Visualization of NIfTI Images

Formats of Images

There are multiple imaging formats. We will use NIfTI:

- NIfTI - Neuroimaging Informatics Technology Initiative (`https://nifti.nimh.nih.gov/nifti-1`)
  - essentially a header and data (binary format)
  - will have extension .nii (uncompressed) or .nii.gz (compressed)
  - we will primarily use compressed NIfTI images
  - we will use 3-D images (4-D and 5-D are possible)
- ANALYZE 7.5 was a precursor to NIfTI
  - had a `hdr` file (header) and `img` file (data)

An Imaging into to R

What is a package?

A package is collection of functions, documentation, data, and tutorials (called vignettes).

- You install a package using the `install.packages` command/function:

```
install.packages("oro.nifti")
```

`install.packages` is a function, `"oro.nifti"` is a character string.

Loading Packages

When you install a package, that means it's downloaded on your computer. That **doesn't** mean that you can use the functions from that package just yet.

- You "load"/import a package into memory using the `library` command

For example, to load the `oro.nifti` package:

```
library(oro.nifti)
```

Now, functions from the `oro.nifti` package can be used.

Some packages we will use

All packages we will discuss are loaded on the RStudio Server:

- `oro.nifti` - reading/writing NIfTI images
- `neurobase` - extends `oro.nifti` and provides helpful imaging functions

Let's load them:

```
library(oro.nifti)
library(neurobase)
```

Reading in NIfTI images: assignment

We will use the `readnii` function (from `neurobase`) to read in a `nifti` object (this is an `R` object).

Here we read in the `training01_01_mprage.nii.gz` file, and assign it to an object called `t1`

```
t1 = readnii("training01_01_mprage.nii.gz")
```

Now, an object `t1` is in memory/the workspace.

Reading in NIfTI images: assignment

In R, you can assign using the equals = or arrow <- (aka assigment operator).

The above command is equivalent to:

```
t1 <- readnii("training01_01_mprage.nii.gz")
```

There are no differences in these 2 commands, but just personal preference.

By default, if you simply pass the object, it is printed, we can also do `print(t1)`:

```
class(t1)
```

```
## [1] "nifti"
## attr(,"package")
## [1] "oro.nifti"
```

```
t1
```

```
## NIfTI-1 format
##   Type             : nifti
##   Data Type        : 4 (INT16)
##   Bits per Pixel   : 16
##   Slice Code       : 0 (Unknown)
##   Intent Code      : 0 (None)
##   Qform Code       : 1 (Scanner_Anat)
##   Sform Code       : 1 (Scanner_Anat)
##   Dimension        : 256 x 256 x 120
```

Help

- To see the documentation for a function, use the `?` symbol before the name of the function. This is a shortcut for the `help` command:
- For example, to see documentation for `readnii`:

```
?readnii
help(topic = "readnii")
```

- To search for help files, use a double `??` or `help.search`:

```
??readnii
help.search(pattern = "readnii")
```

Some Details

- R is case sensitive (e.g. `y` and `Y` are different)
- Commands separated by new line or by a colon (;)
- Use # to comment

You can also be explicit about which package you are using with the `::` operator, where the syntax is `package::function`:

```
t1 = neurobase::readnii("training01_01_mprage.nii.gz")
```

`ms.lesion` Package

The `ms.lesion` package was made for this course. It has all the data we will work with and the outputs from the analyses.

The main function we will use is `get_image_filenames_list_by_subject`. It returns a `list` of filenames of the images for the 5 training and 3 test subjects from the MS lesion challenge 2016 (`http://iacl.ece.jhu.edu/index.php/MSChallenge`)

ms.lesion Package

```
library(ms.lesion)
files = get_image_filenames_list_by_subject()
length(files); names(files);
```

```
## [1] 8
```

```
## [1] "test01"     "test02"     "test03"     "training01"
## [6] "training03" "training04" "training05"
```

```
head(files$training01)
```

```
##
## "/Library/Frameworks/R.framework/Versions/3.3/Resources/
##
##     "/Library/Frameworks/R.framework/Versions/3.3/Resour
##
##  "/Library/Frameworks/R.framework/Versions/3.3/Resources
##
```

MS Lesion

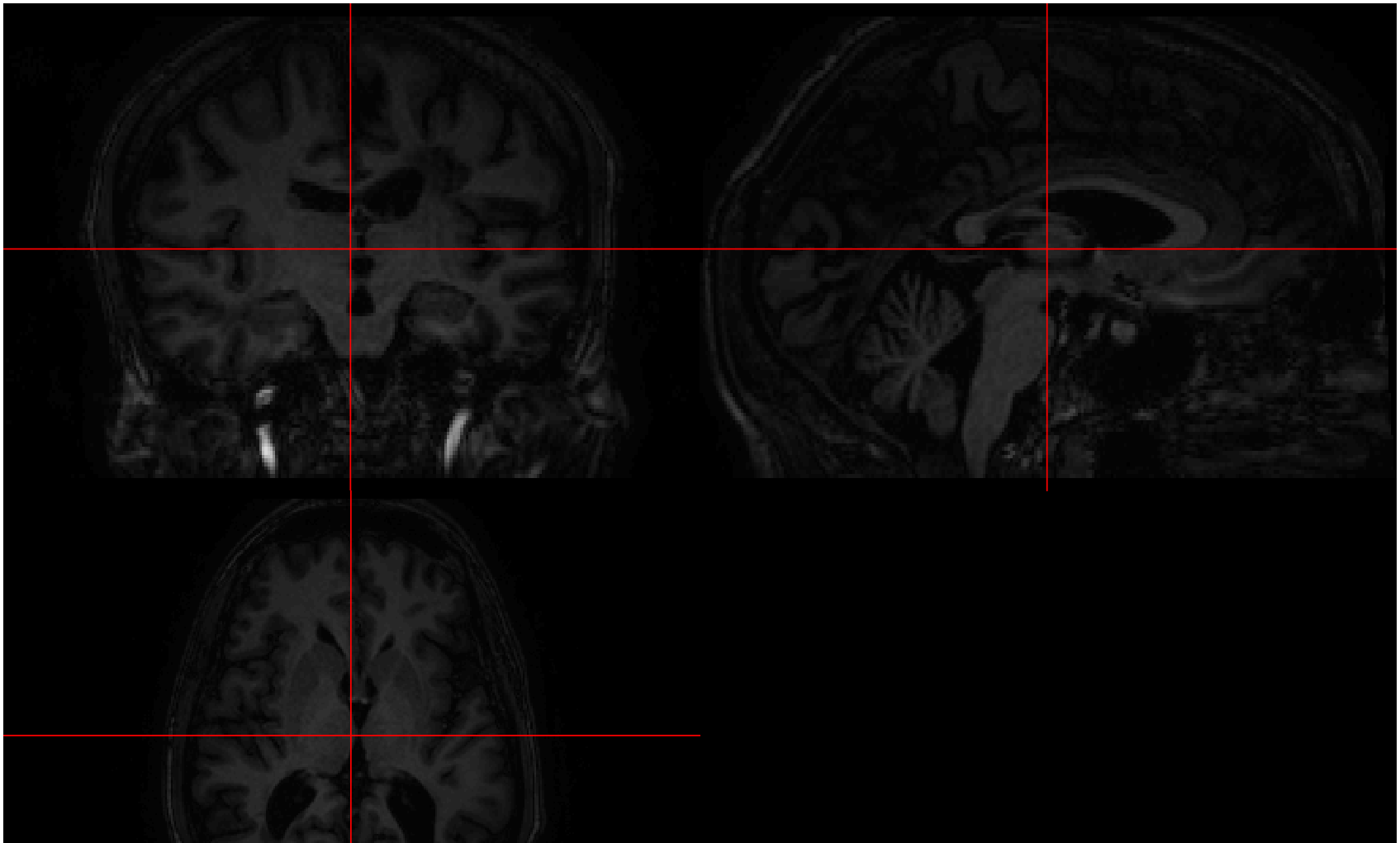
Let's read in the T1 image from a MS lesion data set to visualize:

```
files = files$training01
t1_fname = files["MPRAGE"]
t1 = readnii(t1_fname)
```

Orthographic view

The `oro.nifti::orthographic` function provides great functionality on displaying `nifti` objects in 3 different planes.

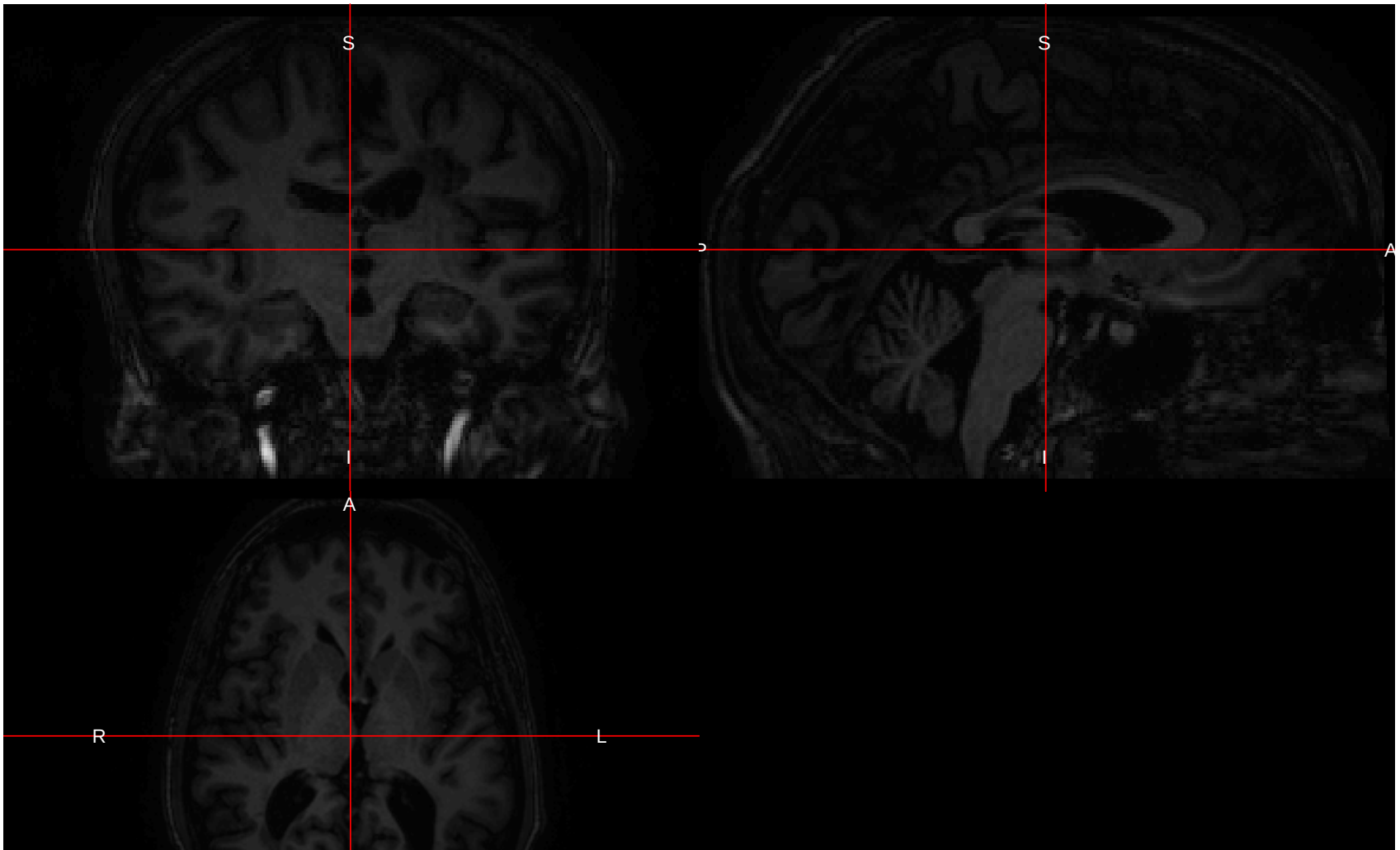
```
oro.nifti::orthographic(t1)
```



Orthographic view with additions

The `neurobase::ortho2` function expands upon this with some different defaults.

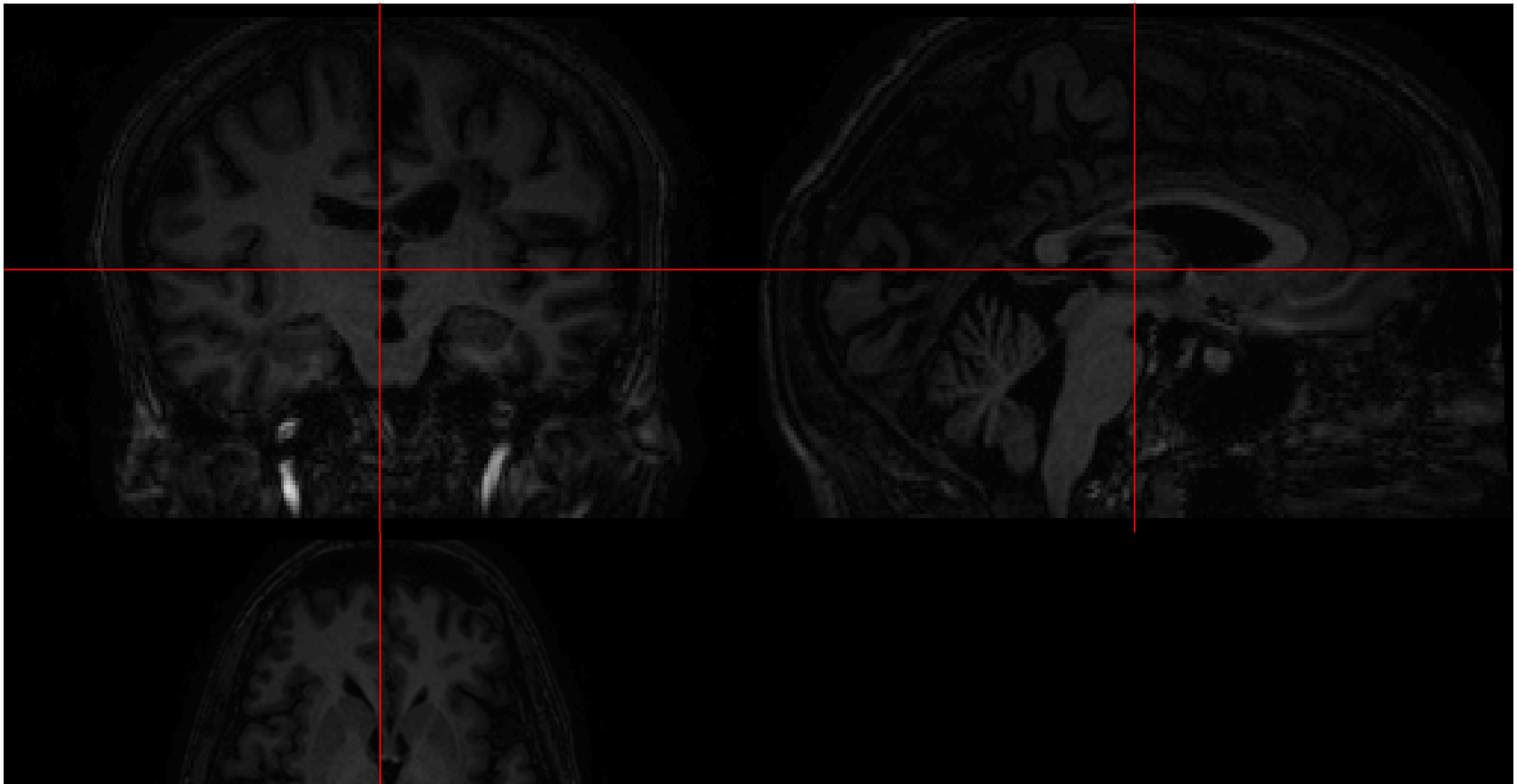
```
neurobase::ortho2(t1)
```



Orthographic view with additions

We see that in `ortho2` there are annotations of the orientation of the image. Again, if the image was not reoriented, then these many not be corrrect. You can turn these off with the `add.orient` argument:

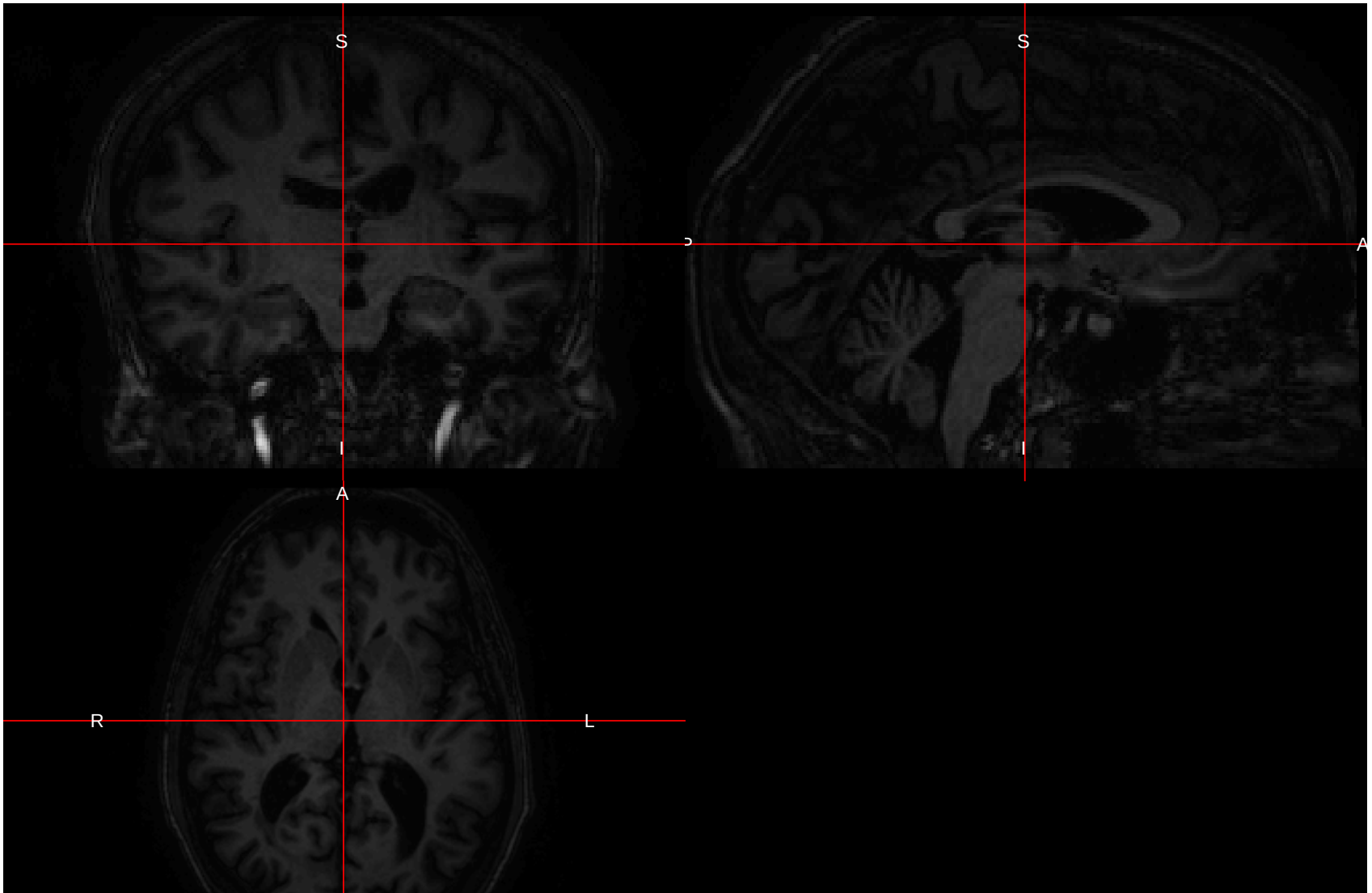
```
neurobase::ortho2(t1, add.orient = FALSE)
```



Bright values

Large intensities can “dampen” the viewing of an image.

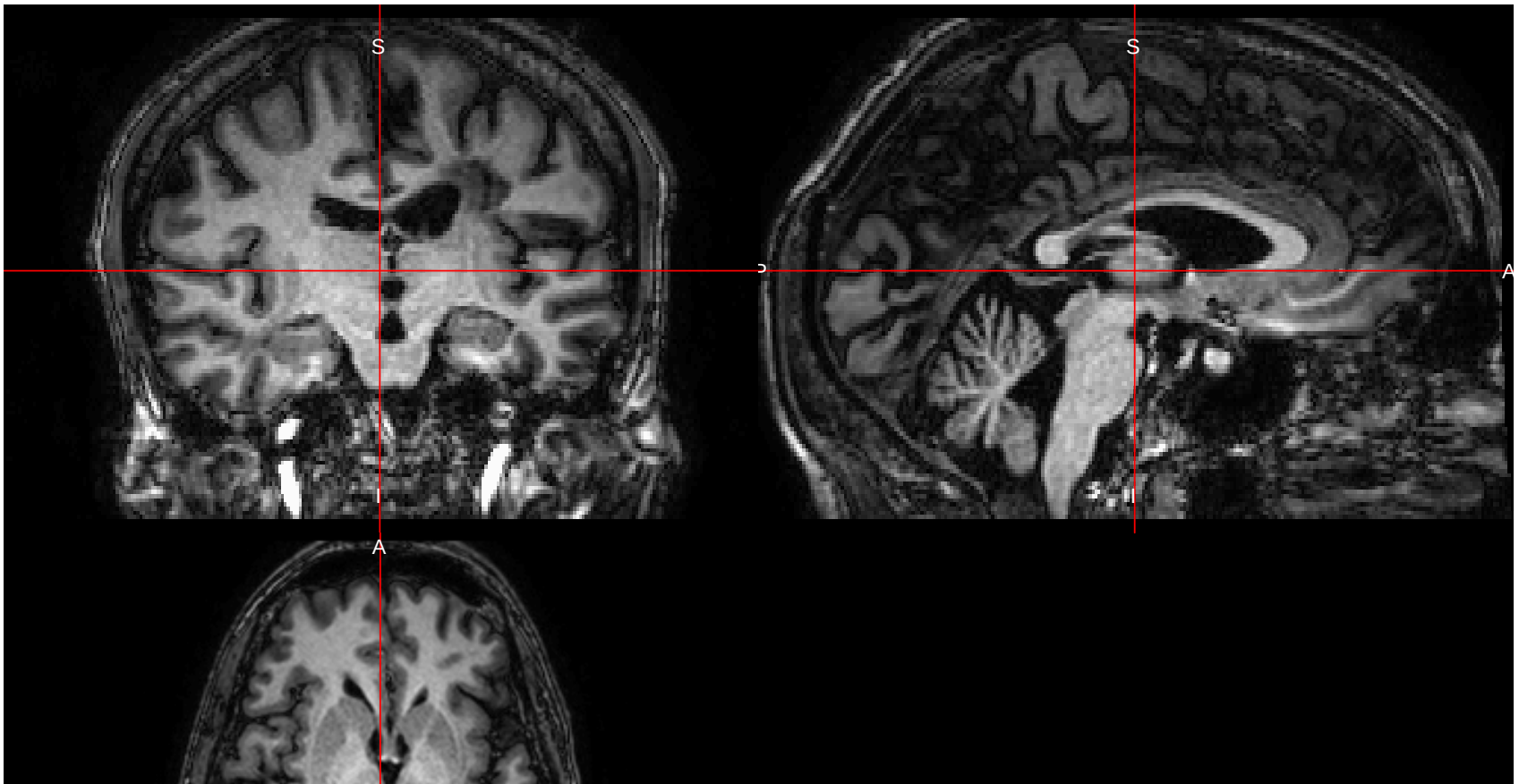
```
ortho2(t1)
```



Bright values

We see a faint outline of the image, but this single large value affects how we view the image. The function `robust_window` calculates quantiles of an image, by default the 0 (min) and 99.9th quantile, and sets values outside of this range to that quantile.

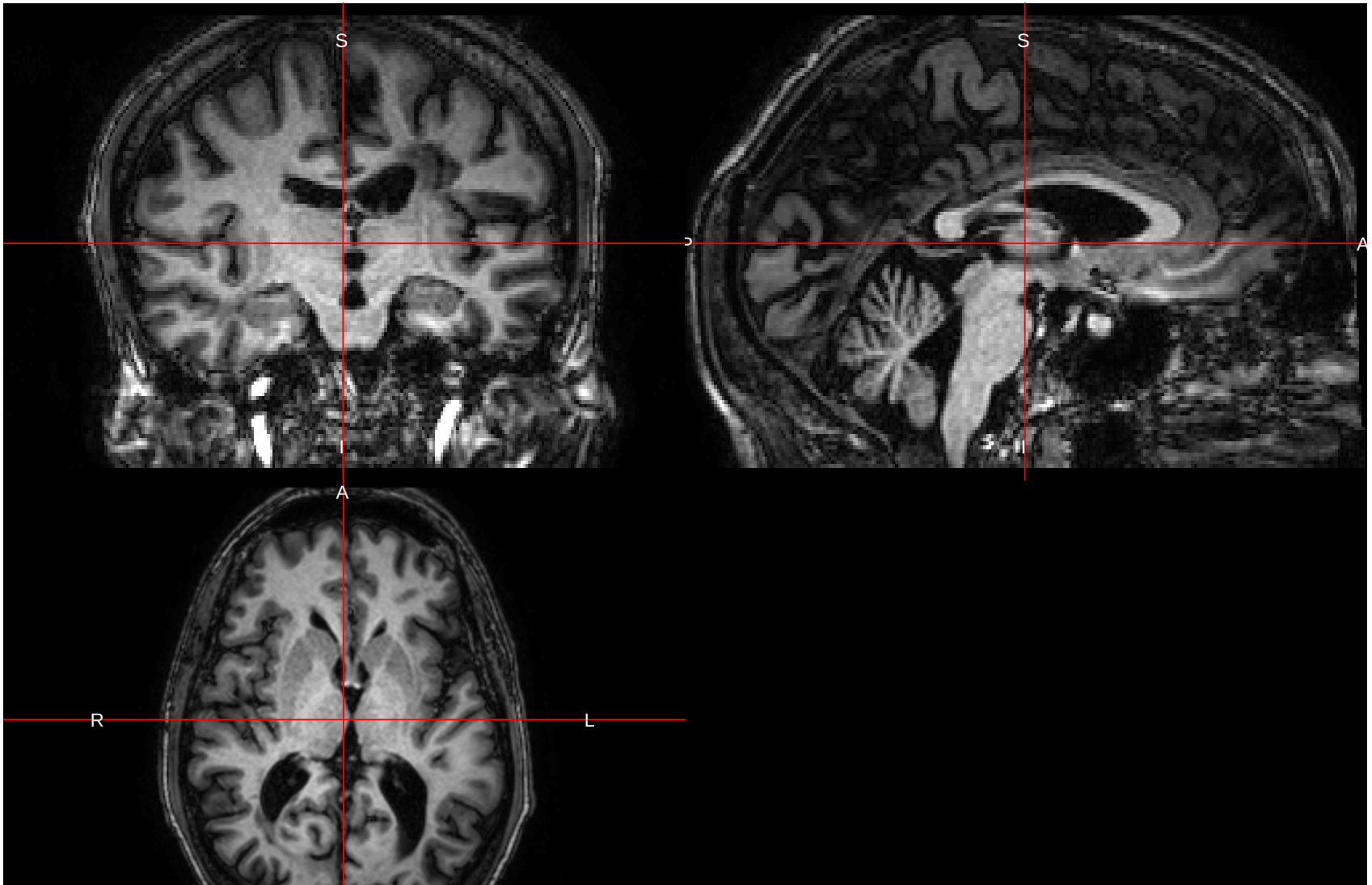
```
ortho2(robust_window(t1))
```



Changing the Windowing

The `zlim` option can also map intensities that can be plotted:

```
ortho2(t1, zlim = quantile(t1, probs = c(0, 0.999)))
```



Differences between `orthographic` and `ortho2`

The above code does not fully illustrate the differences between `orthographic` and `ortho2`. One marked difference is when you would like to "overlay" an image on top of another in an orthograhic view.

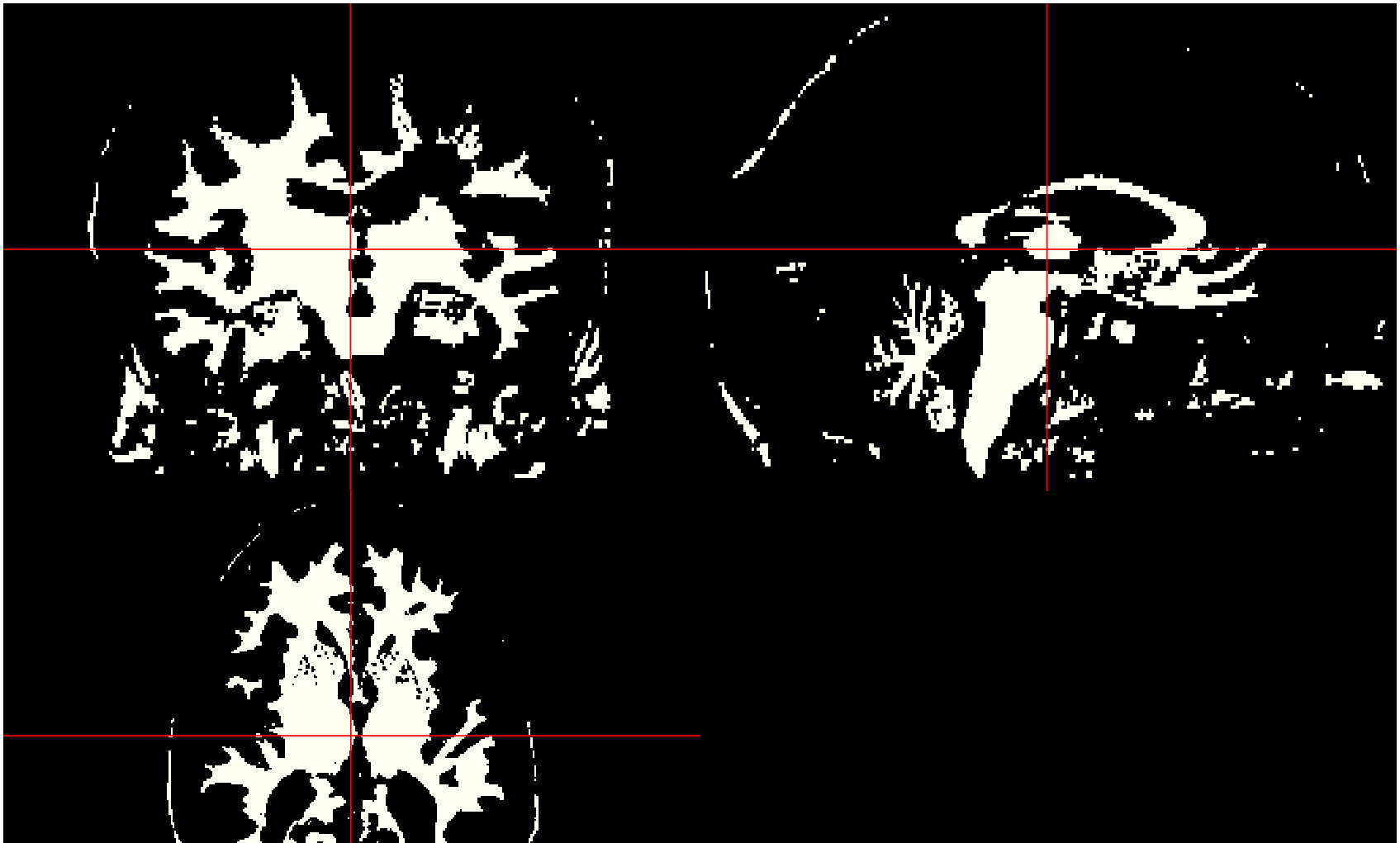
We will be plotting the windowed image for visualization:

```
rt1 = robust_window(t1)
```

Differences between orthographic and ortho2

Here we will highlight voxels greater than the 90th quantile of the image:

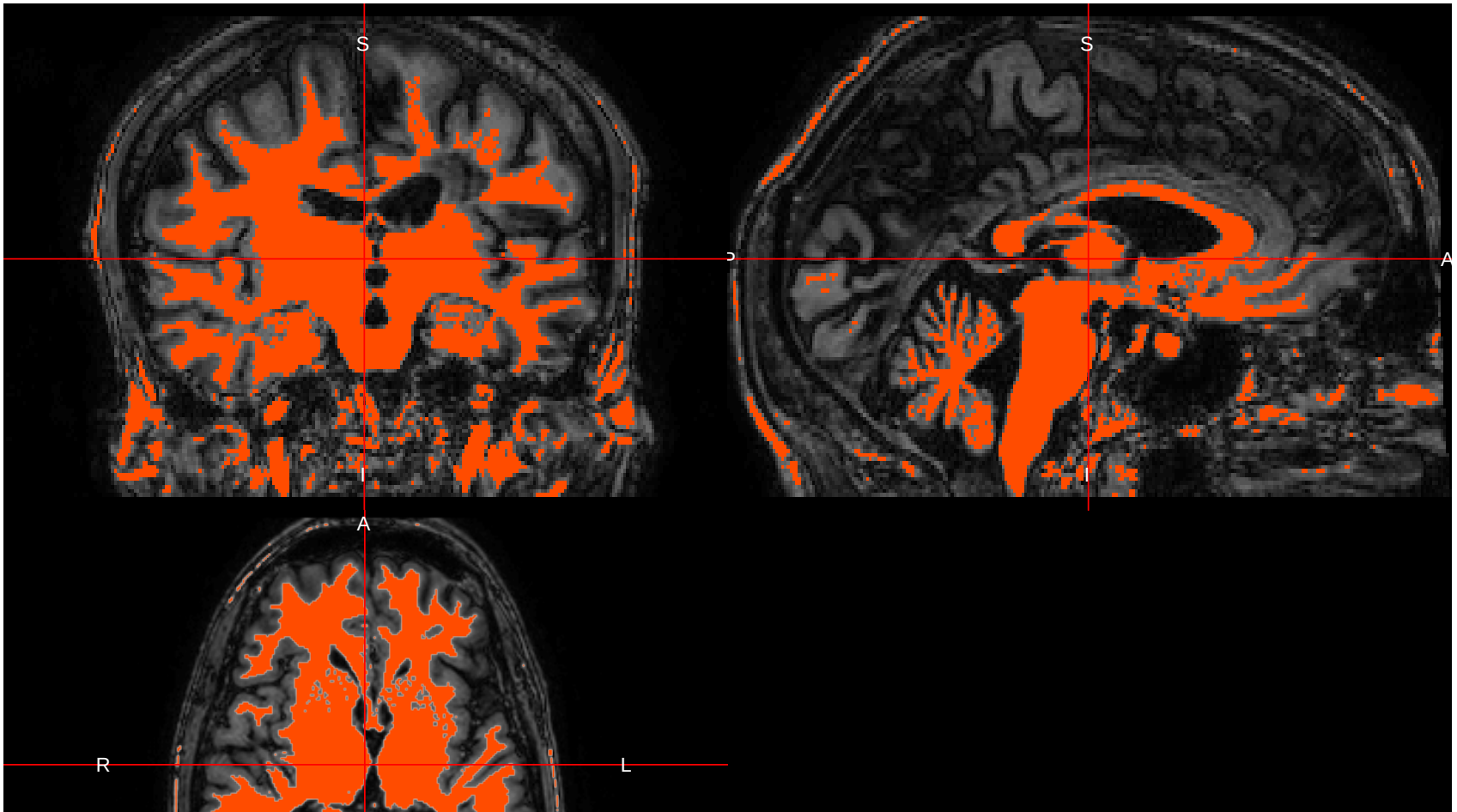
```
orthographic(rt1, y = t1 > quantile(t1, 0.9))
```



Differences between `orthographic` and `ortho2`

We see that the white matter is represented here, but we would like to see areas of the brain that are not over this quantile to be shown as the image. Let us contrast this with:

```
ortho2(rt1, y = t1 > quantile(t1, 0.9))
```



Differences between `orthographic` and `ortho2`

In `R`, `NA` is the placeholder for missing data.

The `ortho2` (and `orthograhic`) function is based on the `graphics::image` function in R, as well as many other functions we will discuss below. When `graphics::image` sees an `NA` value, it does not plot anything there.

We see the image where the mask is `0` shows the original image. This is due to the `NA.y` argument in `ortho2`. The `NA.y` argument in `ortho2` makes it so any values in the `y` argument (in this case the mask) that are equal to zero are turned to `NA`.

Double orthographic view

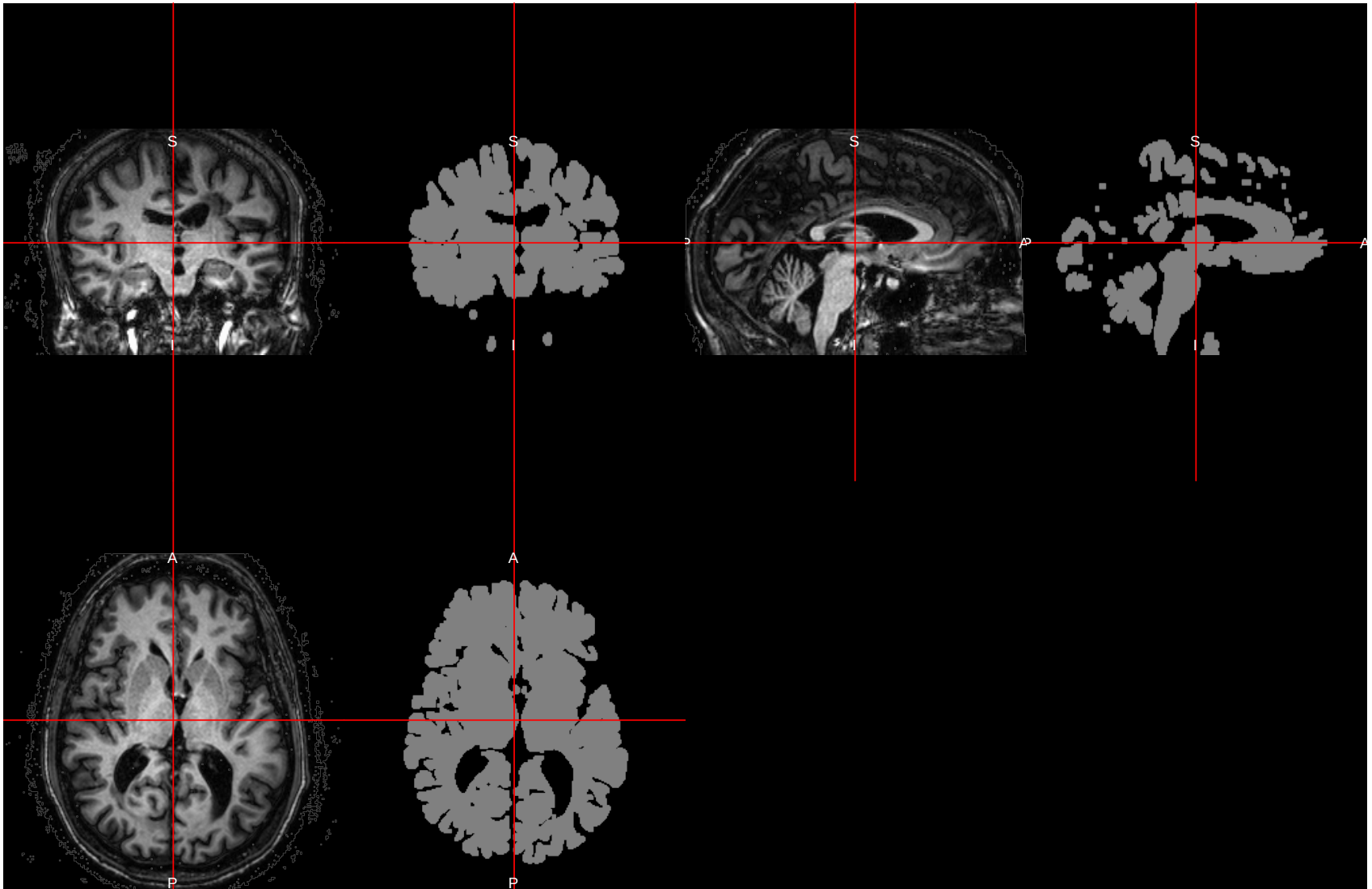
Sometimes you would like to represent 2 images side by side, of the same dimensions and orientation of course. The `double_ortho` function allows you to do this. Let's get a coarse mask of the image:

```
mask = extrantsr::oMask(t1)
```

Double viewing images

We can view the original T1 alongside the brain-extracted image:

```
double_ortho(t1, mask)
```



Single slice view

We may want to view a single slice of an image. The `oro.nifti::image` function can be used here. Note, `graphics::image` exists and `oro.nifti::image` both exist.

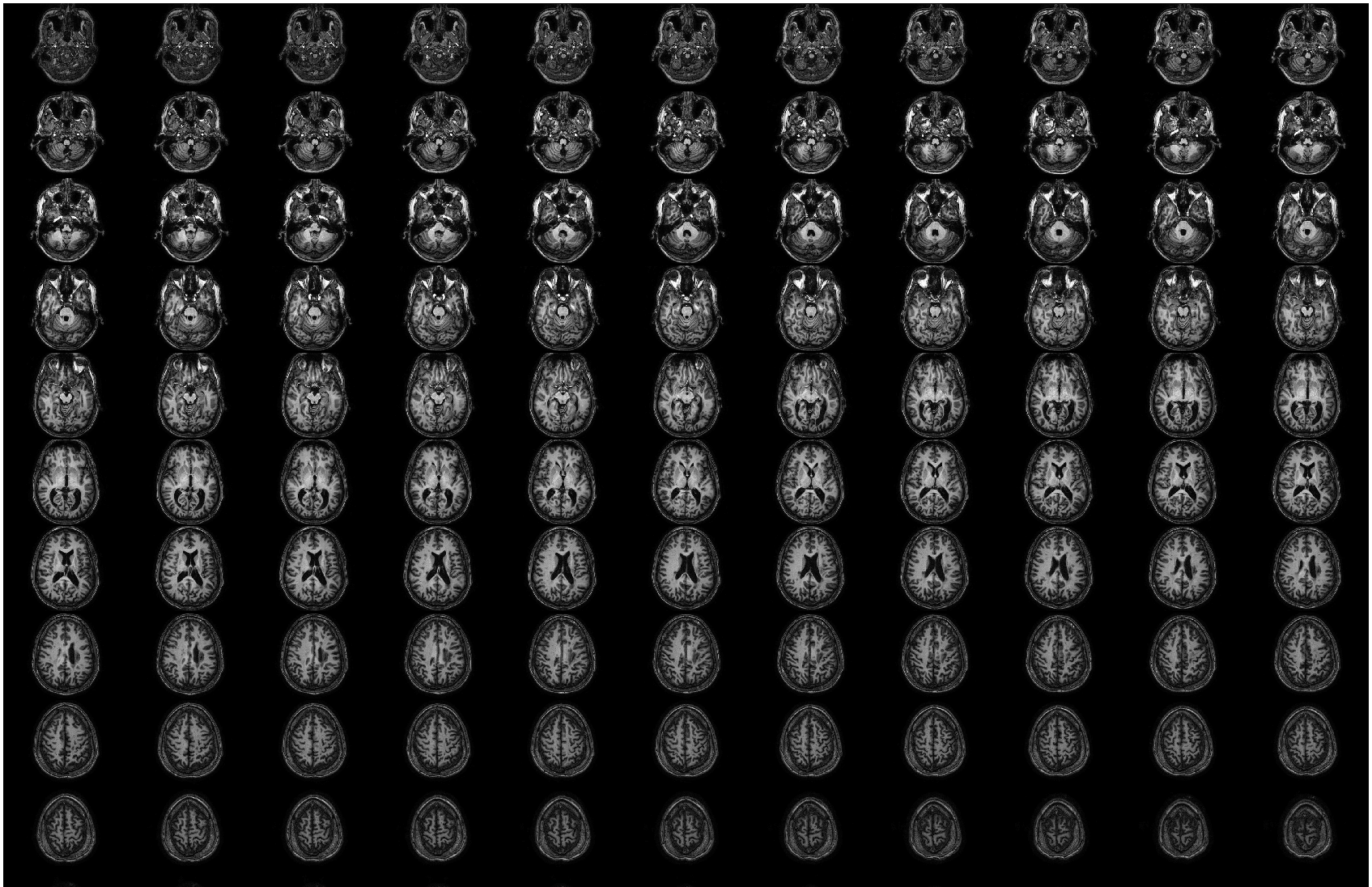
The `oro.nifti::image` allows you to just write `image(nifti_object)`, which performs operations and calls functions using `graphics::image`.

This allows the user to use a "generic" version of `image`, which `oro.nifti` adapted specifically for `nifti` objects. You can see the help for this function in `?image.nifti`.

Plotter

Let's **try** to plot an image of the 80^{th} slice of the T1 image:

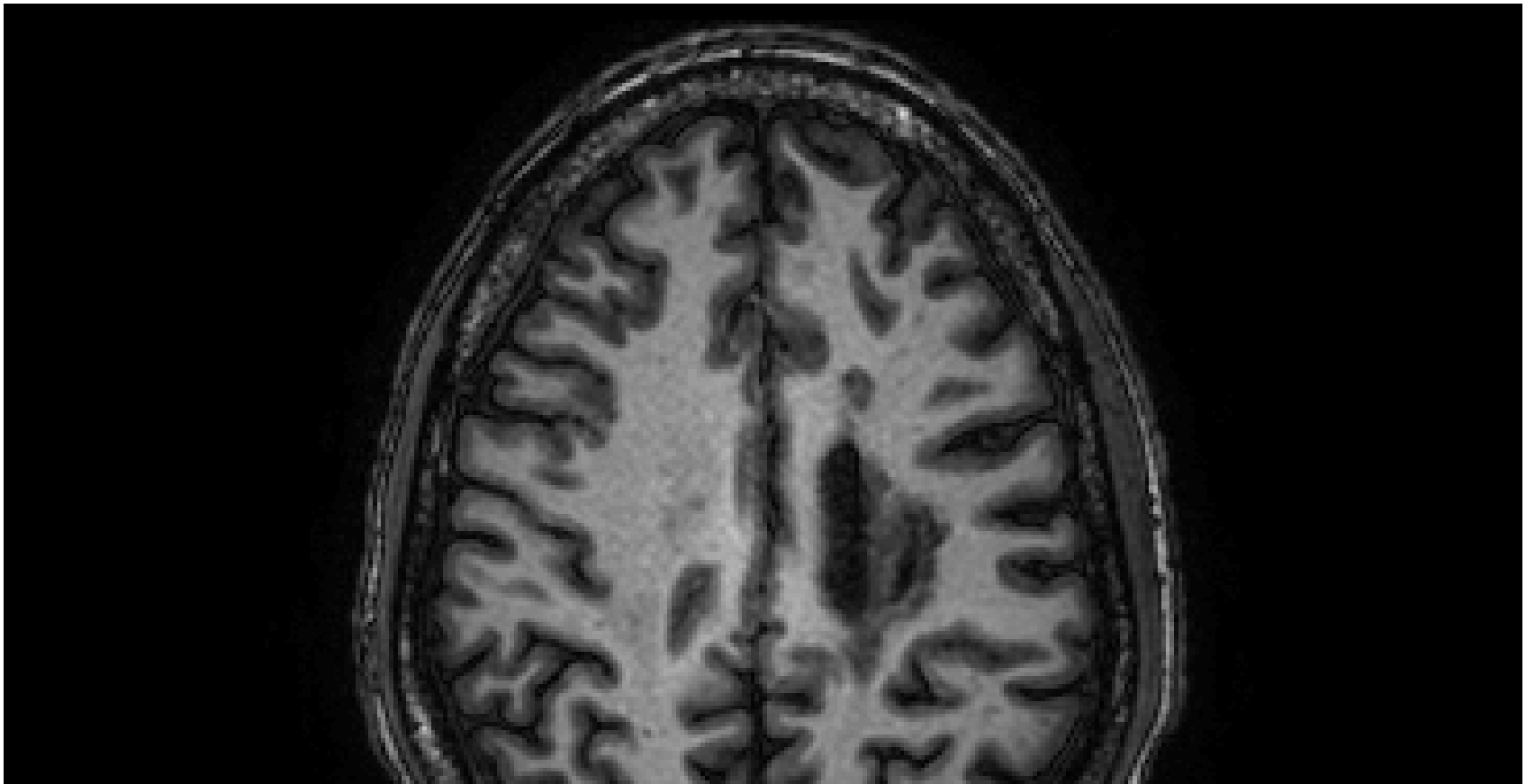
```
image(t1, z = 80)
```



Bad Plot

What happened? Well, the default argument `plot.type` in `image.nifti` is set for `"multiple"`, so that even if you specify a slice, it will plot **all** slices. Here, if we pass `plot.type = "single"`, we get the single slice we want.

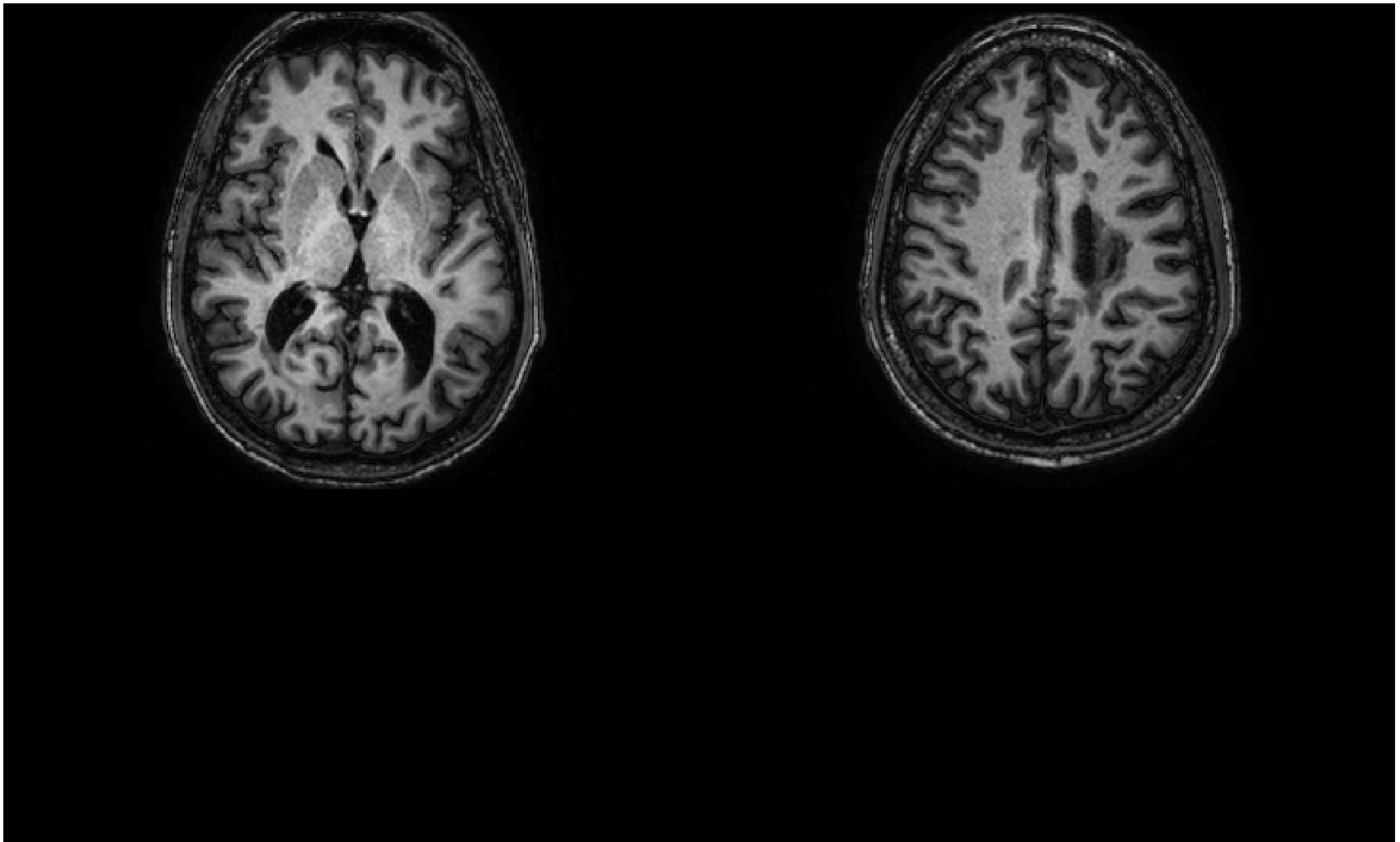
```
image(t1, z = 80, plot.type = "single")
```



Multiple Slices but not All

If we put multiple slices with `plot.type = "single"`, then we will get a view of these 2 slices:

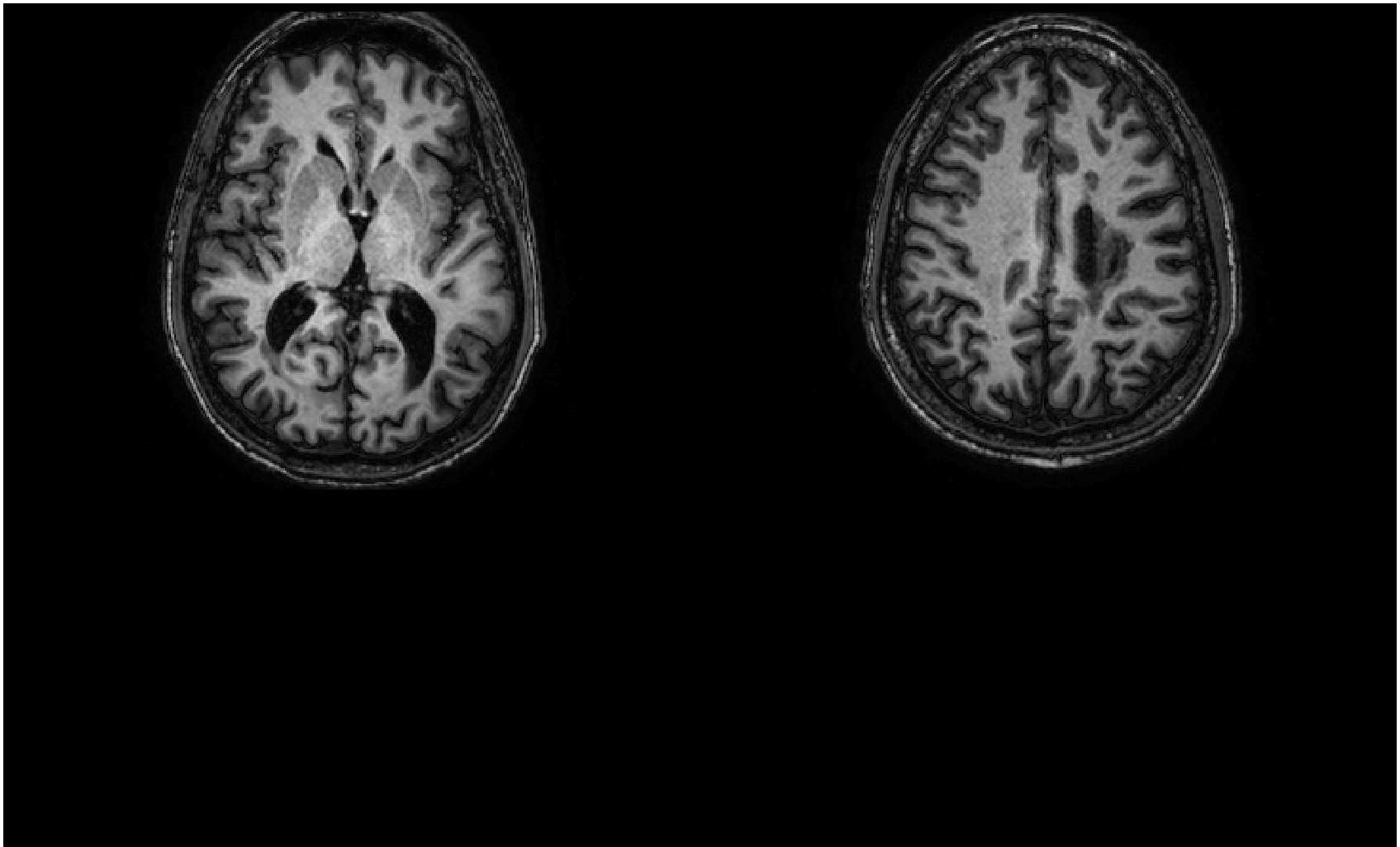
```
image(t1, z = c(60, 80), plot.type = "single")
```



Multiple Slices but not All

In `oro.nifti` version `0.7.4`, we added `slice`, which is the same as `image`, but `plot.type = "single"`.

```
oro.nifti::slice(t1, z = c(60, 80))
```



Overlaying slices

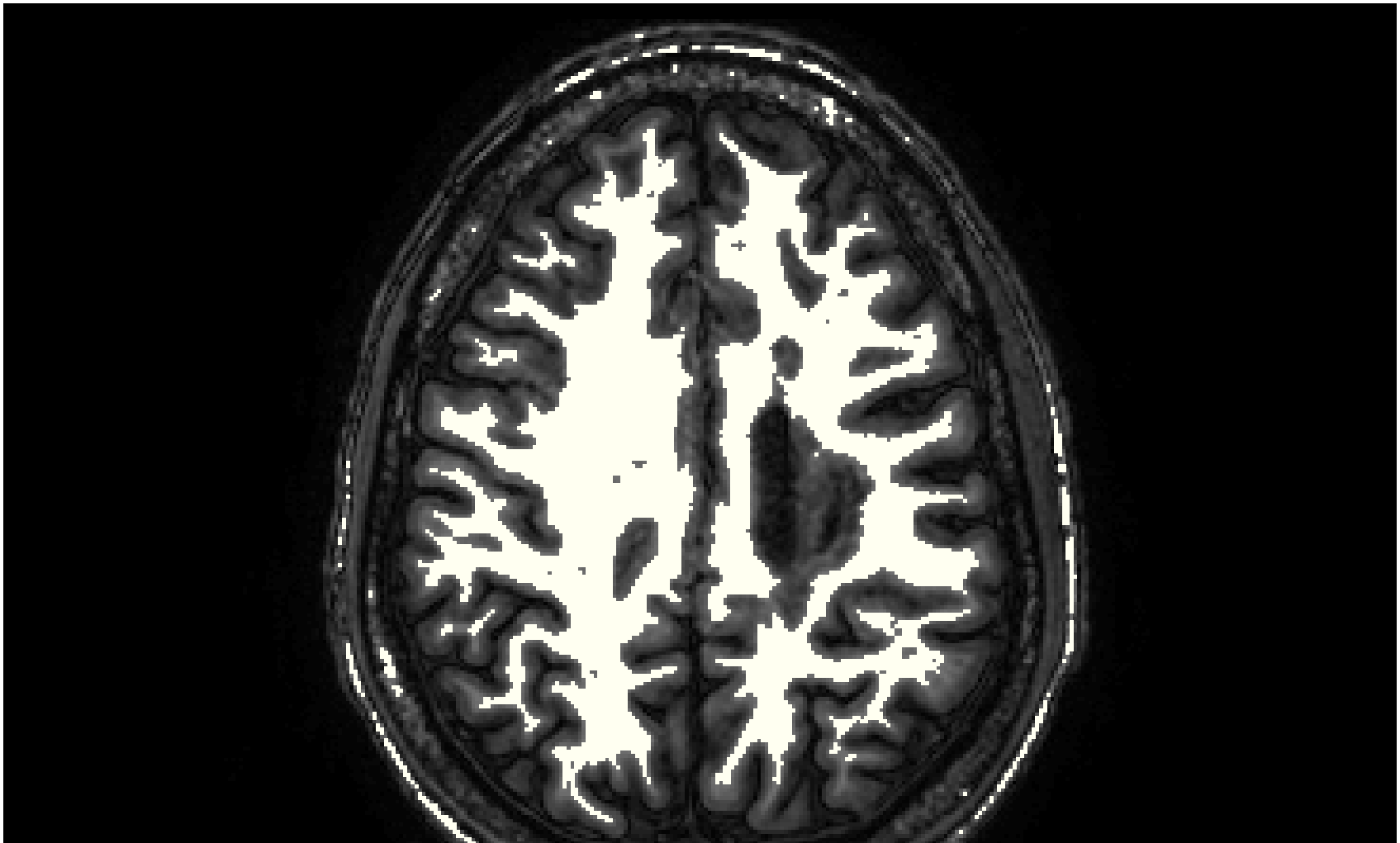
We can also overlay one slice of an image upon another using the `oro.nifti::overlay` function. Here we must specify `plot.type` again for only one slice.

```
overlay(t1, y = t1 > quantile(t1, 0.9), z = 80, plot.type =
```

Overlay2

There is no `overlay2` (at the time of running this) because we added `NA.y` in `oro.nifti` version `0.7.4`.

```
overlay(t1, y = t1 > quantile(t1, 0.9), z = 80, plot.type =
```



Dropping empty dimensions

In some instances, there are extraneous slices to an image. For example, in the Eve template image we read in, it is just the brain. Areas of the skull and extracranial tissue are removed, but the slices remain so that the brain image and the original image are in the same space with the same dimensions. For plotting or further analyses, we can drop these empty dimensions using the `neurobase::dropEmptyImageDimensions` function or `drop_empty_dim` shorthand function.
By default, if one `nifti` is passed to the function and `keep_ind = FALSE`, then the return is a `nifti` object.

```
reduced = dropEmptyImageDimensions(t1)
dim(t1)
```

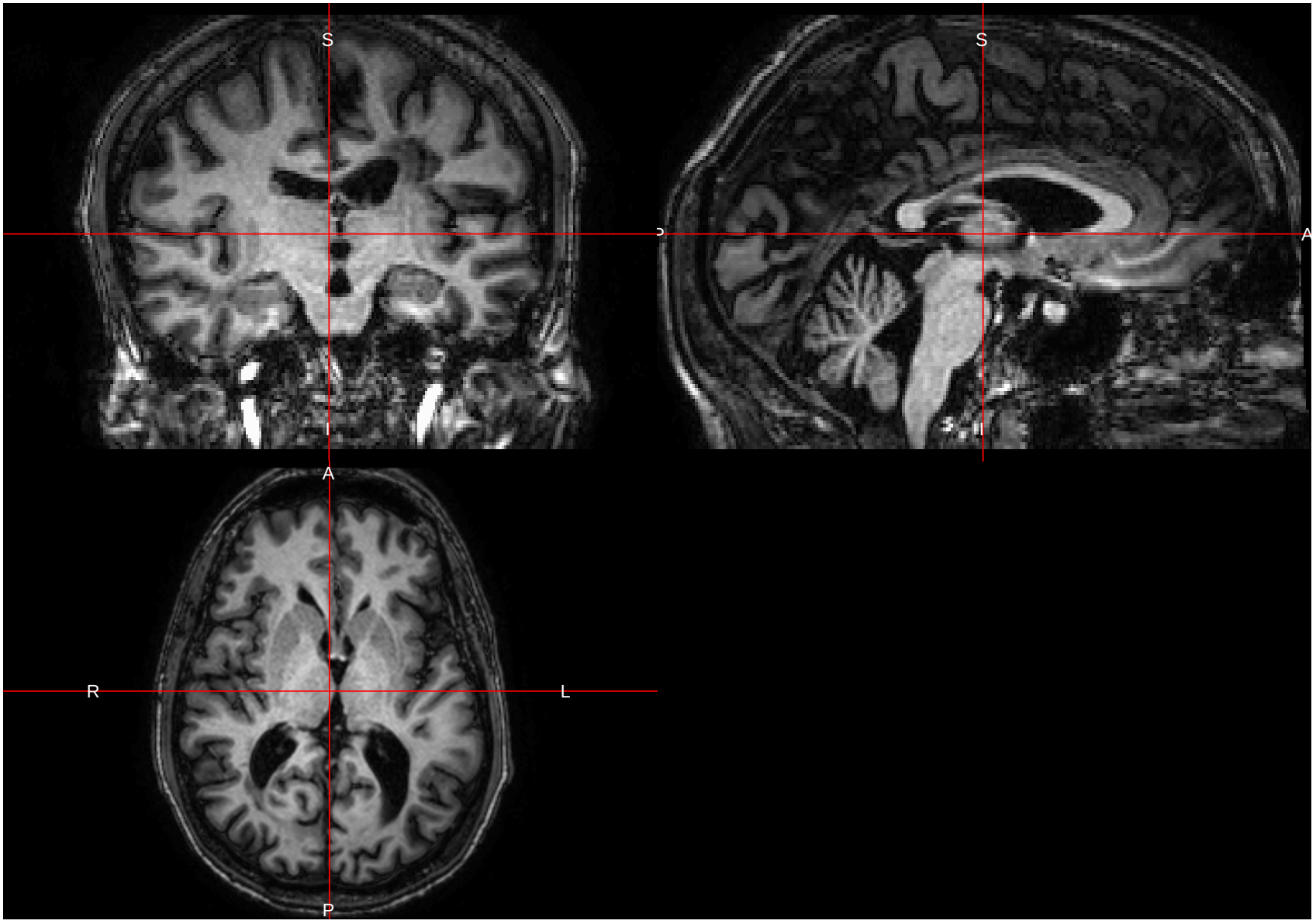
```
## [1] 256 256 120
```

```
dim(reduced)
```

```
## [1] 256 256 120
```

Plotting the Reduced Image

```
ortho2(reduced)
```



Vectorizing a `nifti`

To convert a `nifti` to a `vector`, you can simply use the `c()` operator:

```
vals = c(t1)
class(vals)
```

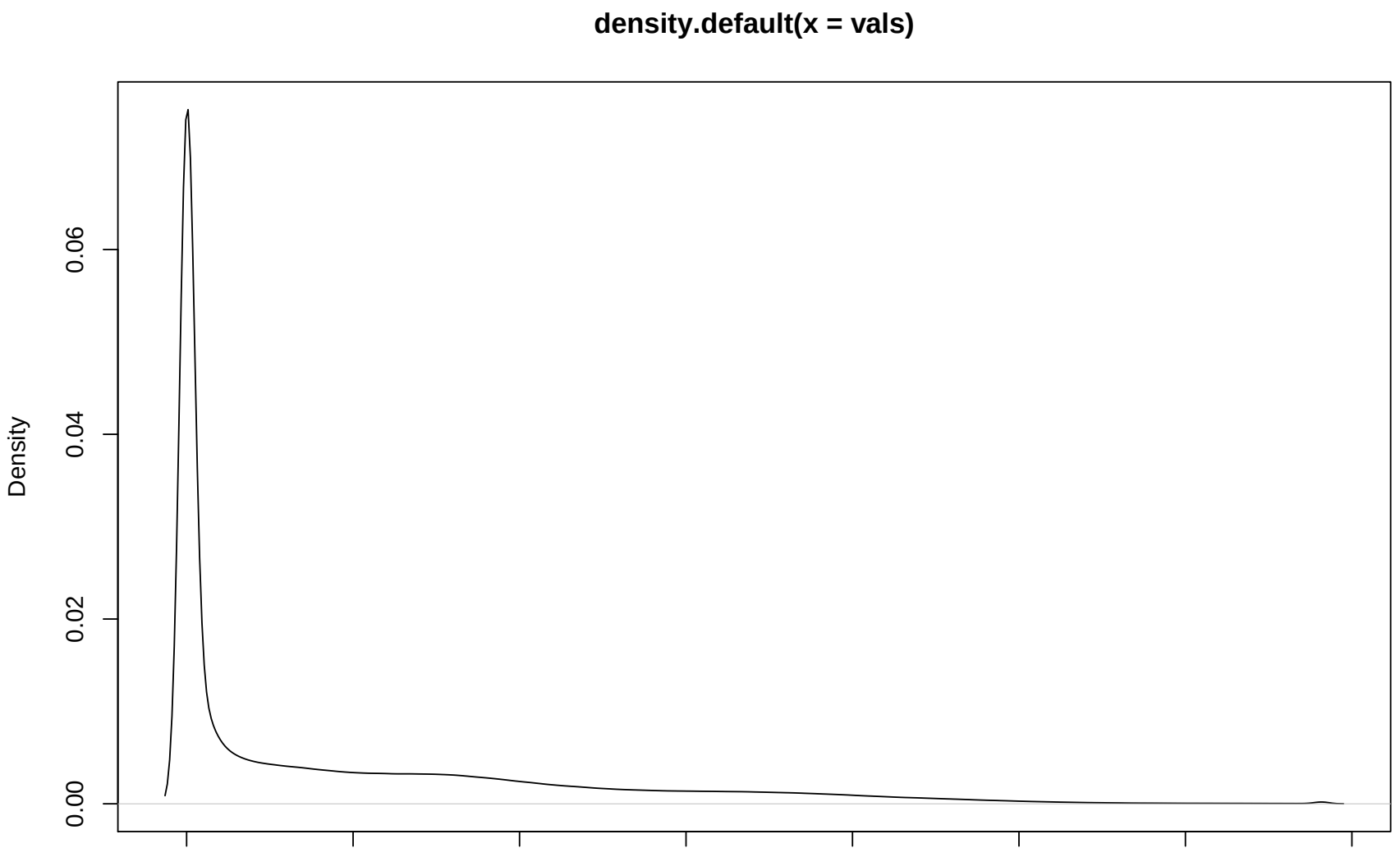
```
## [1] "numeric"
```

Note an `array` can be reconstructed by using `array(vals, dim = dim(eve))` and will be in the correct order as the way R creates vectors and arrays.

Histogram

From these values we can do all the standard plotting/manipulations of data. For example, let's do a marginal density of the values:

```
plot(density(vals))
```



Histogram

You can also pass in a mask to most standard functions:

```
plot(density(t1, mask = t1 > 0))
```