DAMN: Drawn Array's Multistep Normalisation

Goal
Visualisation of expressions of spots in 2-channel microarrays (or any other type of two-channel data).

Dependencies
- tibble (>= 2.1.1),
- dplyr (>= 0.8.0.1),
- shiny (>= 1.6.0),
- shinyFiles (>= 0.9.0),
- ggplot2 (>= 3.3.5),
- magrittr (>= 2.0.1)

Input
Delimited text file(s) containing at least four columns: CH1I, CH1B, CH2I, CH2B.

Output
After importing the data, DAMN allows different ways to visualise your data:

1. **Raw**: No normalisation was performed. Spot intensities are plotted.
2. **MA-regression**: MA-plot („minus“ against „average“) is built;
3. and **normalised** with regression.
4. **Centering**: Results from MA-regression normalisation are centered, i.e. means of all distributions become 0.
5. **Scaling**: Centered distributions are scaled, i.e. their standard deviations become 1.
6. **Distribution normalisation**: one distribution is created from all other distributions.

Installation
```
install.packages(„devtools“) # If you do not have devtools yet.
devtools::install_github("GilaZeus/DAMN")
```

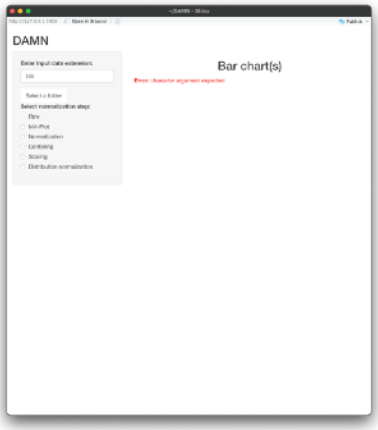
How to work
The simplest way is to start `run_app()`:

```
DAMN::run_app()
```

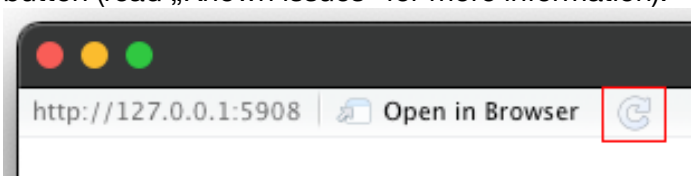
Or if you want to work with other functions:

```
library(DAMN)
run_app()
```

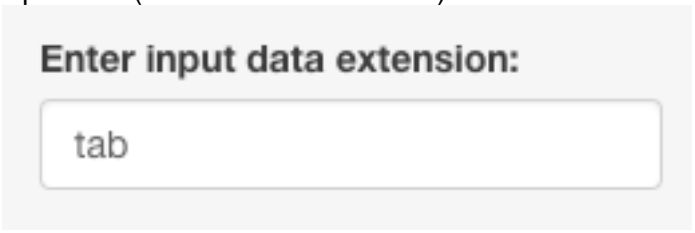
After that a window will appear (or a web-page will be opened, if you work from terminal):



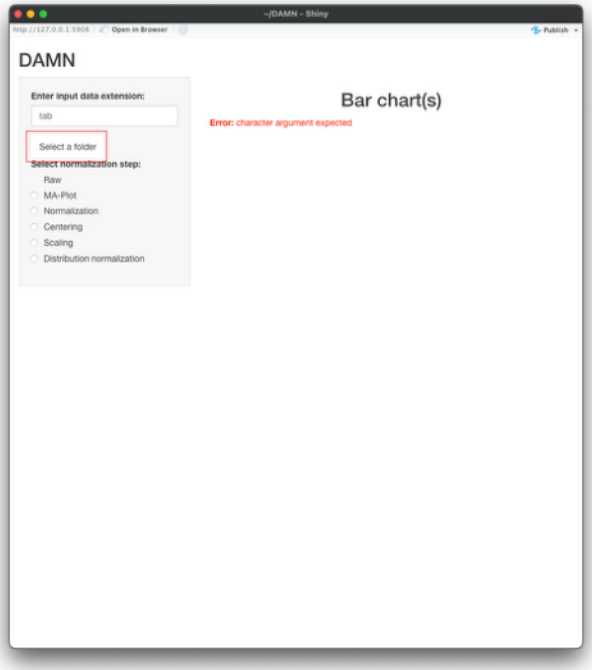
Now press the refresh-button (read „Known issues“ for more information):

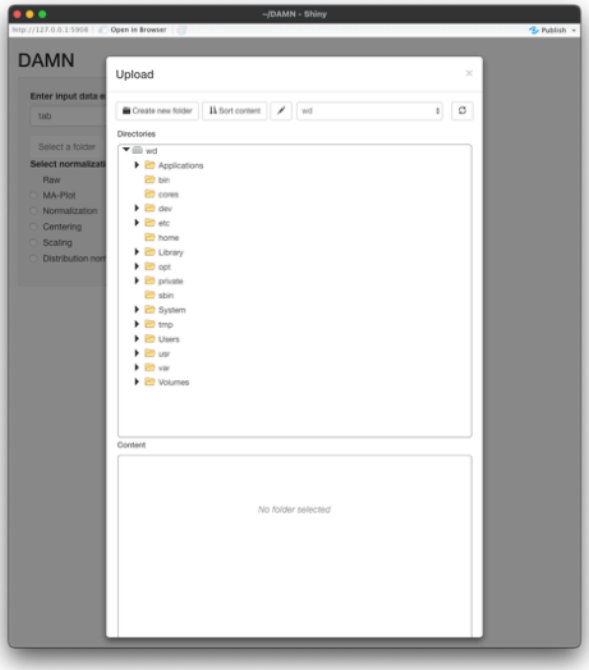


Enter the extension of input files (default value: .tab-files):

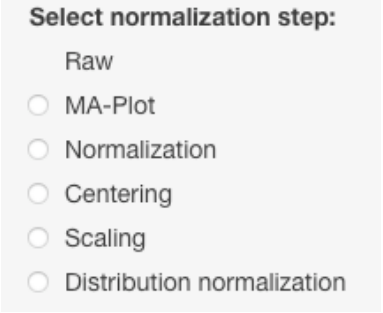


And select a folder with data you want to plot:

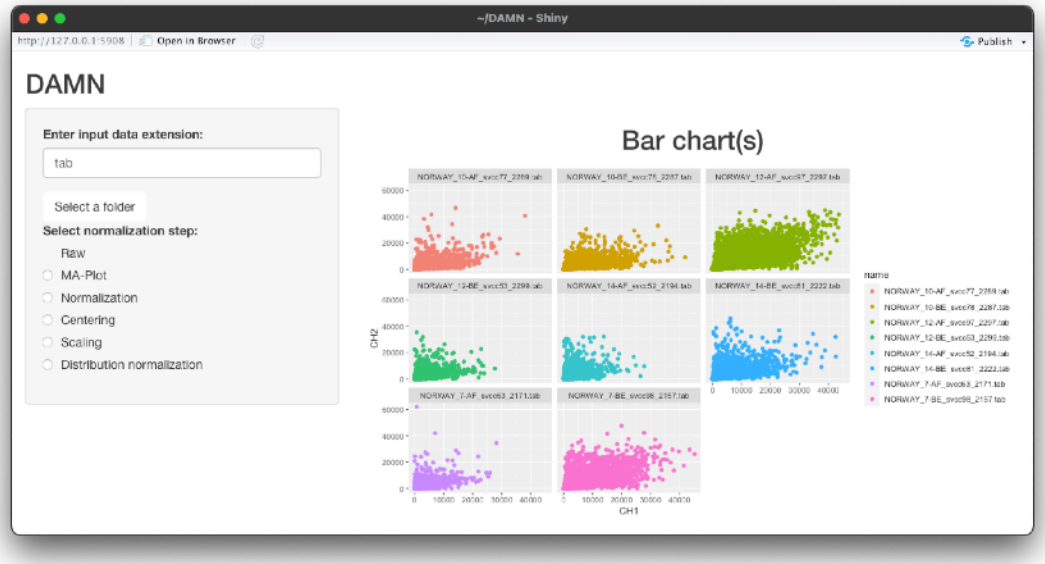


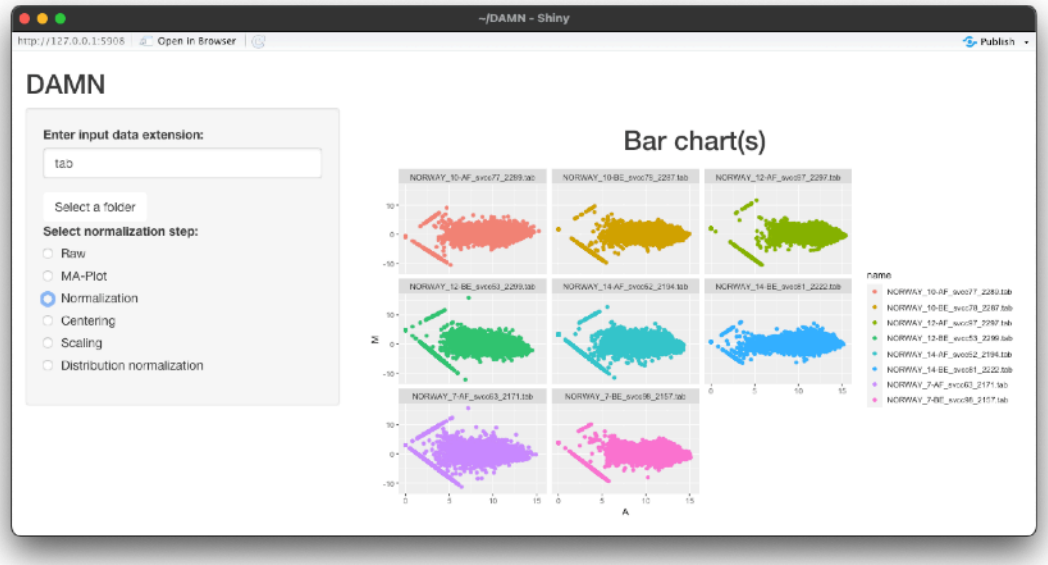


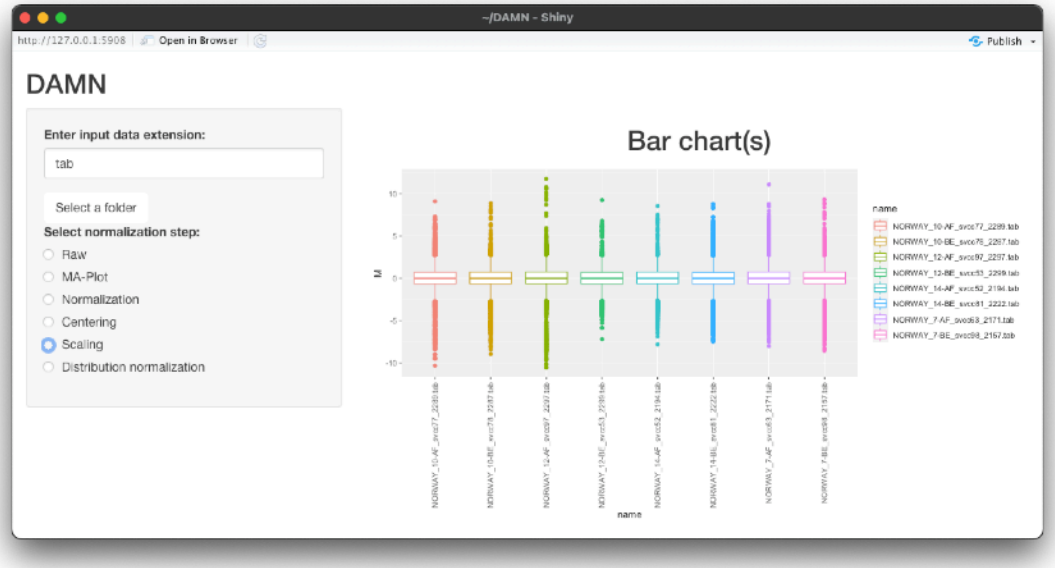
After that you can use the radio buttons to select an appropriate step of normalization:



For example, some graphs for some of the Perou data from our course:







Known issues

Error message after shiny start

Description

„`Error: character argument expected`“ in Shiny window, „`Warning: Error in setwd: character argument expected`“ in the console

Solution

Simply ignore, after selecting a folder it will disappear.

Not reacting „Select a folder“ button

Description

After opening a shiny window with `run_app()`, it isn't possible to select a folder with input data: it does not react on clicking.

Solution

Refresh the shiny window or its web page, after that everything should work properly.

Functions

Run App

Description

Run Shiny App for more comfortable plotting.

Usage

```
run_app(gui = ui, gserver = server)
```

Arguments

`gui:` ui argument of a shiny app.
`gserver:` server argument of a shiny app.

File preparation

Description

Prepare a single microarray dataset for plotting. Reading data and selecting columns of interest.

Usage

```
prepare_single_microarray(file_name)
```

Arguments

`file_name`: Name of a delimited file containing at least SPOT, CH1B, CH1I, CH2I and CH2B columns.

Value

Tibble containing SPOT, CH1B, CH1I, CH2I, CH2B and name columns.

Purge

Description

Remove all columns except M, name and SPOT.

Usage

```
purge(microarray)
```

Arguments

`microarray`: A tibble containing M, name and SPOT columns.

Value

A tibble containing only M, name and SPOT columns.

Merge datasets

Description

Merge *.tab datasets from a directory. Safe them as tibble with M, name and SPOT columns. Secure switching back to the user-defined file directory.

Usage

```
merge_datasets(directory = getwd(), file_ext = "tab", step = "between")
```

Arguments

`directory = getwd()`: Working directory with delimited files.
`file_ext = "tab"`: File extension of delimited files containing M, name and SPOT columns.
`step = "between"`: Determine step of normalization: "between" means between array step.

Value

A tibble containing M, name and SPOT columns, if step = "between", or CH1, CH2, name and SPOT in other cases.

Remove noise

Description

Subtract background noise from every channel.

Usage

```
subtract_background(microarray)
```

Arguments

`microarray`: A tibble containing at least SPOT, CH1B, CH1I, CH2I, CH2B and name columns.

Value

Tibble containing input columns and two new columns CH1 and CH2. All values <= 0 become 1.

Transform channels

Description

Add (M)ean and (A)verage to a tibble containing CH1 and CH2 columns.

Usage

```
mean_average(microarray)
```

Arguments

`microarray`: A tibble containing at least CH1, CH2 and name columns.

Value

Same table with two new columns: M(ean) and A(verage).

Normalization

Description

Normalize mean and average.

Usage

```
normalize(microarray)
```

Arguments

`microarray`: A table containing at least M, A and name columns.

Value

Same table normalized and with NA removed.

Centering

Description

Centering step of between array normalization. Subtract median from each value, so the medians become 0.

Usage

```
center_norm(microarrays)
```

Arguments

`microarrays`: A tibble containing M and name columns.

Value

Same tibble with M column centered.

Scaling

Description

Scaling step of between array normalization. Divide each value through MAD, so std. deviations become 1.

Usage

```
scale_norm(microarrays)
```

Arguments

`microarrays`: A tibble containing M and name columns.

Value

Same tibble with M column scaled.

Distribution normalization

Description

Sort all values in ascending order, build a mean value for each position in the order, so distributions of all experiments become the same.

Usage

```
distribution_norm(microarrays)
```

Arguments

`microarays`: A tibble containing M and name columns.

Value

A new tibble with "number" and M columns, M column is normalized.

Used R functions

Name	Package	Description
`log2()`	`base`	Wrapper for `log()`, computes binary (i.e., base 2) logarithms.
`getwd()`	`base`	Returns an absolute filepath representing the current working directory of the R process.
`setwd()`	`base`	`setwd(dir)` is used to set the working directory to `dir`.
`dir()`	`base`	Produces a character vector of the names of files or directories in the named directory.
`paste()`	`base`	Concatenates vectors after converting to character.
`select()`	`dplyr`	Select (and optionally rename) variables in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. a:f selects all columns from a on the left to f on the right). You can also use predicate functions like `is.numeric` to select variables based on their properties.
`mutate()`	`dplyr`	Adds new variables and preserves existing ones. New variables overwrite existing variables of the same name. Variables can be removed by setting their value to `NULL`.
`group_by()`	`dplyr`	Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group".
`ungroup()`	`dplyr`	Removes grouping.
`add_row()`	`dplyr`	This is a convenient way to add one or more rows of data to an existing data frame.
`ggplot()`	`ggplot2`	Initializes a `ggplot` object. It can be used to declare the input data frame for a graphic and to specify the set of plot aesthetics intended to be common throughout all subsequent layers unless specifically overridden.
`aes()`	`ggplot2`	Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. Aesthetic mappings can be set in `ggplot()` and in individual layers.
`geom_point()`	`ggplot2`	The point geom is used to create scatterplots.
`facet_wrap()`	`ggplot2`	Wraps a 1d sequence of panels into 2d.
`geom_boxplot()`	`ggplot2`	Compactly displays the distribution of a continuous variable. It visualises five summary statistics (the median, two hinges and two whiskers), and all "outlying" points individually.
`theme()`	`ggplot2`	Themes are a powerful way to customize the non-data components of your plots: i.e. titles, labels, fonts, background, gridlines, and legends.
`element_text()`	`ggplot2`	In conjunction with the theme system, the `element_` functions specify the display of how non-data components of the plot are drawn.
`parseDirPath()`	`shinyFiles`	This function takes the value of a `shinyFiles` button input variable and converts it to be easier to work with on the server side.

Name	Package	Description
`lm()`	`stats`	Used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance.
`coef()`	`stats`	A generic function which extracts model coefficients from objects returned by modeling functions. `coefficients` is an alias for it.
`na.omit()`	`stats`	Returns the object with `NA` values removed.
`mad()`	`stats`	Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.
`median()`	`stats`	Compute the sample median.
`as_tibble()`	`tibble`	Turns an existing object, such as a data frame or matrix, into a so-called `tibble`, a data frame with class `tbl_df`.
`tibble()`	`tibble`	`tibble()` constructs a data frame. It is used like `base::data.frame()`, but with a couple of differences.
`read.delim()`	`utils`	Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Used shiny control widgets

Widget	Parameters (only non standard)	Value
`textInput()`	`inputId`	`"extension"`
	`label`	`"Enter input data extension:"`
	`value`	`"tab"`
`shinyFiles::shinyDirButton()`	`id`	`"dir"`
	`label`	`"Select a folder"`
	`title`	`"Upload"`
	`multiple`	`FALSE`
`radioButtons()`	`inputId`	`"step"`
	`label`	`"Select normalization step:"`
	`choices`	`c("Raw" = "raw", "MA-Plot" = "maplot", "Normalization" = "norm", "Centering" = "cent", "Scaling" = "scale", "Distribution normalization" = "distr")`
`h2()`	`...`	`"Bar chart(s)", align = "center"`

Other used shiny functions

Name	Description
`fluidPage()`	Functions for creating fluid page layouts. A fluid page layout consists of rows which in turn include columns. Rows exist for the purpose of making sure their elements appear on the same line (if the browser has adequate width). Columns exist for the purpose of defining how much horizontal space within a 12-unit wide grid it's elements should occupy. Fluid pages scale their components in realtime to fill all available browser width.
`plotOutput()`	Render a `renderPlot()` or `renderImage()` within an application page.
`renderPlot()`	Renders a reactive plot that is suitable for assigning to an output slot.
`shinyApp()`	These functions create Shiny app objects from either an explicit UI/server pair (`shinyApp`), or by passing the path of a directory that contains a Shiny app (`shinyAppDir`).
`shinyFiles::parseDirPath()`	This function takes the value of a shinyFiles button input variable and converts it to be easier to work with on the server side. In the case of file selections and saving the input variable is converted to a data frame (using `parseFilePaths()` or `parseSavePath()` respectively) of the same format as that provided by `shiny::fileInput()`.
`shinyFiles::shinyDirChoose()`	Sets up the required connection to the client in order for the user to navigate the filesystem.
`sidebarLayout(), mainPanel(), sidebarPanel()`	Create a layout (`sidebarLayout()`) with a sidebar (`sidebarPanel()`) and main area (`mainPanel()`). The sidebar is displayed with a distinct background color and typically contains input controls. The main area occupies 2/3 of the horizontal width and typically contains outputs.
`titlePanel()`	Create a panel containing an application title.

Interpretation of the results

Creating packages

To create a package you need two different libraries: `devtools` and `roxygen2`. The installation of both of them can be problematic on macOS systems, but it can be solved with `openssl@1.1` and installing R and Rstudio separately from Anaconda.

After that you need create a new package with `devtools::create(„your_package_name“)` and start creating needed functions in some .R file in /R folder of your project.

Although the task is finished successfully, It is clear how different R is in comparison to Python and Java. The whole process of creating packages is very tricky and requires special code style:

1. It is important to write full paths to the functions you use, because importing with `library()` is not a good choice. It is really easy to forget to put these double colons everywhere.
2. Yo do not need call `source()` or use anything else, if you call your own functions within the package.
3. Comments need special formatting, if you want autodocument your code. For example, here are the comments before the prepare_single_microarray():

```
#' File preparation
#'
#' Prepare a single microarray dataset for plotting.
#' Reading data and selecting columns of interest.
#'
```

```
#' @importFrom magrittr %>%
#' @param file_name Name of a delimited file containing
#'                  at least SPOT, CH1B, CH1I, CH2I and
#'                  CH2B columns.
#' @return Tibble containing SPOT, CH1B, CH1I, CH2I,
#'         CH2B and name columns.
#' @export
```

4. `@export` shows that this function must be documented. It is important to include `@importFrom magrittr %>%`, if you work with pipes.

If you add „`Imports:`“ to your DESCRIPTION file you will not need to worry about dependencies. R will take care of it. For example, during the installation of DAMN R checks availability and versions of the packages mentioned in DESCRIPTION:

```
Imports:
        tibble (>= 2.1.1),
        dplyr (>= 0.8.0.1),
        shiny (>= 1.6.0),
        shinyFiles (>= 0.9.0),
        ggplot2 (>= 3.3.5),
        magrittr (>= 2.0.1)
```

All code can be autodocumented with `devtools::document()` and installed to your R with `devtools::install()`. If you copy your project's files to a git repository, you can install them with `devtools::install_github(„your_username/projectname“)`.

Working with shiny

Shiny provides a simple way for dynamical visualisation. It consists of two parts: UI and server logic. A typical shiny script looks like that:

```
library(shiny)

# Define UI ----
ui <- fluidPage(
      # use some layout
            # define panels in it
                  # use
                  # some
                  # of
                  # many
                  # shiny
                  # widgets
                  # ...
)

# Define server logic ----
server <- function(input, output) {
      # Safe in output some reactive function, e.g. renderPlot(),
      # observe(), etc.
            # Use input to draw, write or etc. something.
}

# Run the app ----
shinyApp(ui = ui, server = server)
```

Pros

- Really easy and intuitive.
- Can create dynamical visualisations even with third-party libraries, most importantly with `ggplot2`.
- Allows code editing on the fly, what can be useful for debugging or UI adjusting.
- Can be opened in every modern browser.

Cons

- Sometimes shiny hides error's traceback.
- Cannot upload directories from the box, you need `shinyFiles` for that.
- `shinyFiles`, in turn, have some problems (see „Known issues").
- Plotting from a script or from the console is more convenient and more familiar for most of the people; shiny is simply redundant.