Saving loon plots

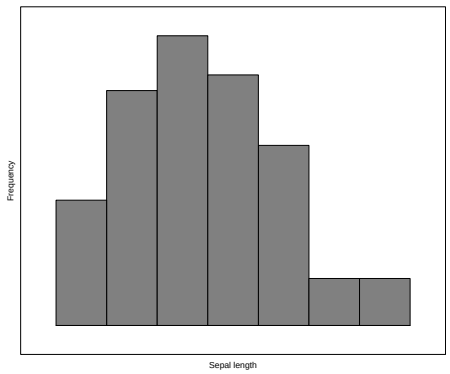
R.W. Oldford

Interactive analysis and static plots

When carrying out an interactive analysis, you might want to save some pictures to include in any later report. You might also want to save the state of some interactive plot so that you can pause your analysis, quit R, and later return to your analysis where you left off.

To illustrate how to do this, consider the classic data set `iris`. Here are three different **linked** plots that you might be display.

```
library(loon)
# First, make sure you always assign a loon plot to a variable
# at the same time that you create it.  This will give you access
# to it later.
#
#  A histogram, the name for the linkingGroup is arbitrary
h <- l_hist(iris$Sepal.Length, color = "grey", xlabel = "Sepal length",
            linkingGroup = "flowers")
#
# A scatterplot
p <- l_plot(iris$Petal.Width, iris$Petal.Length,
            color = "grey", size = 10,
            xlabel = "Petal Width", ylabel = "Petal Length",
            linkingGroup = "flowers")
#
# A serial axes plot using the first 4 columns of iris (i.e. no Species info).
sa <- l_serialaxes(iris[, 1:4],
            color = "grey", linkingGroup = "flowers")
#
# A static version of the plot can be easily produced.
plot(h)
```

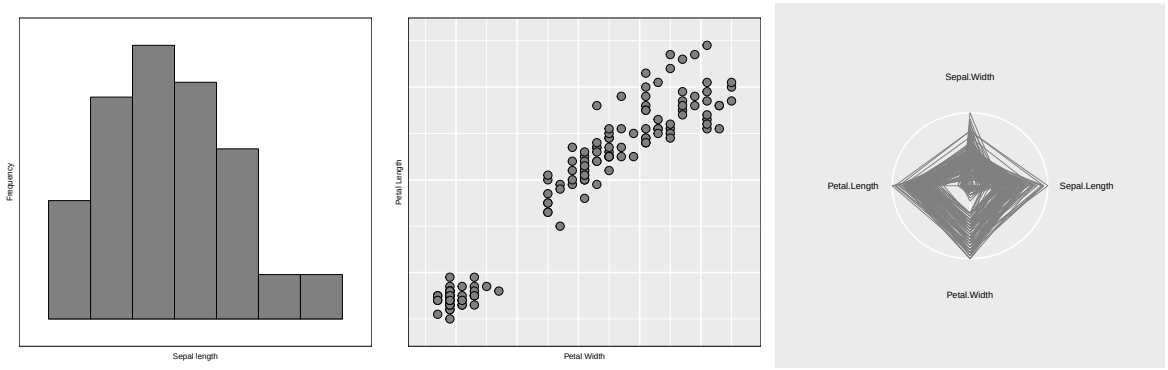


That plot will simply just appear (**as it is**) in the processed `RMarkdown` file.

If you have the `grid` and `gridExtra` packages loaded, you could arrange to see several such plots in one place.

```
library(grid); library(gridExtra)
```

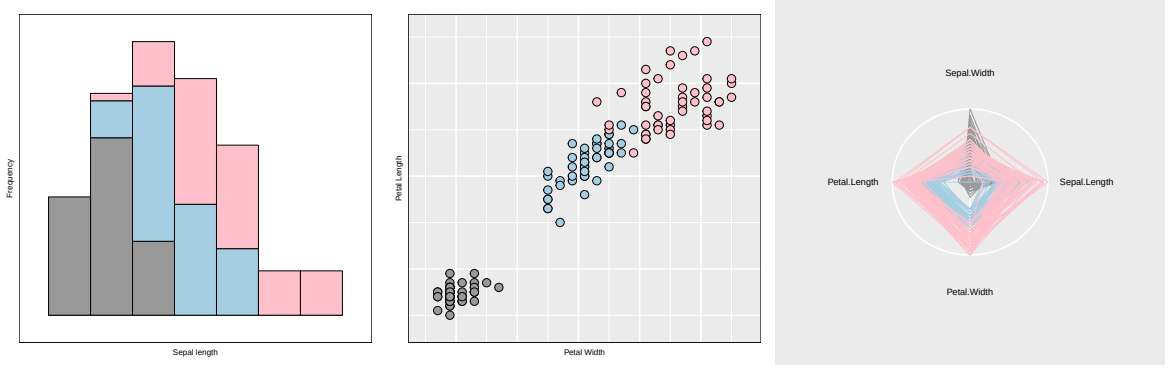
```
# First get the data stuctures corresponding to the static snapshots
# of the current plots.  Because we want to arrange them, we don't
# draw them at first.
plot_h <- plot(h, draw = FALSE)
plot_p <- plot(p, draw = FALSE)
plot_sa <- plot(sa, draw = FALSE)
# Now we draw them
grid.arrange(plot_h, plot_p, plot_sa, nrow = 1)
```



When interaction is only programmatic

This works fine when what you want to display has been entirely determined by programming done inside the `RMarkdown` code. For example, if the colours had been changed programmatically, the linked plots would change and plotting them again would reflect that change.

```
# Change the colour by changing it *programmatically* on the plot
#
h["color"] <- iris$Species  # use species to determine the colours
plot_h <- plot(h, draw = FALSE)
plot_p <- plot(p, draw = FALSE)
plot_sa <- plot(sa, draw = FALSE)
# And draw them again
grid.arrange(plot_h, plot_p, plot_sa, nrow = 1)
```



There are a couple of things to note here.

First, the plots had to be redrawn because they are static snapshots reflecting the what the interactive plot looked like **at the time** the snapshot was taken using `plot()`.

Second, all linked plots changed when the colour was changed only on the histogram.

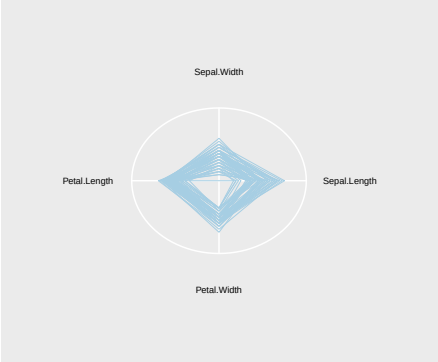
Finally, there are numerous `states` that **can be programmatically accessed and set** on an interactive plot. These states can be found by calling `names()` on the plot in question – e.g. `names(h)` returns the state names for the histogram `h`.

Knowing the states they allow access to values that have been changed, whether programmatically or interactively. In the same way, they can be used to change the states programmatically. For example,

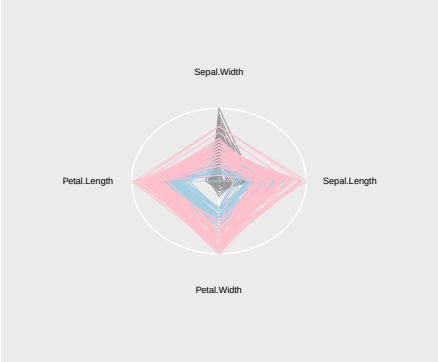
```
head(sa["color"])
```

```
## [1] "#999999999999" "#999999999999" "#999999999999" "#999999999999"
## [5] "#999999999999" "#999999999999"
```

```
sa["active"] <- iris$Species == "versicolor"
plot(sa)
```




```
sa["active"] <- TRUE
plot(sa)
```



A problem with doing things interactively is that any direct manipulation of the plots using the inspector or brushing, panning, zooming, etc. **is not being recorded.** And, if it is not recorded, it will need to be recorded in the `RMarkdown` file by you, as above, if you want the results to appear in the report.

This means you will need to save the state somehow, either as a snapshot or as data in a separate file to be recovered later (saving plot states like colours, selections, etc.).

In either case, information must be stored so that an `RMarkdown` file can read it back in and adjust the plots programmatically.

Saving shapshots for later display.

When you are doing things interactively, you may just want to save a current version of the plot, effectively a snapshot, to a file. Then, **after** you have completed your analysis you can read those back in.

You would save the current picture in a variety of ways.

Export from RStudio

Simply `plot(p)` and select the `Export` button above the static plot. Choose a file name and location for your plot.

Because `plot()` actually turns the loon plot into a `grid` graphic, the picture saves will be close but not necessarily exactly as it appears on screen. Typical differences include font sizes and names.

Use `l_export()`

This is a `loon` function to export a loon plot to a file. On the plus side, what you see is what you get. The loon plot saved to the file will be the same as it appears on the display.

Trouble is, depending on your `OS`, you might not be able to save the picture as any type of image. For example, `pdf` images will be available to nearly all OSes, `png` will not. You do need to specify the file suffix:

```
l_export(p, filename = "myplot_via_l_export.pdf", height = 500, width = 600)
```

Use R's `png()` device

In R, there are a variety of graphics "devices". These identify where Rwill draw a static plot. The call `dev.cur()` will tell you what the current device is.

One such device is `png()` which allows Rplots to be drawn as a `png` graphic. It would work like this

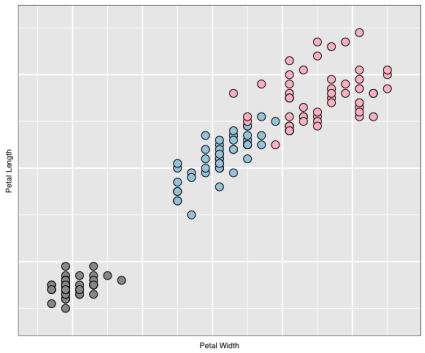
```
# turn the png graphics device on with name of the file
png(file = "myplot_via_R.png", width = 600, height = 500)
# Draw the static plot
plot(p)
# Turn the device off/close the file
dev.off()
```

Including saved graphics in your report

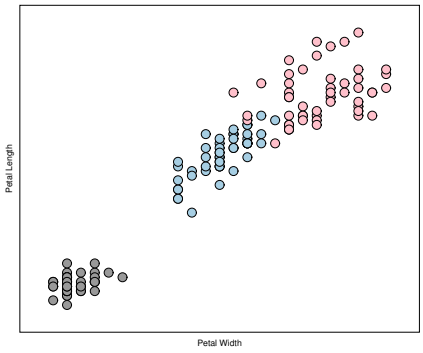
Having saved various snapshots to various files, you can now include them in your Rmarkdown file using the `include_graphics()` function from `knitr`.

Of course, this means that the various files mentioned above were previously saved (not created in the markdown file) and so can now be read into your RMarkdown report as follows:

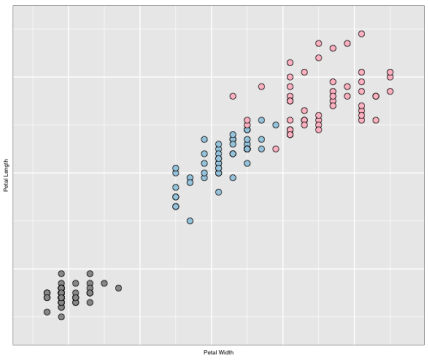
```
# The one exported from RStudio
knitr::include_graphics("myplot_via_RStudio.png")
```




```
# Followed by the one saved using l_export
# (note that background grid is missing)
knitr::include_graphics("myplot_via_l_export.png")
```

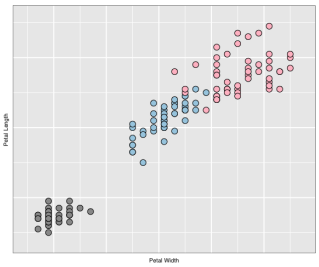



```
# And finally, the one saved using R's png device
knitr::include_graphics("myplot_via_R.png")
```

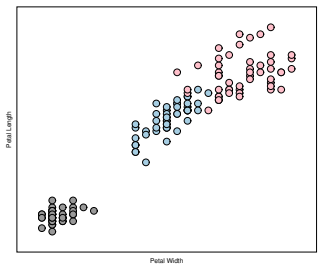


Alternatively, you could use LaTeX{} if you want more control. The following LaTeX{} code will centre the three plots in each column of a tabular lay out

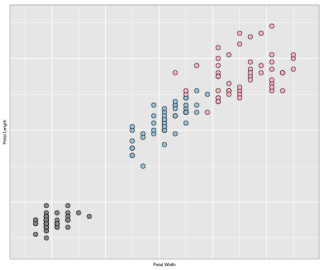
```
\begin{center}
  \begin{tabular}{ccc}
    \includegraphics[height = 0.15\textheight]{myplot_via_RStudio.png} &
    \includegraphics[height = 0.15\textheight]{myplot_via_l_export.pdf} &
    \includegraphics[height = 0.15\textheight]{myplot_via_R.png}
    \\
    {\scriptsize The one exported from \textsf{RStudio}} &
    {\scriptsize The one saved using \texttt{l\_export}} &
    {\scriptsize The one saved using \textsf{R}'s \texttt{png} device}
  \end{tabular}
\end{center}
```



The one exported from RStudio



The one saved using `l_export`



The one saved using R's `png` device

Reconstruct the plot from saved states

loon also provides some functionality to save the *current* states of a loon plot. As with saving snapshots, this would be done at any time during the interactive analysis. Then later, when it is time to write the report, the saved states

of the plot could be read back in and the plot *reconstructed programmatically* to be displayed in the report using `plot()` as above.

The use model is

1. Use `l_saveStates()` to save the current states of the loon plot to a file (repeat as often as desirable, saving to a *different* file each time).
2. Later, use `l_getStates()` to recover the saved states from the named file/
3. Construct a new loon plot of the desired type.
4. Copy the saved states to the new plot using `l_copyStates()` (alternatively, assign selected saved states of the new plot using `$`).
5. Produce a static picture of the new loon plot using `plot()` as above.

Note that only the first step happens in the interactive part of the analysis. Steps 2 through 5 programmatically recover the plot and its picture for incorporation in a report.

Of course in place of step 5, one could use steps 2 through 4 to restart an interactive analysis where one left off.

A simple example

A typical use will be simply recording the plot after some interaction with the display, usually involving linking between two displays, say the scatterplot and histogram above.

We save the states of the plot `p` in a file as

```
l_saveStates(p, file = "p_savedStates")
```

Later, when we construct a new plot (perhaps for a report), the states are recovered from file

```
p_savedStates <- l_getSavedStates(file = "p_savedStates")
```

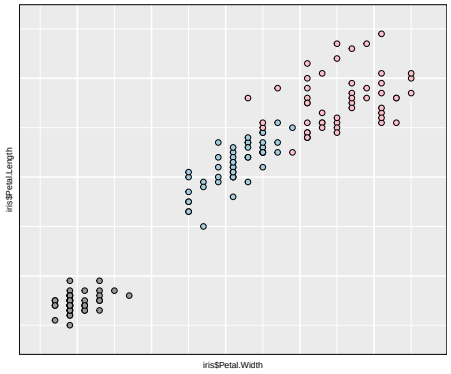
A new plot is constructed and the saved states transferred to it.

```
new_p <- l_plot(iris$Petal.Width, iris$Petal.Length)
l_copyStates(source = p_savedStates, target = new_p)
```

```
## NULL
```

and can be programmatically incorporated into a report as

```
plot(new_p)
```



Again, alternatively `new_p` could simply be a reconstruction so as to continue an interactive analysis from that point.

A more complex example

Again, by default, only the following states related to linking are saved

```
names(p_savedStates)
```

```
## [1] "color"        "active"       "selected"     "linkingKey"   "linkingGroup"
```

Sometimes, the interactive analysis will have been more involved. It could, for example, have involved panning and zooming, changing of glyphs, particular choices for labels and titles, and so on.

For example, through direct manipulation of the loon plot and inspector, the scatterplot `p` might at some point look something like this:

Here, the analyst zoomed in on the species "versicolor", jittered the locations of the points, and changed some of the glyphs to be large closed triangles. This involves many more states, all of which must be saved to reproduce the plot.

The states to be saved can be named explicitly to be most efficient in storage, providing of course one knows which are the states that need to be saved.

All of the states of a loon plot `p` are given by `names(p)`. At the risk of saving more information than necessary, we could choose to save all of the states of `p`, as in

```
l_saveStates(p,
             states = names(p),
             file = "p_focusOnVersicolor")
```

Now all of the states are available when this file is read back in.

```
p_focusOnVersicolor <- l_getSavedStates(file = "p_focusOnVersicolor")
```

When it comes to transferring the states to a new plot there are many choices. We could go with a large collection of display states (more than the default states but still excluding more "basic" states about panning and zooming for example). To see which states, we add the argument `returnNames = TRUE`:

```
l_copyStates(source = p_focusOnVersicolor,
             target = new_p,
             returnNames = TRUE)
```

```
##  [1] "glyph"            "itemLabel"        "showItemLabels"   "title"
##  [5] "showLabels"       "showScales"       "showGuides"       "background"
##  [9] "foreground"       "guidesBackground" "guidelines"       "minimumMargins"
## [13] "labelMargins"     "scalesMargins"    "color"            "selected"
## [17] "active"           "size"             "tag"              "useLoonInspector"
## [21] "selectBy"         "selectionLogic"
```

This excluded "basic" states through the argument `excludeBasicStates` whose default value is `TRUE`.

More, or different, states could be excluded by using the argument `exclude` to list them in a vector. Alternatively, the only states to be transferred could be made explicit by naming them in the value of the argument `states`.

Because the "basic" states were excluded by default, the new plot now looks like

```
plot(new_p)
```

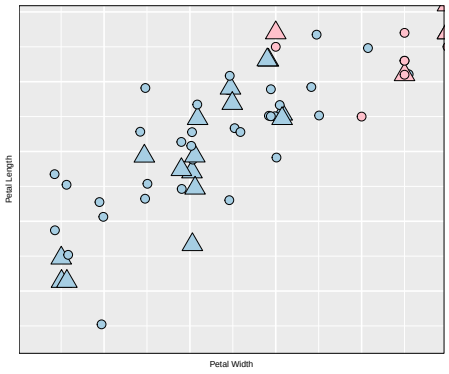
As can be seen, the glyph sizes and shapes were transferred but not the zooming or the jittering. To ensure we get an exact copy, we need to include all of the states, that is to exclude none of them.

```
l_copyStates(source = p_focusOnVersicolor,
             target = new_p,
             excludeBasicStates = FALSE)
```

```
## NULL
```

Now the `new_p` should look much like the original interactive plot did when it was saved.

```
plot(new_p)
```



which it does.