Package 'imply'

November 20, 2026

Version 0.2.0

Date 2026-11-20

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'.

Depends R (>= 4.2.0)

Imports Rcpp, S7 (>= 1.0.0)

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 8.1.1

Config/roxygen2/markdown TRUE

NeedsCompilation yes

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

Maintainer Jon Clayden <code@clayden.org>

Repository CRAN

Date/Publication 2026-11-20 13:30:02 UTC

Contents

asMaskVector 2
denseImage 3
geometry 4
imapply 6
imreduce 8
packedImage 9
parallelism 10
readImageData 11
sparseImage 13
storageDescriptor 14
worldAxes 16

Index **18**

`asMaskVector` *Masks over spatial locations*

Description

A mask selects spatial locations, for example to restrict `voxelApply()` to a region of interest. It may be given in several forms, and this function reduces any of them to one logical value per location.

Usage

```
asMaskVector(mask, spatialDims)
```

Arguments

`mask`	A logical array, a numeric array in which non-zero values are selected, a sparse image whose own mask is used, or any other image, which is treated as a numeric array.
`spatialDims`	The spatial dimensions the mask must cover, or an image or imageGeometry to take them from.

Value

A logical vector with one element per spatial location.

`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,
  unit = NULL,
  geometry = 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 the number the geometry has, if one is given, or otherwise to three, or the dimensionality if that is smaller.
`unit`	The unit of measurement of voxel size and world coordinates.
`geometry`	An imageGeometry, or an image to take one from, giving whatever is not specified explicitly. Its grid must match the leading dimensions of the data.
`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 a `geometry` property holding its imageGeometry.

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

An image geometry describes the spatial grid an image is sampled on, without any of its data: the extent of the grid, the voxel size, the placement of the grid in world space and the unit of measurement. Every image holds one, and any of the functions here may be given either an image or a bare geometry. A bare geometry is useful in its own right, to describe a space that no image has been created in yet, such as the target grid of a resampling.

Usage

```
imageGeometry(
  dims = integer(0),
  voxelSize = NULL,
  worldTransform = NULL,
  unit = NULL
)

isImageGeometry(x)

isImage(x)

geometry(x)

geometry(x) <- value

spatial(x)

voxelSize(x)

voxelSize(x) <- value

worldTransform(x)

worldTransform(x) <- value
```

```
centre(x)

center(x)

extent(x)

sameGeometry(x, y, tolerance = sqrt(.Machine$double.eps))

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

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

Arguments

dims	The extent of the grid, one per spatial dimension.
voxelSize	Voxel size, one per spatial dimension.
worldTransform	A 4x4 affine transform mapping (zero-based) voxel to world coordinates, from which the voxel size is also taken unless it is given separately.
unit	The unit of measurement of voxel size and world coordinates.
x, y	Images or geometries. Anything else with a dimension attribute is treated as having unit voxels at the origin, with the leading three dimensions (or all of them, if fewer) spatial. worldTransform() also accepts a bare 4x4 matrix, which is validated and returned.
value	A replacement value.
tolerance	Numerical tolerance for comparing world transforms.
points	A matrix of points, one per row and one to three columns. Missing columns are taken to be zero. 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. The stored placement is always a rigid transform: rotation or reflection plus translation; no scale and no shear. worldTransform() composes the two into the single 4x4 affine that some 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.

`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

`imageGeometry()` and `geometry()` return an object of S7 class `imageGeometry`, with properties `dims`, `voxelSize`, `orientation` (the rigid part of the world transform) and `unit`. `isImage()` and `isImageGeometry()` return a Boolean value indicating whether their argument is one of the package's image types, or a geometry, respectively. `spatial()` returns the number of spatial dimensions. `voxelSize()` returns a vector of sizes in each spatial dimension. `worldTransform()` returns a numeric affine transform matrix. `centre()` returns the world coordinates of the centre of the grid, always as three values; `center()` is an alias. `extent()` returns the physical size of the grid along each spatial axis. `sameGeometry()` returns a Boolean value, which is `TRUE` if the two grids are the same size and are placed identically in world space, up to `tolerance`; units are compared only if both are known. `toVoxel()` and `fromVoxel()` return matrices of transformed points, one per row. Voxel coordinates have as many columns as `points`, and world or scaled coordinates always have three. 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 subarray 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 or numeric array over the spatial dimensions, an image, or NULL for none; see asMaskVector(). 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()`, `lineApply()` and `sliceApply()` applied to an image, a dense image when the function returns a single value per location, line or slice, respectively, and otherwise as for `imapply()`. The geometry of the result is that of `x` with the axes traversed removed, each remaining location being placed at the centre of the line or slice it summarises, so `centre()` is unchanged. A plain array in gives a plain array out.

---

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,
  unit = NULL,
  geometry = NULL,
  layout = NULL
)

isPackedImage(x)

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

storageType(x)
```

Arguments

`values`	A raw vector holding the packed values.
`dims`	The full dimensions of the image.
`slope, intercept`	Scaling applied to stored values. Chosen automatically when not given.

`spatial`, `voxelSize`, `worldTransform`, `unit`, `geometry`
: Image geometry, as for `denseImage()`.

`layout`
: How the image's axes map onto the order of `values`, as described for `storageLayout()`. The default is the identity, meaning that `values` are in the image's own axis order.

`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.

A packed image can be indexed and assigned to like an array, reading and writing only the values concerned. Assigned values are stored under the image's existing type and scaling, and so rounded to its resolution; a value the scaling cannot reach, or a missing value in an integer type, is an error, since rescaling to fit would change every other value too. Use `asDense()`, or repack with `asPacked()`, to store such values.

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 the 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.

---

readImageData *Reading and writing image data*

Description

readImageData() reads the values of an image from a connection, as described by a storageDescriptor(), and returns an image. writeImageData() writes them. Neither knows anything about file formats, which are the business of packages built on this one: a format package parses any header or metadata, builds a descriptor and a geometry, and calls these to handle the data.

Usage

```
readImageData(
  con,
  descriptor,
  dims,
```

```
  geometry = NULL,
  as = c("auto", "dense", "sparse", "packed"),
  volumes = NULL,
  mask = NULL
)

writeImageData(image, con, descriptor)
```

Arguments

con	A connection, or the path to a file. A connection must be open for binary reading or writing. If its position can be queried, it is moved forwards to the descriptor's offset, or when writing, padded with zeros up to it; otherwise it is taken to be already there. A path is opened, read or written from the start, and closed.
descriptor	A storageDescriptor().
dims	The full dimensions of the image, in its own axis order.
geometry	An imageGeometry, or an image to take one from, for the spatial grid. By default the leading three dimensions (or all of them, if fewer) are spatial, with unit voxels at the origin.
as	The representation wanted: "auto", "dense", "sparse" or "packed".
volumes	Optionally, which volumes to read, as one-based indices into the values held at each location, taken together. The result then has the spatial dimensions followed by one dimension over the volumes read, in the order given. Only data stored a volume at a time can be read this way.
mask	Optionally, the spatial locations to keep, in any form asMaskVector() accepts. Only meaningful for a sparse result.
image	An image or array to write.

Details

Bytes are read through R connections, so compressed streams need nothing special: gzfile() reads gzipped and plain files alike. They are read in chunks, so the only large allocation is the result itself. A gzipped stream cannot be read backwards, so volumes are always read in the order they are stored, whatever order they are asked for in.

The image comes back dense, packed or sparse, according to the as argument. The default, "auto", gives a sparse image if a mask is given; otherwise a packed image if the stored type is one a packedImage can hold, keeping the file's narrow representation and scaling; and otherwise a dense image. The class returned therefore depends on the data, so code that relies on a specific class should ask for it.

A packed image keeps the stream's order, with the layout recording how its axes map onto it, so no data is moved; see reorient(). A dense image is always in its own axis order, so a stream in any other order is rearranged as it is read.

A sparse image keeps exactly the locations selected by mask, if one is given, and otherwise those where any value is non-zero. Only volume-ordered data can be read straight into sparse form, without the dense array ever existing; other layouts, and bit data, are read densely first.

Value

`readImageData()` returns an image. `writeImageData()` returns, invisibly, the descriptor as used, with any scaling it chose filled in, which is what a format package needs to write its header.

Examples

```
image <- denseImage(array(1:24, c(2, 3, 4)))
descriptor <- storageDescriptor("int16", endian = "big")
file <- tempfile()
writeImageData(image, file, descriptor)
readImageData(file, descriptor, dim(image))
```

---

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.

`storedValues()` returns the values exactly as stored, never copying, along with the location of each column as a linear index into the image's own spatial grid. The two differ only for a reoriented image.

Usage

```
sparseImage(
  mask,
  values,
  dim,
  spatial = NULL,
  voxelSize = NULL,
  worldTransform = NULL,
  unit = NULL,
  geometry = NULL,
  layout = NULL
)

isSparseImage(x)

asSparse(x, ...)

maskedMatrix(x)

storedValues(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`	The full dimensions of the image.
`spatial`, `voxelSize`, `worldTransform`, `unit`, `geometry`	Image geometry, as for `denseImage()`.
`layout`	How the image's axes map onto the stored order of locations, as described for `storageLayout()`. For a sparse image it may only reorder and reverse the spatial axes among themselves; the values held at each location are always stored in order. The default is the identity.
`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 location holding `NA` is kept, since `NA` is not zero.

A sparse image can be indexed and assigned to like an array. Assignment keeps the mask accurate: a location given a non-zero value is added, and one left holding only zeros is dropped. That takes a pass over the stored values each time, which suits occasional edits; to set many values, gather them and assign once, rather than one at a time in a loop. As for an array, values are promoted to a wider replacement type.

Value

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

`storageDescriptor` *Storage descriptors*

Description

A storage descriptor says how the values of an image sit in a stream of bytes: their type, byte order and position, any scaling applied to them, and the order of the image's axes in the stream. It deliberately says nothing about any particular file format. A format package parses a header into a descriptor, and `readImageData()` and `writeImageData()` do the rest.

Usage

```
storageDescriptor(
  type,
  endian = "little",
  offset = 0,
  slope = NULL,
  intercept = NULL,
  layout = NULL,
  bitOrder = "lsb"
)
```

Arguments

type	The stored data type, as above.
endian	The byte order of the stream: "little" or "big".
offset	The position of the first value, in bytes from the start of the stream.
slope, intercept	Scaling, as value = stored * slope + intercept. When reading, NULL means no scaling. When writing, NULL means that a scaling is chosen to use the whole range of an integer type, if the data need one.
layout	How the image's axes map onto the order of values in the stream, as described for storageLayout(). NULL means the stream holds the image in its own axis order, first axis fastest.
bitOrder	For "bit" data, whether the first value of each byte is its least or most significant bit.

Details

Types are named by size and signedness: "int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "float32", "float64", "complex64" and "complex128", plus "rgb24" and "rgba32" for colour and "bit" for one bit per value. The C-style names "char", "int", "float" and "double" are accepted too, as "int8", "int32", "float32" and "float64" respectively. C leaves the signedness of char to the platform, so it is fixed as signed here; use "uint8" for unsigned bytes.

When read, integer types of up to 32 bits become R integers unless scaled, and wider integers, floating-point types and scaled values become doubles. 64-bit integers beyond 2^53 cannot be held exactly as doubles and are refused. Colour values become integers from 0 to 255, with an extra trailing dimension of three or four channels. Bits become logical values. Scaling is not applied to complex or colour data.

Value

An object of S7 class storageDescriptor.

Examples

```
storageDescriptor("int16", endian = "big", offset = 352)
```

`worldAxes` *World axes and reorientation*

Description

Every image has an axis order and direction relative to world space. `worldAxes()` reports it as a signed integer for each spatial axis: which world axis it runs along, one to three, negated if its index increases in the negative direction. So `c(-1, 2, 3)` describes an image whose first index increases along the negative first world axis, and whose second and third increase along the second and third. The form is the same as a storage layout's, with world axes in place of storage axes.

Usage

```
worldAxes(x)

reorient(x, to = 1:3)

storageLayout(x)
```

Arguments

`x`	An image, or for `worldAxes()` also a geometry.
`to`	The world axes wanted, in the same form as `worldAxes()` returns: a signed integer per axis, naming each world axis at most once. The default puts every axis along its positive world axis, in order.

Details

Nothing here attaches a meaning to the world axes. Conventions that do, such as the anatomical codes used in neuroimaging, belong in packages built on this one.

`reorient()` permutes and reverses an image's spatial axes so that they run along the world axes given, updating the geometry to match, so that every voxel keeps its position in world space. Only the 48 axis-aligned orientations are reachable this way; whatever oblique rotation is left over stays in the geometry. For an image with fewer than three spatial dimensions, the entries for the world axes it does not span are ignored.

For packed and sparse images this is a change of *view* rather than of data: the image records how its axes map onto its storage, and indexing, `dim()`, the geometry and every apply function work in the view, so no values are moved or copied. A dense image is a plain R array, which base R indexes in memory order, so a dense image is always stored in view order and reorienting one copies its data.

`storageLayout()` returns the layout of an image: one entry per axis, giving the one-based rank of the storage axis it corresponds to (rank one varying fastest), negated where the image runs along that axis backwards. It is the identity for a dense image, and for any image that has not been reoriented or read from a file stored in another order.

Value

`worldAxes()` returns an integer vector with one element per spatial dimension, up to three. `reorient()` returns an image of the same class as `x`, with its axes permuted and reversed as needed.

Examples

```
image <- denseImage(array(1:24, c(2, 3, 4)), worldTransform = diag(c(-1, 1, 1, 1)))
worldAxes(image)
worldAxes(reorient(image))
```

Index

asDense *(*denseImage*)*, 3
asDense(), *10*
asMaskVector, 2
asMaskVector(), *7*, *12*
asPacked *(*packedImage*)*, 9
asSparse *(*sparseImage*)*, 13

base::apply(), *6–9*

canFork *(*parallelism*)*, 10
center *(*geometry*)*, 4
centre *(*geometry*)*, 4
centre(), *8*

denseImage, 3
denseImage(), *10*, *14*

extent *(*geometry*)*, 4

fromVoxel *(*geometry*)*, 4

geometry, *3*, 4
geometry<- *(*geometry*)*, 4
gzfile(), *12*

imageGeometry, *2–4*, *12*
imageGeometry *(*geometry*)*, 4
imapply, 6
imreduce, 8
isDenseImage *(*denseImage*)*, 3
isImage *(*geometry*)*, 4
isImageGeometry *(*geometry*)*, 4
isPackedImage *(*packedImage*)*, 9
isSparseImage *(*sparseImage*)*, 13

lineApply *(*imapply*)*, 6

mask *(*sparseImage*)*, 13
maskedMatrix *(*sparseImage*)*, 13

packedImage, 9, *12*
parallelBackend *(*parallelism*)*, 10
parallelism, *7*, *8*, 10

readImageData, 11
readImageData(), *14*
reorient *(*worldAxes*)*, 16
reorient(), *12*
resolveThreads *(*parallelism*)*, 10

sameGeometry *(*geometry*)*, 4
sliceApply *(*imapply*)*, 6
sparseImage, 13
sparseness *(*sparseImage*)*, 13
spatial *(*geometry*)*, 4
stats::var(), *9*
storageDescriptor, 14
storageDescriptor(), *11*, *12*
storageLayout *(*worldAxes*)*, 16
storageLayout(), *10*, *14*, *15*
storageType *(*packedImage*)*, 9
storedValues *(*sparseImage*)*, 13

toVoxel *(*geometry*)*, 4

voxelApply *(*imapply*)*, 6
voxelApply(), *2*
voxelSize *(*geometry*)*, 4
voxelSize<- *(*geometry*)*, 4

worldAxes, 16
worldTransform *(*geometry*)*, 4
worldTransform<- *(*geometry*)*, 4
writeImageData *(*readImageData*)*, 11
writeImageData(), *14*