Package: imply (via r-universe)

October 15, 2026

Version 0.1.0

Date 2026-09-05

Title Efficiently Apply Functions to Images

Description Infrastructure for handling generalised two and three dimensional images, which may hold multiple values at each spatial location, and efficiently applying functions to them. Dense, compacted and sparse pixel/voxel representations are supported, with one core engine for processing them all. Parallelism is supported via the 'parallel' package, 'libdispatch' and 'OpenMP'.

Imports Rcpp, S7, methods

LinkingTo Rcpp, RcppArray

Suggests tinytest

Encoding UTF-8

License BSD_3_clause + file LICENCE

URL https://github.com/jonclayden/imply

BugReports https://github.com/jonclayden/imply/issues

Config/roxygen2/version 8.1.0

Config/roxygen2/markdown TRUE

NeedsCompilation yes

Author Jon Clayden [cre, aut] (ORCID: <https://orcid.org/0000-0002-6608-0619>)

Maintainer Jon Clayden <code@clayden.org>

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

Date/Publication 2026-09-15 10:40:16 UTC

RemoteUrl https://github.com/cran/imply

RemoteRef HEAD

RemoteSha 9871647734320507f8d1abff3843555d37315c18

Contents

denseImage 2
geometry 3
imapply 5
imreduce 7
packedImage 8
parallelism 9
sparseImage 10

Index **12**

`denseImage` *Dense images*

Description

A dense image is an array carrying image geometry alongside it. It is an S7 class whose parent is the `array` S3 class, which means every storage mode works (double, integer, logical and complex all occur in practice, masks being logical). The data are still a plain R array, so no copy is needed to pass them to compiled code, and the geometry is validated whenever it is set rather than only on construction.

Usage

```
denseImage(
  .data,
  voxelSize = NULL,
  worldTransform = NULL,
  spatial = NULL,
  spaceUnit = NULL,
  timeUnit = NULL,
  template = NULL
)

isDenseImage(x)

asDense(x, ...)
```

Arguments

`.data`	An array, or any atomic vector, which is treated as one-dimensional.
`voxelSize`	Voxel size, one per spatial dimension.
`worldTransform`	A 4x4 affine transform mapping voxel to world coordinates. Decomposed into rotation/translation and voxel size on assignment; see geometry.
`spatial`	The number of leading dimensions that index location rather than the value held at each location. Defaults to three, or the dimensionality if that is smaller.

spaceUnit, timeUnit	Units of measurement.
template	An image to take unspecified geometry from.
x	An image.
...	Further arguments to denseImage().

Details

Properties are stored as ordinary attributes, so compiled code reads them without needing to know anything about S7.

Value

An object of S7 class denseImage representing a dense image, with properties corresponding to the arguments listed above.

Note

S7 qualifies a class name with its package, so the class attribute is "imply::denseImage" and inherits(x, "denseImage") is FALSE. Use isDenseImage() rather than testing the class directly.

---

geometry *Image geometry*

Description

Accessors for the geometry of an image: the number of spatial dimensions, the voxel size and the placement of the image in world space.

Usage

```
isImage(x)

spatial(x)

voxelSize(x)

voxelSize(x) <- value

worldTransform(x)

worldTransform(x) <- value

toVoxel(points, x, type = "world", round = "none", bounds = NULL)

fromVoxel(points, x, type = "world")
```

Arguments

x	An image, or for worldTransform either an image or a 4x4 matrix.
value	A replacement value.
points	A matrix of points, one per row and three columns. Where type is "voxel" (the default convention for fromVoxel()'s input and toVoxel()'s output), these are one-based, as for x[i, j, k].
type	The coordinate convention of points: "voxel" (one-based), "scaled" (millimetres from the one-based origin, ignoring rotation) or "world" (fully transformed).
round	Rounding strategy: "none", "conventional" for nearest neighbour, or "probabilistic" for a stochastic nearest neighbour with probability proportional to proximity.
bounds	Optional image extents, used only by probabilistic rounding to avoid selecting a location beyond the end of the image.

Details

These are deliberately free of any dependency on a file format, and of any assumption that geometry is anatomical: an image with no meaningful spatial interpretation is free to leave it at the identity default and never think about it again.

Voxel size and world placement are stored, and can be set, independently of one another, unlike a NIfTI xform, which bakes voxel size into the same matrix that carries rotation and translation and so has to be kept in sync by hand. Here, voxelSize<- never touches rotation or translation, and the stored placement is always a rigid transform (rotation or reflection plus translation; no scale and no shear) so that the two cannot drift out of agreement.

worldTransform() composes the two into the single 4x4 affine that other packages expect. Setting it back decomposes the matrix into rotation and scale; a matrix that doesn't decompose that way (i.e. one with genuine shear) is rejected rather than silently mangled. In practice a sheared NIfTI sform usually means the field is being used to carry an affine registration or normalisation result (to Talairach or MNI space, say) rather than to describe voxel storage geometry, which is a different kind of information than this package models.

toVoxel() and fromVoxel() use one-based voxel coordinates, matching x[i, j, k] indexing. The affine itself, and the rest of the package's C++ API, is zero-based throughout.

Value

isImage() returns a Boolean value indicating whether its argument is one of the package's image types. spatial() returns the index of the last spatial dimension. voxelSize() returns a vector of sizes in each spatial dimension. worldTransform() returns a numeric affine transform matrix. toVoxel() and fromVoxel() return matrices of transformed points, one per row. The assignment functions are called for their side-effects.

imapply *Apply a function over an image*

Description

imapply() is a memory-efficient analogue of base::apply(). It gives the same answer, but where apply() permutes the whole array into a fresh copy before looping, imapply() gathers each sub-array directly through the stride vector. Peak memory is therefore the input plus the result, rather than twice the input plus the result, which matters once an image is larger than a comfortable fraction of memory.

Usage

```
imapply(x, margin, fun, ..., simplify = TRUE, threads = NULL, progress = FALSE)

voxelApply(
  x,
  fun,
  ...,
  mask = NULL,
  fill = 0,
  simplify = TRUE,
  threads = NULL,
  progress = FALSE
)

lineApply(
  x,
  fun,
  ...,
  axis = 1L,
  simplify = TRUE,
  threads = NULL,
  progress = FALSE
)

sliceApply(
  x,
  fun,
  ...,
  axis = 3L,
  simplify = TRUE,
  threads = NULL,
  progress = FALSE
)
```

Arguments

x	An image or a plain array.
margin	The dimensions to retain, as for base::apply().
fun	A function to apply.
...	Further arguments to fun.
simplify	Whether to simplify the result to an array where possible.
threads	Number of threads to use, or NULL to consult getOption("imply.threads"). See parallelism.
progress	FALSE for none, TRUE for a text progress bar showing the percentage complete and the rate in voxels per second, or a function of (done, total).
mask	For voxelApply(), a logical array over the spatial dimensions, a sparse image whose mask is to be used, or NULL for none. Locations outside it are not visited at all.
fill	The value given to locations outside mask.
axis	For lineApply(), the axis lines run along; for sliceApply(), the axis slices cut across.

Details

The remaining functions differ only in how much of the *space* they hand to fun at a time: a single location for voxelApply(), a one-dimensional line for lineApply(), and a two-dimensional slice for sliceApply().

In every case the values held at each location travel with the unit. So for an image with a time series at each location, voxelApply() passes one series, lineApply() passes a line's worth of series, and sliceApply() passes a slice's worth:

```
dim(x) = 4 x 5 x 6 x 10, spatial = 3

voxelApply(x, f)           f sees  10 values
lineApply(x, f, axis = 1)  f sees  4 x 10
sliceApply(x, f, axis = 3) f sees  4 x 5 x 10
```

axis always names a spatial dimension, but means what the name of each function implies: a line *runs along* its axis, whereas a slice is *cut across* its axis. Use imapply() directly to iterate over a non-spatial dimension, such as applying a function to each volume in a time series.

Value

For imapply(), as base::apply(). For voxelApply(), an image when the function returns a single value per location, otherwise an array.

`imreduce` *Built-in reductions*

Description

`imreduce()` computes a summary over the margins of an image without calling back into R for each one. That matters for two reasons: there is no per-call interpreter overhead, and (since nothing touches R) the loop can run on worker threads. `imapply()` recognises the equivalent base functions and routes calls here automatically where the answer has been shown to be the same.

Usage

```
imreduce(x, margin, what, na.rm = FALSE, threads = NULL)
```

Arguments

`x`	An image, or any array.
`margin`	The dimensions to retain, as for `base::apply()`.
`what`	The reduction: one of `"sum"`, `"mean"`, `"min"`, `"max"`, `"range"`, `"prod"`, `"var"`, `"sd"`, `"which.min"`, `"which.max"`, `"any"`, `"all"` or `"countNA"`.
`na.rm`	Whether to ignore missing values.
`threads`	Number of threads, or `NULL` for the default. See parallelism.

Details

Accumulation is done in double mode whatever the values are stored as, so the answer does not depend on the storage type and error does not compound with the number of values.

Note that for the arithmetic reductions this may not agree with the base equivalent to the last bit: R accumulates `sum()` and `mean()` in long double, which on some platforms is wider than double. Differences are of the order of `1e-16`. `imapply()` therefore routes automatically only where the answers are provably identical (`min`, `max`, `range`, `which.min` and `which.max`) and leaves the rest to be asked for deliberately.

`"var"` is the variance of the values, as `var(as.vector(x))` would give. It is not `stats::var()` applied to a sub-array, which for a matrix returns a covariance matrix.

Value

A vector or array, shaped over `margin` as `base::apply()` would shape it. `"range"` yields two values per call, so it gains a leading dimension.

packedImage *Narrow storage types*

Description

R has no single-precision floating-point type, so a large image held as `double` may cost twice the memory it needs, and potentially roughly twice the time. Raw MRI is commonly 16-bit integer.

Usage

```
packedImage(
  values,
  storageType,
  dims,
  slope = 1,
  intercept = 0,
  spatial = NULL,
  voxelSize = NULL,
  worldTransform = NULL,
  spaceUnit = NULL,
  timeUnit = NULL,
  template = NULL
)

isPackedImage(x)

asPacked(x, type = "float32", slope = NULL, intercept = NULL, ...)

storageType(x)
```

Arguments

`values`	A raw vector holding the packed values.
`dims`, `spatial`, `voxelSize`, `worldTransform`, `spaceUnit`, `timeUnit`	Image geometry, as for `denseImage()`.
`slope`, `intercept`	Scaling applied to stored values. Chosen automatically when not given.
`template`	An image to take unspecified geometry from.
`x`	An image or array.
`type`, `storageType`	One of `"int8"`, `"uint8"`, `"int16"`, `"uint16"`, `"int32"` or `"float32"`.
`...`	Further arguments to `denseImage()`.

Details

Recognising this, a packed image stores its values using a narrow data type, with optional affine scaling, so that an integer type can carry a range it could not otherwise hold: `value = stored * slope + intercept`. When a scaling is needed it is chosen automatically to map the data across the whole of the type's range.

The values live in a raw vector, so they are garbage-collected, serialise, and survive a save and load like any other R object.

Value

An object of S7 class `packedImage` representing an image using a narrow, packed data representation, with properties corresponding to the arguments listed above.

Note

Missing values can only be carried by `float32`; packing data containing `NA` to an integer type is refused rather than silently losing it. Note that `NA` and `NaN` are not distinguished once packed, since the payload that separates them does not survive the narrowing.

`parallelism` *Parallelism*

Description

Work is parallelised in one of two ways, according to what is being run.

Usage

```
parallelBackend()

canFork()

resolveThreads(threads = NULL)
```

Arguments

`threads`	A thread count, or `NULL` to consult `getOption("imply.threads")`. Leaving both unset runs serially.

Details

Compiled kernels run concurrently in process, using Grand Central Dispatch or OpenMP where one or other is available. Which of these was compiled in is reported by `parallelBackend()`.

An R function cannot be called from a worker thread, because R is single-threaded and its interpreter is not reentrant. Applying an R function is therefore parallelised by forking the R session instead, dividing the calls between workers. Forked workers share the image through copy-on-write, so

nothing large is duplicated. Forking is unavailable on Windows, where such work runs serially; the alternative, a socket cluster, would copy the whole image to every worker and so defeat the purpose.

The number of threads may be given per call, or set globally with `options(imply.threads = n)`. Leaving both unset means serial, for compiled kernels and R functions alike. Multi-core use is opt-in.

Value

`parallelBackend()` returns `"libdispatch"`, `"openmp"` or `"none"`. `canFork()` reports whether R functions can be run in parallel.

`sparseImage` *Sparse images*

Description

A sparse image stores only those spatial locations that hold data. Sparsity is over *locations*, not over individual values: a location is either present, in which case the whole vector of values held there is stored, or absent, in which case every one of them is implicitly zero. That generally matches the intent, and it keeps the stored values contiguous.

Usage

```
sparseImage(
  mask,
  values,
  dim,
  spatial = NULL,
  voxelSize = NULL,
  worldTransform = NULL,
  spaceUnit = NULL,
  timeUnit = NULL,
  template = NULL
)

isSparseImage(x)

asSparse(x, ...)

maskedMatrix(x)

sparseness(x)

mask(x)
```

Arguments

`mask`	A raw vector of one bit per location, or a logical vector.
`values`	Packed values, in location order.
`dim`, `spatial`, `voxelSize`, `worldTransform`, `spaceUnit`, `timeUnit`	Image geometry, as for `denseImage()`.
`template`	An image to take unspecified geometry from.
`x`	An image or array.
`...`	Further arguments to `sparseImage()`.

Details

The mask is one bit per location and the values are packed in location order, so finding a value costs a table lookup and a bit count rather than a search. A coordinate list, the other common representation, would store three or four indices alongside every value, and may cost more memory than saves at typical densities.

A location holding `NA` is kept, since `NA` is not zero. Packing an image therefore loses nothing.

Value

An object of S7 class `sparseImage` representing a sparse image, with properties corresponding to the arguments listed above.

Index

asDense *(*denseImage*)*, 2
asPacked *(*packedImage*)*, 8
asSparse *(*sparseImage*)*, 10

base::apply(), *5–7*

canFork *(*parallelism*)*, 9

denseImage, 2
denseImage(), *8*, *11*

fromVoxel *(*geometry*)*, 3

geometry, *2*, 3

imapply, 5
imreduce, 7
isDenseImage *(*denseImage*)*, 2
isImage *(*geometry*)*, 3
isPackedImage *(*packedImage*)*, 8
isSparseImage *(*sparseImage*)*, 10

lineApply *(*imapply*)*, 5

mask *(*sparseImage*)*, 10
maskedMatrix *(*sparseImage*)*, 10

packedImage, 8
parallelBackend *(*parallelism*)*, 9
parallelism, *6, 7*, 9

resolveThreads *(*parallelism*)*, 9

sliceApply *(*imapply*)*, 5
sparseImage, 10
sparseness *(*sparseImage*)*, 10
spatial *(*geometry*)*, 3
stats::var(), *7*
storageType *(*packedImage*)*, 8

toVoxel *(*geometry*)*, 3

voxelApply *(*imapply*)*, 5
voxelSize *(*geometry*)*, 3
voxelSize<- *(*geometry*)*, 3

worldTransform *(*geometry*)*, 3
worldTransform<- *(*geometry*)*, 3