Summer Full Project Final Report

1. Summary of progress report

To recap the aim of the project, *CardinalVis* is a supplementary package for *Cardinal* to provide a graphical user interface (GUI) with a focus on interactive analysis and visualization of mass-spectrometry based imaging experiments. This GUI would assist researchers to more quickly understand their data by enabling rapid exploration of relevant ion images and associate mass spectra. This report will go over the work done on the module “selectVis” and integration of a busy indicator while the server perform computationally expensive operations. Finally, the report will also go over some drawbacks in Plotly, an interactive graphing library, which was intended to be integrated for greater interactivity, but was later dropped.

As described in the proposal, advances in imaging technology have lead to higher resolution imaging, at the cost of larger and complex files. Higher resolution images which can carry multiple samples and span several runs are especially challenging to work with due to the large file sizes and complexity. Building software which can thus scale efficiently to the increasing computation demand is of importance. *CardinalVis* tries to be as scalable as possible while providing flexibility to researchers who want to perform complex imaging experiments. Some processing time is inevitable though, which *CardinalVis* tries to indicate to the user using busy spinners.

2. Build environment

The following is a list of packages and other build information upon which *CardinalVis* is built.

1. Platform - x86_64-pc-linux-gnu (64-bit)
2. R - v3.6.1 (2019-07-05)
3. Cardinal [1] - v2.3.14
4. shiny [2] - v1.3.2
5. shinyWidgets [3] - v0.4.8
6. shinydashboard [4] - v0.7.1

3. Project components

3.1 Module `selectVis`

Cardinal allows researchers to manually select regions-of-interest or pixels on an imaging dataset by using the `selectROI` function. This feature is intended to allow for exploration of a smaller, thus more focused, part of the image. *CardinalVis* provides “selectVis”, a function which tries to imitate the same functionality. “selectVis” can be called from the

command line which launches a shiny dashboard to allow user to select multiple regions, as opposed to one region in “selectROI”, and return a list or factor as requested by the user.

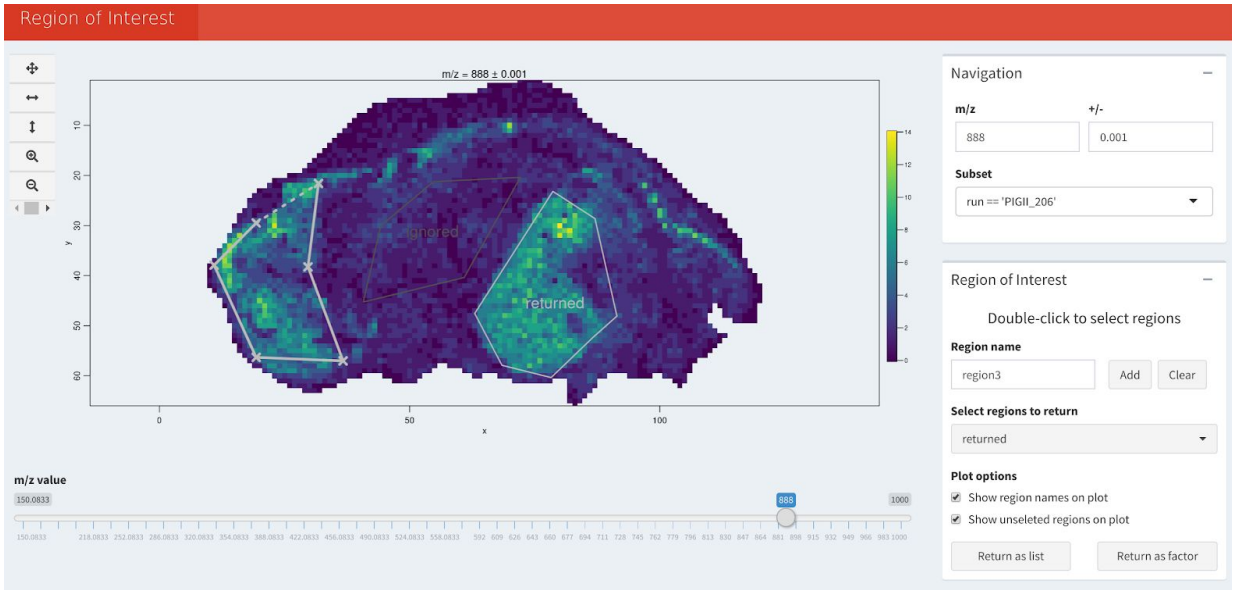

Figure 1: selectVis dashboard UI

3.1.1 Files

R/selectVis.R	- called by user to start the dashboard
R/ui.R	- selectUI function to inflate initial UI
R/server.R	- selectSever server function
R/module-selectView.R	- server function for module
R/module-selectViewUI.R	- UI function for module

3.1.2 Function overview

```
selectVis <- function(dataset, ..., mode) { ... }
```

`dataset`	- The name of an MSImagingExperiment object
`...`	- Additional arguments passed to the underlying `image` function (mz, plusminus, subset, formula, smooth.image, colorscale, fun, contrast.enhance)
`mode`	- Can be “region” or “pixels”

The GUI returns a list of boolean vectors identifying the points that lie within that region. The user can also pick to return a factor produced by `makeFactor` from Cardinal

3.1.3 Implementation

`selectVis` uses `shiny` to create a dashboard as shown in the image above. Internally, `selectVis` creates an application (if called from command line) which inflates the UI -- `selectUI` -- from `ui.R` and runs the server -- `selectServer` -- from `server.R`. The corresponding modules from `module-selectView.R` and `module-selectViewUI.R` are called to complete and tie the UI and server. At this point, the dashboard is ready for interaction.

3.1.4 Interactions

Basic interactions include the ability to update the current m/z (via the numeric input or the slider at the bottom), tolerance values, current subset of the image to view. Image zoom and pan options are also provided to the user.

For region of interest, the dashboard allows the user to double-click on the image to start forming a polygon as a way to pick the region he is interested in. A preview of the complete polygon is shown with solid lines connecting the points clicked and a dashed line showing the closed region - white polygon in above image. Once the user is satisfied, he can rename the region (Region name textbox), discard the region (Clear), or add it to the list of returned regions (Add button). When added, the polygon is completed and the corresponding name is added to the image - polygon “returned” in image. By default, each region is named “regionX” where X corresponds to the position of the region in the list of returns. Finally, the user can select which regions to return using the “Select region to return” dropdown. If a region is deselected, this is reflected in the image by turning the polygon black - polygon “error” in image. Some plot options are also provided to control the elements shown on the current plot. Finally, the user can pick to return the selected regions as a list of boolean vectors or a factor as defined in 3.1.2.

3.1.5 Challenges faced

As mentioned before, current technology allows researchers to record large experiments with multiple runs. To provide a scalable application, *CardinalVis* would have to support selecting multiple regions from across multiple runs. This is achieved by optimally caching the current selection and the state of the experiment when the user confirms a region.

3.2 Busy spinner

Some imaging experiments are large and require a significant amount of processing time. During this processing, the user is left with a blank image with no indication of processing. This could be cause for frustration if the processing time is over a few seconds and a simple way to indicate a busy server would be convenient. To do this, *CadinalVis* implements a busy spinner, that is displayed during long-running operations on the server side. Advancements in CSS have allowed defining animations programmatically avoiding the use of heavy GIF files and images. The spinner used here is taken from CSS loaders [5].

The idea behind busy spinner is to display the “busy” animation whenever the server is busy with a long-running computation and display the ion-image when the server is idle. Much of the inspiration for this is taken from `shinycssloaders` [9]. The problem that restricted the use of `shinycssloaders` was that the spinner was displayed even for small operations, like panning and zooming in the image, causing an impression of delay and overall “slowness” of the application. `shinybusy` [10] implemented a timeout that would help with this, but restricted the animation to a small space in the title bar of the dashboard, which was hard to notice. A way to use the best of both was our best option, a replaceable container with a timeout.

3.2.1 Implementation

As mentioned above, the animation is borrowed from [5], which is provided under the MIT license [6], allowing us to freely use the source with adequate reference. Although the shiny official documentation [7] defines using the `www` subfolder to hold all static CSS files, this method did not work uniformly across browser and runs for our use case, causing unintended side effects. To circumvent this, the animation is defined inline (also mentioned in [7]). This gives us the intended behaviour every time, across all browsers.

`shiny` provides JavaScript events that are triggered by the package when certain conditions are met, a full list of all events and documentation is available at [8]. For our use case, we will need `server:busy` and `server:idle` events that are respectively triggered whenever the server is busy processing and when the server is idle. First, the containers are defined that hold the images and animations. Then, using JavaScript, the application implements an event listener that manipulates these containers whenever these events are triggers, showing and hiding the containers (showing the image container and hiding animation container when server:idle is triggered, and vice-versa). A 500ms timeout is used to allow for small operations (panning, zooming) to not trigger the animation. Essentially, if `server:idle` is triggered within 500ms of `server:busy` being triggered no animation is displayed.

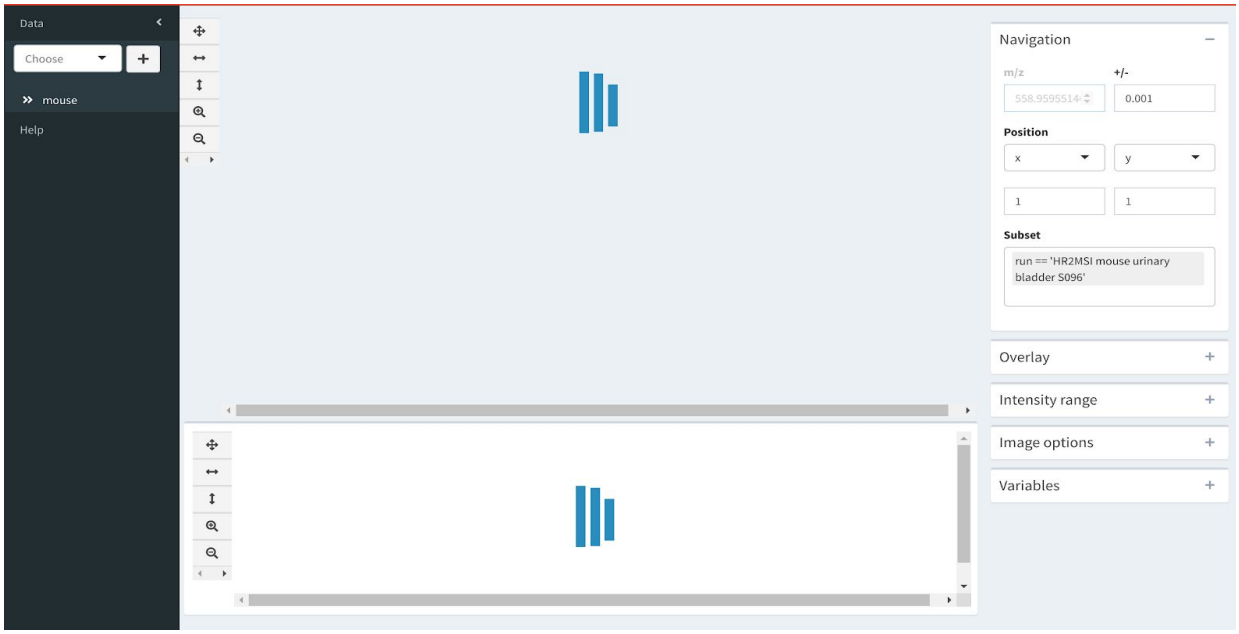

Figure 2: Indication of a busy server in msiVis

3.2.4 Current limitations and future work

Currently all spinners are coupled together causing every animation to be triggered regardless of which plot is actually updating. Implementing decoupled spinners would free the researcher to interacting with other plots/images that might be present on the dashboard while one processes, potentially reducing research time.

3.3 Integration of Plotly for greater interactivity

`Plotly`, an interactive graphing library, provides a convenient wrapper in R for using D3.js to produce plots. One of the main advantages provided is native support for several interactions, like panning, zooming, hover interactivity, etc, while allowing to render beautiful looking plots. The proposal mentioned exploring ways to integrate plotly into the current dashboard. After some experimentation, it was clear that plotly had certain drawbacks that would require significant time and effort to go around.

Two drawbacks that would render plotly inadequate for our use case are listed below:

1. Clicks - Clicking is only possible on an element in plotly. An element can be a line, a point, or any other graph object. But, given the relative density of peaks and the available white space, the user would expect clicking on a point within the white space to behave the same way as clicking the vertically closet point.

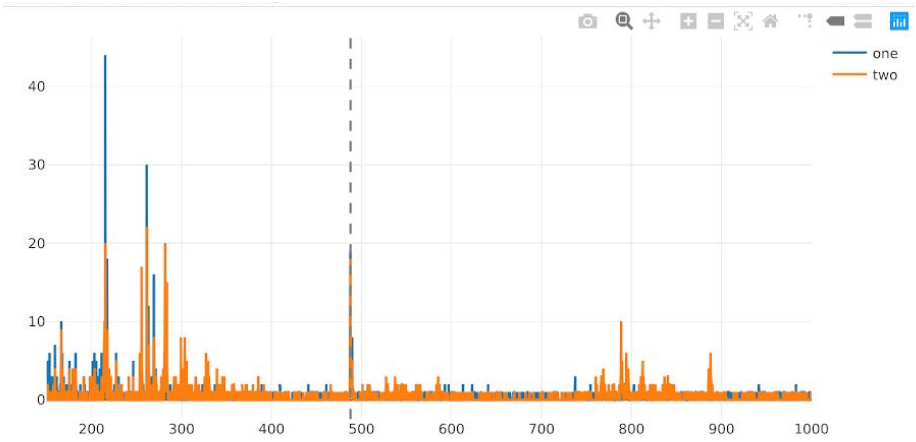

Figure 3: Spectrum plot in plotly

2. Zooming - Plotly support zooming and panning natively and provides and convenient “box” to view the selected area. The problem, however, is that once a click is detected on the plot, the zoom and pan information is lost causing the plot to reset. This behaviour is not desirable and does not help with rapid interaction.

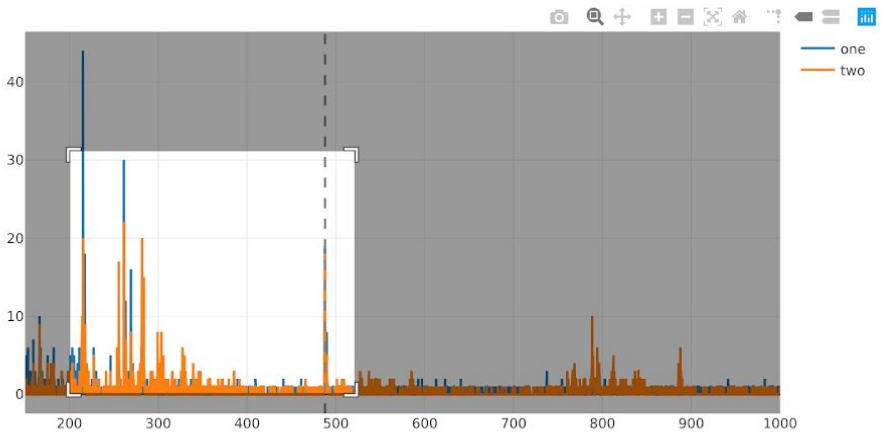

Figure 4: Clipping the spectrum plot in plotly, notice the convenient box.

As mentioned earlier, these drawbacks would require significant time and effort which led to the plan to integrate being dropped.

3.3.1 Possible alternative

Although powerful, Plotly is not the only interactive graphing library available. Exploration of libraries, like Vega and D3.js, will be left for future work. Vega is visualization grammar for creating, saving, and sharing interactive visualization designs. Vega acts like a high-level wrapper to D3.js and thus relies on D3.js. Using D3.js directly could also be favourable if Vega, like Plotly, has drawbacks for this project due to its design.

4. Future work

CardinalVis is still far from complete. There are a number of features that need to be implemented and integrated for this package to be release ready. Some of the work remaining include:

1. Implementation and integration of preprocessing steps allowed by *Cardinal*.
2. Implementation and integration of statistical methods.
3. Linking of modules to allow easier navigation.

5. Acknowledgements

I would like to take a moment to thank Dr. Kylie Bemis and Khoury College for providing me the opportunity to work on this independent project to help the research community. I would also like to thank Dr. Bemis for her patience and support throughout the duration of the project.

References

[1] Cardinal -- https://github.com/kuwisdelu/Cardinal

[2] shiny -- https://github.com/rstudio/shiny
[3] shinyWidgets -- https://github.com/dreamRs/shinyWidgets
[4] shinydashboard -- https://github.com/rstudio/shinydashboard
[5] CSS loaders -- https://projects.lukehaas.me/css-loaders/
[7] license -- https://github.com/lukehaas/css-loaders/blob/step2/LICENSE
[6] Style your apps with CSS -- https://shiny.rstudio.com/articles/css.html
[8] JavaScript Events in Shiny -- https://shiny.rstudio.com/articles/js-events.html
[9] shinybusy - https://github.com/dreamRs/shinybusy
[10] shinycssloaders -- https://github.com/andrewsali/shinycssloaders