.
path	(character(1)) File path to write to or read from.
device	(NULL | character(1) | PJRTDevice) The device on which to place the loaded arrays ("cpu", "cuda", ...) when the active backend is "pjrt", defaulting to default_device().

Details

These are convenience wrappers around nv_serialize() and nv_unserialize() that open and close a file connection.

Value

nv_save(): (NULL)
Invisibly.

nv_read(): (named list of AnvlArray)

See Also

nv_serialize(), nv_unserialize()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
path <- tempfile(fileext = ".safetensors")
nv_save(list(x = x), path)
nv_read(path)
```

nv_scan *Scan (Loop with Per-Step Outputs)*

Description

Runs a fixed-length loop that threads a carry through body while stacking each step's output into preallocated buffers.

At step t, body receives the current carry and the step's slice of xs (taken along axis 1, with that unit axis dropped; a 1-D leaf yields a scalar), and must return list(carry = <same structure as init>, out = <arrays to st
The stacked out buffers gain a new leading axis of size steps.

The whole loop, written out in R:

```
carry <- init
out <- <empty, `steps` rows>
order <- if (reverse) rev(seq_len(steps)) else seq_len(steps)
for (t in order) {
  step <- body(carry, xs[t, ...])  # `x` is NULL when `xs` is empty
  carry <- step$carry
  out[t, ...] <- step$out          # position t, not the loop's position
}
list(carry = carry, out = out)
```

Usage

```
nv_scan(init, xs = NULL, body, steps = NULL, reverse = FALSE)
```

Arguments

init	(arrayish | list()) Initial carry: a single array or a (possibly nested) named list. Every slot must keep a fixed shape and dtype across steps
xs	(arrayish | list() | NULL) Per-step inputs, sliced along axis 1. All leaves must agree on the size of axis 1. NULL or a list with no leaves runs a counted loop over steps steps instead.
body	(function) Step function function(carry, x) returning list(carry = , out =). out may be a single array, a (nested) list of arrays, or NULL (loop for the carry only). Its structure must be identical at every step. x is NULL when xs is empty.
steps	(integer(1) | NULL) Static trip count. Required when xs is empty; otherwise inferred from (and checked against) axis 1 of xs. A trip count of 0 runs no step.
reverse	(logical(1)) If TRUE, steps run t = steps, ..., 1; each step still reads xs at position t and writes its output at position t, so a reverse scan consumes and produces arrays in the original order.

Value

`list(carry = , out = )`: the final carry (same structure as `init`) and the stacked outputs (structure of body's `out`, each leaf gaining a leading axis of size `steps`).

Gradients

Unlike `nv_while()`, a scan is differentiable with `gradient()`, with respect to `init`, `xs` and the arrays body closes over. The forward pass keeps what the backward pass needs of each step, so the memory a gradient takes grows linearly with `steps`.

See Also

`prim_scan()`, `nv_while()`, `nv_cumsum()` for fixed associative scans.

Examples

```
# cumulative sum along axis 1
x <- nv_array(c(1, 2, 3, 4))
nv_scan(
  init = nv_scalar(0),
  xs = x,
  body = function(carry, x) list(carry = carry + x, out = carry + x)
)$out

# a scan is differentiable: the gradient of a cumulative product
f <- function(x) {
  nv_scan(nv_scalar(1), x, function(carry, x) list(carry = carry * x, out = NULL))$carry
}
jit(gradient(f))(nv_array(c(1, 2, 3)))
```

---

nv_sd *Standard Deviation*

Description

Computes the standard deviation along the specified axes.

Usage

```
nv_sd(x, axes = NULL, drop = TRUE, correction = 1L, nan_rm = FALSE)
```

Arguments

x	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
correction	(integer(1)) Degrees of freedom correction. Default is 1 (Bessel's correction).
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Details

Uses Bessel's correction by default (correction = 1), matching R's sd(). Set correction = 0 for population standard deviation.

Value

(arrayish)
Has the input's data type where that is a float, and the default float data type (see default_dtypes()) otherwise. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

See Also

nv_var(), nv_mean()

Examples

```
x <- nv_array(1:5)
# the result is a float, even though the input is an integer
nv_sd(x)

# Bessel's correction by default, correction = 0 for the population value
nv_sd(x, correction = 0L)
```

---

nv_select *Select Elements Along an Axis*

Description

Picks one or more elements along axis axis of x. Use this instead of [or nv_subset when the index to select is provided programmatically.

Usage

```
nv_select(x, axis, index)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis to index into. Negative values count from the end, i.e. -1 refers to the last axis.
index	(arrayish) Scalar or 1-D array of an integer data type, which it keeps – the index takes no part in x's data type.

Value

(arrayish)
Has x's data type. axis is dropped if index was scalar, and otherwise resized to the number of selected elements.

See Also

nv_subset() for general subsetting, prim_static_slice().

Examples

```
# a scalar index drops the axis, an index array keeps it
m <- nv_matrix(1:6, nrow = 2)
nv_select(m, axis = 2L, index = 2L)
nv_select(m, axis = 1L, index = 1L)
nv_select(m, axis = 2L, index = array(c(1L, 3L)))
```

---

nv_seq *Sequence*

Description

Creates a 1-D array with the values from from to to in steps of by, like R's seq(). The sequence counts down when to lies below from, and stops before to when to is not reachable in whole steps: nv_seq(0, 9, by = 2) ends at 8.

nv_seq_like() is a variant where dtype and device default to those of like.

Usage

```
nv_seq(from, to, by = NULL, dtype = NULL, device = NULL)

nv_seq_like(like, from, to, by = NULL, dtype = NULL, device = NULL)
```

Arguments

from, to	(integer(1)) First value and upper (or, when counting down, lower) limit of the sequence.
by	(NULL | integer(1)) Step size, which must be a non-zero whole number pointing from from towards to. NULL (default) uses -1 if from > to and 1 otherwise.
dtype	(NULL | character(1) | DataType) Data type of the result. Can be any numeric data type. NULL (default) uses the default integer data type (see default_dtypes()), since the values are whole. For nv_seq_like(), NULL uses dtype(like).
device	(NULL | character(1) | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or • a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error. The default (NULL) uses default_device().
like	(AnvlArray) Existing array whose attributes are used as defaults (only for nv_seq_like()).

Value

(arrayish)
Has dtype and shape (to - from) %/% by + 1.

See Also

nv_linspace() for a given number of evenly spaced values, nv_iota() for values increasing along an axis of any shape, prim_iota() for the underlying primitive.

Examples

```
nv_seq(3, 7)

# a range that counts down needs no `by`
nv_seq(7, 3)

# `to` is only reached where a whole number of steps lands on it
nv_seq(0, 9, by = 2)
```

```
# a float data type gives the same values as floats
nv_seq(3, 7, dtype = "f32")

# nv_seq_like() takes the data type and device from an existing array
x <- nv_array(c(1, 2, 3), dtype = "f64")
nv_seq_like(x, 1, 5)
```

`nv_serialize` *Serialize Arrays to Raw Bytes*

Description

`nv_serialize()` serializes a named list of arrays into the safetensors format, and `nv_unserialize()` deserializes them. The data type and shape of each array are restored.

Usage

```
nv_serialize(arrays, con = NULL)

nv_unserialize(con, device = NULL)
```

Arguments

`arrays`	(named `list` of `AnvlArray`) Named list of arrays to serialize. Names must be unique.
`con`	(`NULL` | connection | `raw`) For `nv_serialize()`, an optional connection to write to; if `NULL` (default), a raw vector is returned. For `nv_unserialize()`, a connection or raw vector to read from.
`device`	(`NULL` | `character(1)` | `PJRTDevice`) The device on which to place the loaded arrays (`"cpu"`, `"cuda"`, ...) when the active backend is `"pjrt"`, defaulting to `default_device()`.

Value

`nv_serialize()`: (`raw` | `NULL`)
A raw vector if con is `NULL`, otherwise `NULL` invisibly.

`nv_unserialize()`: (named `list` of `AnvlArray`)

See Also

`nv_save()`, `nv_read()`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
raw_data <- nv_serialize(list(x = x))
raw_data
nv_unserialize(raw_data)
```

---

nv_set_seed *Set the Global Random State*

Description

`nv_set_seed()` seeds the global random state, like base R's `set.seed()`. Until it is set, its first use derives it from base R's RNG. `nv_get_rng_state()` and `nv_set_rng_state()` save and restore it, and `local_nv_seed()` and `with_nv_seed()` seed it for a scope.

See the Random Number Generation article for more details.

Usage

```
nv_set_seed(seed)

nv_get_rng_state()

nv_set_rng_state(rng_state)

local_nv_seed(seed, envir = parent.frame())

with_nv_seed(seed, code)
```

Arguments

`seed`	(`integer(1)`) The seed, a 32-bit integer.
`rng_state`	(`AnvlRngState`) A random state, as `nv_get_rng_state()` returns.
`envir`	(`environment`) The scope whose exit restores the global random state.
`code`	(any) An expression to evaluate with the seeded global random state.

Value

`nv_set_seed()`, `local_nv_seed()` and `nv_set_rng_state()` return `NULL`, invisibly; `with_nv_seed()` returns the result of evaluating `code`. `nv_get_rng_state()` returns a copy of the global random state, an `AnvlRngState`.

See Also

Other rng: local_nv_rng_state(), nv_normal, nv_rbinom(), nv_rng_state(), nv_sample(), nv_sample_int(), nv_uniform

Examples

```
nv_set_seed(42L)
nv_runif(3L)
nv_set_seed(42L)
nv_runif(3L)

# save the random state and continue from it later
rng_state <- nv_get_rng_state()
nv_runif(2L)
nv_set_rng_state(rng_state)
nv_runif(2L)

with_nv_seed(1L, nv_runif(2L))
```

---

nv_shift_left *Shift Left*

Description

Element-wise left bit shift.

Usage

```
nv_shift_left(x, shift)
```

Arguments

x	(arrayish) The array whose bits are shifted. Can be any integer data type.
shift	(arrayish) By how many bits to shift each element of x. Brought to x's data type, which it must fit in. Scalars are broadcast.

Value

(arrayish)
Has the inputs' broadcast shape and x's data type.

See Also

prim_shift_left() for the underlying primitive.

Examples

```
x <- nv_array(c(1L, 2L, 4L))
shift <- nv_array(c(1L, 2L, 1L))
nv_shift_left(x, shift)

# the result keeps `x`'s data type, which `shift` is brought to
nv_shift_left(nv_scalar(8L, "i64"), nv_scalar(2L, "i32"))

# a scalar is broadcast
nv_shift_left(x, 1L)
```

`nv_shift_right_arithmetic`
Arithmetic Shift Right

Description

Element-wise arithmetic right bit shift.

Usage

```
nv_shift_right_arithmetic(x, shift)
```

Arguments

`x`	(arrayish) The array whose bits are shifted. Can be any integer data type.
`shift`	(arrayish) By how many bits to shift each element of `x`. Brought to `x`'s data type, which it must fit in. Scalars are broadcast.

Value

(arrayish)
Has the inputs' broadcast shape and `x`'s data type.

See Also

`prim_shift_right_arithmetic()` for the underlying primitive.

Examples

```
x <- nv_array(c(8L, -16L, 32L))
shift <- nv_array(c(1L, 2L, 3L))
nv_shift_right_arithmetic(x, shift)

# the result keeps `x`'s data type, which `shift` is brought to
```

```
nv_shift_right_arithmetic(nv_scalar(-32L, "i64"), nv_scalar(2L, "i32"))

# a scalar is broadcast
nv_shift_right_arithmetic(x, 1L)
```

`nv_shift_right_logical` *Logical Shift Right*

Description

Element-wise logical right bit shift.

Usage

```
nv_shift_right_logical(x, shift)
```

Arguments

`x`	(arrayish) The array whose bits are shifted. Can be any integer data type.
`shift`	(arrayish) By how many bits to shift each element of `x`. Brought to `x`'s data type, which it must fit in. Scalars are broadcast.

Value

(arrayish)
Has the inputs' broadcast shape and `x`'s data type.

See Also

`prim_shift_right_logical()` for the underlying primitive.

Examples

```
x <- nv_array(c(8L, 16L, 32L))
shift <- nv_array(c(1L, 2L, 3L))
nv_shift_right_logical(x, shift)

# the result keeps `x`'s data type, which `shift` is brought to
nv_shift_right_logical(nv_scalar(32L, "i64"), nv_scalar(2L, "i32"))

# a scalar is broadcast
nv_shift_right_logical(x, 1L)
```

nv_sign *Sign*

Description

Element-wise sign function: `-1`, `0` or `1` at the input's data type. An unsigned input holds no negative value, so its sign is `0` or `1`, like base R's `sign()` on a non-negative number. You can also use `sign()`.

Usage

```
nv_sign(x)

## S3 method for class 'AnvlArray'
sign(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

See Also

`prim_sign()` for the underlying primitive, which takes a signed input only.

Examples

```
# the input's data type carries through
x <- nv_array(c(-3, 0, 5))
sign(x)

# an unsigned input is 0 where it is 0 and 1 everywhere else
nv_sign(nv_array(c(0L, 3L), dtype = "ui32"))

# an R value materializes at its default data type
nv_sign(-3)
```

nv_sin *Sine*

Description

Element-wise sine. You can also use `sin()`.

Usage

```
nv_sin(x)

## S3 method for class 'AnvlArray'
sin(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_sin()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, pi / 2, pi))
sin(x)

# an R value materializes at its default data type
nv_sin(0)
```

nv_sinh *Hyperbolic Sine*

Description

Element-wise hyperbolic sine. You can also use `sinh()`.

Usage

```
nv_sinh(x)

## S3 method for class 'AnvlArray'
sinh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`prim_sinh()` for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
sinh(x)

# an R value materializes at its default data type
nv_sinh(1)
```

nv_sinpi *Sine of a Multiple of Pi*

Description

Element-wise `sin(pi * x)`. You can also use `sinpi()`.

Usage

```
nv_sinpi(x)

## S3 method for class 'AnvlArray'
sinpi(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`nv_cospi()`, `nv_tanpi()`, `nv_sin()`

Examples

```
sinpi(nv_array(c(0, 0.5, 1, 1.5)))
```

nv_solve *Solve Linear System*

Description

Solves the linear system `a %*% x = b` for `x`. Uses LU decomposition with partial pivoting internally, so `a` need only be square and non-singular. Unlike `base::solve()`, a singular `a` is not an error: the result then holds non-finite values.

Usage

```
nv_solve(a, b)

## S3 method for class 'AnvlArray'
solve(a, b, ...)
```

Arguments

a	(arrayish) Square non-singular matrix with exactly 2 axes. Can be any numeric data type: a and b are promoted to a common data type and that is then converted to the default float data type (see default_dtypes()) where it is not a float already, since the decomposition is a float one. An R value assumes the other operand's data type within its data type category, and settles on the default float when neither has one.
b	(arrayish) Right-hand side, vector of length n or matrix with n rows. Promoted together with a – see a.
...	No additional arguments.

Details

$$Ax = b$$

$$PA = LU$$

$$LUx = Pb$$

$$Ly = Pb$$

$$Ux = y$$

Value

(arrayish)
The solution x such that a %*% x = b, with b's shape and the operands' common data type – or the default float data type (see default_dtypes()) where that was an integer one.

Shapes

- a: (n, n)
- b: (n,) or (n, k)
- output: same shape as b

See Also

nv_chol(), nv_triangular_solve(), prim_lu()

Examples

```
# the solution has `b`'s shape and the operands' common data type
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
b <- nv_matrix(c(1, 2), nrow = 2, dtype = "f64")
nv_solve(a, b)

# an integer system is solved at the default float data type
nv_solve(nv_matrix(c(3L, 1L, 1L, 2L), nrow = 2), nv_array(c(9L, 8L)))
```

---

nv_sort *Sort*

Description

Sorts an array along an axis.

You can also use `sort()` directly.

Usage

```
nv_sort(x, axis = NULL, decreasing = FALSE, stable = FALSE)

## S3 method for class 'AnvlArray'
sort(x, decreasing = FALSE, ..., axis = NULL)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axis`	(`integer(1)` | `NULL`) Axis along which to sort. Negative values count from the end, i.e. `-1` refers to the last axis. If `NULL` (default), the input is first flattened to a 1-D array, like `base::sort()`.
`decreasing`	(`logical(1)`) If `TRUE`, sort in decreasing order.
`stable`	(`logical(1)`) If `TRUE`, the sort is stable: equal values keep their original relative order along `axis`. Default `FALSE`. Stability is only observable for floats when `-0` / `+0` or `-NaN` / `+NaN` are mixed (they compare equal under the total order used here); for distinct values the result is identical either way.
`...`	Passed on to `nv_sort()`, e.g. `stable`.

Value

(`arrayish`)
Has the input's shape and data type.

NaN handling

NaN values sort to the **end** (ascending) or **beginning** (descending), regardless of sign. +0 and -0 compare equal.

The sort() generic

Like base::sort(), nv_sort() with axis = NULL flattens a multi-axis array into one sorted vector, so sort() on an anvl array agrees with base R (the flatten order does not matter once the elements are sorted). It differs in one respect: base R drops NA by default, whereas NaN is kept and sorted to the end. Pass axis to sort each slice along one axis instead, which keeps the shape.

See Also

prim_sort() for the underlying primitive, nv_order(), nv_top_k(), nv_median(), nv_which_max(), nv_which_min().

Examples

```
# sorting moves elements, so the data type and shape stay
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_sort(x)
sort(x) # via the S3 generic
nv_sort(x, decreasing = TRUE)

m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_sort(m) # one sorted vector, like base R
nv_sort(m, axis = 2L) # each row sorted, shape kept
```

---

nv_sqrt *Square Root*

Description

Element-wise square root. You can also use sqrt().

Usage

```
nv_sqrt(x)

## S3 method for class 'AnvlArray'
sqrt(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_sqrt() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 4, 9))
sqrt(x)

# an R value materializes at its default data type
nv_sqrt(4)
```

---

nv_squeeze *Squeeze*

Description

Removes axes of size 1 from an array. nv_drop() is another spelling of the same function; with the default axes = NULL it drops every size-1 axis, like base::drop().

Usage

```
nv_squeeze(x, axes = NULL)

nv_drop(x, axes = NULL)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to squeeze. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), all axes of size 1 are removed.

Value

(arrayish)
Has x's data type, with the specified axes removed from its shape.

See Also

nv_unsqueeze(), nv_reshape()

Examples

```
# the two size-1 axes are dropped
x <- nv_array(1:6, shape = c(1, 6, 1))
nv_squeeze(x)
nv_drop(x)
```

---

nv_static_slice *Static Slice*

Description

Extracts a slice from an array using static (compile-time) indices. For dynamic indexing, use nv_subset() instead.

Usage

```
nv_static_slice(x, start_indices, end_indices, strides)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
start_indices	(integer()) Start indices (inclusive), one per axis.
end_indices	(integer()) End indices (inclusive), one per axis: the element at end_indices is part of the slice.
strides	(integer()) Step sizes, one per axis. A stride of 1 selects every element.

Value

(arrayish)
Has x's data type and shape ceiling((end_indices - start_indices + 1) / strides) per axis.

See Also

nv_subset(), prim_static_slice() for the underlying primitive.

Examples

```
# elements 2 through 5, the end being inclusive
x <- nv_array(1:10)
nv_static_slice(x, start_indices = 2L, end_indices = 5L, strides = 1L)
```

---

nv_sub *Subtraction*

Description

Subtracts two arrays element-wise. You can also use the - operator.

Usage

```
nv_sub(lhs, rhs)

## S3 method for class 'AnvlArray'
e1 - e2
```

Arguments

lhs, rhs	(arrayish) Two inputs with a common data type. Can be any numeric data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.
e1, e2	(arrayish) The operands of the operator, which it passes on as lhs and rhs.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_sub() for the underlying primitive.

Examples

```
x <- nv_array(c(4, 5, 6))
y <- nv_array(c(1, 2, 3))
nv_sub(x, y)
x - y

# different data types are promoted to their common one
nv_sub(nv_scalar(5, "f32"), nv_scalar(3, "f64"))
```

```
# a scalar is broadcast and an R integer is converted to a float
x - 1L
```

---

nv_sum *Sum Reduction*

Description

Sums array elements along the specified axes. A boolean array is counted, like `base::sum()` does.

Usage

```
nv_sum(x, axes = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
sum(..., na.rm = FALSE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.
...	(arrayish) The arrays to reduce, and, when there is only one, further named arguments of nv_sum(), such as axes.
na.rm	(logical(1)) Forwarded to nv_sum()'s nan_rm argument.

Value

(arrayish)
Has the input's data type, except a boolean input, which is accumulated at the default integer data type (see default_dtypes()). The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

The `sum()` generic

`sum()` reduces over all axes and, like `base::sum()`, takes several data arguments: `sum(x, y)` is the sum of both arrays. `na.rm` becomes `nan_rm`. Beyond base R, named arguments are passed on, so `sum(x, axes = 1L)` reduces a single axis – but only when `x` is the only data argument.

See Also

`prim_sum()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# no axes given: reduce over all of them
nv_sum(x)

# reducing axis 1 removes it, drop = FALSE keeps it at size 1
nv_sum(x, axes = 1L)
nv_sum(x, axes = 1L, drop = FALSE)

# negative axes count from the end
nv_sum(x, axes = -1L)

# NaN propagates unless nan_rm = TRUE
nv_sum(nv_array(c(1, NaN, 3)))
nv_sum(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
nv_sum(nv_array(c(TRUE, FALSE, TRUE))) # counts: 2
```

`nv_svd` *Singular Value Decomposition*

Description

Computes the reduced ("economy") singular value decomposition of a matrix `x` of shape `(m, n)`:

$$A = u \operatorname{diag}(d)\, vt,$$

where `u` has orthonormal columns, `vt` has orthonormal rows, and `d` is the length-`k` (`k = min(m, n)`) vector of non-negative singular values in descending order.

Note: unlike `base::svd()`, which returns the right singular vectors as `v` of shape `(n, k)` (so that `a = u %*% diag(d) %*% t(v)`), this primitive returns them already transposed as `vt` of shape `(k, n)` (matching the underlying LAPACK / cuSOLVER output and avoiding an extra transpose).

Supports any matrix shape on both the host (LAPACK `gesdd`) and CUDA (cuSOLVER `gesvd`) backends. cuSOLVER's `m >= n` requirement is handled transparently via a layout flip for wide matrices.

Usage

```
nv_svd(x)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(named list of three arrayish)
Elements d (length k), u (shape (m, k)), and vt (shape (k, n)). All have the input's data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_svd(), base::svd()

Examples

```
# all three outputs have the input's data type
x <- nv_matrix(c(1, 0, 0, 1, 0, 1), nrow = 3, dtype = "f64")
nv_svd(x)
```

---

nv_tan *Tangent*

Description

Element-wise tangent. You can also use tan().

Usage

```
nv_tan(x)

## S3 method for class 'AnvlArray'
tan(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_tan() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.5, 1))
tan(x)

# an R value materializes at its default data type
nv_tan(0.5)
```

---

nv_tanh *Hyperbolic Tangent*

Description

Element-wise hyperbolic tangent. You can also use tanh().

Usage

```
nv_tanh(x)

## S3 method for class 'AnvlArray'
tanh(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see default_dtypes()). An R value materializes at its default data type and is converted in the same way.

Value

(arrayish)
Has the input's shape, and its data type – or the default float data type (see default_dtypes()) where the input was an integer one.

See Also

prim_tanh() for the underlying primitive.

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
tanh(x)

# an R value materializes at its default data type
nv_tanh(1)
```

`nv_tanpi` *Tangent of a Multiple of Pi*

Description

Element-wise `tan(pi * x)`. You can also use `tanpi()`. Like base R's `base::tanpi()`, it is exact for a whole argument and `NaN` at the half integers, where the tangent has its poles.

Usage

```
nv_tanpi(x)

## S3 method for class 'AnvlArray'
tanpi(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any numeric data type: a float keeps its own, and an integer one is converted to the default float data type (see `default_dtypes()`). An R value materializes at its default data type and is converted in the same way.

Value

(`arrayish`)
Has the input's shape, and its data type – or the default float data type (see `default_dtypes()`) where the input was an integer one.

See Also

`nv_sinpi()`, `nv_cospi()`, `nv_tan()`

Examples

```
tanpi(nv_array(c(0, 0.25, 0.5, 1)))
```

`nv_tcrossprod` *Transpose Cross Product (Matrix)*

Description

Computes `x %*% t(y)`. If `y` is missing, computes `x %*% t(x)`. Above rank 2 the last two axes are the matrix and the leading ones are batch axes, as in `nv_matmul()`: only the matrix is transposed.

Usage

```
nv_tcrossprod(x, y = NULL)

## S3 method for class 'AnvlArray'
tcrossprod(x, y = NULL, ...)
```

Arguments

`x`	(`arrayish`) An array with at least 2 axes, as for `base::tcrossprod()`. Can be any numeric data type; `x` and `y` are promoted to a common data type.
`y`	(`arrayish` | `NULL`) Optional second array. If `NULL`, uses `x`.
`...`	No additional arguments.

Value

(`arrayish`)
Has the operands' common data type, and the shape of `x %*% t(y)`.

See Also

`nv_crossprod()`, `nv_matmul()`

Examples

```
# `x %*% t(x)`, so a 2x3 gives a 2x2
x <- nv_matrix(1:6, nrow = 2, dtype = "f32")
nv_tcrossprod(x)
```

nv_top_k *Top-K Elements*

Description

Returns the k largest values over one or more axes, sorted in decreasing order.

Usage

```
nv_top_k(x, k, axes = NULL, indices = FALSE)
```

Arguments

x	(arrayish) One input, with at least 1 axis. Can be any numeric data type. An R value materializes at its default data type.
k	(integer(1)) Number of top elements to return. Must be a whole number between 1 and the number of elements axes holds together; a fractional or logical k is refused rather than truncated.
axes	(integer() | NULL) Axes to take the top k over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), ranks over every axis.
indices	(logical(1)) If FALSE (default), returns just the top-k values. If TRUE, returns list(values = ..., indices = ...) where indices holds the position of each top-k value.

Value

(arrayish | named list of two arrayish)
One array when indices = FALSE, a named list of values and indices when indices = TRUE. The values have the input's data type and the indices the default integer data type (see default_dtypes()). Both have the input's shape with axes replaced by a single axis of size k, sitting where the first of them was; values are sorted decreasing along that axis.

Ranking several axes

Taking the top k over several axes ranks all of their elements together, so nv_top_k(x, k) equals nv_top_k(nv_flatten(x), k). The indices then index the column-major flattening of those axes – the order nv_flatten() produces – rather than any single axis.

NaN handling

NaN ranks larger than any finite value (so it appears first in the top-k output); -NaN ranks smaller. Unlike nv_sort(), the sign bit is not canonicalized.

See Also

prim_top_k() for the underlying primitive, nv_sort().

Examples

```
# the values keep the input's data type, the indices the default integer
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_top_k(x, k = 3L)
nv_top_k(x, k = 3L, indices = TRUE)

m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_top_k(m, k = 2L, axes = 2L) # the top 2 of each row
nv_top_k(m, k = 2L) # the top 2 of the whole matrix
```

---

nv_trace *Matrix Trace*

Description

Computes the trace (sum of diagonal elements) of a 2-D array.

Usage

```
nv_trace(x)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any data type. An R value materializes at its default data type.

Value

(arrayish)
A scalar with x's data type, except a boolean input, which is counted at the default integer data type (see default_dtypes()).

See Also

nv_extract_diag(), nv_diag()

Examples

```
# the diagonal is summed to a scalar
x <- nv_array(c(1, 0, 0, 0, 2, 0, 0, 0, 3), shape = c(3, 3))
nv_trace(x)
```

nv_triangular_solve *Triangular Solve*

Description

Solves a triangular system of linear equations. When `left = TRUE`, returns `x` such that `op(a) %*% x = b`. When `left = FALSE`, returns `x` such that `x %*% op(a) = b`. Here `op` is `a` or `t(a)` depending on `transpose`.

Usage

```
nv_triangular_solve(
  a,
  b,
  left = TRUE,
  lower = TRUE,
  unit_diag = FALSE,
  transpose = FALSE
)
```

Arguments

`a`	(`arrayish`) Triangular coefficient matrix with at least 2 axes. The last two axes must be equal; any leading axes are batch axes. Can be any numeric data type: `a` and `b` are promoted to a common data type and that is then converted to the default float data type (see `default_dtypes()`) where it is not a float already, since the solve is a float one. An R value assumes the other operand's data type within its data type category, and settles on the default float when neither has one.
`b`	(`arrayish`) Right-hand side. For `a` of shape `(B..., n, n)`, `b` may be either: • full rank — shape `(B..., n, k)` when `left = TRUE`, or `(B..., k, n)` when `left = FALSE`; • one rank less, shape `(B..., n)`, meaning a single column (`left = TRUE`) or row (`left = FALSE`) per batch — it is reshaped internally and the reshape is undone on the result so the output rank matches `b`. `b`'s batch axes (`B...`) must match `a`'s exactly. It is promoted together with `a` – see `a`.
`left`	(`logical(1)`) If `TRUE` (default), solve `op(a) %*% x = b`; if `FALSE`, solve `x %*% op(a) = b`.
`lower`	(`logical(1)`) Whether `a` is lower or upper triangular. Defaults to `TRUE`.
`unit_diag`	(`logical(1)`) If `TRUE`, the diagonal of `a` is treated as all ones (and the actual values on the diagonal are ignored). Defaults to `FALSE`.

transpose (logical(1))
If TRUE, solve with t(a) in place of a. Defaults to FALSE.

Details

As a convenience, b may have one fewer axis than a (a single right-hand side per batch, shape (B..., n) for a of shape (B..., n, n)). It is reshaped internally to a column (left = TRUE) or row (left = FALSE) and reshaped back on the way out. Because we don't broadcast, this is not ambiguous (as it would be for NumPy).

Differentiation is only implemented for a single system: a gradient() of a batched solve errors.

Value

(arrayish)
The solution x, with b's shape and the operands' common data type – or the default float data type (see default_dtypes()) where that was an integer one.

See Also

nv_solve(), nv_chol(), prim_triangular_solve()

Examples

```
L <- nv_matrix(c(2, 1, 0, 3), nrow = 2, dtype = "f32")
b <- nv_matrix(c(4, 3), nrow = 2, dtype = "f32")
nv_triangular_solve(L, b)
```

---

nv_tril *Lower Triangular Matrix*

Description

Returns the lower triangular part of a 2-D array, setting elements above the specified diagonal to zero.

Usage

```
nv_tril(x, diagonal = 0L)
```

Arguments

x (arrayish)
One input, with exactly 2 axes. Can be any data type. An R value materializes at its default data type.

diagonal (integer(1))
Diagonal offset: the kept region is col - row <= diagonal. 0 (default) keeps the main diagonal and everything below it, a positive value keeps that many diagonals above it as well, and a negative one drops the main diagonal and -diagonal - 1 below it.

Value

(arrayish)
Has the same shape and data type as x.

See Also

nv_triu(), nv_lower_tri()

Examples

```
# elements above the main diagonal become zero
x <- nv_fill(1, c(3, 3))
nv_tril(x)
```

---

nv_triu *Upper Triangular Matrix*

Description

Returns the upper triangular part of a 2-D array, setting elements below the specified diagonal to zero.

Usage

```
nv_triu(x, diagonal = 0L)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any data type. An R value materializes at its default data type.
diagonal	(integer(1)) Diagonal offset: the kept region is col - row >= diagonal. 0 (default) keeps the main diagonal and everything above it, a negative value keeps that many diagonals below it as well, and a positive one drops the main diagonal and diagonal - 1 above it.

Value

(arrayish)
Has the same shape and data type as x.

See Also

nv_tril(), nv_upper_tri()

Examples

```
# elements below the main diagonal become zero
x <- nv_fill(1, c(3, 3))
nv_triu(x)
```

---

nv_trunc *Truncate*

Description

Element-wise truncation (round toward zero). You can also use `trunc()`.

Usage

```
nv_trunc(x)

## S3 method for class 'AnvlArray'
trunc(x, ...)
```

Arguments

x	(arrayish) One input. Can be any numeric data type: a float is rounded and keeps its own, and an integer one is already whole and is returned unchanged. An R value materializes at its default data type and is treated in the same way.
...	Not used; must be empty.

Value

(arrayish)
Has the input's shape and data type.

See Also

`nv_floor()`, `nv_ceiling()`, `nv_round()`.

Examples

```
# the input's data type carries through
x <- nv_array(c(1.2, 2.7, -1.5))
trunc(x)
trunc(nv_array(1:3)) # an integer array is already whole
```

nv_uniform *The Uniform Distribution*

Description

Density (nv_dunif), distribution function (nv_punif), quantile function (nv_qunif), and random generation (nv_runif) for the Uniform distribution on the interval from min to max.

Usage

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

nv_punif(q, min = 0, max = 1, lower_tail = TRUE, log_p = FALSE)

nv_qunif(p, min = 0, max = 1, lower_tail = TRUE, log_p = FALSE)

nv_runif(shape, min = 0, max = 1, dtype = NULL)
```

Arguments

x, q	(arrayish) Quantiles at which to evaluate the density (x) or the distribution function (q).
min, max	(arrayish) Lower and upper limits of the distribution. Either scalars, or arrays of exactly the same shape as x/q/p (or the sample, for nv_runif), in which case the interval varies elementwise and each element of x/q/p is evaluated against, or each draw made from, its own min/max.
log, log_p	(logical(1)) If TRUE, the densities/probabilities are given as logarithms. For nv_qunif this describes the input p.
lower_tail	(logical(1)) If TRUE (default), probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
p	(arrayish) Probabilities at which to evaluate the quantile function. Values outside $[0, 1]$ give NaN.
shape	(integer()) Shape of the result.
dtype	(NULL | character(1) | DataType) Floating point data type of the sample. The default (NULL) takes it from min and max, and uses the default float type where both are R values.

Details

The Uniform distribution has probability density function:

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

and zero elsewhere, where a is `min` and b is `max`.

All four are univariate functions evaluated elementwise, returning one value per element of `x`/`q`/`p` (or of the sample). Non-scalar `min`/`max` therefore give a separate univariate Uniform per element, *not* a multivariate Uniform over the hyper-rectangle $\prod_i [a_i, b_i]$. For that, reduce over the result: `nv_prod(nv_dunif(x, min, max))`, or `nv_sum(nv_dunif(x, min, max, log = TRUE))` on the log scale.

Value

(`arrayish`)
`nv_dunif()`, `nv_punif()`, and `nv_qunif()` return an `arrayish` with the same shape as `x`/`q`/`p` and the float data type described under Data Types.

`nv_runif()` returns the sample, an `arrayish` of shape `shape` and the data type described under `dtype`.

Data Types

`nv_dunif()`, `nv_punif()` and `nv_qunif()` compute at a float data type and return it. If `x`/`q`/`p` is a float, they compute at its data type; an R double settles on the default float. An integer `x`/`q`/`p` (an array or an R integer) is converted to the widest float among the arrays given for `min` or `max`, or to the default float if there is none. Integers beyond 2^{24} are not all exact at `f32`, so a large integer may change in that conversion.

An R value for `min` or `max` takes the computing data type. An array is promoted to it, so a wider one (e.g. `f64` for an `f32` `x`) is an error.

Random State

The draws come from the active random state: the global random state (see `nv_set_seed()`), or the random state that `with_nv_rng_state()` or `local_nv_rng_state()` make the default for a scope. They advance it in place.

Random generation

`nv_runif` draws uniforms on the open interval $(0, 1)$ and scales them to (a, b). Rounding to the data type can still give a or b where the interval is narrow next to its magnitude, e.g. for `nv_runif(n, 1e6, 1e6 + 1)` at `f32`.

`min` and `max` are `arrayish`, so they may vary across the sample: they are applied to the draws after they have been reshaped to `shape`, and so may either be scalars or have exactly that shape. As in base R's `runif()`, an element whose `min` equals its `max` is that value, and one whose `min` or `max` is not finite, or whose `max` is less than its `min`, is `NaN`. The random state is advanced regardless.

See Also

Other rng: `local_nv_rng_state()`, `nv_normal`, `nv_rbinom()`, `nv_rng_state()`, `nv_sample()`, `nv_sample_int()`, `nv_set_seed()`

Examples

```
x <- nv_array(c(-0.5, 0, 0.25, 1, 1.5))
nv_dunif(x)
nv_dunif(x, min = -1, max = 2)
nv_dunif(x, log = TRUE)

# `min`/`max` may vary elementwise, giving one univariate Uniform per
# element rather than a single distribution over a hyper-rectangle
lower <- nv_array(c(-1, -1, 0, 0, 1))
upper <- nv_array(c(0, 1, 1, 2, 2))
nv_dunif(x, min = lower, max = upper)

nv_punif(x)
nv_punif(x, min = -1, max = 2)
nv_punif(x, lower_tail = FALSE)
nv_punif(x, log_p = TRUE)

p <- nv_array(c(0.025, 0.5, 0.975))
nv_qunif(p)
nv_qunif(p, min = -1, max = 2)
nv_qunif(p, lower_tail = FALSE)
nv_qunif(nv_array(c(-700, -2, -0.1), dtype = "f64"), log_p = TRUE)

with_nv_seed(42L, nv_runif(c(2, 3)))

# `min`/`max` may also be arrays of the same shape as the sample
lower <- nv_array(matrix(c(0, 10, 20, 30, 40, 50), nrow = 2))
nv_runif(c(2, 3), min = lower, max = lower + 1)
```

`nv_unsqueeze` *Unsqueeze*

Description

Inserts axes of size 1 at the specified positions.

Usage

```
nv_unsqueeze(x, axes)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axes`	(`integer()`) Positions of the new axes in the *result*, which has `naxes(x) + length(axes)` axes. Negative values count from the end of the result, i.e. `-1` appends a new axis at the end.

Value

(arrayish)
Has x's data type, with an extra axis of size 1 in its shape for each of axes.

See Also

nv_squeeze(), nv_reshape()

Examples

```
# a size-1 axis is inserted, at the front or at the back
x <- nv_array(c(1, 2, 3))
nv_unsqueeze(x, axes = 1L)
nv_unsqueeze(x, axes = -1L)

# several at once
nv_unsqueeze(x, axes = c(1L, 3L))
```

---

nv_upper_tri *Upper Triangular Mask*

Description

Returns a boolean matrix that is TRUE on and above the given diagonal, mirroring base R's upper.tri(). Use nv_triu() to zero out the other triangle of an existing array instead.

Usage

```
nv_upper_tri(shape, diagonal = 1L, device = NULL)

nv_upper_tri_like(like, shape = NULL, diagonal = 1L, device = NULL)
```

Arguments

shape	(integer()) Shape of the result: exactly two axis sizes, since the result is a matrix.
diagonal	(integer(1)) Diagonal offset, with the same meaning as in nv_triu(). The default 1 excludes the main diagonal, matching upper.tri(); use 0 to include it, matching upper.tri(diag = TRUE).
device	(NULL | character(1) | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or

- a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error.

The default (NULL) uses default_device().

like (AnvlArray)
Existing array whose attributes are used as defaults (only for nv_upper_tri_like()).

Value

(arrayish)
Has the given shape and boolean data type.

See Also

nv_lower_tri(), nv_triu(), prim_iota() for the underlying primitive.

Examples

```
# a boolean mask, whatever the array it is later used with
nv_upper_tri(c(3, 3))
nv_upper_tri(c(3, 3), diagonal = 0L)
x <- nv_fill(0, shape = c(3, 3))
nv_upper_tri_like(x)
```

---

nv_var *Variance*

Description

Computes the variance along the specified axes.

Usage

```
nv_var(x, axes = NULL, drop = TRUE, correction = 1L, nan_rm = FALSE)
```

Arguments

x (arrayish)
One input. Can be any data type. An R value materializes at its default data type.

axes (integer() | NULL)
Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.

drop (logical(1))
Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

correction (integer(1))
Degrees of freedom correction. Default is 1 (Bessel's correction).

nan_rm (logical(1))
How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Details

Uses Bessel's correction by default (correction = 1), matching R's var(). Set correction = 0 for population variance.

Value

(arrayish)
Has the input's data type where that is a float, and the default float data type (see default_dtypes()) otherwise. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

See Also

nv_sd(), nv_mean()

Examples

```
x <- nv_array(1:5)
# the result is a float, even though the input is an integer
nv_var(x)

# Bessel's correction by default, correction = 0 for the population variance
nv_var(x, correction = 0L)

# NaN propagates unless nan_rm = TRUE
nv_var(nv_array(c(1, NaN, 3, 5)))
nv_var(nv_array(c(1, NaN, 3, 5)), nan_rm = TRUE)
```

---

nv_which_max *Index of the Maximum*

Description

Returns the index of the maximum value over one or more axes. Ties are broken by returning the smallest index.

Usage

```
nv_which_max(x, axes = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

(arrayish)
Has the default integer data type (see default_dtypes()) regardless of the input's, and the input's shape with axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Reducing several axes

nv_which_max() is the index to nv_max()'s value: called with the same axes and drop, it points at the element whose value nv_max() returns. Reducing several axes ranks their elements together, and the result indexes the column-major flattening of those axes – the order nv_flatten() produces and base::which.max() reports – which is also the order ties are broken in.

NaN handling

With nan_rm = FALSE (default), if any entry being reduced is NaN, the returned index points at the first such NaN. With nan_rm = TRUE, NaN entries are skipped.

See Also

nv_which_min(), nv_max().

Examples

```
# the index comes out at the default integer data type
nv_which_max(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_which_max(m) # indexes the flattened matrix
nv_which_max(m, axes = 2L) # one index per row
nv_which_max(nv_array(c(1, NaN, 3)))
nv_which_max(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

nv_which_min *Index of the Minimum*

Description

Returns the index of the minimum value over one or more axes. Ties are broken by returning the smallest index.

Usage

```
nv_which_min(x, axes = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer() | NULL) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis. If NULL (default), reduces over all axes.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.
nan_rm	(logical(1)) How to handle NaN values in float inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

(arrayish)
Has the default integer data type (see default_dtypes()) regardless of the input's, and the input's shape with axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Reducing several axes

nv_which_min() is the index to nv_min()'s value: called with the same axes and drop, it points at the element whose value nv_min() returns. Reducing several axes ranks their elements together, and the result indexes the column-major flattening of those axes – the order nv_flatten() produces and base::which.min() reports – which is also the order ties are broken in.

NaN handling

With nan_rm = FALSE (default), if any entry being reduced is NaN, the returned index points at the first such NaN. With nan_rm = TRUE, NaN entries are skipped.

See Also

nv_which_max(), nv_min().

Examples

```
# the index comes out at the default integer data type
nv_which_min(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_which_min(m) # indexes the flattened matrix
nv_which_min(m, axes = 2L) # one index per row
nv_which_min(nv_array(c(2, NaN, 1, 3)))
nv_which_min(nv_array(c(2, NaN, 1, 3)), nan_rm = TRUE)
```

---

nv_while *While Loop*

Description

Executes a functional while loop.

Usage

```
nv_while(init, cond, body)
```

Arguments

`init`	(named `list()`) Named list of initial state values, a tree in the sense of pjrt's `RTree`. Each leaf becomes a parameter of the loop's sub-graphs. R values are materialized at their default data type.
`cond`	(`function`) Condition function returning a scalar boolean. Receives the state values as arguments.
`body`	(`function`) Body function returning the updated state as a named list with the same structure, data types and shapes as `init`. Nothing is promoted: a loop-carried state is meant to be heterogeneous, so each member keeps its own data type across iterations.

Value

(named `list`)
A tree of the loop-carried arrays – see `RTree` – in its final state after the loop terminates, with `init`'s structure, data types and shapes.

Gradients

`gradient()` cannot differentiate a while loop: its trip count is only known at run time, so there is no static size for the per-iteration states the backward pass needs. `nv_scan()`, whose number of steps is static, is differentiable.

See Also

prim_while() for the underlying primitive.

Examples

```
# the loop state is a named list, and each member keeps its data type
nv_while(
  init = list(i = nv_scalar(0L), total = nv_scalar(0L)),
  cond = function(i, total) i < 5L,
  body = function(i, total) list(
    i = i + 1L,
    total = total + i
  )
)
```

---

nv_xor *Bitwise XOR*

Description

Element-wise bitwise XOR – a logical XOR on a boolean input, and a bit-by-bit one on an integer. For *logical* inputs, you can also use `xor`.

Usage

```
nv_xor(x, y)
```

Arguments

x, y	(arrayish) Two inputs with a common data type. Can be any integerish data type. Scalars are broadcast. An R value takes the other operand's data type when that is in its own or a higher category. Otherwise it settles on its default data type, and the operands meet at their common data type.

Value

(arrayish)
Has the inputs' broadcast shape and common data type.

See Also

prim_xor() for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
nv_xor(x, y)

# different data types are promoted to their common one
nv_xor(nv_scalar(12L, "i32"), nv_scalar(10L, "i64"))

# a scalar is broadcast
nv_xor(x, TRUE)
```

---

peek_dtype *Peek at a Data Type*

Description

The data type `x` would take if it materialized. Relevant for R objects and their `RData` trace-time analogon: for those it is the default of the active backend (see `default_dtypes()`), which the value has not materialized at yet.

Usage

```
peek_dtype(x)
```

Arguments

x	(`arrayish` | `AbstractArray`) The value to ask about.

Value

(`xlamisc::DataType`)
The data type `x` has, or the default data type it would materialize at if it is still a bare R value.

See Also

`as_anvl_arrays()`, RData, shape()

Examples

```
peek_dtype(1.5)
peek_dtype(1L)
peek_dtype(nv_array(1:3, dtype = "i8"))
```

`platform.AnvlArray` *Get the Platform Name of an Array or Buffer*

Description

Returns the name of the hardware platform (e.g. `"cpu"`, `"cuda"`) the data lives on. This is not the backend; see `backend()` for that.

Usage

```
## S3 method for class 'AnvlArray'
platform(x, ...)

platform(x, ...)
```

Arguments

`x`	(`AnvlArray` | `PJRTBuffer`) An array or buffer.
`...`	Additional arguments passed to methods (unused).

Details

Implemented via the generic `pjrt::platform()`.

Value

`(character(1))`

See Also

`pjrt::platform()`

Examples

```
x <- nv_array(1:4, dtype = "f32")
platform(x)
```

prim_abs *Primitive Absolute Value*

Description

Element-wise absolute value.

Usage

```
prim_abs(x)
```

Arguments

x	(arrayish) One input. Can be any signed numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_abs(), specified under abs.

See Also

nv_abs(), abs()

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 2, -3))
prim_abs(x)

# an R value materializes at its default data type
prim_abs(-1)
```

`prim_acos` *Primitive Arc Cosine*

Description

Element-wise inverse cosine.

Usage

```
prim_acos(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_acos()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.acos.

See Also

`nv_acos()`, `acos()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_acos(x)

# an R value materializes at its default data type
prim_acos(0.5)
```

prim_acosh *Primitive Inverse Hyperbolic Cosine*

Description

Element-wise inverse hyperbolic cosine.

Usage

```
prim_acosh(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_acosh(), an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.acosh.

See Also

nv_acosh(), acosh()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 2, 10))
prim_acosh(x)

# an R value materializes at its default data type
prim_acosh(2)
```

`prim_add` *Primitive Addition*

Description

Adds two arrays element-wise.

Usage

```
prim_add(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_add()`, specified under add.

See Also

`nv_add()`, `+`

Examples

```
# two R values: both take an R double's default data type
prim_add(1, 2)

# the R value is built at the array's data type instead
prim_add(1, nv_scalar(2, "f64"))
```

prim_all *Primitive All Reduction*

Description

Performs logical AND along the specified axes.

Usage

```
prim_all(x, axes, drop = TRUE)
```

Arguments

x	(arrayish) One input. Must be a boolean or an R logical.
axes	(integer()) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

Value

(arrayish)
Has the boolean data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reduce(), specified under reduce. The reducer is hlo_and().

See Also

nv_all()

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
# TRUE where every element is TRUE along axis 1
prim_all(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_all(x, axes = 1L, drop = FALSE)
```

---

prim_and *Primitive Bitwise And*

Description

Element-wise bitwise AND – a logical AND on a boolean input, and a bit-by-bit one on an integer.

Usage

```
prim_and(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any integerish data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_and(), specified under and.

See Also

nv_and(), &

Examples

```
# two R values: both take an R integer's default data type
prim_and(12L, 10L)

# the R value is built at the array's data type instead
prim_and(12L, nv_scalar(10L, "i64"))
```

`prim_any` *Primitive Any Reduction*

Description

Performs logical OR along the specified axes.

Usage

```
prim_any(x, axes, drop = TRUE)
```

Arguments

`x`	(`arrayish`) One input. Must be a boolean or an R logical.
`axes`	(`integer()`) Axes to reduce over. Negative values count from the end, i.e. `-1` refers to the last axis.
`drop`	(`logical(1)`) Whether to drop the reduced axes: removed from the output shape if `TRUE`, set to 1 if `FALSE`.

Value

(`arrayish`)
Has the boolean data type. The shape is the input's with the reduced axes removed (`drop = TRUE`) or set to 1 (`drop = FALSE`).

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_reduce()`, specified under reduce. The reducer is `hlo_or()`.

See Also

nv_any()

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
# TRUE where any element is TRUE along axis 1
prim_any(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_any(x, axes = 1L, drop = FALSE)
```

---

prim_asin *Primitive Arc Sine*

Description

Element-wise inverse sine.

Usage

```
prim_asin(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_asin(), an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.asin.

See Also

nv_asin(), asin()

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_asin(x)

# an R value materializes at its default data type
prim_asin(0.5)
```

`prim_asinh` *Primitive Inverse Hyperbolic Sine*

Description

Element-wise inverse hyperbolic sine.

Usage

```
prim_asinh(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_asinh()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.asinh.

See Also

`nv_asinh()`, `asinh()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_asinh(x)

# an R value materializes at its default data type
prim_asinh(1)
```

`prim_atan` *Primitive Arc Tangent*

Description

Element-wise inverse tangent.

Usage

```
prim_atan(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_atan()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.atan.

See Also

`nv_atan()`, `atan()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_atan(x)

# an R value materializes at its default data type
prim_atan(1)
```

`prim_atan2` *Primitive Arctangent 2*

Description

Element-wise atan2 operation: the angle (in radians) between the positive x-axis and the point `(x, y)`.

The operands are named `y` and `x`, in that order, after `base::atan2()`, rather than `lhs` / `rhs`.

Usage

```
prim_atan2(y, x)
```

Arguments

`y`, `x`	(`arrayish`) Ordinate and abscissa of the point, of the same shape. Can be any float data type. `y` and `x` must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_atan2()`, specified under atan2.

See Also

`nv_atan2()`

Examples

```
# two R values: both take an R double's default data type
prim_atan2(1, 1)

# the R value is built at the array's data type instead
prim_atan2(1, nv_scalar(1, "f64"))
```

`prim_atanh` *Primitive Inverse Hyperbolic Tangent*

Description

Element-wise inverse hyperbolic tangent.

Usage

```
prim_atanh(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_atanh()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.atanh.

See Also

`nv_atanh()`, `atanh()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-0.5, 0, 0.5))
prim_atanh(x)

# an R value materializes at its default data type
prim_atanh(0.5)
```

---

prim_bitcast_convert *Primitive Bitcast Conversion*

Description

Reinterprets the bits of an array as a different data type without modifying the underlying data.

Usage

```
prim_bitcast_convert(x, dtype)
```

Arguments

x	(arrayish) One input. Can be any data type except bool. An R value materializes at its default data type.
dtype	(character(1) | DataType) Any target data type except bool. One of the same bit width as the input's leaves the shape unchanged; a narrower one adds a *leading* axis holding the pieces; a wider one consumes the first axis, whose size must equal the ratio of the two widths. The pieces of one element are therefore adjacent in the column-major element order nv_flatten() reads, and a narrowing conversion lays the bytes out the way as_raw() writes them.

Value

(arrayish)
Has the given dtype, and the shape described under dtype.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_bitcast_convert(), specified under bitcast_convert. StableHLO puts the lane axis last, so a width-changing conversion is lowered with one hlo_transpose() that rotates it to the front, or off the front, depending on the direction.

See Also

nv_bitcast_convert()

Examples

```
# same width: the bits are reread, the shape stays
prim_bitcast_convert(nv_array(1L, dtype = "i32"), dtype = "f32")

# narrower: a leading axis holds the four bytes of each i32
prim_bitcast_convert(nv_array(1L, dtype = "i32"), dtype = "i8")

# wider: the first axis is consumed, and its size must be the width ratio
prim_bitcast_convert(nv_array(rep(1L, 4), dtype = "i8"), dtype = "i32")
```

---

prim_broadcast_in_axes

Primitive Broadcast

Description

Broadcasts an array to a new shape by replicating the data along new or size-1 axes.

Usage

```
prim_broadcast_in_axes(x, shape, broadcast_axes)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
shape	(integer()) Target shape. The size of each mapped axis must either match the size of the corresponding axis of x, or that axis of x must have size 1.
broadcast_axes	(integer()) Maps each axis of x to an axis of the output. Must have length equal to the number of axes of x.

Value

(arrayish)
Has the same data type as the input and the given shape.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_broadcast_in_dim(), specified under broadcast_in_dim.

See Also

nv_broadcast_to()

Examples

```
# axis 1 of the input becomes axis 2 of the result, which repeats it
x <- nv_array(c(1, 2, 3))
prim_broadcast_in_axes(x, shape = c(2, 3), broadcast_axes = 2L)
```

---

prim_cbrt *Primitive Cube Root*

Description

Element-wise cube root.

Usage

```
prim_cbrt(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_cbrt(), specified under cbrt.

See Also

nv_cbrt()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 8, 27))
prim_cbrt(x)

# an R value materializes at its default data type
prim_cbrt(8)
```

`prim_ceiling` *Primitive Ceiling*

Description

Element-wise ceiling.

Usage

```
prim_ceiling(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_ceil()`, specified under ceil.

See Also

`nv_ceiling()`, `ceiling()`

Examples

```
# the input's data type carries through
x <- nv_array(c(1.2, 2.7, -1.5))
prim_ceiling(x)

# an R value materializes at its default data type
prim_ceiling(1.2)
```

`prim_chol` *Primitive Cholesky Decomposition*

Description

Computes the Cholesky decomposition of a symmetric positive-definite matrix. Axes before the last two are batch axes.

Usage

```
prim_chol(x, lower = FALSE)
```

Arguments

`x`	(`arrayish`) One input, with at least 2 axes, the last two of equal size (a square matrix); any leading axes are batch axes. Can be any float data type. An R value materializes at its default data type.
`lower`	(`logical(1)`) If `FALSE` (default, matching base R's `base::chol()`), compute the upper triangular factor `U` such that `x = t(U) %*% U`. If `TRUE`, compute the lower triangular factor `L` such that `x = L %*% t(L)`.

Details

Differentiation is only implemented for a single matrix: the `reverse` rule errors on a batched input (an input with more than 2 axes).

Value

(`arrayish`)
Has the same shape and data type as the input. The values in the triangle not specified by `lower` are implementation-defined. A matrix that is not positive definite factors to `NaN`.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to a "potrf" hlo_custom_call() (backed by LAPACK on CPU and cuSOLVER on CUDA) + postprocessing.

References

Murray I (2016). "Differentiation of the Cholesky decomposition." *arXiv preprint arXiv:1602.07527*.

Walter S (2012). *Structured higher-order algorithmic differentiation in the forward and reverse mode with application in optimum experimental design*. Ph.D. thesis, Mathematisch-Naturwissenschaftliche Fakult"at II.

See Also

nv_solve()

Examples

```
# create a positive-definite matrix
x <- nv_matrix(c(4, 2, 2, 3), nrow = 2, dtype = "f32")
prim_chol(x, lower = TRUE)
```

---

prim_clamp *Primitive Clamp*

Description

Clamps every element of x to the range [min, max].

Usage

```
prim_clamp(x, min, max)
```

Arguments

x	(arrayish) The array to clamp. Can be any data type. x, min and max must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
min, max	(arrayish) Lower and upper bound. Each must be scalar or the same shape as x, and shares its data type.

Value

(arrayish)
Has x's shape and the data type the operands agreed on.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_clamp(), specified under clamp.

See Also

nv_clamp()

Examples

```
x <- nv_array(c(-1, 0.5, 2))
# the R bounds take x's data type
prim_clamp(x, 0, 1)

# an integer array takes integer bounds
prim_clamp(nv_array(1:5), 0L, 3L)

# the f64 bound settles it: x and max are built at f64 too
prim_clamp(1, nv_scalar(0, "f64"), 2)
```

---

prim_concatenate *Primitive Concatenate*

Description

Concatenates arrays along an axis.

Usage

```
prim_concatenate(..., axis)
```

Arguments

...	(arrayish) Arrays to concatenate. Can be of any data type. Must all have the same number of axes, and the same shape except along axis. All inputs must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
axis	(integer(1)) Axis along which to concatenate. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(arrayish)
Has the data type the inputs agreed on. The output shape matches the inputs in all axes except `axis`, which is the sum of the input sizes along that axis.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_concatenate()`, specified under concatenate.

See Also

`nv_concatenate()`

Examples

```
# the inputs already agree on a data type; axis 1 grows to 6
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
prim_concatenate(x, y, axis = 1L)
```

`prim_convert` *Primitive Convert Data Type*

Description

Converts the elements of an array to a different data type. An R value in the target's category (an R integer at `i8`, say) is materialized at the target directly and checked for out-of-range or missing values. Any other R value (e.g. an R double at an integer data type) is built at `f64`, `i32` or `bool` according to its storage type and converted like an array, without that check.

Usage

```
prim_convert(x, dtype)
```

Arguments

`x`	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
`dtype`	(`character(1)` | `DataType`) Target data type. Can be any data type; the conversion is a value conversion, so it may lose precision or wrap around.

Value

(arrayish)
Has the given dtype and the input's shape.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_convert(), specified under convert.

See Also

nv_convert()

Examples

```
# the values are preserved, the data type changes
x <- nv_array(c(1L, 2L, 3L))
prim_convert(x, dtype = "f32")
```

---

prim_convolution *Primitive Convolution*

Description

General N-D windowed convolution. The axis arguments say which axis of each operand plays which role, so any layout can be described. Most users want nv_conv1d() / nv_conv2d() / nv_conv3d() instead.

Usage

```
prim_convolution(
  x,
  kernel,
  x_batch_axis,
  x_feature_axis,
  x_spatial_axes,
  kernel_input_feature_axis,
  kernel_output_feature_axis,
  kernel_spatial_axes,
  output_batch_axis,
  output_feature_axis,
```

```
  output_spatial_axes,
  window_strides,
  padding,
  x_dilation,
  kernel_dilation,
  feature_group_count = 1L,
  batch_group_count = 1L,
  precision = "highest"
)
```

Arguments

`x`	(arrayish) Input, e.g. `[batch, channels, *spatial]`. Can be any data type. `x` and `kernel` must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
`kernel`	(arrayish) Kernel, e.g. `[out_ch, in_ch/groups, *spatial]`. Shares `x`'s data type – see `x`.
`x_batch_axis`, `x_feature_axis`	(integer(1)) Batch and feature axis of `x`.
`x_spatial_axes`	(integer()) Spatial axes of `x`.
`kernel_input_feature_axis`, `kernel_output_feature_axis`	(integer(1)) Input and output feature axis of `kernel`.
`kernel_spatial_axes`	(integer()) Spatial axes of `kernel`.
`output_batch_axis`, `output_feature_axis`	(integer(1)) Batch and feature axis of the output.
`output_spatial_axes`	(integer()) Spatial axes of the output.
`window_strides`	(integer()) Stride per spatial axis.
`padding`	(matrix) `[n_spatial, 2]` of `(low, high)` padding.
`x_dilation`, `kernel_dilation`	(integer()) Input/kernel dilation.
`feature_group_count`, `batch_group_count`	(integer(1)) Grouping.

precision (character(1))
One of "highest", "high", "default".

Value

(arrayish)
Has the data type x and kernel agreed on. Its shape is given by the output axis arguments: the batch axis holds x's batch size divided by batch_group_count, the feature axis kernel's output feature size, and each spatial axis the number of window positions along it.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_convolution(), specified under convolution. The axis numbers are converted on the way down.

See Also

nv_conv1d(), nv_conv2d(), nv_conv3d()

Examples

```
# a 1-D convolution in NCW layout: one batch, one channel, width 5,
# convolved with a width-3 kernel, giving 3 window positions
x <- nv_array(1:5, shape = c(1, 1, 5), dtype = "f32")
kernel <- nv_array(c(1, 0, -1), shape = c(1, 1, 3), dtype = "f32")
prim_convolution(
  x, kernel,
  x_batch_axis = 1L, x_feature_axis = 2L, x_spatial_axes = 3L,
  kernel_output_feature_axis = 1L, kernel_input_feature_axis = 2L,
  kernel_spatial_axes = 3L,
  output_batch_axis = 1L, output_feature_axis = 2L, output_spatial_axes = 3L,
  window_strides = 1L, padding = matrix(0L, nrow = 1, ncol = 2),
  x_dilation = 1L, kernel_dilation = 1L
)
```

---

prim_cos *Primitive Cosine*

Description

Element-wise cosine.

Usage

```
prim_cos(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_cosine(), specified under cosine.

See Also

nv_cos(), cos()

Examples

```
# the input's data type carries through
x <- nv_array(c(0, pi / 2, pi))
prim_cos(x)

# an R value materializes at its default data type
prim_cos(0)
```

---

prim_cosh *Primitive Hyperbolic Cosine*

Description

Element-wise hyperbolic cosine.

Usage

```
prim_cosh(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_cosh(), an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.cosh.

See Also

nv_cosh(), cosh()

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_cosh(x)

# an R value materializes at its default data type
prim_cosh(1)
```

---

prim_cummax *Primitive Cumulative Maximum*

Description

Running maximum of array elements along a single axis along with the index of the last occurrence of the running maximum. At output position j, the values output is max(input[1:j]) and the indices output is the largest i in 1:j with input[i] == values[j] (last-occurrence tiebreak).

Usage

```
prim_cummax(x, axis)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(named list of two arrayish)
Elements values, the running maximum at the input's data type, and indices, the running argmax at the default integer data type (see default_dtypes()). Both have the input's shape.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_reduce_window(), specified under reduce_window. The window is variadic over (values, iota), so the index of the running extremum is carried alongside it.

See Also

nv_cummax()

Examples

```
# `values` keeps the input's data type, `indices` is the default integer
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
prim_cummax(x, axis = 1L)
```

---

prim_cummin *Primitive Cumulative Minimum*

Description

Running minimum of array elements along a single axis along with the index of the last occurrence of the running minimum. At output position j, the values output is min(input[1:j]) and the indices output is the largest i in 1:j with input[i] == values[j] (last-occurrence tiebreak).

Usage

```
prim_cummin(x, axis)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(named list of two arrayish)
Elements values, the running minimum at the input's data type, and indices, the running argmin at the default integer data type (see default_dtypes()). Both have the input's shape.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_reduce_window(), specified under reduce_window. The window is variadic over (values, iota), so the index of the running extremum is carried alongside it.

See Also

nv_cummin()

Examples

```
# `values` keeps the input's data type, `indices` is the default integer
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
prim_cummin(x, axis = 1L)
```

---

prim_cumprod *Primitive Cumulative Product*

Description

Cumulative product of array elements along a single axis. Output position j along axis equals the product of input positions 1:j.

A boolean input is accumulated with a logical AND, so the result is a running AND. nv_cumprod() multiplies zeroes and ones instead.

Usage

```
prim_cumprod(x, axis)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo

StableHLO

Lowers to hlo_reduce_window(), specified under reduce_window. The reducer is hlo_multiply().

See Also

nv_cumprod()

Examples

```
# the accumulation keeps the input's data type
x <- nv_matrix(1:6, nrow = 2)
prim_cumprod(x, axis = 1L)
```

---

prim_cumsum *Primitive Cumulative Sum*

Description

Cumulative sum of array elements along a single axis. Output position j along axis equals the sum of input positions 1:j.

A boolean input is accumulated with a logical OR, so the result is a running OR rather than a running count. nv_cumsum() counts instead.

Usage

```
prim_cumsum(x, axis)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis along which to accumulate. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_reduce_window(), specified under reduce_window. The reducer is hlo_add().

See Also

nv_cumsum()

Examples

```
# the accumulation keeps the input's data type
x <- nv_matrix(1:6, nrow = 2)
prim_cumsum(x, axis = 1L)
```

---

prim_custom_gradient *Primitive Custom Gradient*

Description

Calls fn on args, but differentiates the result with rule instead of through the operations fn performs. Use it where those operations give no gradient – e.g. a prim_while() loop, which gradient() cannot differentiate – or an inaccurate one, while the derivative has a formula of its own.

Usage

```
prim_custom_gradient(args, fn, rule)
```

Arguments

`args` (named `list()`)
The arguments of `fn`: `arrayish` values, or trees of them in the sense of pjrt's `RTree`, by name. `rule` gives the gradient with respect to these. No entry may be named `out`, `grad` or `required`, the names under which `rule` receives the result, its gradient and which gradients are required. An R value materializes at its default data type, as in `prim_while()`.

`fn` (`function`)
Computes the result from the entries of `args`, which it receives as arguments. Returns an array, or a tree of them.

`rule` (`function | anvl_rule_reverse`)
Computes the gradient. A function receives the entries of `args` as arguments, plus `out`, the result of `fn`, `grad`, the gradient with respect to `out`, of the same structure, data types and shapes, and `required`, a named `logical()` with one element per entry of `args`: `TRUE` where the gradient with respect to that entry is required, i.e. where any of its arrays requires one. Returns a named list with the gradient with respect to each required entry – for an array `x`, the sum over the elements of `out` of their `grad` times their derivative with respect to `x` – of `x`'s structure, data types and shapes. An entry that is not required, e.g. an integer, which never is, can be left out or be `NULL`.

Alternatively, a `rule_reverse()` with a `forward` pass, which replaces `fn` where the result is differentiated, so that the backward pass can read what the forward pass computed. Its passes are those of a primitive's rule, except that `inputs` are the entries of `args` by name rather than one array per operand, `params` is an empty list, and `outputs`, `grads`, `required` and the returned gradients are those of a function above.

Value

(`arrayish | list`)
The result of `fn`.

Gradients

`rule` is traced when the gradient is, once the function being differentiated has been traced. It can close over constants – R values and arrays – but not over a traced value: pass that through `args` instead.

`fn` may close over traced values, but `gradient()` cannot differentiate with respect to those, as `rule` only gives the gradient with respect to `args`. Pass such a value through `args` instead.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers the operations of `fn` in place. `rule` is not lowered: a gradient is traced, as anvl operations, before the program is lowered.

See Also

`gradient()`

Examples

```
# x^n by repeated multiplication: a while loop, which gradient() cannot
# differentiate on its own
pow_loop <- function(x, n) {
  prim_while(
    init = list(i = 1L, y = x),
    cond = function(i, y) i < n,
    body = function(i, y) list(i = i + 1L, y = y * x)
  )$y
}
pow <- function(x, n) {
  prim_custom_gradient(
    list(x = x, n = n),
    fn = pow_loop,
    rule = function(x, n, out, grad, required) list(x = grad * n * x^(n - 1L))
  )
}
x <- nv_scalar(2)
pow(x, 3L)
# the derivative 3 * x^2
jit(gradient(pow, wrt = "x"))(x, 3L)
# and, as it is computed from x, the second derivative 6 * x
dpow <- function(x, n) gradient(pow, wrt = "x")(x, n)$x
jit(gradient(dpow, wrt = "x"))(x, 3L)

# With a forward pass, x^(n - 1) is computed once, for the result and for
# its gradient
pow_fwd <- function(x, n) {
  prim_custom_gradient(
    list(x = x, n = n),
    fn = pow_loop,
    rule = rule_reverse(forward = function(inputs, params, required) {
      p <- inputs$x^(inputs$n - 1L)
      list(
        outputs = p * inputs$x,
        backward = function(inputs, outputs, grads, params, required) {
          list(x = grads * inputs$n * p)
        }
      )
    })
  )
}
jit(gradient(pow_fwd, wrt = "x"))(x, 3L)
```

prim_digamma *Primitive Digamma*

Description

Element-wise digamma function (logarithmic derivative of the gamma function).

Usage

```
prim_digamma(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_digamma(), an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.digamma.

See Also

nv_digamma(), digamma()

Examples

```
# the input's data type carries through
x <- nv_array(c(0.5, 1, 2, 5))
prim_digamma(x)

# an R value materializes at its default data type
prim_digamma(2)
```

prim_div *Primitive Division*

Description

Divides two arrays element-wise.

Usage

```
prim_div(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any numeric data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_divide(), specified under divide.

See Also

nv_div(), /

Examples

```
# two R values: both take an R double's default data type
prim_div(10, 4)

# the R value is built at the array's data type instead
prim_div(10, nv_scalar(4, "f64"))
```

`prim_dot_general` *Primitive Dot General*

Description

General dot product of two arrays, supporting contraction over arbitrary axes and batching.

Usage

```
prim_dot_general(
  lhs,
  rhs,
  contracting_axes,
  batching_axes,
  precision = "highest"
)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type whose shapes are constrained by `contracting_axes` and `batching_axes` rather than having to match. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.
`contracting_axes`	(`list(integer(), integer())`) A list of two integer vectors specifying which axes of `lhs` and `rhs` to contract over. The contracted axes must have matching sizes.
`batching_axes`	(`list(integer(), integer())`) A list of two integer vectors specifying which axes of `lhs` and `rhs` are batch axes. These must have matching sizes.
`precision`	(`character(1)`) Controls the trade-off between speed and numerical accuracy of the operation. One of `"highest"` (default), `"high"` or `"default"`. Only the StableHLO backend honors this; it is ignored by the quickr backend.

Value

(`arrayish`)
Has the data type the operands agreed on. The output shape is the batch axes followed by the remaining (non-contracted, non-batched) axes of `lhs`, then `rhs`.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_dot_general(), specified under dot_general.

See Also

nv_matmul(), %*%

Examples

```
# contracting a 2x3 with a 3x2 gives a 2x2 at the operands' data type
x <- nv_matrix(1:6, nrow = 2)
y <- nv_matrix(1:6, nrow = 3)
prim_dot_general(x, y,
  contracting_axes = list(2L, 1L),
  batching_axes = list(integer(0), integer(0))
)
```

---

prim_dynamic_slice *Primitive Dynamic Slice*

Description

Extracts a slice from an array whose start position is determined at runtime via array-valued indices. The slice shape (slice_sizes) is a fixed R integer vector.

Use prim_static_slice() instead when all indices are known at compile time and you need stride support.

Usage

```
prim_dynamic_slice(x, ..., slice_sizes)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
...	(arrayish) Scalar start indices of an integer data type, one per axis of x. They are brought to one data type among themselves, never x's.
slice_sizes	(integer()) Size of the slice in each axis. Must have length equal to naxes(x) and satisfy 1 <= slice_sizes <= shape(x) per axis.

Value

(arrayish)
Has the input's data type and shape slice_sizes.

Out of Bounds Behavior

Start indices are clamped before the slice is extracted: adjusted_start_indices = clamp(1, start_indices, shape(x) - slice_sizes + 1). This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_dynamic_slice(), specified under dynamic_slice.

See Also

prim_static_slice(), prim_dynamic_update_slice(), prim_scatter(), prim_gather(), nv_subset(), [

Examples

```
# 1-D: extract 3 elements starting at position 3
x <- nv_array(1:10)
start <- nv_scalar(3L)
prim_dynamic_slice(x, start, slice_sizes = 3L)

# 2-D: extract a 2x2 block from a matrix
x <- nv_matrix(1:12, nrow = 3, ncol = 4)
row_start <- nv_scalar(2L)
col_start <- nv_scalar(1L)
prim_dynamic_slice(x, row_start, col_start, slice_sizes = c(2L, 2L))
```

---

prim_dynamic_update_slice

Primitive Dynamic Update Slice

Description

Returns a copy of x with a slice replaced by update at a runtime-determined position. This is the write counterpart of prim_dynamic_slice(): dynamic slice reads a block from an array, while dynamic update slice writes a block into an array.

Usage

```
prim_dynamic_update_slice(x, update, ...)
```

Arguments

x	(arrayish) The array to write into. Can be any data type. x and update must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
update	(arrayish) The values to write at the specified position. Must have the same number of axes as x, with shape(update) <= shape(x) per axis. Shares x's data type.
...	(arrayish) Scalar start indices of an integer data type, one per axis of x. They are brought to one data type among themselves, never x's.

Value

(arrayish)
Has the same data type and shape as x.

Out of Bounds Behavior

Start indices are clamped before the update is written: adjusted_start_indices = clamp(1, start_indices, shape(x) - shape(update) + 1). This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_dynamic_update_slice(), specified under dynamic_update_slice.

See Also

prim_dynamic_slice(), prim_scatter(), prim_gather(), nv_subset_assign(), [<-

Examples

```
# 1-D: overwrite two elements starting at position 2
x <- nv_array(1:5)
update <- nv_array(c(10L, 20L))
start <- nv_scalar(2L)
prim_dynamic_update_slice(x, update, start)
```

```
# 2-D: write a 2x2 block into a 3x4 matrix
x <- nv_fill(0L, shape = c(3, 4))
update <- nv_matrix(c(1L, 2L, 3L, 4L), nrow = 2, ncol = 2)
row_start <- nv_scalar(2L)
col_start <- nv_scalar(3L)
prim_dynamic_update_slice(x, update, row_start, col_start)
```

`prim_eigh` *Primitive Symmetric Eigendecomposition*

Description

Computes the eigendecomposition of a symmetric matrix `x` of shape `(n, n)`:

$$A = \text{vectors}\,\text{diag}(\text{values})\,\text{vectors}^\top.$$

Only the lower triangle of `x` is read. The columns of `vectors` are the (orthonormal) eigenvectors and `values` is the length-`n` vector of (real) eigenvalues in ascending order. Output names and order match `base::eigen()`, which unlike this primitive also handles non-symmetric matrices.

Usage

```
prim_eigh(x)
```

Arguments

`x`	(`arrayish`) One input, a symmetric square matrix with exactly 2 axes. Can be any float data type. An R value materializes at its default data type.

Value

(named `list` of two `arrayish`)
Elements `values` (length `n`) and `vectors` (shape `(n, n)`). Both have the input's data type.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `hlo_custom_call()` with target `"eigh"`.

See Also

`nv_eigh()`

Examples

```
# `values` and `vectors` both have the input's data type
x <- nv_array(c(2, 1, 1, 2), shape = c(2, 2), dtype = "f64")
prim_eigh(x)
```

`prim_eq` *Primitive Equal*

Description

Element-wise equality comparison.

Usage

```
prim_eq(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(`arrayish`)
Has the inputs' shape and boolean data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_compare()`, specified under compare. The comparison direction is `EQ`.

See Also

`nv_eq()`, `==`

Examples

```
# two R values: both take an R double's default data type
prim_eq(1, 1)

# the R value is built at the array's data type instead
prim_eq(1, nv_scalar(1, "f64"))
```

---

prim_erf *Primitive Error Function*

Description

Element-wise error function `erf(x) = (2 / sqrt(pi)) * integral_0^x exp(-t^2) dt`.

Usage

```
prim_erf(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_erf()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.erf.

See Also

`nv_erf()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_erf(x)

# an R value materializes at its default data type
prim_erf(1)
```

`prim_erf_inv` *Primitive Inverse Error Function*

Description

Element-wise inverse error function (the inverse of `erf` on `(-1, 1)`).

Usage

```
prim_erf_inv(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_erf_inv()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.erf_inv.

See Also

`nv_erf_inv()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-0.5, 0, 0.5))
prim_erf_inv(x)

# an R value materializes at its default data type
prim_erf_inv(0.5)
```

`prim_erfc` *Primitive Complementary Error Function*

Description

Element-wise complementary error function `erfc(x) = 1 - erf(x)`.

Usage

```
prim_erfc(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_erfc()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.erfc.

See Also

`nv_erfc()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_erfc(x)

# an R value materializes at its default data type
prim_erfc(1)
```

---

prim_exp *Primitive Exponential*

Description

Element-wise exponential.

Usage

```
prim_exp(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_exponential(), specified under exponential.

See Also

nv_exp(), exp()

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 1, 2))
prim_exp(x)

# an R value materializes at its default data type
prim_exp(1)
```

`prim_expm1` *Primitive Exponential Minus One*

Description

Element-wise exp(x) - 1, more accurate for small x.

Usage

```
prim_expm1(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_exponential_minus_one()`, specified under exponential_minus_one.

See Also

`nv_expm1()`

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.001, 1))
prim_expm1(x)

# an R value materializes at its default data type
prim_expm1(0.001)
```

`prim_fill` *Primitive Fill*

Description

Creates an array of a given shape and data type, filled with a scalar value. The advantage of using this function instead of e.g. doing `nv_array(1, shape = c(100, 100))` is that lowering of `prim_fill()` is efficiently represented in the compiled program, while the latter uses 100 * 100 * 4 bytes of memory.

Usage

```
prim_fill(value, shape, dtype, device = NULL)
```

Arguments

`value`	(`numeric(1)`) Scalar value to fill the array with. It has to be something `dtype` can hold: a whole number in its range for an integer data type, a non-negative one for an unsigned integer, and a logical or `0` / `1` for `bool`.
`shape`	(`integer()`) Shape of the output array.
`dtype`	(`character(1)` | `DataType`) Data type of the result.
`device`	(`NULL` | `character(1)` | `device`) The device the data lives on, given either as: • a *device string* naming the platform (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), which is resolved against the backend in use, or • a *device object* as returned by `nv_device()`: a `PJRTDevice` for the `"pjrt"` backend or a `quickr_device` for the `"quickr"` backend. It must belong to the active backend (`active_backend()`); a device of another backend is an error. The default (`NULL`) uses `default_device()`.

Value

(`arrayish`)
Has the given `shape` and `dtype`.

Implemented Rules

- `stablehlo`
- `quickr`

StableHLO

Lowers to `hlo_tensor()`, stablehlo's constant builder, specified under constant.

See Also

`nv_fill()`

Examples

```
# the R double is built at the requested data type
prim_fill(3.14, shape = c(2, 3), dtype = "f32")
```

`prim_floor` *Primitive Floor*

Description

Element-wise floor.

Usage

```
prim_floor(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to hlo_floor(), specified under floor.

See Also

nv_floor(), floor()

Examples

```
# the input's data type carries through
x <- nv_array(c(1.2, 2.7, -1.5))
prim_floor(x)

# an R value materializes at its default data type
prim_floor(1.2)
```

---

prim_gather *Primitive Gather*

Description

Gathers slices from the x array at positions specified by start_indices. Each index vector in start_indices identifies a starting position in x, and a slice of size slice_sizes is extracted from that position. The gathered slices are assembled into the output array.

This is the inverse of prim_scatter(): gather reads slices from a array at given indices, while scatter writes slices into an array at given indices.

Usage

```
prim_gather(
  x,
  start_indices,
  slice_sizes,
  offset_axes,
  collapsed_slice_axes,
  x_batching_axes,
  start_indices_batching_axes,
  start_index_map,
  index_vector_axis,
  indices_are_sorted = FALSE,
  unique_indices = FALSE
)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
start_indices	(arrayish) Array of starting indices, of an integer data type, which it keeps – the indices take no part in x's. Contains index vectors that map to positions in x via start_index_map. The axis specified by index_vector_axis holds the index vectors.
slice_sizes	(integer()) Size of the slice to gather from x in each axis. Must have length equal to naxes(x).
offset_axes	(integer()) Axes in the output that correspond to the non-collapsed slice axes of x.
collapsed_slice_axes	(integer()) Axes of x that are collapsed (removed) from the slice. The corresponding entries in slice_sizes must be 1. Together with offset_axes and x_batching_axes, these must account for all axes of x.
x_batching_axes	(integer()) Axes of x that are batch axes. Use integer(0) when there are no batch axes.
start_indices_batching_axes	(integer()) Axes of start_indices that correspond to batch axes. Must have the same length as x_batching_axes.
start_index_map	(integer()) Maps each component of an index vector in start_indices to an axis of x. For example, start_index_map = 1L means each index vector indexes into the first axis of x. This is prim_scatter()'s scatter_axes_to_x_axes.
index_vector_axis	(integer(1)) Axis of start_indices that contains the index vectors. If set to naxes(start_indices) + 1, each scalar element of start_indices is treated as a length-1 index vector.
indices_are_sorted	(logical(1)) Whether indices are guaranteed to be sorted. Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually sorted. Default FALSE.
unique_indices	(logical(1)) Whether indices are guaranteed to be unique (no duplicates). Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually unique. Default FALSE.

Value

(arrayish)
Has the input's data type. The output shape is composed of the offset axes (from the slice) and the remaining axes from `start_indices`. See the underlying StableHLO function for more details.

Out of Bounds Behavior

Start indices are clamped before the slice is extracted: `clamp(1, start_index, shape(x) - slice_sizes + 1)`. This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_gather()`, specified under gather. The axis and index arguments are intricate; anvl states them in its own terms and converts on the way down.

See Also

`prim_scatter()`, `nv_subset()`, `nv_subset_assign()`, `[`, `[<-`

Examples

```
# gather rows 1 and 3 from a 3x3 matrix
x <- nv_matrix(1:9, nrow = 3)
indices <- nv_matrix(c(1L, 3L), ncol = 1)
prim_gather(
  x, indices,
  slice_sizes = c(1L, 3L),
  offset_axes = 2L,
  collapsed_slice_axes = 1L,
  x_batching_axes = integer(0),
  start_indices_batching_axes = integer(0),
  start_index_map = 1L,
  index_vector_axis = 2L
)
```

prim_ge *Primitive Greater Than or Equal*

Description

Element-wise greater than or equal comparison.

Usage

```
prim_ge(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and boolean data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_compare(), specified under compare. The comparison direction is GE.

See Also

nv_ge(), >=

Examples

```
# two R values: both take an R double's default data type
prim_ge(2, 1)

# the R value is built at the array's data type instead
prim_ge(2, nv_scalar(1, "f64"))
```

prim_gt *Primitive Greater Than*

Description

Element-wise greater than comparison.

Usage

```
prim_gt(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and boolean data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_compare(), specified under compare. The comparison direction is GT.

See Also

nv_gt(), >

Examples

```
# two R values: both take an R double's default data type
prim_gt(2, 1)

# the R value is built at the array's data type instead
prim_gt(2, nv_scalar(1, "f64"))
```

`prim_if` *Primitive If*

Description

Conditional execution of one of two branches based on a scalar boolean predicate, mirroring R's `if` construct: it branches between two *functions* and evaluates only the selected one. Its arguments are named after that construct, where `prim_ifelse()` – which selects element-wise between two *arrays* – is named after `ifelse()`.

Usage

```
prim_if(pred, true, false)
```

Arguments

`pred`	(`arrayish`) Predicate deciding which branch to execute. Must be a scalar of the boolean data type, or an R logical.
`true`, `false`	(`function()`) Zero-argument functions for the true and false branches. Both must return outputs of the same structure, data types and shapes. As with `prim_ifelse()`, whose two values must already agree, nothing is promoted: branches that disagree are an error.

Value

(`arrayish | list`)
Result of the executed branch: an array, or a tree of them in the sense of pjrt's `RTree` – a `list`, nested arbitrarily – with the structure, data types and shapes both branches share.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_if()`, specified under if.

See Also

`nv_if()`, `prim_ifelse()`

Examples

```
# both branches must return the same structure, data types and shapes
prim_if(nv_scalar(TRUE), \() nv_scalar(1), \() nv_scalar(2))
```

`prim_ifelse` *Primitive Ifelse*

Description

Element-wise selection based on a boolean predicate, mirroring R's `ifelse()`. For each element, returns the corresponding element from `yes` where `test` is `TRUE` and from `no` where `test` is `FALSE`.

`prim_if()` is the other conditional: it mirrors R's `if` construct and branches between two *functions*, evaluating only the selected one.

Usage

```
prim_ifelse(test, yes, no)
```

Arguments

`test`	(`arrayish`) Predicate array. Must be a boolean or an R logical, and scalar or the same shape as `yes`.
`yes`, `no`	(`arrayish`) Values to select from, of the same shape. Can be any data type. `yes` and `no` must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.

Value

(`arrayish`)
Has the shape of `yes` and `no`, and the data type they agreed on.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_select()`, specified under select.

See Also

nv_ifelse(), prim_if()

Examples

```
# the result takes the branches' data type; `test` only selects
test <- nv_array(c(TRUE, FALSE, TRUE))
prim_ifelse(test, nv_array(c(1, 2, 3)), nv_array(c(4, 5, 6)))
```

---

prim_iota *Primitive Iota*

Description

Creates an array with values increasing along the specified axis.

Usage

```
prim_iota(axis, shape, dtype, start = 1L, device = NULL)
```

Arguments

axis	(integer(1)) Axis along which values increase. Negative values count from the end of shape, i.e. -1 refers to the last axis.
shape	(integer()) Shape of the output array.
dtype	(character(1) | DataType) Data type of the result. Can be any numeric data type, boolean being the one exception.
start	(integer(1)) Starting value.
device	(NULL | character(1) | device) The device the data lives on, given either as: • a *device string* naming the platform (e.g. "cpu", "cuda", "cuda:<n>"), which is resolved against the backend in use, or • a *device object* as returned by nv_device(): a PJRTDevice for the "pjrt" backend or a quickr_device for the "quickr" backend. It must belong to the active backend (active_backend()); a device of another backend is an error. The default (NULL) uses default_device().

Value

(arrayish)
Has the given dtype and shape.

Implemented Rules

- stablehlo
- quickr

StableHLO

Lowers to hlo_iota(), specified under iota.

See Also

nv_iota()

Examples

```
# the sequence is built at the requested data type
prim_iota(axis = 1L, shape = 5L, dtype = "i32")
```

---

prim_is_finite *Primitive Is Finite*

Description

Element-wise check if values are finite (not Inf, -Inf, or NaN).

Usage

```
prim_is_finite(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the same shape as the input and boolean data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_is_finite(), specified under is_finite.

See Also

nv_is_finite()

Examples

```
# the result is boolean, whatever float data type the input has
x <- nv_array(c(1, Inf, NaN, -Inf, 0))
prim_is_finite(x)

# an R value materializes at its default data type before the test
prim_is_finite(1)
```

---

prim_le *Primitive Less Than or Equal*

Description

Element-wise less than or equal comparison.

Usage

```
prim_le(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and boolean data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_compare(), specified under compare. The comparison direction is LE.

See Also

nv_le(), <=

Examples

```
# two R values: both take an R double's default data type
prim_le(1, 2)

# the R value is built at the array's data type instead
prim_le(1, nv_scalar(2, "f64"))
```

---

prim_lgamma *Primitive Log-Gamma*

Description

Element-wise natural logarithm of the absolute value of the gamma function.

Usage

```
prim_lgamma(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_lgamma(), an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.lgamma.

See Also

nv_lgamma(), lgamma()

Examples

```
# the input's data type carries through
x <- nv_array(c(0.5, 1, 2, 5))
prim_lgamma(x)

# an R value materializes at its default data type
prim_lgamma(2)
```

`prim_log` *Primitive Logarithm*

Description

Element-wise natural logarithm.

Usage

```
prim_log(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_log(), specified under log.

See Also

nv_log(), log()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 2.718, 7.389))
prim_log(x)

# an R value materializes at its default data type
prim_log(2)
```

`prim_log1p` *Primitive Log Plus One*

Description

Element-wise log(1 + x), more accurate for small x.

Usage

```
prim_log1p(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_log_plus_one()`, specified under log_plus_one.

See Also

`nv_log1p()`

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.001, 1))
prim_log1p(x)

# an R value materializes at its default data type
prim_log1p(0.001)
```

`prim_lt` *Primitive Less Than*

Description

Element-wise less than comparison.

Usage

```
prim_lt(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(`arrayish`)
Has the inputs' shape and boolean data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_compare()`, specified under compare. The comparison direction is `LT`.

See Also

`nv_lt()`, `<`

Examples

```
# two R values: both take an R double's default data type
prim_lt(1, 2)

# the R value is built at the array's data type instead
prim_lt(1, nv_scalar(2, "f64"))
```

`prim_lu` *Primitive LU Decomposition*

Description

Computes the partial-pivoted LU decomposition of a matrix `x`:

$$PA = LU,$$

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. `L` (with implicit unit diagonal) and `U` are packed into a single `LU` output matching LAPACK's `getrf` layout. P is returned in two equivalent forms: `pivots` (LAPACK's sequential row-swap encoding) and `permutation` (an explicit permutation vector).

Usage

```
prim_lu(x)
```

Arguments

`x`	(`arrayish`) One input, with exactly 2 axes. Can be any float data type. An R value materializes at its default data type.

Value

(named `list` of three `arrayish`)
Elements `LU (m, n)` with the input's data type; `pivots (k,)` at the default integer data type (see `default_dtypes()`) with `k = min(m, n)` (sequential row swaps: row `i` was exchanged with row `pivots[i]` during elimination step `i`); and `permutation (m,)` at that same data type, a permutation vector for P such that `(P %*% A)[i, ]` equals `A[permutation[i], ]`.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to a `"lu"` `hlo_custom_call()` (backed by LAPACK on CPU and cuSOLVER on CUDA) for `LU` and `pivots`, followed by a `hlo_while()` loop that converts `pivots` to `permutation` in-graph.

See Also

nv_lu()

Examples

```
# `LU` keeps the input's data type; the pivots are the default integer
x <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
prim_lu(x)
```

---

prim_max *Primitive Max Reduction*

Description

Finds the maximum of array elements along the specified axes.

Usage

```
prim_max(x, axes, drop = TRUE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer()) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reduce(), specified under reduce. The reducer is hlo_maximum().

See Also

nv_max()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# reducing axis 1 removes it from the shape, and the data type is kept
prim_max(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_max(x, axes = 1L, drop = FALSE)

# reducing every axis gives a scalar
prim_max(x, axes = c(1L, 2L))
```

---

prim_min *Primitive Min Reduction*

Description

Finds the minimum of array elements along the specified axes.

Usage

```
prim_min(x, axes, drop = TRUE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer()) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_reduce()`, specified under reduce. The reducer is `hlo_minimum()`.

See Also

`nv_min()`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# reducing axis 1 removes it from the shape, and the data type is kept
prim_min(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_min(x, axes = 1L, drop = FALSE)

# reducing every axis gives a scalar
prim_min(x, axes = c(1L, 2L))
```

`prim_mul` *Primitive Multiplication*

Description

Multiplies two arrays element-wise.

Usage

```
prim_mul(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_multiply(), specified under multiply.

See Also

nv_mul(), *

Examples

```
# two R values: both take an R double's default data type
prim_mul(2, 3)

# the R value is built at the array's data type instead
prim_mul(2, nv_scalar(3, "f64"))
```

---

prim_ne *Primitive Not Equal*

Description

Element-wise inequality comparison.

Usage

```
prim_ne(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and boolean data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_compare(), specified under compare. The comparison direction is NE.

See Also

nv_ne(), !=

Examples

```
# two R values: both take an R double's default data type
prim_ne(1, 2)

# the R value is built at the array's data type instead
prim_ne(1, nv_scalar(2, "f64"))
```

---

prim_negate *Primitive Negation*

Description

Negates an array element-wise.

Usage

```
prim_negate(x)
```

Arguments

x	(arrayish) One input. Can be any numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_negate(), specified under negate.

See Also

nv_negate(), unary -

Examples

```
# the input's data type carries through
x <- nv_array(c(1, -2, 3))
prim_negate(x)

# an R value materializes at its default data type
prim_negate(1)
```

---

prim_not *Primitive Bitwise Not*

Description

Element-wise bitwise NOT – a logical NOT on a boolean input, and a bit-by-bit one on an integer.

Usage

```
prim_not(x)
```

Arguments

x	(arrayish) One input. Can be any integerish data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_not(), specified under not.

See Also

nv_not()

Examples

```
# on a boolean this is a logical negation
x <- nv_array(c(TRUE, FALSE, TRUE))
prim_not(x)

# on an integer it complements every bit, so `12L` becomes `-13`
prim_not(nv_array(12L))
```

---

prim_or *Primitive Bitwise Or*

Description

Element-wise bitwise OR – a logical OR on a boolean input, and a bit-by-bit one on an integer.

Usage

```
prim_or(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any integerish data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_or(), specified under or.

See Also

nv_or(), |

Examples

```
# two R values: both take an R integer's default data type
prim_or(12L, 10L)

# the R value is built at the array's data type instead
prim_or(12L, nv_scalar(10L, "i64"))
```

---

prim_pad *Primitive Pad*

Description

Pads an array with a given padding value.

Usage

```
prim_pad(
  x,
  padding_value,
  edge_padding_low,
  edge_padding_high,
  interior_padding
)
```

Arguments

x	(arrayish) The array to pad. Can be any data type. x and padding_value must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.

padding_value (arrayish)
: Scalar value to use for padding. Shares x's data type.

edge_padding_low
: (integer())
Amount of padding to add at the start of each axis.

edge_padding_high
: (integer())
Amount of padding to add at the end of each axis.

interior_padding
: (integer())
Amount of padding to add between elements in each axis.

Value

(arrayish)
Has the data type the operands agreed on. Each axis grows by edge_padding_low + edge_padding_high, plus interior_padding between every pair of elements; negative edge padding trims (see hlo_pad()).

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_pad(), specified under pad.

See Also

nv_pad()

Examples

```
x <- nv_array(1:3)
# one element before, two after
prim_pad(x, 0L, edge_padding_low = 1L, edge_padding_high = 2L, interior_padding = 0L)

# interior padding goes between the elements
prim_pad(x, 0L, 0L, 0L, 1L)

# negative edge padding trims instead
prim_pad(x, 0L, -1L, 0L, 0L)

# one padding amount per axis
prim_pad(nv_matrix(1:4, nrow = 2), 0L, c(1L, 0L), c(0L, 1L), c(0L, 0L))

# the R padding value is built at x's data type
prim_pad(nv_array(c(1.5, 2.5), dtype = "f64"), 0, 1L, 1L, 0L)
```

prim_plogis *Primitive Logistic (Sigmoid)*

Description

Element-wise logistic sigmoid: 1 / (1 + exp(-x)), i.e. `stats::plogis()` with the default location and scale.

Usage

```
prim_plogis(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_logistic()`, specified under logistic.

See Also

`nv_plogis()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-2, 0, 2))
prim_plogis(x)

# an R value materializes at its default data type
prim_plogis(2)
```

`prim_pmax` *Primitive Parallel Maximum*

Description

Element-wise maximum of two arrays, like `base::pmax()`.

Usage

```
prim_pmax(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_maximum()`, specified under maximum.

See Also

`nv_pmax()`

Examples

```
# two R values: both take an R double's default data type
prim_pmax(1, 5)

# the R value is built at the array's data type instead
prim_pmax(1, nv_scalar(5, "f64"))
```

prim_pmin *Primitive Parallel Minimum*

Description

Element-wise minimum of two arrays, like base::pmin().

Usage

```
prim_pmin(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Two inputs of the same data type and shape. Can be any data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_minimum(), specified under minimum.

See Also

nv_pmin()

Examples

```
# two R values: both take an R double's default data type
prim_pmin(1, 5)

# the R value is built at the array's data type instead
prim_pmin(1, nv_scalar(5, "f64"))
```

`prim_popcnt` *Primitive Population Count*

Description

Element-wise population count (number of set bits).

Usage

```
prim_popcnt(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any integer data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_popcnt()`, specified under popcnt.

See Also

`nv_popcnt()`

Examples

```
# the set bits are counted, at the input's own integer data type
x <- nv_array(c(7L, 3L, 15L))
prim_popcnt(x)
```

prim_pow *Primitive Power*

Description

Raises x to the power of y element-wise.

Usage

```
prim_pow(x, y)
```

Arguments

x, y	(arrayish) Two inputs of the same data type and shape. Can be any numeric data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_power(), specified under power.

See Also

nv_pow(), ^

Examples

```
# two R values: both take an R double's default data type
prim_pow(2, 3)

# the R value is built at the array's data type instead
prim_pow(2, nv_scalar(3, "f64"))
```

`prim_print` *Primitive Print*

Description

Prints an array value to the console during execution and returns the input unchanged. This is useful for debugging JIT-compiled code. Bare R inputs print at their category's default data type.

Usage

```
prim_print(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.

Value

(`arrayish`)
Returns the input unchanged.

Order of prints

A print is currently not ordered with respect to other prints. Due to compiler reordering it is possible that the order of the prints differs from their location in the code.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_custom_call()`.

See Also

`nv_print()`

Examples

```
# the value is printed and handed back unchanged
x <- nv_array(c(1, 2, 3))
prim_print(x)
# bare R inputs are printed at their category's default data type
prim_print(1L)
```

prim_prod *Primitive Product Reduction*

Description

Multiplies array elements along the specified axes.

A boolean input is reduced with a logical AND, so the result is a boolean. nv_prod() multiplies zeroes and ones instead.

Usage

```
prim_prod(x, axes, drop = TRUE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer()) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reduce(), specified under reduce. The reducer is hlo_multiply().

See Also

nv_prod()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# reducing axis 1 removes it from the shape, and the data type is kept
prim_prod(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_prod(x, axes = 1L, drop = FALSE)

# reducing every axis gives a scalar
prim_prod(x, axes = c(1L, 2L))
```

`prim_psigamma` *Primitive Psigamma*

Description

Element-wise psigamma function: the `deriv`-th derivative of the digamma function, i.e. the `(deriv + 1)`-th derivative of the log-gamma function. Both `x` and `deriv` must have the same shape; `deriv` typically holds non-negative integer values.

Usage

```
prim_psigamma(x, deriv)
```

Arguments

`x`, `deriv`	(`arrayish`) Two inputs of the same shape. Can be any float data type. `x` and `deriv` must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_polygamma()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.polygamma.

See Also

nv_psigamma()

Examples

```
# both operands are floats, as the primitive requires
x <- nv_array(c(0.5, 1, 2))
deriv <- nv_array(c(1, 1, 2))
prim_psigamma(x, deriv)
```

---

prim_qr *Primitive QR Decomposition*

Description

Computes the reduced QR decomposition of a matrix x:

$$A = QR,$$

where Q has orthonormal columns ($Q^\top Q = I$) and R is upper triangular. For an $m \times n$ input with $k = \min(m, n)$, Q has shape $m \times k$ and R has shape $k \times n$.

Usage

```
prim_qr(x)
```

Arguments

x	(arrayish) One input, with exactly 2 axes. Can be any float data type. An R value materializes at its default data type.

Value

(named list of two arrayish)
Elements Q (shape (m, k)) and R (shape (k, n)), where (m, n) = shape(x) and k = min(m, n). Both have the input's data type.

Implemented Rules

- stablehlo

StableHLO

Lowers to a "geqrf" + "orgqr" hlo_custom_call() pair (backed by LAPACK on CPU and cuSOLVER on CUDA) + postprocessing.

See Also

nv_qr()

Examples

```
# `Q` is 3x2 and `R` 2x2, both at the input's data type
x <- nv_array(1:6, shape = c(3, 2), dtype = "f32")
prim_qr(x)
```

---

prim_reduce *Primitive Generic Reduce*

Description

Reduces an array along the specified axes using a user-supplied associative reducer. `reducer` and `init` must satisfy the constraints in the "Associativity Requirement" section below.

Usage

```
prim_reduce(x, init, axes, reducer, drop = TRUE)
```

Arguments

`x`	(arrayish) The array to reduce. Can be any data type. `x` and `init` must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
`init`	(arrayish) Scalar initial value, with no axes. Must be the neutral element w.r.t. `reducer`, and shares `x`'s data type – see `x`.
`axes`	(`integer()`) Axes to reduce over. Negative values count from the end, i.e. `-1` refers to the last axis.
`reducer`	(`function(lhs, rhs)`) Binary reducer producing a scalar of the same data type as `x`. Its two arguments are passed by position, so they may carry any names. Must be associative (see "Associativity Requirement").
`drop`	(`logical(1)`) If `TRUE` (default) the reduced axes are removed; if `FALSE` they are kept with size 1.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (`drop = TRUE`) or set to 1 (`drop = FALSE`).

Associativity Requirement

The order in which `reducer` is applied across the reduction window is implementation-defined. If the reducer is not associative, the result is ill-defined. Furthermore, `init` must be the neutral element for this reducer. Because arithmetic in a float data type is non-associative, the output of the reduction can differ between backends (GPU, CPU), even if the underlying mathematical function (like `+`) is associative.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `hlo_reduce()`, specified under reduce. The body is `reducer`.

See Also

`prim_sum()`, `prim_max()`

Examples

```
# `init` shares `x`'s data type, and the reduced axis disappears
x <- nv_array(c(1, 2, 3, 4))
prim_reduce(x, init = nv_scalar(0), axes = 1L, reducer = prim_add)
prim_reduce(x, init = nv_scalar(1), axes = 1L, reducer = prim_mul)
```

`prim_remainder` *Primitive Remainder*

Description

Element-wise remainder. The result has the sign of the dividend, which is what StableHLO's `remainder` does. Base R's `%%` takes the sign of the divisor instead and is available via `nv_mod()`.

Usage

```
prim_remainder(x, y)
```

Arguments

`x`, `y`	(`arrayish`) Two inputs of the same data type and shape. Can be any numeric data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_remainder(), specified under remainder.

See Also

nv_remainder()

Examples

```
# two R values: both take an R double's default data type
prim_remainder(1, -3)

# the R value is built at the array's data type instead
prim_remainder(1, nv_scalar(-3, "f64"))

# the sign follows the dividend, where base R's %% follows the divisor
1 %% -3
```

---

prim_reshape *Primitive Reshape*

Description

Reshapes an array to a new shape without changing the underlying data. The elements keep their column-major order, exactly as base R's dim<- does.

Usage

```
prim_reshape(x, shape)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
shape	(integer()) Target shape. Must have the same number of elements as x. At most one entry may be -1, in which case its extent is inferred from the remaining entries and the number of elements of x.

Value

(arrayish)
Has the same data type as the input and the given shape.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reshape(), specified under reshape. The lowering wraps it in two hlo_transpose() calls that reverse every axis, which is what turns stablehlo's row-major reshape into a column-major one; each is skipped where that side has at most one axis.

See Also

nv_reshape()

Examples

```
# the elements keep their column-major order; the data type is untouched
x <- nv_array(1:6)
prim_reshape(x, shape = c(2, 3))
```

---

prim_rev *Primitive Reverse*

Description

Reverses the order of elements along specified axes.

Usage

```
prim_rev(x, axes)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer()) Axes to reverse. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(arrayish)
Has the input's data type and shape.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reverse(), specified under reverse.

See Also

nv_rev()

Examples

```
# the order along axis 1 is flipped
x <- nv_array(c(1, 2, 3, 4, 5))
prim_rev(x, axes = 1L)
```

---

prim_round *Primitive Round*

Description

Rounds the elements of an array to the nearest integer.

Usage

```
prim_round(x, method = "nearest_even")
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.
method	(character(1)) Rounding method. "nearest_even" (default) rounds to the nearest even integer on a tie, "afz" rounds away from zero on a tie.

Value

(arrayish)
Has x's data type and shape.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_round_nearest_even(), specified under round_nearest_even. With method = "afz" it lowers to hlo_round_nearest_afz() instead, specified under round_nearest_afz.

See Also

nv_round()

Examples

```
# the input's data type carries through
x <- nv_array(c(1.4, 2.5, 3.6))
prim_round(x)

# an R value materializes at its default data type
prim_round(2.5)
```

---

prim_rsqrt *Primitive Reciprocal Square Root*

Description

Element-wise reciprocal square root.

Usage

```
prim_rsqrt(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_rsqrt(), specified under rsqrt.

See Also

nv_rsqrt()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 4, 9))
prim_rsqrt(x)

# an R value materializes at its default data type
prim_rsqrt(4)
```

---

prim_sample_bits *Primitive Random Bits*

Description

Draws random bits from the active random state (see nv_set_seed()) with the ThreeFry-2x32 counter-based generator of Salmon et al. (2011), and advances that random state by one step.

See the Random Number Generation article for more details.

Usage

```
prim_sample_bits(shape, dtype)
```

Arguments

shape	(integer()) Shape of the result.
dtype	(character(1) | DataType) An unsigned integer data type: "ui8", "ui16", "ui32" or "ui64".

Value

(arrayish)
The random bits, of data type dtype and shape shape. Every value is made from an encrypted 64-bit block of its own: a "ui64" value is the block, and a narrower value the xor of the block's two 32-bit words, truncated to its low bits, as JAX's jax.random.bits() does by default. The value at a position thus does not depend on the number of values drawn, so a draw of 3 is the start of a draw of 6. The draws fill shape in row-major order, as JAX's do, so a draw of c(2, 3) is not a draw of 6 filled into c(2, 3) in R's column-major order.

Random State

The draws come from the active random state: the global random state (see nv_set_seed()), or the random state that with_nv_rng_state() or local_nv_rng_state() make the default for a scope. They advance it in place.

Implemented Rules

- stablehlo

References

Salmon J, Moraes M, Dror R, Shaw D (2011). "Parallel Random Numbers: As Easy as 1, 2, 3." In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. doi:10.1145/2063384.2063405.

Claessen K, Pałka M (2013). "Splittable Pseudorandom Number Generators using Cryptographic Hashing." In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, 47–58. doi:10.1145/2503778.2503784.

Bradbury J, Frostig R, Hawkins P, Johnson M, Katariya Y, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.11.2, http://github.com/jax-ml/jax.

See Also

nv_runif(), nv_rnorm()

Examples

```
nv_get_rng_state()
prim_sample_bits(2L, "ui32")
# The counter is incremented by 1
nv_get_rng_state()
# A random state of its own is drawn from within with_nv_rng_state():
custom_rng <- nv_rng_state(1L)
custom_rng
with_nv_rng_state(custom_rng, prim_sample_bits(2L, "ui32"))
# And it is still updated
custom_rng
# And does not affect the global RNG
nv_get_rng_state()
```

prim_scan *Primitive Scan*

Description

Runs body a fixed number of times, threading a carry through the steps and stacking each step's outputs along a new leading axis. Step t receives the carry and, for every array in xs, its slice at position t along axis 1 with that axis dropped.

Usage

```
prim_scan(init, xs, body, steps, reverse = FALSE)
```

Arguments

init	(list()) Initial carry: a (possibly nested) list of arrays. Every leaf keeps its shape and data type across steps. It must not contain a random state: to draw from one in body, make it the default before the loop with local_nv_rng_state().
xs	(list()) Per-step inputs: a (possibly nested) list of arrays sliced along axis 1, all of size steps along it. An empty list runs a counted loop. It must not contain a random state.
body	(function) Step function function(carry, x) returning list(carry = , out =), where carry has the structure of init and out is a (possibly nested) list of arrays or NULL. x is NULL when xs is empty.
steps	(integer(1)) Static trip count; the size of axis 1 of every array in xs. 0 runs no step and returns init with zero-length stacked outputs.
reverse	(logical(1)) If TRUE, steps run from steps down to 1; each step still reads xs at its own position and writes its output there.

Value

list(carry = , out =): the final carry and the stacked outputs, each leaf of out gaining a leading axis of size steps.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to `hlo_while()` over a counter, the carry, the output buffers and `xs`, with `hlo_dynamic_slice()` reading each step's inputs and `hlo_dynamic_update_slice()` writing its outputs.

See Also

`nv_scan()`, `prim_while()`

Examples

```
prim_scan(
  init = list(s = nv_scalar(0)),
  xs = list(x = nv_array(c(1, 2, 3))),
  body = function(carry, x) {
    s <- carry$s + x$x
    list(carry = list(s = s), out = s)
  },
  steps = 3L
)
```

---

prim_scatter *Primitive Scatter*

Description

Produces a result array identical to `x` except that slices at positions specified by `scatter_indices` are updated with values from the `update` array. When multiple indices point to the same location, the `update_fn` function determines how to combine the values (by default the new value replaces the old one).

This is the inverse of `prim_gather()`: gather reads slices from an array at given indices, while scatter writes slices into an array at given indices.

Usage

```
prim_scatter(
  x,
  scatter_indices,
  update,
  update_window_axes,
  inserted_window_axes,
  x_batching_axes,
  scatter_indices_batching_axes,
  scatter_axes_to_x_axes,
  index_vector_axis,
  indices_are_sorted = FALSE,
  unique_indices = FALSE,
  update_fn = "replace"
)
```

Arguments

x (arrayish)
The base array to scatter into. Can be any data type. x and update must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.

scatter_indices (arrayish)
Array of indices, of an integer data type, which it keeps – the indices take no part in x's. Contains index vectors that map to positions in x via scatter_axes_to_x_axes. The axis specified by index_vector_axis holds the index vectors.

update (arrayish)
Update values array. Shares x's data type – see x.

update_window_axes (integer())
Axes of update that are window axes, i.e. they correspond to the slice being written into x.

inserted_window_axes (integer())
Axes of x whose slices have size 1 and are inserted (not present) in the update window. Together with update_window_axes and x_batching_axes, these must account for all axes of x.

x_batching_axes (integer())
Axes of x that are batch axes. Use integer(0) when there are no batch axes.

scatter_indices_batching_axes (integer())
Axes of scatter_indices that correspond to batch axes. Must have the same length as x_batching_axes.

scatter_axes_to_x_axes (integer())
Maps each component of an index vector in scatter_indices to an axis of x. For example, scatter_axes_to_x_axes = 1L means each index vector indexes into the first axis of x. This is prim_gather()'s start_index_map.

index_vector_axis (integer(1))
Axis of scatter_indices that contains the index vectors. If set to naxes(scatter_indices) + 1, each scalar element of scatter_indices is treated as a length-1 index vector.

indices_are_sorted (logical(1))
Whether indices are guaranteed to be sorted. Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually sorted. Default FALSE.

unique_indices (logical(1))
Whether indices are guaranteed to be unique (no duplicates). Setting to TRUE

may improve performance but produces undefined behavior if the indices are not actually unique. Default FALSE.

update_fn (character(1) | function)
How an update combines with the value in x it lands on. One of

- "replace" (the default): the update replaces the value,
- "add", "sub", "mul": the value plus, minus or times the update,
- "max", "min": the larger or the smaller of the two,

or a binary function f(old, new) of two scalars at x's data type, which returns one. Only a named update_fn can be differentiated, see section Gradients.

Value

(arrayish)
Has the same data type and shape as x.

Out of Bounds Behavior

If a computed result index falls outside the bounds of x, the update for that index is silently ignored.

Update Order

When multiple indices in scatter_indices map to the same element of x, update_fn combines each of their updates with it in turn, so "add" adds them all up. The order is implementation-defined and may vary between plugins ("cpu", "cuda"), which matters for "replace", where one of them is the one that ends up written.

Gradients

gradient() differentiates a scatter whose update_fn is one of the names, but not a function, even one that computes the same.

- With "replace", an update gets the gradient only where it is the one that was written.
- With "mul", the gradient with respect to update needs unique_indices = TRUE.
- With "max" and "min", x and the updates that tie for the result share its gradient equally.

An update that falls outside of x, and is therefore ignored, gets a gradient of zero.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_scatter(), specified under scatter. The axis and index arguments are intricate; anvl states them in its own terms and converts on the way down.

See Also

prim_gather(), nv_subset(), nv_subset_assign(), [, [<-

Examples

```
# scatter values 10 and 30 into positions 1 and 3 of a zero vector
x <- nv_array(c(0, 0, 0, 0, 0))
indices <- nv_matrix(c(1L, 3L), ncol = 1)
updates <- nv_array(c(10, 30))
prim_scatter(
  x, indices, updates,
  update_window_axes = integer(0),
  inserted_window_axes = 1L,
  x_batching_axes = integer(0),
  scatter_indices_batching_axes = integer(0),
  scatter_axes_to_x_axes = 1L,
  index_vector_axis = 2L
)
```

---

prim_shift_left *Primitive Shift Left*

Description

Element-wise left bit shift.

Usage

```
prim_shift_left(x, shift)
```

Arguments

x	(arrayish) The array whose bits are shifted. Can be any integer data type.
shift	(arrayish) By how many bits to shift each element of x. Has the same data type and shape as x. An R value takes the other operand's data type, and its default data type when neither operand has one.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_shift_left(), specified under shift_left.

See Also

nv_shift_left()

Examples

```
# two R values: both take an R integer's default data type
prim_shift_left(8L, 2L)

# the R value is built at the array's data type instead
prim_shift_left(8L, nv_scalar(2L, "i64"))
```

---

prim_shift_right_arithmetic

Primitive Arithmetic Shift Right

Description

Element-wise arithmetic right bit shift.

Usage

```
prim_shift_right_arithmetic(x, shift)
```

Arguments

x	(arrayish) The array whose bits are shifted. Can be any integer data type.
shift	(arrayish) By how many bits to shift each element of x. Has the same data type and shape as x. An R value takes the other operand's data type, and its default data type when neither operand has one.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to `hlo_shift_right_arithmetic()`, specified under shift_right_arithmetic.

See Also

`nv_shift_right_arithmetic()`

Examples

```
# two R values: both take an R integer's default data type
prim_shift_right_arithmetic(-32L, 2L)

# the R value is built at the array's data type instead
prim_shift_right_arithmetic(-32L, nv_scalar(2L, "i64"))
```

`prim_shift_right_logical` *Primitive Logical Shift Right*

Description

Element-wise logical right bit shift.

Usage

```
prim_shift_right_logical(x, shift)
```

Arguments

`x`	(`arrayish`) The array whose bits are shifted. Can be any integer data type.
`shift`	(`arrayish`) By how many bits to shift each element of `x`. Has the same data type and shape as `x`. An R value takes the other operand's data type, and its default data type when neither operand has one.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to hlo_shift_right_logical(), specified under shift_right_logical.

See Also

nv_shift_right_logical()

Examples

```
# two R values: both take an R integer's default data type
prim_shift_right_logical(32L, 2L)

# the R value is built at the array's data type instead
prim_shift_right_logical(32L, nv_scalar(2L, "i64"))
```

---

prim_sign *Primitive Sign*

Description

Element-wise sign.

Usage

```
prim_sign(x)
```

Arguments

x	(arrayish) One input. Can be any signed numeric data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_sign(), specified under sign.

See Also

nv_sign(), sign()

Examples

```
# the input's data type carries through
x <- nv_array(c(-3, 0, 5))
prim_sign(x)

# an R value materializes at its default data type
prim_sign(-3)
```

---

prim_sin *Primitive Sine*

Description

Element-wise sine.

Usage

```
prim_sin(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_sine(), specified under sine.

See Also

nv_sin(), sin()

Examples

```
# the input's data type carries through
x <- nv_array(c(0, pi / 2, pi))
prim_sin(x)

# an R value materializes at its default data type
prim_sin(0)
```

`prim_sinh` *Primitive Hyperbolic Sine*

Description

Element-wise hyperbolic sine.

Usage

```
prim_sinh(x)
```

Arguments

`x`	(`arrayish`) One input. Can be any float data type. An R value materializes at its default data type.

Value

(`arrayish`)
Has the input's shape and data type.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_sinh()`, an op of the CHLO dialect, a higher-level companion to StableHLO that is lowered to it during compilation. See chlo.sinh.

See Also

`nv_sinh()`, `sinh()`

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_sinh(x)

# an R value materializes at its default data type
prim_sinh(1)
```

`prim_sort` *Primitive Sort*

Description

Sorts arrays along the given axis.

Sorting is determined by the *first array* only: it is the sort key, and any additional arrays are reordered with the same permutation that sorts the first. This enables idioms like *argsort* (sort `x` paired with an `iota` and read off the second output) and key-value sorts (sort `keys` paired with `values`).

1-D slices along `axis` are sorted independently; other axes are preserved.

Usage

```
prim_sort(xs, axis, decreasing = FALSE, stable = FALSE)
```

Arguments

`xs`	(`list` of `arrayish`) One or more arrays to sort. The first is the sort key; the rest are carried along under the same permutation. All must share the same shape.
`axis`	(`integer(1)`) Axis along which to sort. Negative values count from the end, i.e. `-1` refers to the last axis.
`decreasing`	(`logical(1)`) If `TRUE`, sort the key in decreasing order (largest first). Default `FALSE`. Additional arrays are reordered by the same permutation regardless.
`stable`	(`logical(1)`) If `TRUE`, the sort is stable: the relative order of equal *keys* is preserved. Default `FALSE`.

Value

(`list` of `arrayish`)
One sorted output per element of `xs`, in the same order, each with the shape and data type of its input.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_sort(), specified under sort. The comparator uses hlo_compare() (LT for ascending, GT for descending) on the first array. For float keys it uses compare_type = "TOTALORDER" and canonicalizes -0/+0 and -NaN/+NaN to their positive form before comparing, so all NaN values land at one end of the result regardless of sign. Integer keys use SIGNED / UNSIGNED as appropriate.

See Also

nv_sort(), nv_order(), nv_top_k(), nv_median()

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5))
prim_sort(list(x), axis = 1L)[[1L]]

# sort indices by the values (argsort): pair x with iota and read off
# the second result
idx <- nv_iota(axis = 1L, shape = 5L, dtype = "i64")
out <- prim_sort(list(x, idx), axis = 1L)
out[[1L]] # sorted x
out[[2L]] # permutation indices
```

---

prim_sqrt *Primitive Square Root*

Description

Element-wise square root.

Usage

```
prim_sqrt(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_sqrt(), specified under sqrt.

See Also

nv_sqrt(), sqrt()

Examples

```
# the input's data type carries through
x <- nv_array(c(1, 4, 9))
prim_sqrt(x)

# an R value materializes at its default data type
prim_sqrt(4)
```

---

prim_static_slice *Primitive Static Slice*

Description

Extracts a slice from an array using static (compile-time) indices. All indices and strides are fixed R integers, and both ends are inclusive.

Use prim_dynamic_slice() instead when the start position must be computed at runtime (e.g. depends on array values).

Usage

```
prim_static_slice(x, start_indices, end_indices, strides)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
start_indices	(integer()) Start indices (inclusive), one per axis. Must satisfy 1 <= start_indices <= end_indices + 1 per axis, where start_indices == end_indices + 1 selects an empty axis.

`end_indices` (integer())
End indices (inclusive), one per axis. Must satisfy `end_indices <= shape(x)` per axis. Unlike StableHLO's exclusive `limit_indices`, the element at `end_indices` is part of the slice.

`strides` (integer())
Step sizes, one per axis. Must be `>= 1`. A stride of 1 selects every element; a stride of 2 selects every other element, etc.

Value

(arrayish)
Has the same data type as the input and shape `ceiling((end_indices - start_indices + 1) / strides)`.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_slice()`, specified under slice.

See Also

`prim_dynamic_slice()`, `prim_scatter()`, `prim_gather()`, `nv_subset()`, `[`

Examples

```
# 1-D: extract elements 2 through 5, the end being inclusive
x <- nv_array(1:10)
prim_static_slice(x, start_indices = 2L, end_indices = 5L, strides = 1L)

# 1-D: every other element using strides
x <- nv_array(1:10)
prim_static_slice(x, start_indices = 1L, end_indices = 10L, strides = 2L)

# 2-D: extract the submatrix of rows 1-3 and columns 2-4
x <- nv_matrix(1:12, nrow = 3, ncol = 4)
prim_static_slice(x,
  start_indices = c(1L, 2L),
  end_indices = c(3L, 4L),
  strides = c(1L, 1L)
)
```

`prim_sub` *Primitive Subtraction*

Description

Subtracts two arrays element-wise.

Usage

```
prim_sub(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Two inputs of the same data type and shape. Can be any numeric data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(`arrayish`)
Has the inputs' shape and data type.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `hlo_subtract()`, specified under subtract.

See Also

`nv_sub()`, `-`

Examples

```
# two R values: both take an R double's default data type
prim_sub(5, 3)

# the R value is built at the array's data type instead
prim_sub(5, nv_scalar(3, "f64"))
```

prim_sum *Primitive Sum Reduction*

Description

Sums array elements along the specified axes.

A boolean input is reduced with a logical OR, so the result is a boolean rather than a count. nv_sum() counts instead.

Usage

```
prim_sum(x, axes, drop = TRUE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axes	(integer()) Axes to reduce over. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) Whether to drop the reduced axes: removed from the output shape if TRUE, set to 1 if FALSE.

Value

(arrayish)
Has the input's data type. The shape is the input's with the reduced axes removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_reduce(), specified under reduce. The reducer is hlo_add().

See Also

nv_sum()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
# reducing axis 1 removes it from the shape, and the data type is kept
prim_sum(x, axes = 1L)

# drop = FALSE keeps the reduced axis at size 1 instead
prim_sum(x, axes = 1L, drop = FALSE)

# reducing every axis gives a scalar
prim_sum(x, axes = c(1L, 2L))
```

---

prim_svd *Primitive Singular Value Decomposition*

Description

Computes the reduced ("economy") singular value decomposition of a matrix `x` of shape `(m, n)`:

$$A = u \operatorname{diag}(d)\, vt,$$

where `u` has orthonormal columns, `vt` has orthonormal rows, and `d` is the length-`k` (`k = min(m, n)`) vector of non-negative singular values in descending order.

Note: unlike `base::svd()`, which returns the right singular vectors as `v` of shape `(n, k)` (so that `a = u %*% diag(d) %*% t(v)`), this primitive returns them already transposed as `vt` of shape `(k, n)` (matching the underlying LAPACK / cuSOLVER output and avoiding an extra transpose).

Supports any matrix shape on both the host (LAPACK `gesdd`) and CUDA (cuSOLVER `gesvd`) backends. cuSOLVER's `m >= n` requirement is handled transparently via a layout flip for wide matrices.

Usage

```
prim_svd(x)
```

Arguments

`x`	(`arrayish`) One input, with exactly 2 axes. Can be any float data type. An R value materializes at its default data type.

Value

(named `list` of three `arrayish`)
Elements `d` (length `k`), `u` (shape `(m, k)`), and `vt` (shape `(k, n)`). All have the input's data type.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to hlo_custom_call() with target "svd".

See Also

nv_svd()

Examples

```
# all three outputs have the input's data type
x <- nv_array(c(1, 0, 0, 1, 0, 1), shape = c(3, 2))
prim_svd(x)
```

---

prim_tan *Primitive Tangent*

Description

Element-wise tangent.

Usage

```
prim_tan(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_tan(), specified under tan.

See Also

nv_tan(), tan()

Examples

```
# the input's data type carries through
x <- nv_array(c(0, 0.5, 1))
prim_tan(x)

# an R value materializes at its default data type
prim_tan(0.5)
```

---

prim_tanh *Primitive Hyperbolic Tangent*

Description

Element-wise hyperbolic tangent.

Usage

```
prim_tanh(x)
```

Arguments

x	(arrayish) One input. Can be any float data type. An R value materializes at its default data type.

Value

(arrayish)
Has the input's shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_tanh(), specified under tanh.

See Also

nv_tanh(), tanh()

Examples

```
# the input's data type carries through
x <- nv_array(c(-1, 0, 1))
prim_tanh(x)

# an R value materializes at its default data type
prim_tanh(1)
```

`prim_top_k` *Primitive Top-K*

Description

Returns the `k` largest values along the last axis, sorted in decreasing order, and with `indices = TRUE` their indices into that axis as well.

For other axes, transpose so the target axis is last, call `prim_top_k()`, then transpose back. `nv_top_k()` does this.

Usage

```
prim_top_k(x, k, indices)
```

Arguments

`x`	(`arrayish`) One input, with at least 1 axis. Can be any numeric data type. An R value materializes at its default data type.
`k`	(`integer(1)`) Number of top elements. Must satisfy `1 <= k <= shape(x)[naxes(x)]`.
`indices`	(`logical(1)`) Whether to also return the indices of the top elements. Without them the order among tied values is unspecified, which lets the lowering pick the cheapest selection for the platform.

Value

(named `list` of one or two `arrayish`)
Element `values`, the top-k values at the input's data type, and, when `indices` is `TRUE`, `indices`, their indices along the last axis at the default integer data type (see `default_dtypes()`). Both have the input's shape with the last axis replaced by `k`. With indices, ties are broken by lower index first.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to hlo_top_k(). Without indices on CUDA it lowers to an unstable descending hlo_sort() of the values followed by an hlo_slice(), which is what the CHLO op expands to there minus the index operand and the stability the ties no longer need; XLA's CPU backend has a dedicated top-k kernel, so it keeps hlo_top_k().

See Also

nv_top_k(), prim_sort()

Examples

```
# `values` keeps the input's data type, `indices` is the default integer
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
prim_top_k(x, k = 3L, indices = FALSE)
prim_top_k(x, k = 3L, indices = TRUE)
```

---

prim_transpose *Primitive Transpose*

Description

Permutes the axes of an array.

Usage

```
prim_transpose(x, perm)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
perm	(integer()) Specifies the new ordering of axes. Must be a permutation of axes(x), the axis indices of x. Negative values count from the end, i.e. -1 refers to the last axis.

Value

(arrayish)
Has the input's data type and shape shape(x)[perm].

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to hlo_transpose(), specified under transpose.

See Also

nv_aperm(), t()

Examples

```
# the 2x3 becomes a 3x2, keeping its data type
x <- nv_matrix(1:6, nrow = 2)
prim_transpose(x, perm = c(2L, 1L))
```

---

prim_triangular_solve *Primitive Triangular Solve*

Description

Solves a system of linear equations with a triangular coefficient matrix. When left_side is TRUE, solves op(a) %*% x = b for x. When left_side is FALSE, solves x %*% op(a) = b for x. Axes before the last two are batch axes and must match between a and b (no broadcasting). Here op is A or A^T depending on transpose_a.

Usage

```
prim_triangular_solve(a, b, left_side, lower, unit_diagonal, transpose_a)
```

Arguments

a	(arrayish) Triangular coefficient matrix with at least 2 axes. The last two axes must be equal (square matrix); any leading axes are batch axes. Can be any float data type. a and b must have the same data type. An R value among them assumes the data type of the others when it is in its data type category, and its default data type when none of them has one.
b	(arrayish) Right-hand side. Same rank as a (rank >= 2), with matching leading batch axes. The size of a's last two (square) axes must equal b's second-to-last axis when left_side = TRUE, or b's last axis when left_side = FALSE. Shares a's data type – see a.
left_side	(logical(1)) If TRUE, solve op(a) %*% x = b. If FALSE, solve x %*% op(a) = b.
lower	(logical(1)) If TRUE, a is lower triangular. If FALSE, a is upper triangular.

`unit_diagonal` (`logical(1)`)
If `TRUE`, assume diagonal elements of `a` are 1.

`transpose_a` (`logical(1)`)
If `TRUE`, solve with `t(a)` in place of `a`.

Details

Differentiation is only implemented for a single system: the `reverse` rule errors on batched operands (operands with more than 2 axes).

Value

(`arrayish`)
Has `b`'s shape and the data type `a` and `b` agreed on.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

On CPU, an `f32` or `f64` solve lowers to a `"trsm"` `hlo_custom_call()` (backed by BLAS). Otherwise it lowers to `hlo_triangular_solve()`, specified under triangular_solve, which XLA runs through cuBLAS on CUDA.

References

Giles M (2008). "An extended collection of matrix derivative results for forward and reverse mode automatic differentiation." Oxford University Computing Laboratory.

See Also

`nv_solve()`

Examples

```
# solve L %*% x = b where L is lower triangular
L <- nv_matrix(c(2, 1, 0, 3), nrow = 2, dtype = "f32")
b <- nv_matrix(c(4, 3), nrow = 2, dtype = "f32")
prim_triangular_solve(L, b,
  left_side = TRUE, lower = TRUE,
  unit_diagonal = FALSE, transpose_a = FALSE
)
```

`prim_which_max` *Primitive Index of the Maximum*

Description

Returns the index of the maximum value along a single axis. Ties are broken by returning the smallest index.

Usage

```
prim_which_max(x, axis, drop = TRUE)
```

Arguments

`x`	(`arrayish`) One input. Can be any data type. An R value materializes at its default data type.
`axis`	(`integer(1)`) Axis along which to find the index of the maximum. Negative values count from the end, i.e. `-1` refers to the last axis.
`drop`	(`logical(1)`) If `TRUE` (default) the reduced axis is removed; if `FALSE` it is kept with size 1.

Value

(`arrayish`)
Has the default integer data type (see `default_dtypes()`) regardless of the input's, and the input's shape with `axis` removed (`drop = TRUE`) or set to 1 (`drop = FALSE`).

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `hlo_reduce()`, specified under reduce. The reduction is variadic over `(values, indices)`, with a (value > value | (value == value & idx < idx)) selector.

See Also

`prim_which_min()`, `nv_which_max()`

Examples

```
# the index comes out at the default integer data type
prim_which_max(nv_array(c(3, 1, 4, 1, 5)), axis = 1L)
```

prim_which_min *Primitive Index of the Minimum*

Description

Returns the index of the minimum value along a single axis. Ties are broken by returning the smallest index.

Usage

```
prim_which_min(x, axis, drop = TRUE)
```

Arguments

x	(arrayish) One input. Can be any data type. An R value materializes at its default data type.
axis	(integer(1)) Axis along which to find the index of the minimum. Negative values count from the end, i.e. -1 refers to the last axis.
drop	(logical(1)) If TRUE (default) the reduced axis is removed; if FALSE it is kept with size 1.

Value

(arrayish)
Has the default integer data type (see default_dtypes()) regardless of the input's, and the input's shape with axis removed (drop = TRUE) or set to 1 (drop = FALSE).

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to hlo_reduce(), specified under reduce. The reduction is variadic over (values, indices), with a (value < value | (value == value & idx < idx)) selector.

See Also

prim_which_max(), nv_which_min()

Examples

```
# the index comes out at the default integer data type
prim_which_min(nv_array(c(3, 1, 4, 1, 5)), axis = 1L)
```

`prim_while` *Primitive While Loop*

Description

Repeatedly executes body while cond returns `TRUE`, like R's `while` loop. The loop state is initialized with `init` and passed through each iteration; it is the only thing carried from one iteration to the next.

Usage

```
prim_while(init, cond, body)
```

Arguments

`init`	(`named list()`) Named list of initial state values, a tree in the sense of pjrt's `RTree`. Each leaf becomes a parameter of the loop's sub-graphs. R values are materialized at their default data type.
`cond`	(`function`) Condition function that receives the current state as arguments and outputs whether to continue the loop.
`body`	(`function`) Body function that receives the current state as arguments and returns a named list with the same structure, data types and shapes as `init`. Nothing is promoted: a loop-carried state is meant to be heterogeneous, so each member keeps its own data type across iterations.

Value

(named `list`)
A tree of the loop-carried arrays – see `RTree` – with the same structure, data types and shapes as `init`, in its final state after the loop terminates.

Implemented Rules

- `stablehlo`
- `quickr`

StableHLO

Lowers to `hlo_while()`, specified under while.

See Also

`nv_while()`

Examples

```
# the loop state is a named list, and each member keeps its data type
prim_while(
  init = list(i = nv_scalar(0L), total = nv_scalar(0L)),
  cond = function(i, total) i <= 5L,
  body = function(i, total) list(
    i = i + 1L,
    total = total + i
  )
)
```

`prim_xor` *Primitive Bitwise Xor*

Description

Element-wise bitwise XOR – a logical XOR on a boolean input, and a bit-by-bit one on an integer.

Usage

```
prim_xor(x, y)
```

Arguments

x, y	(arrayish) Two inputs of the same data type and shape. Can be any integerish data type. R values take the other operand's data type when it is in their data type category, and their default data type when neither operand has one. An R value outside the other operand's category is an error, as are two R values of different storage types.

Value

(arrayish)
Has the inputs' shape and data type.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `hlo_xor()`, specified under xor.

See Also

nv_xor()

Examples

```
# two R values: both take an R integer's default data type
prim_xor(12L, 10L)

# the R value is built at the array's data type instead
prim_xor(12L, nv_scalar(10L, "i64"))
```

---

promotion_rule *Promotion Rules*

Description

Functions for materializing R values as arrays and promoting inputs. Most commonly used via the .promote argument of as_anvl_arrays().

promotion_common() brings every input to their common data type (common_dtype()). An R value takes the data type the arrays meet at when that is in its own or a higher category, and otherwise contributes its default data type.

promotion_like() brings the inputs to the data type of a selected input. If the selected input is an R value, its default data type is used.

promotion_dtype() brings the inputs to the specified data type.

promotion_rdata_common() brings the *R values* to the data type of the arrays among the inputs, which must all have the same one. An R value must be in that data type's category (a double can e.g. *not* become an integer). When all inputs are R values, they settle on their shared default data type; R values of different storage types are an error. This rule is commonly used in primitives expecting homogeneous inputs for one or more argument subsets.

promotion_grouped() applies several rules to disjoint subsets. The rules must all refer to arguments by name, or all by position.

promotion_rule() creates a new promotion rule. It takes in arrayish values and outputs a list of data types, with NULL indicating no conversion.

Usage

```
promotion_common(on = NULL, fallback = NULL)

promotion_like(arg, on = NULL, coerce = FALSE)

promotion_dtype(dtype, on = NULL, coerce = FALSE)

promotion_rdata_common(on = NULL)
```

```
promotion_grouped(...)

promotion_rule(fn, kind, on = NULL, ...)
```

Arguments

on	(NULL | character() | numeric()) Subset of arguments to apply a rule to. Indicated either via position or argument name. For promotion_rule(), on only declares which arguments the rule covers (which promotion_grouped() needs to check that its rules are disjoint); fn itself must restrict itself to them.
fallback	(NULL | xlamisc::DataType | character(1)) The data type to settle on when *every* input is a bare R value, in place of the default those would materialize at on their own. NULL (default) leaves them their default.
arg	(character(1) | numeric(1)) Which input to take the data type from: its name in the as_anvl_arrays() call, or its position. Naming it needs the call's arguments to be named.
coerce	(logical(1)) Bring an input to the target even where that is not a promotion, instead of raising an error. Two things are refused without it: a float reaching an integer data type, which no category crosses to on its own (an R double at i32, or an f32 array at i32), and narrowing a value the target cannot hold (an f64 array at f32). The default is FALSE.
dtype	(xlamisc::DataType | character(1)) The data type to bring the inputs to.
...	For promotion_grouped(): (PromotionRule) The rules to apply to disjoint argument subsets. For promotion_rule(): (any) Further fields stored in the rule's spec attribute next to on, e.g. for format() to show.
fn	(function) The rule.
kind	(character(1)) What the rule is, for printing: it shows as <{kind}>, so give it the name of the function that builds it.

Value

(PromotionRule)

See Also

as_anvl_arrays(), nv_promote_to_common(), common_dtype()

Examples

```
promotion_common()(list(pi, nv_scalar(2L, "i64")))
promotion_common(fallback = "f64")(list(1, 2))
promotion_common(c(1, 2))(list(-3, 4, 1))


promotion_like("x", coerce = TRUE)(list(x = nv_scalar(1, "f32"), nv_scalar(1, "f64")))
# without `coerce`, a target the input cannot hold is refused
try(promotion_like("x")(list(x = nv_scalar(1, "f32"), nv_scalar(1, "f64"))))


promotion_dtype("f64")(list(1, nv_scalar(2, "f32")))


promotion_rdata_common()(list(nv_scalar(1, "f64"), 2))
try(promotion_rdata_common()(list(nv_scalar(1L, "i32"), 2.5)))


rule <- promotion_grouped(
  promotion_dtype("f64", on = "x"),
  promotion_like("x", on = "y")
)
rule
rule(list(x = 1, y = nv_scalar(2L, "i32"), z = 3L))


# every input at the widest float in the call, and never below f32.
widest_float <- promotion_rule(
  function(args) {
    widths <- vapply(args, function(a) {
      dt <- peek_dtype(to_abstract(a))
      if (xlamisc::is_dtype_float(dt)) xlamisc::dtype_width(dt) else 0L
    }, integer(1))
    rep(list(as_dtype(paste0("f", max(c(32L, widths))))), length(args))
  },
  "widest_float"
)
widest_float
as_anvl_arrays(nv_array(1L), 2.5, nv_array(1, dtype = "f64"), .promote = widest_float)
```

`quickr_device` *Quickr Device*

Description

Device descriptor for the quickr backend. The only supported `type` is `"cpu"`.

Usage

```
quickr_device(x = "cpu")
```

Arguments

x	(character(1)) Device type. Currently only supports "cpu".

Value

(QuickrDevice)

See Also

nv_device(), AnvlBackendQuickr().

RData *R Data Class*

Description

The AbstractArray of a non-static R value that was passed to a jit-compiled function. It is special, because it does not have a data type, i.e., calling dtype() on it results in an error.

Usage

```
RData(shape, r_type)
```

Arguments

shape	(stablehlo::Shape | integer()) The shape of the value: () for a length-1 vector, its dim() for an R array.
r_type	(character(1)) The R storage type: "double", "integer" or "logical".

Details

It only exists *during* tracing, never in a finalized graph.

Value

(RData)

Extractors

Just like AbstractArray, with the exception that dtype() errs.

See Also

AbstractArray, AnvlGraph

Examples

```
RData(c(2, 3), "double")
# is equivalent to
nv_aval("double", c(2, 3))
# below, the `RData` input is materialized in 32 and 64-bit precisions, so the input
# dtype becomes f64.
# by NOT converting RData to their default data type we prevent loss of precision
# (avoiding a double -> default data type -> f64 round trip)
graph <- trace_fn(function(x) {
    print(x)
    list(x + nv_scalar(1, "f64"), x + nv_scalar(1, "f32"))
  }, list(x = nv_aval("double", c())))
)
print(graph)
# the actual inputs to the compiled program
graph$inputs
# the R storage types of the inputs passed as R values; NA for inputs passed
# as AnvlArrays
graph$rdata_types
```

`rule_reverse` *Reverse Rule*

Description

Construct a reverse-mode autodiff rule for a primitive. Provide exactly one of `backward` and `forward`.

Pass `backward` when the primitive's forward statement can run unmodified, which covers most use cases. It has the signature `function(inputs, outputs, grads, params, required)` and returns a `list` with one entry per input: that input's gradient, or `NULL` where `required` says it is not needed.

Pass `forward` when a slightly different forward pass enables a more efficient backward pass. It has the signature `function(inputs, params, required)`, where `required` says which inputs need a gradient, and returns `list(outputs = , backward = )`: the forward results and a closure with the signature of `backward` above, which can use intermediate values of the forward pass via lexical scoping.

A rule with a `forward` pass can also be the `rule` of `prim_custom_gradient()`.

Usage

```
rule_reverse(backward = NULL, forward = NULL)
```

Arguments

`backward`	(`NULL` | `function`) Backward hook for the default case.
`forward`	(`NULL` | `function`) Alternative forward hook that returns both the outputs and a backward closure.

Value

(anvl_rule_reverse)

See Also

graph_gradient()

Examples

```
# the rule of prim_negate()
rule_reverse(function(inputs, outputs, grads, params, required) {
  list(if (required[[1L]]) prim_negate(grads[[1L]]))
})
```

---

shape *Get the Shape of an Array*

Description

Returns the shape of an array as an integer() vector.

Usage

```
shape(x, ...)
```

Arguments

x	(arrayish) An array-like object.
...	Additional arguments passed to methods (unused).

Details

An R value has a shape when it is arrayish: a length-1 vector is a scalar with shape integer(), and an R array has its dim(). Any other R vector is an error; use nv_array() to make it an array.

This is implemented via the generic xlamisc::shape().

Value

(integer())

Examples

```
x <- nv_array(1:4, dtype = "f32")
shape(x)
shape(nv_array(1:6, shape = c(2, 3)))
# a length-1 R vector is a scalar
shape(1)
shape(TRUE)
# an R array has its `dim()`
shape(array(1:6, dim = c(2, 3)))
# any other R vector has no shape
try(shape(c(2, 3)))
```

Shape *Create a Shape Object*

Description

Constructs a Shape, the axis sizes of an array. A Shape *is* its integer vector, with a class attached, so `length()` is the number of axes and `shape[i]` is the size of axis `i`.

Usage

```
Shape(dims = integer())
```

Arguments

`dims`	An `integer()` vector of axis sizes (>= 0). `NA` marks an axis whose size is only known at run time.

Value

(Shape)

See Also

`shape()`, `stablehlo::Shape()`

Examples

```
Shape(c(2L, 3L))
```

`stablehlo` *Lower a Graph to StableHLO*

Description

Converts a traced `AnvlGraph` into the StableHLO intermediate representation (IR), building a `stablehlo::Func` with the stablehlo package. Each statement of the graph is translated to its corresponding StableHLO op. The result can be serialized to MLIR text via `stablehlo::repr()` and subsequently compiled to an XLA executable with `pjrt::pjrt_compile()`.

The rule for translating a primitive to stablehlo is `prim_<name>[["stablehlo"]]`.

The arguments of the resulting function are, in this order: the graph's constants, the graph's inputs, the inputs of the graph's side effects (such as the key and the counter of the global random state, see `nv_set_seed()`), and the phantom donated inputs (when `donate_unaliased_outputs = TRUE`), one per output that is not already aliased to a donated input, in the order of the outputs. Its results are the graph's outputs alone: it returns no value for a side effect, as the host advances it after the run by the number of its uses, e.g. the random state by the number of draws (see the side effects section of the Internals article).

This is a low-level function; most users should use `jit()` instead.

Usage

```
stablehlo(
  graph,
  id = "main",
  captured = list(),
  donate = character(),
  donate_unaliased_outputs = FALSE,
  platform = NULL
)
```

Arguments

`graph`	(`AnvlGraph`) The graph to lower (e.g. produced by `trace_fn()`).
`id`	(`character(1)`) The id of the resulting StableHLO function. Use `"main"` (the default) for a top-level lowering (returning from the `main` function finalizes the module) and `""` for a closure/region lowering (e.g. a while body or a scatter update computation) that builds an anonymous nested function inside an enclosing build.
`captured`	(`list(FuncValue)`) Values of the enclosing function to bind the graph's last `length(captured)` inputs to. The lowered function reads them the way an MLIR region captures a value from above, instead of declaring them as inputs. Used for the subgraphs of higher-order primitives (e.g. `nv_while`), whose trailing inputs are the values they close over. The graph's constants always become inputs, and `GraphLiteral`s are always inlined.

donate (character())
Names of the arguments whose buffers should be donated. Donated buffers can be aliased with outputs of the same type, enabling in-place operations.

donate_unaliased_outputs
(logical(1))
If TRUE and the current target platform is "cpu", append a phantom donated input for every output that isn't already aliased to a user-donated input. They come after all other arguments. This is needed internally so R keeps track of the CPU buffers memory in order to know when to garbage collect.

platform (NULL | character(1))
Target platform name (e.g. "cpu", "cuda"). Stored on a process-wide global during the call so that platform-aware lowering rules (queried via current_platform()) can branch on it. NULL (the default) leaves the current value untouched — recursive calls from higher-order primitives inherit the platform of the enclosing call.

Value

(list)
Of length 4:

- the stablehlo::Func
- The graph's constants: the GraphValues holding ConcreteArrays, whose data must be passed as the leading inputs at execution time.
- A list of phantom-output specs, one per phantom donated input appended when donate_unaliased_outputs = TRUE. Each entry is a list(dtype, shape) describing the buffer the executor must allocate. Empty when no phantoms were added.
- The other stablehlo::Funcs the function calls, which a program must define next to it: functions the lowering rules build once and call wherever they need them, e.g. the ThreeFry block cipher of prim_sample_bits(). Empty for most graphs, and always for a nested lowering (id = ""), whose calls the top-level lowering collects.

See Also

trace_fn(), jit(), current_platform()

Examples

```
# the closed-over array `x` becomes the constant input %0, before the
# graph's own input `y` (%1)
x <- nv_array(c(1, 2))
graph <- trace_fn(function(y) y + x, list(y = nv_aval("f32", shape = 2)))
out <- stablehlo(graph)
out[[1L]]
out[[2L]]

# a donated input is aliased with an output of the same type
graph <- trace_fn(
  function(a, b) list(a + b, a * b),
```

```
  list(a = nv_aval("f32", 3), b = nv_aval("f32", 3))
)
stablehlo(graph, donate = "a")[[1L]]

# on CPU, a phantom input is appended for every output not aliased yet, and
# the third element describes the buffers to allocate for them
out <- stablehlo(graph, donate_unaliased_outputs = TRUE, platform = "cpu")
out[[1L]]
out[[3L]]
```

---

subgraphs *Get Subgraphs from Higher-Order Primitive*

Description

Extracts the subgraphs from the parameters of a statement that applies a higher-order primitive, such as the branches of `prim_if()` or the body of `prim_while()`.

This is not recursive: only the subgraphs held directly by `statement` are returned, not the ones nested in the statements of those subgraphs. Call `subgraphs()` on their statements to descend further.

Usage

```
subgraphs(statement)
```

Arguments

`statement`	(`GraphStatement`) The statement.

Value

(named `list(AnvlGraph)`)
The subgraphs, named after the parameters that hold them. Empty for a primitive that is not higher-order.

---

to_abstract *Convert to Abstract Array*

Description

Convert an object to its abstract array representation (`AbstractArray`).

Usage

```
to_abstract(x, pure = FALSE)
```

Arguments

x	(arrayish | AbstractArray) Object to convert.
pure	(logical(1)) Whether to convert to a pure AbstractArray and not e.g. RData or ConcreteArray.

Value

(AbstractArray)
A ConcreteArray for an AnvlArray, an RData for an R value, the abstract array of a GraphBox, and x itself for an AbstractArray. With pure = TRUE, a plain AbstractArray of the same shape and data type.

Examples

```
# an R value becomes `RData`: it has no data type of its own yet
to_abstract(1.5)
to_abstract(1L)
to_abstract(TRUE)

# an AnvlArray becomes a ConcreteArray
to_abstract(nv_array(1:4))

# use pure = TRUE to strip subclass info
to_abstract(nv_array(1:4), pure = TRUE)
```

---

trace_fn *Trace an R Function into a Graph*

Description

Executes f with abstract array arguments and records every primitive operation into an AnvlGraph.

The resulting graph can be lowered to StableHLO (via stablehlo()) or transformed (e.g. via graph_gradient()).

When f uses a side effect, the graph records it in its side_effects field rather than among its inputs and outputs (see AnvlGraph).

Usage

```
trace_fn(
  f,
  args = NULL,
  desc = NULL,
  args_flat = NULL,
  in_tree = NULL,
  optimize = FALSE
)
```

Arguments

f	(function) The function to trace. Must not be a JitFunction (i.e. already jitted).
args	(list of (AnvlArray | AbstractArray)) The (unflattened) arguments to the function. Mutually exclusive with the args_flat/in_tree pair.
desc	(NULL | GraphDescriptor) Optional descriptor. When NULL (default), a new descriptor is created.
args_flat	(list) Flattened arguments. Must be accompanied by in_tree.
in_tree	(RTree) Tree structure describing how args_flat maps back to f's arguments.
optimize	(logical(1) | character()) Which graph optimization passes to run on the traced graph before returning it. TRUE runs all passes, FALSE (default) runs none, and a character vector selects a subset by name. The available passes are: • "inline_scalars": replace scalar-shaped constants with inline literals. • "remove_unused_constants": drop constants not referenced by the graph. jit() always traces with all passes enabled.

Value

(AnvlGraph) Contains the traced operations.

See Also

stablehlo() to lower the graph, jit() for end-to-end compilation.

Examples

```
graph <- trace_fn(function(x, y) x + y,
  args = list(x = nv_array(1, dtype = "f32"), y = nv_array(2, dtype = "f32")))
graph
```

---

with_dtypes *Run a Function at Given Data Types*

Description

with_dtypes() wraps f into a function that works at the data types dtypes names: on each call every array argument of a category dtypes names is converted to that data type, the defaults (see default_dtypes()) are set to the float / int entries for the duration of the call, and every returned array of a named category is converted as well. For example, with_dtypes(nv_add, c(float = "f64")) is an nv_add() that computes floats at f64.

A category dtypes does not name is left alone, in the arguments, in the body and in the result.

Usage

```
with_dtypes(f, dtypes)
```

Arguments

f	(function) The function to wrap.
dtypes	(named character()	named list()) A mapping of the data type categories (float, int and uint) to data types, e.g. c(float = "f64", int = "i64"). Each may be a string or a DataType. A category it does not name is left as it is.

Details

Note that f itself can also change the default data types, which overrides the defaults configured by with_dtypes().

Value

A function with the same arguments as f, a JitFunction if f was one.

See Also

default_dtypes(), local_default_dtypes()

Examples

```
add_f64 <- with_dtypes(nv_add, c(float = "f64"))
# an `f32` argument is converted, and the result comes back as `f64`
dtype(add_f64(nv_array(1, dtype = "f32"), 2.5))
# A category that is not named is untouched
dtype(add_f64(nv_array(1L, dtype = "i32"), 2L))
# `uint` is converted too, but sets no default
dtype(with_dtypes(nv_add, c(uint = "ui32"))(nv_array(1L, dtype = "ui8"), 2L))
```

Index

!.AnvlArray *(nv_not)*, 158
!=.AnvlArray *(nv_ne)*, 153
∗ **rng**
    local_nv_rng_state, 62
    nv_normal, 155
    nv_rbinom, 178
    nv_rng_state, 182
    nv_sample, 185
    nv_sample_int, 186
    nv_set_seed, 194
    nv_uniform, 220
*.AnvlArray *(nv_mul)*, 152
+.AnvlArray *(nv_add)*, 68
-.AnvlArray *(nv_sub)*, 207
/.AnvlArray *(nv_div)*, 108
<.AnvlArray *(nv_lt)*, 143
<=.AnvlArray *(nv_le)*, 135
==.AnvlArray *(nv_eq)*, 113
>.AnvlArray *(nv_gt)*, 127
>=.AnvlArray *(nv_ge)*, 126
[.AnvlArray, 10
[<-.AnvlArray, 12
%*%.AnvlArray *(nv_matmul)*, 145
%/%.AnvlArray *(nv_floor_div)*, 124
%%.AnvlArray *(nv_mod)*, 151
&.AnvlArray *(nv_and)*, 70
^.AnvlArray *(nv_pow)*, 169

abs(), *232*
abs.AnvlArray *(nv_abs)*, 65
AbstractArray, *32*, *38*, *40*, *43*, *49*, *51*, *57*, *78*, *230*, *349*, *355–357*
AbstractArray *(nv_aval)*, 78
acos(), *233*
acos.AnvlArray *(nv_acos)*, 66
acosh(), *234*
acosh.AnvlArray *(nv_acosh)*, 67
active backend, *35*
active_backend, 13
active_backend(), *9*, *15*, *34*, *47*, *50*, *54*, *56*, *58*, *60*, *106*, *111*, *121*, *122*, *131*, *137*, *142*, *192*, *224*, *276*, *285*
all.AnvlArray *(nv_all)*, 69
anvl *(anvl-package)*, 9
anvl-package, 9
AnvlArray, 14, *15*, *16*, *18*, *19*, *22–25*, *28–30*, *32*, *38*, *43*, *54*, *57*, *110*, *111*, *121*, *122*, *131*, *137*, *142*, *148*, *187*, *192*, *193*, *224*, *231*, *356*, *357*
AnvlBackend, *18*, *19*
AnvlBackend(), *18*, *19*, *106*
AnvlBackendPjrt, 18
AnvlBackendPjrt(), *19*
AnvlBackendQuickr, 19
AnvlBackendQuickr(), *18*, *349*
AnvlGraph, *43*, *44*, *46*, *48*, *49*, *349*, *353*, *356*, *357*
AnvlPrimitive, *20*, *43*
AnvlPrimitive *(new_primitive)*, 64
AnvlPrimitiveDef, 20, *43*, *65*
any.AnvlArray *(nv_any)*, 71
aperm.AnvlArray *(nv_aperm)*, 72
apply_promotion, 20
arr, 21
array, *26*
array(), *80*
arrayish, *11–13*, 22, *25*, *26*, *30*, *36*, *37*, *43*, *57*, *64–81*, *83–105*, *107–109*, *112–174*, *176–186*, *188–192*, *195–208*, *210–227*, *229*, *230*, *232–261*, *263–277*, *279–316*, *318*, *321–343*, *345*, *346*, *351*, *356*
as-AnvlArray, 23
as.double(), *27*
as.double.AnvlArray *(as-AnvlArray)*, 23
as.integer.AnvlArray *(as-AnvlArray)*, 23
as.integer64.AnvlArray *(as-AnvlArray)*, 23

as.logical.AnvlArray *(as-AnvlArray)*, 23
as.vector.AnvlArray *(as-AnvlArray)*, 23
as_anvl_array, 25
as_anvl_array(), *25*
as_anvl_arrays *(as_anvl_array)*, 25
as_anvl_arrays(), *25*, *230*, *346, 347*
as_array, *17*
as_array *(as_array.AnvlArray)*, 26
as_array(), *18, 19*, *24*
as_array.AnvlArray, 26
as_dtype, 27
as_dtype(), *52*
as_raw, 28
as_raw(), *83*, *244*
asin(), *239*
asin.AnvlArray *(nv_asin)*, 73
asinh(), *240*
asinh.AnvlArray *(nv_asinh)*, 74
atan(), *241*
atan.AnvlArray *(nv_atan)*, 75
atanh(), *243*
atanh.AnvlArray *(nv_atanh)*, 77
await, 29
await(), *29*
axes, 30
axes(), *16*
axes(x), *339*

backend, 30
backend(), *106*, *231*
base::all(), *69*
base::any(), *72*
base::aperm(), *72, 73*
base::as.vector(), *24*
base::atan2(), *76*, *242*
base::c(), *91*
base::cbind(), *79, 80*
base::chol(), *248*
base::crossprod(), *99*
base::cummax(), *100*
base::cummin(), *101*
base::cumprod(), *102*
base::cumsum(), *103*
base::determinant(), *105*
base::dim(), *37*
base::drop(), *205*
base::eigen(), *112*, *269*
base::length(), *56*
base::max(), *147*
base::min(), *151*
base::order(), *160*
base::pmax(), *164*, *303*
base::pmin(), *165*, *304*
base::prod(), *171*
base::psigamma(), *173*
base::range(), *177, 178*
base::rbind(), *79, 80*
base::rev(), *182*
base::solve(), *130*, *201*
base::sort(), *203, 204*
base::sum(), *208, 209*
base::svd(), *210*
base::t(), *73*
base::tanpi(), *212*
base::tcrossprod(), *213*
base::which.max(), *226*
base::which.min(), *227*
bit64::as.integer64(), *23*
bit64::integer64, *23, 24*, *27*
broadcast, *79*, *91*, *129*, *173*

c(), *22*
c.AnvlArray *(nv_concatenate)*, 90
category, *68*, *70*, *109*, *113*, *124*, *126, 127*, *135*, *143*, *151–153*, *159*, *164, 165*, *169*, *180*, *207*, *229*
cbind.AnvlArray *(nv_bind)*, 79
ceiling(), *247*
ceiling.AnvlArray *(nv_ceiling)*, 87
chol.AnvlArray *(nv_chol)*, 88
common data type, *68*, *70*, *76*, *109*, *113*, *124*, *126, 127*, *135*, *143*, *151–153*, *159*, *164, 165*, *169*, *180*, *207*, *229*
common_dtype, 31
common_dtype(), *39*, *346, 347*
ConcreteArray, 32, *32*, *78*, *354*, *356*
cos(), *255*
cos.AnvlArray *(nv_cos)*, 96
cosh(), *256*
cosh.AnvlArray *(nv_cosh)*, 97
cospi.AnvlArray *(nv_cospi)*, 98
crossprod.AnvlArray *(nv_crossprod)*, 98
cummax.AnvlArray *(nv_cummax)*, 99
cummin.AnvlArray *(nv_cummin)*, 100
cumprod.AnvlArray *(nv_cumprod)*, 102
cumsum.AnvlArray *(nv_cumsum)*, 103
current descriptor, *43*
current_descriptor, 33

current_descriptor(), *46*, *61*
current_platform, 33
current_platform(), *354*

data type category, *76*, *173*, *202*, *216*, *235*, *237*, *242*, *249*, *250*, *253*, *264*, *265*, *268*, *270*, *281*, *282*, *284*, *287*, *291*, *295*, *296*, *299*, *300*, *303*, *304*, *306*, *309*, *311*, *312*, *321*, *333*, *340*, *345*
DataType, *15*, *35*, *38*, *60*, *83*, *95*, *120*, *122*, *131*, *137*, *156*, *179*, *186*, *192*, *220*, *244*, *251*, *276*, *285*, *317*, *358*
default data type, *11*, *12*, *15*, *65–68*, *70*, *72*, *74*, *75*, *77*, *83*, *86–90*, *92–94*, *96–98*, *100–105*, *107–109*, *112–117*, *120*, *122–127*, *130*, *132–136*, *138–141*, *143–147*, *149–154*, *158–160*, *163–165*, *168–171*, *174*, *176*, *177*, *180*, *181*, *183–185*, *189*, *191*, *198–201*, *203–208*, *210–212*, *214*, *215*, *217–219*, *222*, *224*, *226–230*, *232–235*, *237*, *239–251*, *253*, *255–260*, *263–266*, *268–275*, *277*, *279*, *281*, *282*, *284*, *286–300*, *302–316*, *321*, *323–328*, *330*, *331*, *333–340*, *342–345*
default device, *16*
default float, *82*, *108*, *119*, *156*, *167*, *221*
default float type, *156*, *220*
default integer type, *179*, *186*
default_device, 34
default_device(), *9*, *15*, *18*, *53*, *59*, *60*, *111*, *121*, *122*, *131*, *137*, *142*, *187*, *192*, *193*, *224*, *276*, *285*
default_dtypes, 35
default_dtypes(), *9*, *14*, *18*, *39*, *47*, *54*, *57*, *58*, *60*, *61*, *66*, *67*, *74–77*, *87*, *89*, *96–98*, *100–105*, *108*, *112*, *114–117*, *120*, *125*, *130*, *136–141*, *144*, *148*, *160*, *163*, *171*, *173*, *174*, *184*, *190*, *192*, *199–202*, *204*, *205*, *208*, *210–212*, *214–217*, *225–227*, *230*, *257*, *258*, *292*, *338*, *342*, *343*, *357*, *358*
default_float *(*default_dtypes*)*, 35
default_int *(*default_dtypes*)*, 35
determinant.AnvlArray *(*nv_determinant*)*, 105

device, *15*, *25*, 36, *53*, *111*, *120*, *122*, *131*, *137*, *142*, *192*, *223*, *276*, *285*
device(), *16*, *18*
digamma(), *263*
digamma.AnvlArray *(*nv_digamma*)*, 107
dim(), *57*
dim.AnvlArray, 37
donated, *13*
dtype, 37
dtype(), *16*, *40*
dtype_categories, 38, *138*

eq_type, 40
exp(), *274*
exp.AnvlArray *(*nv_exp*)*, 116
expm1.AnvlArray *(*nv_expm1*)*, 117

floor(), *278*
floor.AnvlArray *(*nv_floor*)*, 123
format(), *347*

gamma.AnvlArray *(*nv_gamma*)*, 125
gradient, 41
gradient(), *13*, *44*, *47*, *82*, *89*, *128*, *189*, *217*, *228*, *260–262*, *322*
graph_desc_add, 42
graph_desc_add(), *20*, *21*, *46*, *47*, *65*
graph_gradient, 43
graph_gradient(), *41*, *351*, *356*
graph_to_quickr_r_function, 44
GraphBox, *15*, *22*, *23*, *25*, *38*, *43*, 45, *46*, *78*, *148*, *356*
GraphDescriptor, *33*, *43*, *45*, 46, *61*
GraphLiteral, *45*, 48, *48*
GraphLiteral(), *48*
GraphNode, *45*, 48
GraphStatement, *20*, *42*, *46*, 49
GraphValue, *45*, *46*, *48*, 49, *78*, *354*
GraphValue(), *48*

hlo_abs(), *232*
hlo_acos(), *233*
hlo_acosh(), *234*
hlo_add(), *51*, *235*, *260*, *334*
hlo_and(), *236*, *237*
hlo_asin(), *239*
hlo_asinh(), *240*
hlo_atan(), *241*
hlo_atan2(), *242*

hlo_atanh(), *243*
hlo_bitcast_convert(), *244*
hlo_broadcast_in_dim(), *246*
hlo_cbrt(), *246*
hlo_ceil(), *247*
hlo_clamp(), *250*
hlo_compare(), *270*, *281*, *282*, *287*, *291*, *297*, *330*
hlo_concatenate(), *251*
hlo_convert(), *252*
hlo_convolution(), *254*
hlo_cosh(), *256*
hlo_cosine(), *255*
hlo_custom_call(), *249*, *269*, *292*, *307*, *310*, *336*, *341*
hlo_digamma(), *263*
hlo_divide(), *264*
hlo_dot_general(), *266*
hlo_dynamic_slice(), *267*, *320*
hlo_dynamic_update_slice(), *268*, *320*
hlo_erf(), *271*
hlo_erf_inv(), *272*
hlo_erfc(), *273*
hlo_exponential(), *274*
hlo_exponential_minus_one(), *275*
hlo_floor(), *278*
hlo_gather(), *280*
hlo_if(), *283*
hlo_iota(), *51*, *286*
hlo_is_finite(), *286*
hlo_lgamma(), *288*
hlo_log(), *289*
hlo_log_plus_one(), *290*
hlo_logistic(), *302*
hlo_maximum(), *294*, *303*
hlo_minimum(), *295*, *304*
hlo_multiply(), *259*, *296*, *308*
hlo_negate(), *298*
hlo_not(), *299*
hlo_or(), *238*, *300*
hlo_pad(), *301*
hlo_polygamma(), *309*
hlo_popcnt(), *305*
hlo_power(), *306*
hlo_reduce(), *236*, *238*, *294*, *295*, *308*, *312*, *334*, *342*, *343*
hlo_reduce_window(), *257–260*
hlo_remainder(), *313*
hlo_reshape(), *314*
hlo_reverse(), *315*
hlo_round_nearest_afz(), *316*
hlo_round_nearest_even(), *316*
hlo_rsqrt(), *317*
hlo_scatter(), *322*
hlo_select(), *284*
hlo_shift_left(), *324*
hlo_shift_right_arithmetic(), *325*
hlo_shift_right_logical(), *326*
hlo_sign(), *326*
hlo_sine(), *327*
hlo_sinh(), *328*
hlo_slice(), *332*, *339*
hlo_sort(), *330*, *339*
hlo_sqrt(), *331*
hlo_subtract(), *333*
hlo_tan(), *336*
hlo_tanh(), *337*
hlo_tensor(), *57*, *277*
hlo_top_k(), *339*
hlo_transpose(), *244*, *314*, *340*
hlo_triangular_solve(), *341*
hlo_while(), *292*, *320*, *344*
hlo_xor(), *345*

ifelse(), *128*, *129*, *283*, *284*
install_anvl, 50
IotaArray, 51, *51*, *78*
is.finite.AnvlArray *(*nv_is_finite*)*, 132
is.infinite.AnvlArray *(*nv_is_infinite*)*, 133
is.nan.AnvlArray *(*nv_is_nan*)*, 134
is_arrayish *(*arrayish*)*, 22
is_arrayish(), *22*
is_device, 52
is_dtype, 52
is_dtype(), *28*

jit, 53
jit(), *11*, *12*, *18*, *19*, *22*, *25*, *41*, *44*, *45*, *47*, *55*, *56*, *60*, *62*, *65*, *106*, *182*, *353*, *354*, *357*
jit_cache_size, 55
jit_cache_size(), *54*

length(), *37*
length.AnvlArray, 56
lgamma(), *288*

lgamma.AnvlArray *(nv_lgamma)*, 136
LiteralArray, *48*, 57, *57*, *78*
local_backend, 58
local_backend(), *9*, *14*, *18*, *19*, *54*, *60*, *61*
local_backend(quickr), *19*
local_default_device, 59
local_default_device(), *9*, *34*
local_default_dtypes, 60
local_default_dtypes(), *35*, *358*
local_descriptor, 61
local_descriptor(), *47*
local_nv_rng_state, 62
local_nv_rng_state(), *156*, *157*, *179*, *182*, *183*, *185*, *186*, *195*, *221*, *318*, *319*
local_nv_seed *(nv_set_seed)*, 194
local_platform *(current_platform)*, 33
local_platform(), *33*
log(), *289*
log.AnvlArray *(nv_log)*, 138
log10.AnvlArray *(nv_log10)*, 139
log1p.AnvlArray *(nv_log1p)*, 140
log2.AnvlArray *(nv_log2)*, 141

map_tree, 63
map_tree(), *29*
max.AnvlArray *(nv_max)*, 146
mean.AnvlArray *(nv_mean)*, 147
median.AnvlArray *(nv_median)*, 148
min.AnvlArray *(nv_min)*, 150

naxes, 64
naxes(), *16*, *30*
nelts(), *57*
neq_type *(eq_type)*, 40
new_primitive, 64
new_primitive(), *21*, *42*
nv_abs, 65
nv_abs(), *232*
nv_acos, 66
nv_acos(), *233*
nv_acosh, 67
nv_acosh(), *234*
nv_add, 68
nv_add(), *235*
nv_all, 69
nv_all(), *69*, *236*
nv_and, 70
nv_and(), *237*
nv_any, 71
nv_any(), *71*, *239*
nv_aperm, 72
nv_aperm(), *340*
nv_array, *15*, *110*
nv_array *(AnvlArray)*, 14
nv_array(), *18*, *19*, *106*, *109*, *111*, *351*
nv_array_like *(AnvlArray)*, 14
nv_asin, 73
nv_asin(), *239*
nv_asinh, 74
nv_asinh(), *240*
nv_atan, 75
nv_atan(), *241*
nv_atan2, 76
nv_atan2(), *242*
nv_atanh, 77
nv_atanh(), *243*
nv_aval, 78
nv_bind, 79
nv_binomial, 80
nv_bitcast_convert, 83
nv_bitcast_convert(), *245*
nv_bool *(nv_dtype_constructors)*, 109
nv_broadcast_arrays, 84, *86*
nv_broadcast_arrays(), *86*
nv_broadcast_scalars, 85
nv_broadcast_scalars(), *85*, *86*
nv_broadcast_to, 86
nv_broadcast_to(), *85*, *246*
nv_cbind *(nv_bind)*, 79
nv_cbrt, 87
nv_cbrt(), *246*
nv_ceiling, 87
nv_ceiling(), *219*, *247*
nv_chol, 88
nv_chol(), *202*, *217*
nv_clamp, 89
nv_clamp(), *250*
nv_concatenate, 90
nv_concatenate(), *80*, *251*
nv_conv1d, 91
nv_conv1d(), *94*, *95*, *252*, *254*
nv_conv2d, 93
nv_conv2d(), *92*, *95*, *252*, *254*
nv_conv3d, 94
nv_conv3d(), *92*, *94*, *252*, *254*
nv_convert, 95
nv_convert(), *17*, *24*, *27*, *31*, *39*, *40*, *83*, *111*,

252
nv_cos, 96
nv_cos(), *98*, *255*
nv_cosh, 97
nv_cosh(), *256*
nv_cospi, 98
nv_cospi(), *201*, *212*
nv_crossprod, 98
nv_crossprod(), *213*
nv_cummax, 99
nv_cummax(), *257*
nv_cummin, 100
nv_cummin(), *258*
nv_cumprod, 102
nv_cumprod(), *258, 259*
nv_cumsum, 103
nv_cumsum(), *189*, *259, 260*
nv_dbinom *(*nv_binomial*)*, 80
nv_det, 104
nv_det(), *105, 106*
nv_determinant, 105
nv_determinant(), *104*
nv_device, 106
nv_device(), *15*, *30*, *34*, *111*, *121, 122*, *131*, *137*, *142*, *192*, *224*, *276*, *285*, *349*
nv_dexp *(*nv_exponential*)*, 118
nv_diag, 107
nv_diag(), *120, 121*, *215*
nv_digamma, 107
nv_digamma(), *263*
nv_div, 108
nv_div(), *125*, *264*
nv_dnorm *(*nv_normal*)*, 155
nv_dpois *(*nv_poisson*)*, 166
nv_drop *(*nv_squeeze*)*, 205
nv_dtype_constructors, 109
nv_dunif *(*nv_uniform*)*, 220
nv_eigh, 112
nv_eigh(), *269*
nv_empty *(*AnvlArray*)*, 14
nv_empty(), *15*
nv_empty_like *(*AnvlArray*)*, 14
nv_eq, 113
nv_eq(), *270*
nv_erf, 114
nv_erf(), *271*
nv_erf_inv, 114
nv_erf_inv(), *272*
nv_erfc, 115
nv_erfc(), *273*
nv_exp, 116
nv_exp(), *274*
nv_expm1, 117
nv_expm1(), *275*
nv_exponential, 118
nv_extract_diag, 119
nv_extract_diag(), *215*
nv_eye, 120
nv_eye_like *(*nv_eye*)*, 120
nv_fill, *17*, 121
nv_fill(), *57*, *277*
nv_fill_like *(*nv_fill*)*, 121
nv_flatten, 123
nv_flatten(), *83*, *91*, *214*, *226, 227*, *244*
nv_float32 *(*nv_dtype_constructors*)*, 109
nv_float64 *(*nv_dtype_constructors*)*, 109
nv_floor, 123
nv_floor(), *219*, *278*
nv_floor_div, 124
nv_floor_div(), *108*, *152*
nv_gamma, 125
nv_ge, 126
nv_ge(), *281*
nv_get_rng_state *(*nv_set_seed*)*, 194
nv_gt, 127
nv_gt(), *282*
nv_if, 128
nv_if(), *47*, *129*, *283*
nv_ifelse, 129
nv_ifelse(), *128*, *142*, *285*
nv_int16 *(*nv_dtype_constructors*)*, 109
nv_int32 *(*nv_dtype_constructors*)*, 109
nv_int64 *(*nv_dtype_constructors*)*, 109
nv_int8 *(*nv_dtype_constructors*)*, 109
nv_inv, 130
nv_iota, *17*, 131
nv_iota(), *51*, *138*, *192*, *286*
nv_iota_like *(*nv_iota*)*, 131
nv_is_finite, 132
nv_is_finite(), *133, 134*, *287*
nv_is_infinite, 133
nv_is_infinite(), *134*
nv_is_nan, 134
nv_is_nan(), *133*
nv_le, 135
nv_le(), *288*

nv_lgamma, 136
nv_lgamma(), *125*, *288*
nv_linspace, 137
nv_linspace(), *132*, *192*
nv_linspace_like *(*nv_linspace*)*, 137
nv_log, 138
nv_log(), *139*, *141*, *289*
nv_log10, 139
nv_log10(), *141*
nv_log1p, 140
nv_log1p(), *290*
nv_log2, 141
nv_log2(), *139*
nv_lower_tri, 142
nv_lower_tri(), *218*, *224*
nv_lower_tri_like *(*nv_lower_tri*)*, 142
nv_lt, 143
nv_lt(), *291*
nv_lu, 144
nv_lu(), *293*
nv_matmul, 145
nv_matmul(), *98, 99*, *213*, *266*
nv_matrix *(*AnvlArray*)*, 14
nv_max, 146
nv_max(), *146*, *178*, *226*, *294*
nv_mean, 147
nv_mean(), *148*, *190*, *225*
nv_median, 148
nv_median(), *149*, *177*, *204*, *330*
nv_min, 150
nv_min(), *150*, *178*, *227*, *295*
nv_mod, 151
nv_mod(), *125*, *179, 180*, *312*
nv_mul, 152
nv_mul(), *296*
nv_ne, 153
nv_ne(), *297*
nv_negate, 154
nv_negate(), *298*
nv_normal, *62*, 155, *179*, *183*, *185, 186*, *195*, *221*
nv_not, 158
nv_not(), *299*
nv_or, 159
nv_or(), *300*
nv_order, 160
nv_order(), *204*, *330*
nv_outer, 161
nv_pad, 162
nv_pad(), *301*
nv_pbinom *(*nv_binomial*)*, 80
nv_pexp *(*nv_exponential*)*, 118
nv_plogis, 163
nv_plogis(), *302*
nv_pmax, 164
nv_pmax(), *303*
nv_pmin, 165
nv_pmin(), *304*
nv_pnorm *(*nv_normal*)*, 155
nv_poisson, 166
nv_popcnt, 168
nv_popcnt(), *305*
nv_pow, 169
nv_pow(), *306*
nv_ppois *(*nv_poisson*)*, 166
nv_print, 170
nv_print(), *307*
nv_prod, 171
nv_prod(), *171*, *308*
nv_promote_to_common, 172
nv_promote_to_common(), *25*, *39*, *347*
nv_psigamma, 173
nv_psigamma(), *310*
nv_punif *(*nv_uniform*)*, 220
nv_qbinom *(*nv_binomial*)*, 80
nv_qexp *(*nv_exponential*)*, 118
nv_qnorm *(*nv_normal*)*, 155
nv_qpois *(*nv_poisson*)*, 166
nv_qr, 174
nv_qr(), *311*
nv_quantile, 175
nv_quantile(), *149*, *176*
nv_qunif *(*nv_uniform*)*, 220
nv_range, 177
nv_rbind *(*nv_bind*)*, 79
nv_rbinom, 178
nv_rbinom(), *62*, *82*, *157*, *183*, *185, 186*, *195*, *221*
nv_read *(*nv_save*)*, 187
nv_read(), *16*, *193*
nv_remainder, 179
nv_remainder(), *152*, *313*
nv_reshape, 180
nv_reshape(), *206*, *223*, *314*
nv_rev, 181
nv_rev(), *315*

nv_rng_state, 182
nv_rng_state(), *62*, *157*, *179*, *185, 186*, *195*, *221*
nv_rnorm *(*nv_normal*)*, 155
nv_rnorm(), *157*, *318*
nv_round, 183
nv_round(), *219*, *316*
nv_rsqrt, 184
nv_rsqrt(), *317*
nv_runif *(*nv_uniform*)*, 220
nv_runif(), *318*
nv_sample, 185
nv_sample(), *62*, *157*, *179*, *183*, *186*, *195*, *221*
nv_sample_int, 186
nv_sample_int(), *62*, *157*, *179*, *183*, *185*, *195*, *221*
nv_save, 187
nv_save(), *16*, *193*
nv_scalar, *15*, *111*
nv_scalar *(*AnvlArray*)*, 14
nv_scalar_like *(*AnvlArray*)*, 14
nv_scan, 188
nv_scan(), *47*, *62*, *228*, *320*
nv_sd, 189
nv_sd(), *225*
nv_select, 190
nv_select(), *175*
nv_seq, *17*, 191
nv_seq(), *51*, *132*, *138*
nv_seq_like *(*nv_seq*)*, 191
nv_serialize, *17*, 193
nv_serialize(), *16*, *187*
nv_set_rng_state *(*nv_set_seed*)*, 194
nv_set_seed, 194
nv_set_seed(), *62*, *156, 157*, *179*, *182, 183*, *185, 186*, *221*, *317, 318*, *353*
nv_shift_left, 195
nv_shift_left(), *324*
nv_shift_right_arithmetic, 196
nv_shift_right_arithmetic(), *325*
nv_shift_right_logical, 197
nv_shift_right_logical(), *326*
nv_sign, 198
nv_sign(), *327*
nv_sin, 199
nv_sin(), *201*, *327*
nv_sinh, 200
nv_sinh(), *328*
nv_sinpi, 201
nv_sinpi(), *98*, *212*
nv_solve, 201
nv_solve(), *89*, *104*, *106*, *130*, *217*, *249*, *341*
nv_sort, 203
nv_sort(), *149*, *160*, *177*, *214, 215*, *330*
nv_sqrt, 204
nv_sqrt(), *331*
nv_squeeze, 205
nv_squeeze(), *223*
nv_static_slice, 206
nv_sub, 207
nv_sub(), *333*
nv_subset *(*[.AnvlArray*)*, 10
nv_subset(), *13*, *191*, *206*, *267*, *280*, *323*, *332*
nv_subset_assign *(*[<-.AnvlArray*)*, 12
nv_subset_assign(), *11*, *268*, *280*, *323*
nv_sum, 208
nv_sum(), *148*, *208*, *334*
nv_svd, 209
nv_svd(), *336*
nv_tan, 210
nv_tan(), *212*, *337*
nv_tanh, 211
nv_tanh(), *337*
nv_tanpi, 212
nv_tanpi(), *98*, *201*
nv_tcrossprod, 213
nv_tcrossprod(), *99*
nv_top_k, 214
nv_top_k(), *204*, *330*, *338, 339*
nv_trace, 215
nv_trace(), *120*
nv_transpose *(*nv_aperm*)*, 72
nv_triangular_solve, 216
nv_triangular_solve(), *202*
nv_tril, 217
nv_tril(), *142*, *218*
nv_triu, 218
nv_triu(), *218*, *223, 224*
nv_trunc, 219
nv_uint16 *(*nv_dtype_constructors*)*, 109
nv_uint32 *(*nv_dtype_constructors*)*, 109
nv_uint64 *(*nv_dtype_constructors*)*, 109
nv_uint8 *(*nv_dtype_constructors*)*, 109
nv_uniform, *62*, *157*, *179*, *183*, *185, 186*, *195*, 220
nv_unserialize *(*nv_serialize*)*, 193

nv_unserialize(), *16*, *187*
nv_unsqueeze, 222
nv_unsqueeze(), *206*
nv_upper_tri, 223
nv_upper_tri(), *142*, *218*
nv_upper_tri_like *(*nv_upper_tri*)*, 223
nv_var, 224
nv_var(), *190*
nv_which_max, 225
nv_which_max(), *204*, *227*, *342*
nv_which_min, 227
nv_which_min(), *204*, *226*, *343*
nv_while, 228
nv_while(), *47*, *62*, *82*, *189*, *344*
nv_xor, 229
nv_xor(), *346*

peek_dtype, 230
peek_dtype(), *21*, *25*, *38, 39*
pjrt::as_pjrt_device(), *18*
pjrt::await(), *29*
pjrt::install_pjrt(), *50*
pjrt::map_tree(), *63*
pjrt::pjrt_buffer(), *18*
pjrt::pjrt_compile(), *18*, *53*, *353*
pjrt::platform(), *231*
pjrt::pmap_tree(), *63*
PJRTBuffer, *231*
PJRTDevice, *15*, *36*, *111*, *121, 122*, *131*, *137*, *142*, *187*, *192, 193*, *224*, *276*, *285*
platform *(*platform.AnvlArray*)*, 231
platform(), *16*
platform.AnvlArray, 231
pmap_tree *(*map_tree*)*, 63
prim_abs, 232
prim_abs(), *66*
prim_acos, 233
prim_acos(), *66*
prim_acosh, 234
prim_acosh(), *67*
prim_add, 235
prim_add(), *68*
prim_all, 236
prim_all(), *69*
prim_and, 237
prim_and(), *70*
prim_any, 238
prim_any(), *72*
prim_asin, 239
prim_asin(), *74*
prim_asinh, 240
prim_asinh(), *75*
prim_atan, 241
prim_atan(), *75*
prim_atan2, 242
prim_atan2(), *76*
prim_atanh, 243
prim_atanh(), *77*
prim_bitcast_convert, 244
prim_bitcast_convert(), *83*
prim_broadcast_in_axes, 245
prim_broadcast_in_axes(), *86*
prim_cbrt, 246
prim_cbrt(), *87*
prim_ceiling, 247
prim_ceiling(), *88*
prim_chol, 248
prim_chol(), *89*
prim_clamp, 249
prim_clamp(), *90*
prim_concatenate, 250
prim_concatenate(), *91*
prim_convert, 251
prim_convert(), *96*
prim_convolution, 252
prim_convolution(), *92*, *94, 95*
prim_cos, 254
prim_cos(), *96*
prim_cosh, 255
prim_cosh(), *97*
prim_cummax, 256
prim_cummax(), *100*
prim_cummin, 257
prim_cummin(), *101*
prim_cumprod, 258
prim_cumprod(), *102*
prim_cumsum, 259
prim_cumsum(), *103*
prim_custom_gradient, 260
prim_custom_gradient(), *350*
prim_digamma, 263
prim_digamma(), *108*
prim_div, 264
prim_div(), *109*
prim_dot_general, 265
prim_dot_general(), *145*
prim_dynamic_slice, 266

prim_dynamic_slice(), *267*, *268*, *331*, *332*
prim_dynamic_update_slice, 267
prim_dynamic_update_slice(), *267*
prim_eigh, 269
prim_eigh(), *112*
prim_eq, 270
prim_eq(), *113*
prim_erf, 271
prim_erf(), *114*
prim_erf_inv, 272
prim_erf_inv(), *115*
prim_erfc, 273
prim_erfc(), *116*
prim_exp, 274
prim_exp(), *116*
prim_expm1, 275
prim_expm1(), *117*
prim_fill, 276
prim_fill(), *43*, *122*, *276*
prim_floor, 277
prim_floor(), *124*
prim_gather, 278
prim_gather(), *267*, *268*, *320*, *321*, *323*, *332*
prim_ge, 281
prim_ge(), *126*
prim_gt, 282
prim_gt(), *127*
prim_if, 283
prim_if(), *128*, *284*, *285*, *355*
prim_ifelse, 284
prim_ifelse(), *21*, *129*, *283*
prim_iota, 285
prim_iota(), *43*, *51*, *132*, *142*, *192*, *224*
prim_is_finite, 286
prim_is_finite(), *132*
prim_le, 287
prim_le(), *135*
prim_lgamma, 288
prim_lgamma(), *136*
prim_log, 289
prim_log(), *139*
prim_log1p, 290
prim_log1p(), *140*
prim_lt, 291
prim_lt(), *143*
prim_lu, 292
prim_lu(), *104*, *106*, *144*, *202*
prim_max, 293
prim_max(), *147*, *312*
prim_min, 294
prim_min(), *151*
prim_mul, 295
prim_mul(), *153*
prim_ne, 296
prim_ne(), *154*
prim_negate, 297
prim_negate(), *154*
prim_not, 298
prim_not(), *158*
prim_or, 299
prim_or(), *159*
prim_pad, 300
prim_pad(), *162*
prim_plogis, 302
prim_plogis(), *163*
prim_pmax, 303
prim_pmax(), *164*
prim_pmin, 304
prim_pmin(), *165*
prim_popcnt, 305
prim_popcnt(), *168*
prim_pow, 306
prim_pow(), *169*
prim_print, 307
prim_print(), *170*
prim_prod, 308
prim_prod(), *172*
prim_psigamma, 309
prim_psigamma(), *173*
prim_qr, 310
prim_qr(), *174*
prim_reduce, 311
prim_reduce(), *21*
prim_remainder, 312
prim_remainder(), *152*, *180*
prim_reshape, 313
prim_reshape(), *181*
prim_rev, 314
prim_rev(), *182*
prim_round, 315
prim_round(), *183*
prim_rsqrt, 316
prim_rsqrt(), *184*
prim_sample_bits, 317
prim_sample_bits(), *354*
prim_scan, 319

prim_scan(), *189*
prim_scatter, 320
prim_scatter(), *12*, *21*, *267*, *268*, *278–280*, *332*
prim_shift_left, 323
prim_shift_left(), *195*
prim_shift_right_arithmetic, 324
prim_shift_right_arithmetic(), *196*
prim_shift_right_logical, 325
prim_shift_right_logical(), *197*
prim_sign, 326
prim_sign(), *198*
prim_sin, 327
prim_sin(), *199*
prim_sinh, 328
prim_sinh(), *200*
prim_sort, 329
prim_sort(), *21*, *149*, *160*, *204*, *339*
prim_sqrt, 330
prim_sqrt(), *205*
prim_static_slice, 331
prim_static_slice(), *191*, *206*, *266*, *267*
prim_sub, 333
prim_sub(), *207*
prim_sum, 334
prim_sum(), *209*, *312*
prim_svd, 335
prim_svd(), *210*
prim_tan, 336
prim_tan(), *211*
prim_tanh, 337
prim_tanh(), *211*
prim_top_k, 338
prim_top_k(), *215*
prim_transpose, 339
prim_transpose(), *73*
prim_triangular_solve, 340
prim_triangular_solve(), *217*
prim_which_max, 342
prim_which_max(), *343*
prim_which_min, 343
prim_which_min(), *342*
prim_while, 344
prim_while(), *21*, *229*, *260*, *261*, *320*, *355*
prim_xor, 345
prim_xor(), *229*
prod.AnvlArray *(*nv_prod*)*, 171
promoted to a common data type, *79*, *91–94*, *99*, *129*, *145*, *161*, *173*, *202*, *213*, *216*
promotion rule, *20*
promotion_common *(*promotion_rule*)*, 346
promotion_common(), *172*
promotion_dtype *(*promotion_rule*)*, 346
promotion_grouped *(*promotion_rule*)*, 346
promotion_grouped(), *347*
promotion_like *(*promotion_rule*)*, 346
promotion_rdata_common *(*promotion_rule*)*, 346
promotion_rdata_common(), *20*
promotion_rule, *21*, *25*, 346
PromotionRule, *347*

qr.AnvlArray *(*nv_qr*)*, 174
quantile.AnvlArray *(*nv_quantile*)*, 175
quickr_device, *15*, *36*, *106*, *111*, *121*, *122*, *131*, *137*, *142*, *192*, *224*, *276*, *285*, 348
quickr_device(cpu), *19*

range.AnvlArray *(*nv_range*)*, 177
raw, *28*, *193*
rbind.AnvlArray *(*nv_bind*)*, 79
RData, *38*, *40*, *78*, *170*, *230*, 349, *356*
rev.AnvlArray *(*nv_rev*)*, 181
RTree, *47*, *128*, *228*, *261*, *283*, *344*, *357*
rule_reverse, 350
rule_reverse(), *44*, *261*

saveRDS(), *16*
sd(), *190*
seq(), *191*
set.seed(), *194*
Shape, 352
shape, 351
shape(), *16*, *30*, *37*, *230*, *352*
sign(), *327*
sign.AnvlArray *(*nv_sign*)*, 198
sin(), *327*
sin.AnvlArray *(*nv_sin*)*, 199
sinh(), *328*
sinh.AnvlArray *(*nv_sinh*)*, 200
sinpi.AnvlArray *(*nv_sinpi*)*, 201
solve.AnvlArray *(*nv_solve*)*, 201
sort.AnvlArray *(*nv_sort*)*, 203
sqrt(), *331*
sqrt.AnvlArray *(*nv_sqrt*)*, 204

stablehlo, 353
stablehlo(), *18*, *33*, *34*, *356*, *357*
stablehlo::Func, *353*, *354*
stablehlo::repr(), *353*
stablehlo::Shape, *51*, *57*, *78*, *349*
stablehlo::Shape(), *352*
stats::median(), *149*
stats::pbeta(), *81*
stats::pgamma(), *167*
stats::plogis(), *163*, *302*
stats::quantile(), *177*
subgraphs, 355
sum.AnvlArray *(*nv_sum*)*, 208

t(), *340*
t.AnvlArray *(*nv_aperm*)*, 72
tan(), *337*
tan.AnvlArray *(*nv_tan*)*, 210
tanh(), *337*
tanh.AnvlArray *(*nv_tanh*)*, 211
tanpi.AnvlArray *(*nv_tanpi*)*, 212
tcrossprod.AnvlArray *(*nv_tcrossprod*)*, 213
to_abstract, 355
to_abstract(), *78*
trace_fn, 356
trace_fn(), *45–47*, *353*, *354*
trigamma.AnvlArray *(*nv_psigamma*)*, 173
trunc.AnvlArray *(*nv_trunc*)*, 219

utils::install.packages(), *50*

value_and_gradient *(*gradient*)*, 41
value_and_gradient(), *44*
var(), *225*

with_backend *(*local_backend*)*, 58
with_backend(), *9*, *14*, *54*, *61*
with_default_device *(*local_default_device*)*, 59
with_default_device(), *9*
with_default_dtypes *(*local_default_dtypes*)*, 60
with_default_dtypes(), *35*
with_dtypes, 357
with_dtypes(), *35*, *54*
with_nv_rng_state *(*local_nv_rng_state*)*, 62
with_nv_rng_state(), *156*, *179*, *182*, *185*, *186*, *221*, *318*
with_nv_seed *(*nv_set_seed*)*, 194
withr::defer(), *33*

xlamisc::as_array(), *26*
xlamisc::as_dtype(), *28*
xlamisc::as_raw(), *28*
xlamisc::DataType, *15*, *28*, *31*, *51*, *57*, *78*, *230*, *347*
xlamisc::device(), *36*
xlamisc::dtype(), *38*
xlamisc::dtype_category(), *38*
xlamisc::is_dtype(), *52*
xlamisc::naxes(), *64*
xlamisc::shape(), *351*