Reproducible and programmatic generation of neuroimaging visualisations

Sidhant Chopra, Löic Labache, Winnie Orchard, [...]

09/03/2022

Introduction

Visualising neuroimaging data is one of the primary ways in which we evaluate data quality, interpret results and communicate findings. These visualisations are commonly produced using graphic interface-based (GUI) tools where individual images are opened and display settings are manually changed until the desired output is reached. The choice to use GUI-based software's has been driven by a perception of convenience, flexibility and accessibility. However now there are also many code-based software packages which are well-documented and don't often require high-level knowledge of programming, making them more accessible to the neuroimaging community. These tools are flexible and allow for the generation of reproducible, high-quality and publication-ready brain visualisations in only a few lines of code (Figure 1), especially within the R, Python and Matlab environments.

This article will cover three major advantages of using code-generated visualisations over GUI based tools: replicability, flexibility and integration. First, by generating figures using code, you increase the replicability of your figures for yourself, your collaborators and your readers. Second, code-based software provides more precise controls over visualisation settings, while also benefiting from advantages of programming, such as being able to iteratively generate multiple figures. Finally, being able to integrate figure generation into your analysis scripts reduces chances of errors, with more advanced tools now allowing for seamless weaving together of prose and code. Here, we review the advantages of learning and using programmatic neuroimaging visualisations, focusing on benefits to replicability, flexibility and intergration. We conclude by introducing examples of R-packages which allow for visualisation of statistics at region-of-interest (ROI), voxel, vertex and edge level data, followed by a brief discussion of limitations and functionality gaps in code-based brain visualisation tools.

Replicability

In recent years, there have been multiple large-scale efforts empirically demonstrating the lack of reproducibility of findings from neuroimaging data (cite poldrack). One common solution proposed for achieving robust and reliable discoveries has been to encourage scientific output which can be transparently evaluated and independently replicated. In practise, what this usually means is openly sharing detailed methods, materials, code and data. While there is a trend towards increasing transparency and code sharing in neuroimaging research (cite), the sharing of code used to generate figures which include brain renderings and spatial maps has been relatively neglected. This gap in reproducibility is partly driven by the fact that brain figures are often created using a manual process of tinkering with sliders, buttons and overlays on a GUI, concluding with a screenshot and sometimes beatification in image processing software like Illustrator. This process inherently makes neuroimaging visualisations difficult, if not impossible to replicate, even by the authors who created them.

Given that brain figures regularly form the centrepiece of interpretation within a paper, conference presentations and news reports, making sure they can be reliably regenerated is crucial for knowledge generation.

By writing and sharing code used to generate brain visualisations, a direct and traceable link is established between the underlying data and scientific figure. While the code that produces a replicable figure doesn't reflect the validity of the scientific finding or the accuracy of the content figure, it instils transparency and robustness while demonstrating a desire to further scientific knowledge. Some would even consider publishing figures which cannot be replicated as closer to advertising, rather than science (cite blog).

Flexibility

Being able to exactly replicate your figures via code has serious advantages beyond good open science practises. In particular, the ability to reprogram inputs (such as statistical maps) and settings (such as colour schemes and visual orientations) can streamline your entire scientific workflow. Changing inputs and settings via code allows for the easy production of multiple figures, such as those resulting from multiple analyses which require similar visualisations. While some GUI-based tools do offer command-line access to generate visualisations, they can lack flexibility to easily generate publication ready figures and can lose the benefits, such as iteration, provided by your preferred programming environment. A simple for-loop or copying and pasting the code with the input and/or setting-of-interest modified can be a powerful method for exploring visualisation options or creating multi-panel figures. Likewise, an arduous request from a reviewer to alter the data processing or analysis becomes less of a burden when the figure can be re-generated with a few lines of code, as opposed to re-pasting and re-illustrating them. Having a code-base with reprogrammable inputs can mean that the generation of visualisations requires less time, energy and effort than manual GUI-based generation. This also makes it easier to make figures for each subsequent project that you work on. Keep in mind that the gains of writing code for your figures are cumulative, and in addition to improving your programming, you start to build a code-base for figure generation that you can continue to reuse and share through your scientific career.

Precise controls via code over visualization settings, such as colour schemes, legend placement and camera angles, can provide you with much greater flexibility over visualisations. Nonetheless, part of the appeal of GUI-based tools is that the pre-sets for such settings can provide a useful starting point and reduce the decision burden on the user. However, similar pre-sets are often available in the from of default settings in most code-based packages, negating the need for the user to manually enter each and every choice of setting required for making the image. Most code-based tools also come with detialed documentation, or even entire papers (cite ggseg and ciftitools) which provide examples of figures which can be used as starting points or templates for new users. As the popularity of code-sharing for figure increases, there will be an cornucopia of templates that can be used as starting points for new figures.

While brain visualisations are often thought of as the end results of analyses, they also form a vital part of quality control of imaging data. Tools to automatically detect artefacts, de-noise the data and generate derivatives are becoming more robust, but we are not yet at the stage where visualising the data during processing is no longer need. Nonetheless, when working with large datasets such as Human Connectome Project (cite) or UK BioBank (cite), it is unfeasible to use traditional GUI-based tool to visually check the data. The time it takes to open a single file and achieve the desired visualisation settings vastly compounds when working with large datasets. Knowing how to programmatically generate brain visualisations can allow you to iterate your visualisation code over each image of a large datasets making checking the quality of each data processing or analysis step accessible and achievable. The visual outputs of each iteration can be complied into accessible documents which can be easily scrolled, with more advanced usage allowing for the creation of interactive HTML reports, similar to those created by tools like mriQC (cite). Increasing capacity to conduct visual quality control on larger datasets can increase the identification of processing errors and result in more reliable and valid findings from your data.

Integrative and Interactive reporting

Often programming languages such as R, Python and Matlab are used for the analysis in neuroimaging studies, but the brain visualisation resulting from these analyses is outsourced to separate GIU-based visualisation tools such as FSLeyes, Freeview or ITK-snap. Switching from your analysis environment to a

GUI-based visualisation process be a cumbersome deviation from the scientific workflow. This can make debugging errors more difficult, as you have to regularly switch program to visually examine the results of any modifications to analyses. Using the brain visualisation tools that already exist within your chosen programming environment can provide instant visual feedback on the impact of modifications to processing or analyses.

Increasingly popular tools such as R-Markdown, Quarto and Jupyter Notebook allow for the mixing of prose and code in a single script, resulting in fully reproducible and publication ready papers. By using code-based tools available within your preferred environment, brain visualisations can be directly integrated and embedded within a paper or report. Some journals that publish neuroimaging studies are moving towards allowing reproducible manuscripts, including reproducible figures (cite e-life,f1000,plos), with some even allowing on-demand re-running of code using cloud services (cite).

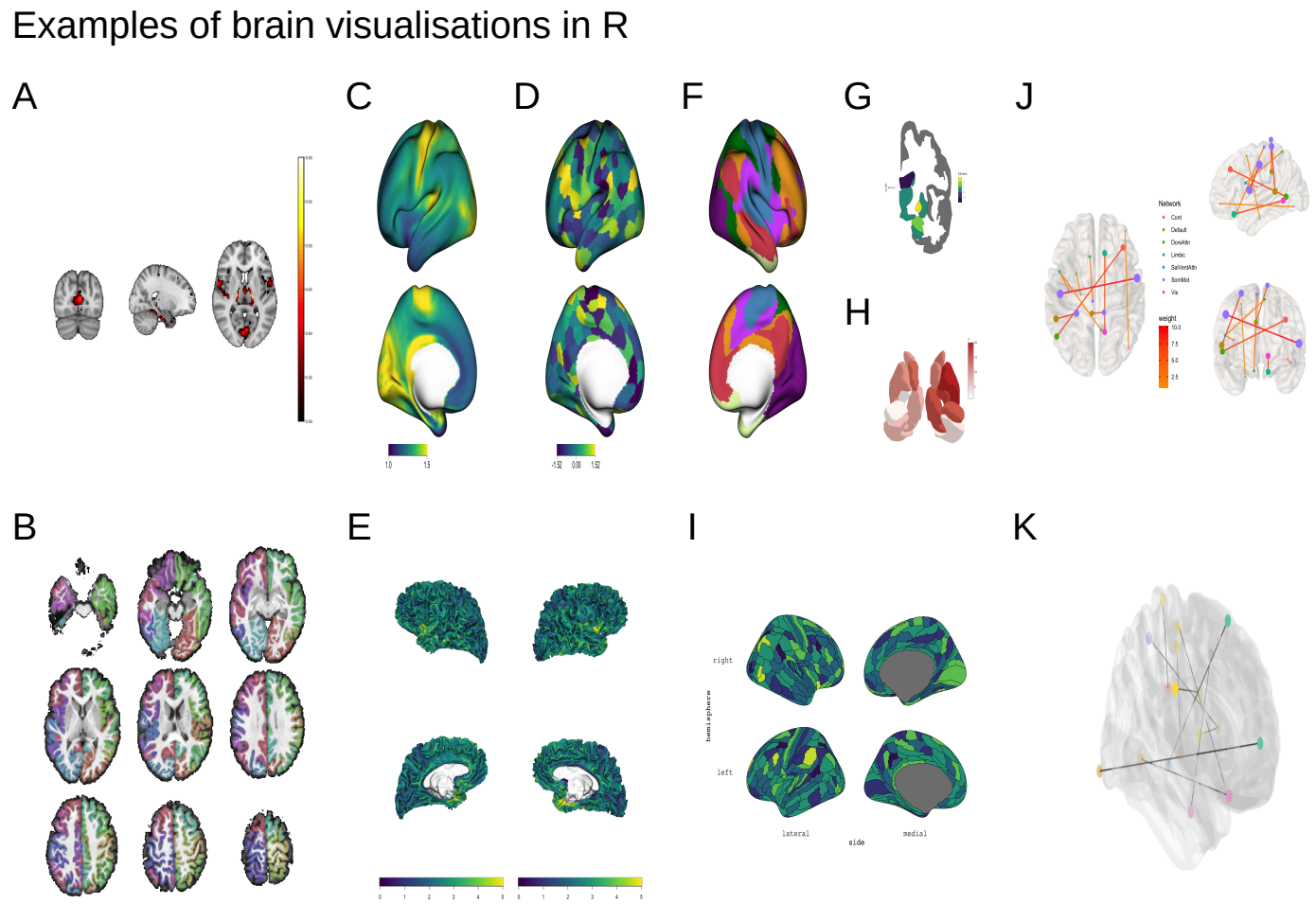
Neuroimaging data are often spatially 3D and can have multiple time points, adding a 4th dimension (e.g. fMRI data). Thus, communicating findings or evaluating quality using static 2D slices is challenging. While well curated 3D renderings can help with spatial localisation (cite madan), in the end, static 2D images can only provide an incomplete representation of 3D or 4D data. A added advantage of some of code-based tools is that you can generate 'rich' media like interactive figures or animations, which can improve scientific communication of findings. Many code-based software tools now allow for the generation of animated brain visualisations (cite blog) or interactive widgets which allow users to zoom, rotate and scroll through slices. Linking to or even embedding these videos or interactive figures in papers can greatly enhance the communication of findings.

Neuroimaging data are often spatially 3D and can have multiple time points, adding a 4th dimension (e.g. fMRI data). Thus, communicating findings or evaluating quality using static 2D slices is challenging, and may not be the best representation of the data, or the interpretations. While well-curated 3D renderings can help with spatial localisation (see madan), in the end, static images can only provide an incomplete representation of the data, and forces researchers to choose the best angle to show, which often involves compromising one result for another. An added advantage of some of the code-based tools is that you can generate 'rich' media like interactive widgets – figures or animations, which allow users to zoom, rotate and scroll through slices. Interacting with a figure in this way can improve scientific communication of findings. Linking to or even embedding these videos or interactive figures in papers can greatly enhance the communication of findings and make your paper more engaging for the reader. Such rich brain visualisations lend themselves to being embedded or shared on science communication mediums other than academic papers - such as presentations, web-sites and social media - all of which can promote the communication of your research with peers and a reach larger audiences.

Examples of neuroimaging visualisation packages available in R

The following four sections provide brief examples packages and functions available in R for visualising voxel, vertex, ROI and edge-level data. A more didactic introduction to these tools is provided in a collection of beginner friendly R-Notebooks.

Figure 1 [Fix image size ratios]



[Figure 1 - Add Caption and remove panel F]

Voxel-level visualisations. The `oro.nifti` package allows for the reading, writing, manipulation and visualisation of voxel-level imaging data of either nifti, analyze or afni formats. In particular, the `image`, `slice` and `orthographic` functions allow users to display the desired number of slices in the desired orientations, while also providing precise control over overlaid images, colour scales, legend placement and other aesthetic settings (Fig 1A). Smoothing and re-sampling of images is sometimes required for visualisation purposes, which can be achieved via the `ANTsR` /`extrantsr` or `fslr` packages.

Vertex-level visualisations. The recently developed `fsbrain` package allows for the visualisation of vertex-level and atlas-level data obtained using Freesurfer. It contains many flexible functions for visualising individual-level and group-level measures such as cortical thickness, volume or surface area (Fig 1B). The package also comes with extensive guides and documentations to assist users (cite).

CIFTI or 'grey-ordinate' data is becoming a popular format for structural and functional imaging data, as it combines vertex-level data of the cortex with voxel-level data fo the cerebellum and sub-cortex. In R, CIFTI data can now be read, visualised and manipulated using the well-documented `ciftiTools` package (Fig 1C; cite).

ROI-level visualisations. Statistical parameters can be mapped onto discreet cortical and sub-cortical regions using the `ggseg` and `ggseg3d` package. These packages flexibly generates aesthetic renderings of cortical, sub-cortical and white matter ROIs in 2D and 3D (Fig 1D). Usually, ROI-level interpretations are dependent on a standardised atlas or parcellation schemes the brain, accordingly this package and its add-ons (`ggExtra`) provides a large array of commonly used atlases and some functionality for users to contribute other atlases. Additionally, being part of the 'Grammar of Graphics' frame work allows for integration with packages such as `gganimate`, enabling users to animate and visualise dynamic changes in spatial and temporal statistics across ROIs. The `ciftiTools` package also allows for surface and voxel-level visualisation of ROIs for the cortex and sub-cortex respectively (Fig 1E). Flexibly allowing CIFTI format atlases and ROIs to be displayed.

Edge-level visualisations. Analyses which divide the brain into discrete regions and examine pair-wise dependences between regional phenotypes are becoming increasingly common. Often the results of these dependency analyses are statistics relating to 'edges' or links between any two regions. The package `brainconn` allows users to plot brain regions as nodes of a network graph, and dependences between regions as edges of that graph, in both 2D and interactive 3D (Figure 1D). Similar to `ggseg`, this package comes within multiple commonly used atlases and contains functionality for users to enter their own atlas schemes.

Limitations and functionality gaps in R packages for brain visualisation

- While most of these tools do not require a strong knowledge of programming, there can still be a steeper learning curve when compared to GUI-based tools.
- For quick and interactive viewing of single images, sometimes GUI tools can be faster (although, fsleyes can be called from python and R environments with ease)
- Visualising data-types such as streamlines from DWI-based tractography are still not well represented in code-based tools.
- Lack of user friendly tools which can render non-standard or uncommon atlases or ROIs. Although, with `ciftitools`, this becomes easier, as long as atlas follows the CIFTI convention.
- 3D renderings of non-standard ROIs like unique subcortical schemes or brain stem are still difficult to do. Can to use complex workarounds like ITKsnap + pyVista.

Supplement (Code used to generate figure)

```
#Figure 1A
#voxel 1

# ADD COMMENTS TO EXPLAIN EACH LINE

library(neurobase)
library(ggplotify)

template <- readnii('data/MNI152_T1_1mm_brain.nii.gz') #load the nifti file into R

effect <- readnii("data/MNI152_effect_size.nii.gz")

#threshold the effect at the desired cut-off
effect[effect<0.3] <- NA

#Plot the effect on the brain template
breaks = seq(.3,.6, by=0.005)

png("data/fig1a.png")
ortho2(x = template,
       y = effect,
       crosshairs = F,
       bg = "white",
       NA.x = T,
       col.y= hotmetal(),
       xyz = c(70,50,80),
       useRaster = T,
       ycolorbar = TRUE,
       mfrow = c(1,3)) +
  colorbar(breaks,col = hotmetal(n=length(breaks)-1),
           text.col = "black", labels = TRUE, maxleft = 0.95)
dev.off()
```

```
#Figure 1B
#voxel 2

library(neurobase)
library(scales)

dti <- readnii('data/sub-001__t1_warped.nii.gz') #load the nifti file into R

atlas <-  readnii('data/sub-001_schaefer300n7_aseg_to_dwispace_gm_rois.nii.gz')

png('data/fig1b.png')
overlay(x=robust_window(dti), y=atlas,
        plot.type = "single",
        z=c(seq(30,110,10)),
        col.y =  alpha(rainbow(300), 0.3),
        plane = "axial",
        useRaster = T,
        bg = "white",
```

```
        NA.x=T,
        zlim.y =c(1,300))

      #  mar = c(0.1, 0.1, 1.5, 0.1)) + #marget of bottom, left, top, right borders
dev.off()
```

```
#vertex 1
#Figure 1B
library(rgl)
library(ggplot2)
# Load the package and point to the Connectome Workbench
suppressPackageStartupMessages(library(ciftiTools))

ciftiTools.setOption("wb_path", "/Applications/workbench/")

# Read and visualize a CIFTI file

cifti_fname <- ciftiTools::ciftiTools.files()$cifti["dtseries"]
surfL_fname <- ciftiTools.files()$surf["left"]

cii <- read_cifti(
  cifti_fname, brainstructures="left",
  surfL_fname=surfL_fname)


view_xifti_surface(cii,
                   hemisphere = "left",
                   view = "both",
                   idx = 1,
                   colors = viridis::viridis(n = 100),
                   zlim = c(1,1.5),
                   legend_embed = T,
                   cex.title = 2)

rgl.snapshot("data/fig1c.png")
rgl.close()
```

```
#Vertex 2
#Figure 1X - Remove white space

#weird angle

library(fsbrain)

#download_optional_data()
subjects_dir = get_optional_data_filepath("subjects_dir")

subject_id = 'subject1'

colourmap = colorRampPalette(viridis::viridis(n = 100))


vis.subject.morph.native(subjects_dir = subjects_dir,
```

```
                          subject_id = 'subject1',
                          measure = 'thickness',
                          hemi='both',
                          views = 't9',
                          draw_colorbar = "horizontal",
                          cortex_only = T, surface = "white",
                          makecmap_options = list('colFn'=colourmap))
rgl.snapshot("data/fig1d.png")
rgl.close()
```

```
#ROI1
#Figure 1e
suppressPackageStartupMessages(library(ciftiTools))
parc <- load_parc("Yeo_7")
view_xifti_surface(parc,
                   legend_embed = T,
                   hemisphere = "right")

rgl.snapshot("data/fig1e.png")

rgl.close()
```

```
#ROI1
#Figure 1X - combine with previous figure

set.seed(1993) #set a random seed (good practice for reproducibility)

parc <- load_parc("Schaefer_400")

ramdom_metric <- rnorm(400)

cii <- move_from_mwall(cii, NA)

parc_vec <- c(as.matrix(parc))

xii_metric <- c(NA, ramdom_metric)[parc_vec + 1]

xii1 <- select_xifti(cii, 1)

xii_metric <- newdata_xifti(xii1, xii_metric)

plot2 <- view_xifti_surface(xii_metric,
                            colors = viridis::viridis(n = 400),
                            borders = "black",
                            hemisphere = "left", )

rgl.snapshot("data/fig1f.png")

rgl.close()
```

```
#ROI2

#devtools::install_github("LCBC-UiO/ggsegGlasser")
```

```r
library(ggseg)
library(ggplot2)
library(ggsegGlasser)

set.seed(1993) #set a random seed (good practice for reproducibility)


base_atlas <- as.data.frame(na.omit(cbind(glasser$data$region, glasser$data$hemi)))

colnames(base_atlas) <- c("region", "hemi")

Effect <- rnorm(dim(base_atlas)[1]) #generate a random numbers for each roi in the atlas

base_atlas <- cbind(Effect, base_atlas)

cortex <- ggseg(atlas = glasser,
                .data = base_atlas,
                mapping=aes(fill=Effect),
                position="stacked",
                colour="black",
                size=.2,
                show.legend = F,
                plot.background = "white") + scale_fill_viridis_b()

sub_base_atlas <- as.data.frame(na.omit(cbind(aseg$data$region, aseg$data$hemi)))

colnames(sub_base_atlas) <- c("region", "hemi")

Effect <- rnorm(dim(sub_base_atlas)[1])

sub_base_atlas <- cbind(Effect, sub_base_atlas)

subcortex <- ggseg(atlas = aseg,
                   .data = sub_base_atlas,
                   hemisphere = "right",
                   mapping=aes(fill=Effect),
                   position="stacked",
                   colour="black",
                   size=.2,
                   show.legend = T,
                   plot.background = "white") +
  scale_fill_viridis_b()

ggsave(filename = 'data/fig1g1.png', plot = cortex, device = 'png', bg = 'white')
ggsave(filename = 'data/fig1g2.png', plot = subcortex, device = 'png', bg = 'white')
```

```r
make_ggseg3d <- function(attribute,
                         colour.pal = c("light yellow",
                                        "orange",
                                        "red",
                                        "dark red"),
                         hide.colourbar=FALSE,
                         output.png=FALSE,
```

```
                          file.name="ggseg_3d.png") {
  # Inputs:
  #' attribute = This must be a numeric vector in this: Left-Thalamus, Left-Caudate, Left-Putamen, Left

  library(ggseg3d)
  library(tidyr)
  library(dplyr)
  #remove(aseg_3d)
  aseg_3d <- aseg_3d
  aseg_3d <- tidyr::unnest(aseg_3d, cols = c(ggseg_3d))

  attribute.ggseg3d <- c(rep(NA, 4), attribute[1:4], rep(NA, 3), attribute[5:7],
                         rep(NA, 5), attribute[8:14],  rep(NA, 6))

  data <- dplyr::mutate(aseg_3d, attribute =  attribute.ggseg3d)
  #remove NA regions
  aseg_3d[which(is.na(data$attribute)),] <- NA
  aseg_3d <- tidyr::drop_na(aseg_3d)

  data[which(is.na(data$attribute)),] <- NA
  data <- tidyr::drop_na(data)
  data$attribute[data$attribute==0]<-NA #make 0 valus NA so they are set as grey in ggseg3d

  scene=list(camera = list(eye = list(x = 0, y = 1, z = -2.25)),
             aspectratio = list(x=1.6,y=1.6,z=1.6))


  plot <- ggseg3d::ggseg3d(.data = data,
                           atlas = aseg_3d,
                           colour = "attribute",
                           text = "attribute",
                           palette = colour.pal)
  plot <- remove_axes(plot)
  plot <- plotly::layout(plot,
                         scene = scene,
                         width = 600, height = 600)

  if(hide.colourbar==TRUE){
    plot <- plotly::hide_colorbar(plot)
  }

  if( output.png==TRUE){
    plotly::orca(plot, file =  file.name)
  }

  return(plot)

}

set.seed(1993)
volume <- sample(1:20, 14) #randomly generated data for 14 structures
make_ggseg3d(volume,
             colour.pal = c("white", "firebrick"),
```

```
             output.png = T, file.name =
               "data/fig1g3.png")
```

```
#Edge (also add 3d version)
#Fig1X

#remotes::install_github("sidchop/brainconn")
library(brainconn)

conmat <- example_weighted_undirected
degree <- rowSums(conmat)[which(rowSums(conmat)!=0)]
fig1h <- brainconn(atlas ="schaefer300_n7",
                   conmat=conmat,
                   node.size = degree/2,
                   view="ortho",
                   edge.color.weighted = T,
                   background.alpha = 0.4,
                   show.legend = F,
                   edge.color = scale_edge_colour_gradient2(low='yellow', mid='orange', high='red'))
ggsave(filename = 'data/fig1h.png', plot = fig1h, device = 'png', bg = 'white')


brainconn3D(atlas ="schaefer300_n7",
            conmat=conmat,
            show.legend = F)
```