SVGMapping tutorial

Raphaël Champeimont and Jean-Christophe Aude

CEA, iBiTecS, Integrative Biology Laboratory,
F-91191 Gif Sur Yvette, France.

jean-christophe.aude@cea.fr

May 31, 2011

Abstract

Here we present how to use `SVGMapping`, an R package that aims at displaying experimental data on custom-made SVG images. For example, it allows you to create an image showing a cellular mechanism with relevant genes displayed as circles, and colored according to their expression level in a microarray experiment.

SVG is an image format that has the property of being vector-based, meaning that an SVG image can be scaled as big as you want without loosing quality. Moreover, it is becoming increasingly popular with the possibility to integrate SVG files in HTML (a new feature of HTML 5) and to dynamically modify SVG content using JavaScript. We suggest to create your SVG drawings by using the free Inkscape[1] software.

Contents

[1] http://inkscape.org/

1 Quickstart

First we need to load the library into the current workspace.

```
> library(SVGMapping)
```

During this tutorial we will use the sample data set provided with this package. This data set contains expression levels of a subset of genes taken from an experiment in which authors have measured the transcriptional response of *Saccharomyces cerevisiae* to aeration after anaerobic growth (the complete original data set is available on the GEO[2] website under the accession number `GSE7140`). It also contains a yeast genes annotation matrix. More details about these two variables will be given later in the tutorial.

```
> data(yeastExprData)      # Expression data from geo:GSE7140
> data(yeastAnnotMatrix)   # Annotations from SGD
```

Typical `SVGMapping` usage involves three steps. First, one has to load the source template. This is done using the `loadSVG()` function. On this example we will use a yeast TCA cycle pathway sketch.

```
> TCAtemplate <- system.file("extdata/example.svg", package="SVGMapping")
> mysvg <- loadSVG(TCAtemplate)
```

The second step concerns the data *mapping* itself. This task is done using the `mapDataSVG()` function. A key feature of this package is its ability to add multiple information on the same template by iteratively calling this function. On this very basic example only one call to this function will be necessary. Here expression log-ratios will be used to select background colors. Annotations with fold-change values will be used to build the javascript tooltip windows. Full details about the parameters of this function will be given in the next paragraphs.

```
> logratios <- yeastExprData[,1]
> foldchange <- ifelse(logratios>=0, 2^(logratios), -2^-logratios)
> mapDataSVG(mysvg,
+            numData=logratios,
+            tooltipData=foldchange,
+            annotation=yeastAnnotMatrix)
```

The last step is straightforward: we use the `saveSVG()` function to store the altered template on the filesystem. Here this will create a file named `output1.svg` inside the current directory. To view an SVG file, you can use a modern Web browser (*ie* compliant with the latest W3C standards) that supports the SVG image format. Safari 5, Google Chrome or Firefox 4 are currently supported browsers that runs on Windows, OS/X and Linux. Another method is to use a dedicated drawing software such as Adobe Illustrator or Inkscape. Here one should see a drawing similar to the one on figure˜1, with a tooltip window appearing when your mouse cursor is over the circles (`MDH1` on this figure) that depict genes.

```
> saveSVG(mysvg, file="output1.svg")
```

Alternatively, one can call `showSVG(mysvg)` to automatically create a temporary file and open it in the default browser.

[2] http://www.ncbi.nlm.nih.gov/geo

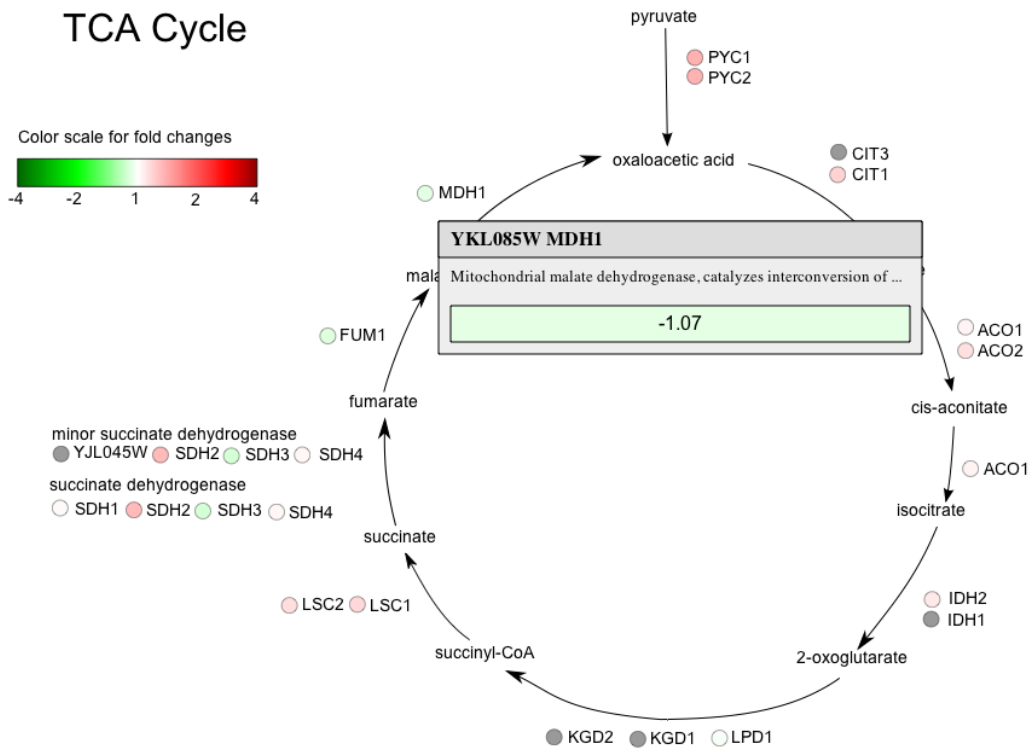

Figure 1: Screenshot of the produced "output1.svg" file

Hereafter we detail the template formatting and mapped data structure used to do this mapping . First, the template SVG file was created using Inkscape (a freely available program to edit SVG images). You retrieve the full path of the template using this command below.

```
> system.file("extdata/example.svg", package = "SVGMapping")
```

If you open this file with Inkscape, you'll observe that, unlike the output file, circles for genes are all colored in grey. Remember that graphical shapes are identified using the *label* attributes. This attribute can be modified using the *object properties* dialog window (see the *Object* menu). If you open this window for grey circles you will notice labels related to yeast genes (*ie* `YKL085W`, `YPL262W`). `SVGMapping` uses these labels as a key index to select rows in the data matrix (*numData*) to retrieve expression values. For example, circles related to the genes `MDH1` and `FUM1` are labeled using related Orf names `YKL085W` and `YPL262W`. Thus, expression values using these Orf names will be used to select the corresponding color for filling the shape (this is the default action of the `mapDataSVG()` function). You can retrieve these expression values (as $\log_2(R/G)$) using:

```
> yeastExprData[c("YKL085W", "YPL262W"), 1]
```

```
   YKL085W    YPL262W
-0.1025448 -0.1268348
```

By default these values are converted to colors using a green-white-red gradient in the range $[-2, 2]$ (see the color scale on figure 1). You can customize the color scale and mapping range by passing optional arguments to `mapDataSVG` (more information on colors in section 4.1 on page˜8).

You can observe that the displayed value on the tooltip window for the MDH1 gene (see figure 1) is -1.07 and differ from the value used to map colors -0.1025448. Indeed, we have proceeded to a systematic conversion of $\log_2(R/G)$ to fold changes using the formula:

$$\text{fold.change}(lgr) = \begin{cases} 2^{lgr} & \text{if } lgr \geq 0, \\ -2^{-lgr} & \text{otherwise} \end{cases}$$

We did this conversion because users sometimes prefer fold-changes over $\log_2$-ratios.

The other thing you can see on the tooltip is a name and description of the gene. This information is provided by the annotation parameter. In our example these annotations are taken from the yeast2.db R package˜[2], which we converted to a matrix. The content of the first three rows of this matrix is:

```
> substring(yeastAnnotMatrix[1:3, ], 0, 80)
```

```
        name
YAL001C "YAL001C TFC3"
YAL002W "YAL002W VPS8"
YAL003W "YAL003W EFB1"
        description
YAL001C "Largest of six subunits of the RNA polymerase III transcription initiation facto"
YAL002W "Membrane-associated protein that interacts with Vps21p to facilitate soluble vac"
YAL003W "Translation elongation factor 1 beta; stimulates nucleotide exchange to regenera"
        url
YAL001C "http://www.yeastgenome.org/cgi-bin/locus.fpl?locus=YAL001C"
YAL002W "http://www.yeastgenome.org/cgi-bin/locus.fpl?locus=YAL002W"
YAL003W "http://www.yeastgenome.org/cgi-bin/locus.fpl?locus=YAL003W"
```

As you can obsereve yeastAnnotMatrix is indexed by the same *labels* set in the SVG template. Besides the annotations displayed in the tooltip window, it contains an URL to redirect the navigator when clicking on gene circles[3].

2 Single condition modes

2.1 Line colors

It is possible to change line (stroke) colors, as illustrated by this example. Arrows were labeled using Inkscape with names like *'from-pyruvate'*. As you can see in the sample code below, by calling mapDataSVG several times it is possible to combine different information in the same file.

First we load the template:

```
> mysvg <- loadSVG(TCAtemplate)
```

Then, we color gene associated circles using expression levels (see section 1 on page˜2).

[3] In the example, URLs point to the "Saccharomyces Genome Database" website˜[1]

```
> mapDataSVG(mysvg,
+            numData=logratios,
+            tooltipData=foldchange,
+            annotation=yeastAnnotMatrix)
```

For each reaction, we assign an arbitrary "expression level" equal to the mean of the expression levels of the involved enzymes.

```
> reacdata <- c(mean(yeastExprData[c("YGL062W","YBR218C"),1]),
+                    yeastExprData["YKL085W",1],
+                    yeastExprData["YPL262W",1]
+               )
> names(reacdata) <- c("from-pyruvate",
+                      "from-malate",
+                      "from-fumarate")
```

The `reacdata` matrix is then used to map the **stroke** color of reaction lines, using the same color gradient as microarray (after removing lightest colors).

```
> mapDataSVG(mysvg,
+            numData=reacdata, mode="stroke",
+            tooltipData=NULL,
+            col=microarrayColors[c((1:333),(666:1000))])
```

Finally we save the modified SVG file in the current directory under the name *'output3.svg'* (see figure 2).

```
> saveSVG(mysvg, file="output3.svg")
```

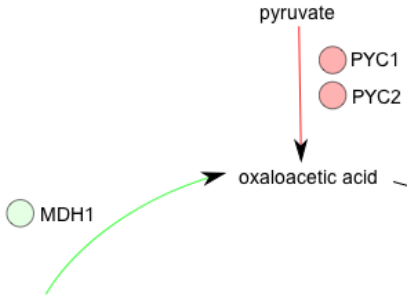

Figure 2: Screenshot of the produced "output3.svg" file

2.2 Filling Erlenmeyers

This package features a mode to display filled flasks where the liquid level is given in the data matrix, as a real number between 0 and 1. The following example illustrates this usage, and produces the result that can been seen on figure˜3 (the data used there is completely fake).

First we load a new template that contains the Erlenmeyers shapes besides the metabolites names.

```
> TCAtemplate2 <- system.file("extdata/example2.svg", package="SVGMapping")
> mysvg <- loadSVG(TCAtemplate2)
```

Our "fake" dataset contains "fake" metabolites concentration (in $mmol.l^{-1}$ unit) for "real" metabolites.

```
> mydata <- c(10, 25, 50)
> names(mydata) <- c("citrate", "cis-aconitate", "isocitrate")
```

The first `mapDataSVG()` instruction will set the filling colors and tooltips displayed values (*ie* we just add the unit).

```
> mapDataSVG(mysvg,
+            numData=mydata, mode="fill",
+            col=c("#FF0000","#00FF00","#0000FF"),
+            colrange=range(mydata),
+            tooltipData=paste(mydata, "mmol/L"))
```

The second `mapDataSVG()` instruction will set the filling heights using scaled values (in the range $[0, 1]$). Resulting image is saved in the current directory under the name *'output4.svg'* (see figure 3).

```
> mapDataSVG(mysvg,
+            numData=mydata/100, mode="partial-fill")
> saveSVG(mysvg, file="output4.svg")
```

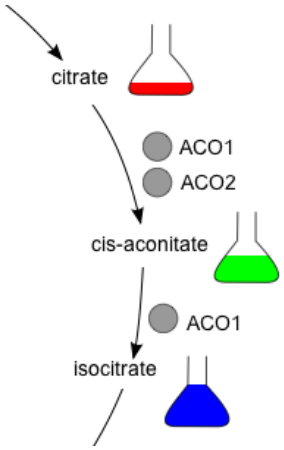

Figure 3: Screenshot of the produced "output4.svg" file

As you can see, it is possible to combine color change and filling level, but the color needs to be changed first. Indeed, the `partial-fill` mode will use the color defined by the `fill` mode.

Of course any SVG shape will work, so you can use this feature to simulate the filling of any closed graphical shape (*eg* battery energy level).

Note, however, that `SVGMapping` only defines the relative height, and does not know the surface which is actually filled, so depending on the shape, the surface filled will not necessarily be proportional to the value in the data matrix. It is your responsibility to apply a correction the numeric values if needed.

2.3 Other features

It is also possible to change opacity and line width, see the manual page of `mapDataSVG` for more information.

Functions are also provided if you want to manually edit the SVG data, see `setAttributeSVG` and `setStyleSVG`. As `SVGMapping` is built upon the XML R package, you can directly customize the SVG XML instructions.

3 Multi-conditions modes

3.1 Pie charts

Our program has a mode to replace circles by pie charts, with each slice corresponding to a different experiment. Here we illustrate this by showing expression data at different times (0, 5, 10, 20, 60, 120 minutes).

Let's load the TCA template previously used.

```
> mysvg <- loadSVG(TCAtemplate)
```

This time we will use all columns of the `yeastExprData` matrix (one column per time).

```
> logratios <- yeastExprData
> foldchange <- ifelse(logratios>=0, 2^(logratios), -2^-logratios)
```

Then we proceed as previously using the *pie* mode:

```
> mapDataSVG(mysvg,
+            numData=logratios, mode="pie",
+            tooltipData=foldchange,
+            annotation=yeastAnnotMatrix)
> saveSVG(mysvg, file="output2.svg")
```

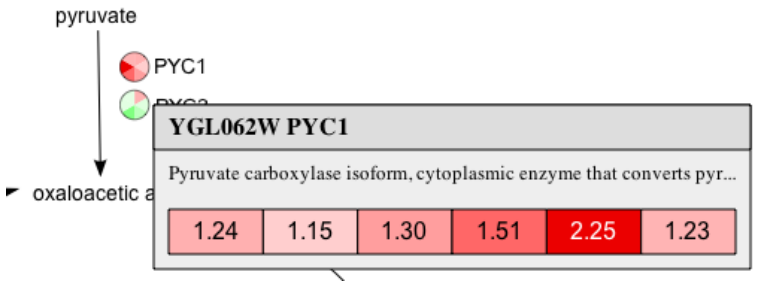

Figure 4: Screenshot of the produced "output2.svg" file

Advice: To avoid confusing the user with many irrelevant colors, you might want to replace log-ratios that are not significant by zeros in order to have them displayed in white.

3.2 Color stripes

An alternative to pie charts when comparing several experiments is to use color stripes. The following example illustrates this function and gives the result shown in figure˜5.

```
> mysvg <- loadSVG(TCAtemplate2)
> mydata <- matrix(c(-2, -1.5, -1, -0.5, 2, 1.5, 1, 0.5), ncol=4, byrow=TRUE)
> rownames(mydata) <- c("malate", "fumarate")
> mapDataSVG(mysvg, mydata, mode="fill")
> saveSVG(mysvg, file="output5.svg")
```

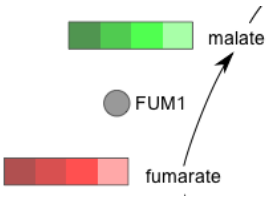

Figure 5: Screenshot of the produced "output5.svg" file

As you can see, no mode is specified so the *'fill'* mode is used by default. In the case of a multi-column data matrix, it will automatically switch to color stripes. It can be seen as a generalization of the *'fill'* mode. Note, however, that color stripes are implemented as gradients while simple color filling is just a color specification. Therefore only the single condition *'fill'* mode can be used as a pre-coloring step if followed by the *'partial-fill'* mode.

4 Advanced settings

4.1 Colors

All functions that produce colors in `mapDataSVG` call `computeExprColors` to compute colors from numeric data. This function uniformly maps a range $[a, b]$ to a list of colors, by making the assumption that the first color should be mapped to a and the last to b. When using `mapDataSVG`, these settings are called `col` and `colrange`. The default is $a = -2$ and $b = 2$.

If you want to display a continuous variable using colors, you need to provide a list of colors with a gradient containing all possible colors in your range. The default gradient, `microarrayColors`, contains 1000 colors that go from green to red (see figure˜1).

```
> mysvg <- loadSVG(TCAtemplate2)
> mapDataSVG(mysvg, matrix(seq(-2,2,by=0.5), nrow=1, dimnames=list("malate", NULL)))
> showSVG(mysvg)
```

You can specify a different list of colors. Note that it is recommended to provide a long list of colors so that `computeExprColors` can select the right colors with a maximum precision. A gradient with 1000

colors is a typical good length because it provides a high precision. Increasing it even more makes no sense since there is a finite number of possible colors in a computer, and a human being cannot make the difference between infinitely near colors.

On this example below we use a custom rainbow gradient rather than the default microarray green-white-red gradient.

```
> mysvg <- loadSVG(TCAtemplate2)
> mapDataSVG(mysvg, matrix(seq(-2,2,by=0.5), nrow=1, dimnames=list("malate", NULL)),
+            col=substring(rainbow(1000), 0, 7))
> showSVG(mysvg)
```

The `substring` call is required because the `rainbow()` function returns a list of colors in the `#RRGGBBAA` format while SVG uses the `#RRGGBB` format.

You might also be interested in **discrete** color mapping (where colors are selected using their position in the *col* list):

```
> mysvg <- loadSVG(TCAtemplate2)
> mapDataSVG(mysvg, matrix(c(1,2,3,2,3,1,2), nrow=1, dimnames=list("malate", NULL)),
+            col=c("#CC0000", "#00CC00", "#0000CC"), colrange=c(1,3))
> showSVG(mysvg)
```

References

[1] SGD project. "Saccharomyces Genome Database" `http://downloads.yeastgenome.org/`.

[2] Marc Carlson, Seth Falcon, Herve Pages, and Nianhua Li. *yeast2.db: Affymetrix Yeast Genome 2.0 Array annotation data (chip yeast2).* R package version 2.4.5.