Open Source Computational Framework in R for Studying Biological Rhythms via Analysis of Behavioral Phase and Circular Statistics

Boyan S. Kostadinov[*] Hannah L. Pettibone[†] Ausra Pranevicius[‡] Evardra Bell[§]
Xiaona Zhou[¶] Orie T. Shafer[‖] Maria P. Fernandez[**]

Contents

[*]Co-first author and lead technical contact, Mathematics Department, NYC College of Technology, City University of New York. Brooklyn, NY 11201, U.S.A., bkostadinov@citytech.cuny.edu
[†]Co-first author, Advanced Science Research Center, The Graduate Center, City University of New York. New York City, NY 10031, U.S.A.
[‡]Department of Neuroscience and Behavior, Barnard College of Columbia University. New York City, NY 10027, U.S.A.
[§]Department of Neuroscience and Behavior, Barnard College of Columbia University. New York City, NY 10027, U.S.A.
[¶]Mathematics Department, NYC College of Technology, City University of New York. Brooklyn, NY 11201, U.S.A.
[‖]Advanced Science Research Center, The Graduate Center, City University of New York. New York City, NY 10031, U.S.A.
[**]Corresponding author, Department of Neuroscience and Behavior, Barnard College of Columbia University. New York City, NY 10027, U.S.A., mfernand@barnard.edu

1 Introduction

This protocol represents the extended methods supplement for the STAR Protocols paper with the same title.

All data analysis and visualizations in this protocol should be **fully reproducible**. We provide the data files and the source **RMarkdown** (**Rmd**) file with the complete **R** code and text narrative, which can be knitted inside **RStudio** into a PDF or HTML file for a full reproduction of all results and figures. The source **Rmd** file together with the data files are posted in the **GitHub** repository Fernandez et al. [2]. For the circular phase analysis and visualizations, we use the actual phases generated for the study in Fernandez et al. [3].

2 Programming Environment

We use **R** as the programming environment and **RStudio** as the IDE. The following freely available software installers are available for Windows, Mac, and Linux.

- MiKTeX installers (for PDF output): http://miktex.org/download
- R installers: http://cran.stat.ucla.edu/
- RStudio installers: https://www.rstudio.com/

Note that **MiKTeX** is required only for generating PDF documents from the source Rmd file. After you load the source Rmd document inside **RStudio**, you must first install and load the following **R** packages:

```
library(tidyverse)
library(signal)
library(pracma)
library(circular)
library(scales)
library(reshape2)
library(svglite)
```

As an introduction to **R, RStudio**, **ggplot2** and the **tidyverse** collection of R packages for doing modern data science, we refer the reader to Wickham and Grolemund [5].

3 Phases Analysis on LD5

3.1 Initial Data Processing

We use the **DAMFileScan111X** (V1.11) to scan **DAMSystem3** and **DAMSystemMB** Monitor data (text) files and aggregate the data into 15 min bins for a certain time range. Note that file names must be of the form `MonitorNNN.txt` or `MonitorNNN.csv`, where `NNN = 1 to 120`, and files must consist of 42 columns, tab or comma delimited, containing data for 32 flies. In particular, we used the **DAMFileScan111X** program to scan a tab delimited full monitor text file and save the fly activity in 15 min bins, for the last, fifth day of the LD cycle, into the text file `LD5APm15mCtM016.txt`.

We are interested in computing the evening phases for the entire sample of 32 flies on the last (5th) day of the LD cycle (LD5). Since for this experiment, *lights-on* is 10:00AM and *lights-off* is 10:00PM, we want to save the data into 15 min bins, within 12 hours around 10:00PM on LD5. For this purpose, we load the full monitor text file into the **DAMFileScan111X** program, then select the first bin to be 10:15AM on LD5 and the last bin to be 10:00AM the next day (DD1). Note that the bin `10:15AM` represents the fly activity between 10:00AM and 10:15AM, and the `10:00AM` bin represents the fly activity between 09:45AM and 10:00AM. The resulting output is saved in the text file `LD5APm15mCtM016.txt`.

Table 1: The head of the LD5 data.

X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12
401340	6 Jul 20	10:15:00	1	14	0	0	0	0	1	10	45
401355	6 Jul 20	10:30:00	1	14	0	0	0	0	1	35	37
401370	6 Jul 20	10:45:00	1	14	0	0	0	0	1	30	25
401385	6 Jul 20	11:00:00	1	14	0	0	0	0	1	24	35

Next, we use the statistical programming language **R** and the Integrated Development Environment (IDE) **RStudio** (both open source and free) to load the text file `LD5APm15mCtM016.txt` and do some initial processing before we start computing the evening phases. We use the **tidyverse** collection of R packages for modern data analysis and visualizations developed by RStudio.

The R code below shows how we load the text file `LD5APm15mCtM016.txt` and save it to the `data` dataframe. Note that since this is a tab delimited file we use the **readr** (a part of the **tidyverse**) function `read_delim()` with second argument `"\t"` and `col_names = FALSE`. If the `col_types` argument is not set to specify character type for column three (`X3`) then the 3rd column will be saved as a time column (of type `hms`), given its structure.

```r
# load the text data file
data <- read_delim("LD5APm15mCtM016.txt",delim="\t",col_names = FALSE)
```

The dataframe `data` has 96 observations for the 96 bins covering the 24 hours that we have selected (centered at 10:00PM on LD5), divided into 15 min bins. It has 42 variables, where the last 32 variables (columns) are representing the 32 flies. In Table 1, we show the first 12 columns and the first 4 rows of the data. Column 11 (`X11`) represents the first fly.

```r
# from data select the first 12 columns then print the first 4 rows
data %>%
  select(1:12) %>%
  head(4) %>%
  knitr::kable(caption = "The head of the LD5 data.")
```

Note that the pipe operator `%>%` used for composition of functions plays a very prominent role in the **tidyverse**, and the `select()` function is from the **dplyr** package in **tidyverse**. The last pipe above chains the selected data into the `kable()` function from the **knitr** package to print Table 1.

The fly activity for the 32 flies is given in the last 32 columns, so we need to remove the first 10 columns, but it is useful to keep for now column 3, which is the column of times that specify the bins. We also rename the resulting first column from its default name `X3` (it was the 3rd column of times in the original data) to `time`.

```r
# keep only columns 3 and then 11:42
data <- data %>% select(c(3,11:42)) %>% rename(time = X3)
```

Now, `data` has 33 columns for `time` and the 32 flies. It may also be useful to rename the fly columns `2:33` as `fly1`, `fly2`, etc. For this purpose, we define a function to rename the fly columns. The function `flyname(x)` takes as an argument `x` the name of a fly column, say `X11` (for the first fly), and it returns the name `fly1`. We use the `str_sub()` function from the **stringr** package in the **tidyverse**, which extracts a substring.

```r
# to rename the fly columns
flyname <- function(x){
  flyindex <- as.numeric(str_sub(x,2,3)) - 10
  return(paste("fly",flyindex,sep=""))
}
```

Using the `flyname` function, we can rename the fly columns `2:33` as `fly1`, `fly2`, etc.

Table 2: The LD5 data around 10:00PM, showing rows 47:50.

time	state	fly1	fly2	fly3	fly4	fly5	fly6	fly7	fly8	fly9	fly10
21:45:00	light	37	33	30	22	30	23	27	55	37	23
22:00:00	light	22	25	25	18	19	23	33	41	30	18
22:15:00	dark	38	33	40	36	34	30	49	77	56	36
22:30:00	dark	34	33	42	40	32	51	41	71	54	32

```r
# rename fly columns
data <- rename_with(data, flyname, .cols = 2:33)
```

For plotting, it is useful to add to the data a new variable `state` for the 96 bins with 48 `light` and 48 `dark` values, depending on which part of the light/dark cycle the bin belongs.

```r
# add a state variable after time
data <- data %>% mutate(state = c(rep("light",48), rep("dark",48)), .after=time)
```

Table 2 shows the first 12 columns and a slice of rows `47:50` of the resulting data, created with the code:

```r
# from data select the first 12 columns and then take a slice of rows 47:50
data %>%
  select(1:12) %>%
  slice(47:50) %>%
  knitr::kable(caption="The LD5 data around 10:00PM, showing rows 47:50.")
```

Note that the state changes from light to dark at bin 49 that corresponds to time `22:15`, which captures the fly activity between 10:00PM and 10:15PM. This is an important observation for computing the evening phase.

3.2 Visualizing the fly activity

We use the **ggplot2** package of the **tidyverse** for all visualization purposes. For more details on using **ggplot2**, we refer the reader to the book Wickham and Grolemund [5].

For illustration purposes, we visualize the activity of `fly1` by using bars centered at the corresponding bins obtained by `seq_along(fly1)`, set as the `x` variable. The bars are colored in light or dark gray based on their `state` value (with `fill=state`). The two shades of gray are provided manually by their HEX values `#b2b2b2` and `#dedede`. The time of the day under darkness is also shaded in gray using the `annotate("rect",...)` function, where we used appropriate coordinates for `xmin` and `xmax` to make the shaded area consistent with the positioning of the bars. We also add the labels `ZT0` and `ZT12` at the appropriate positions. Note that in the title we paste the name `fly1` using the name of the 3rd column given by `names(data)[3]`. The code below creates Figure 1.

```r
data %>%
  ggplot(aes(x=seq_along(fly1),y=fly1)) +
  geom_col(aes(fill=state),col="black", alpha=0.8) +
  scale_fill_manual(values = c("#b2b2b2", "#dedede"), guide = FALSE) +
  annotate("rect",xmin=48.5,xmax=96.5,ymin=0,ymax=Inf,alpha=0.3) +
  annotate("rect",xmin=-Inf,xmax=0.5,ymin=0,ymax=Inf,alpha=0.3) +
  scale_x_continuous(name="",breaks=c(0.5,48.5,96.5),labels=c("ZT0","ZT12","ZT0")) +
  ggtitle(paste("Activity of an AP male",names(data)[3],"on LD5")) +
  ylab("Activity (beam crosses/min)") +
  theme_bw()
```

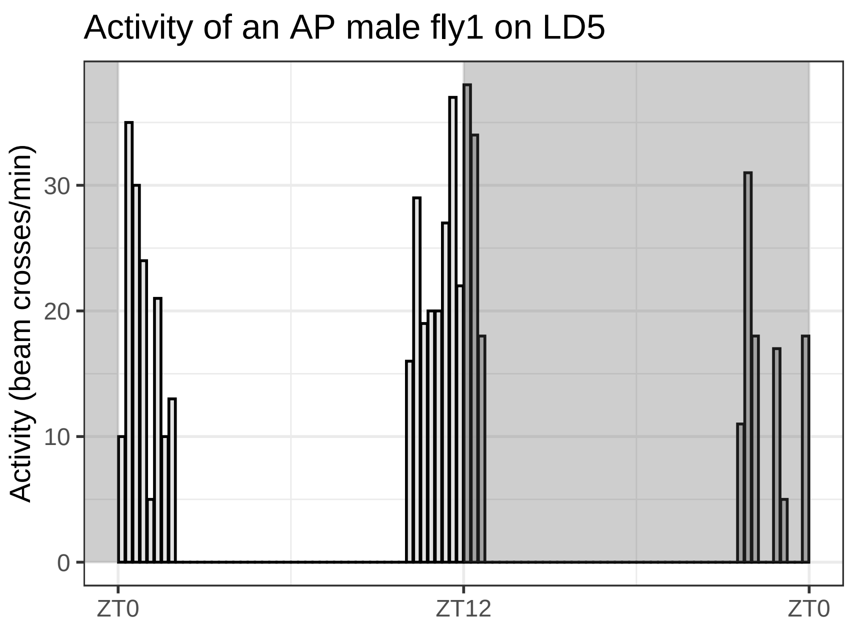

Figure 1: Plot of fly activity centered at ZT12.

3.3 Average activity plot

We want to visualize the average activity of all flies, excluding dead flies, on LD5, which is the last day of LD for our data. This means that we need to compute averages across all live flies for each bin in the data, which amounts to row-wise averages being applied to the `data` data frame. The **dplyr** package in the **tidyverse**, and R in general, are particularly well suited to performing operations over columns, but performing operations over rows is harder. The **dplyr** package allows for grouping the data row-wise using the `rowwise()` function, which offers a more general approach that will work for any summary function. However, there is already a built-in function `rowMeans()` for computing row averages without having to group the data into rows as it operates on the data frame as whole.

Before we compute the row averages, we need to find out which flies are dead. We define a fly to be "dead" if it has zero activity in every single bin. In particular, having a non-zero activity in one bin only would classify the fly as "alive". A simple check then would be to sum the total activity of the fly and if it is zero, we define it as "dead".

Below is the function `dead_flyname()`, which takes the full data frame `data` and returns the names of the dead flies with zero total activity. The key line of code is `summarise(data,across(3:34,sum))`, where we apply the `sum()` function to columns `3:34`, which represent the 32 flies (the first two columns are `time` and `state`). Note that even though `live` is a vector with numbers that represent the total activity for that particular fly, applying the logical negation operator `!live` will coerce the non-zero numbers to the logical value `TRUE` and the zero numbers to the logical value `FALSE`, and then negate. This way `!live` will become a logical vector with `TRUE` values only for the "dead" flies. We can then extract the names of the "dead" flies from the names of all flies with simple indexing `fly_names[!live]`.

```
dead_flyname<- function(data){
  live <- summarise(data,across(3:34,sum)) # sum data across columns 3:34 (flies only)
  fly_names <- names(data[,c(-1,-2)]) # column names of data without first 2 columns
  return(fly_names[!live]) # return the names of dead flies
}
```

We can now call this function to get the names of the "dead" flies:

```r
dead_flyname(data)
```

```
## [1] "fly28" "fly29" "fly30" "fly31" "fly32"
```

In addition to knowing the names of the "dead" flies, it is very useful to have the index vector of all live flies. The function `live_flyindex()` below does exactly that. The key new line of code is `2 + which(as.logical(live))`, which returns the index vector of all live flies in the original data frame, since `which(as.logical(live))` is the index vector of all live flies within all 32 flies, and adding 2 to all indices will shift them to represent all live flies within the original `data` that has two extra columns.

```r
live_flyindex <- function(data){
  live <- summarise(data,across(3:34,sum)) # sum data across columns 3:34 (flies only)
  return(2 + which(as.logical(live))) # indices of live flies in original data (+2)
}
```

Thus, the index vector of all live flies is the following:

```r
alive <- live_flyindex(data) # index vector of all live flies
show(alive)
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## [26] 28 29
```

Finally, we can compute the average activity across all live flies on LD5 and visualize it. For this purpose, we need to update the `data` data frame and add a new variable `avg` to it (after `state`), which is the average activity across live flies only, computed with `rowMeans(across(alive)`, where `alive` is the index vector of all live flies. This can be done in one line of code:

```r
data <- data %>% mutate(avg = rowMeans(across(alive)), .after=state)
```

We can now create the average activity plot, shown in Figure 2, using the new `avg` variable.

```r
data %>%
  ggplot(aes(x=seq_along(avg),y=avg)) +
  geom_col(aes(fill=state),col="black", alpha=0.8) +
  scale_fill_manual(values = c("#b2b2b2", "#dedede"), guide = FALSE) +
  annotate("rect",xmin=48.5,xmax=96.5,ymin=0,ymax=Inf,alpha=0.3) +
  annotate("rect",xmin=-Inf,xmax=0.5,ymin=0,ymax=Inf,alpha=0.3) +
  scale_x_continuous(name="",breaks=c(0.5,48.5,96.5),labels=c("ZT0","ZT12","ZT0")) +
  ggtitle("Average activity of live flies on LD5") +
  ylab("Average activity (beam crosses/min)") +
  theme_bw()
```

We can save the average activity plot as a `svg` file with `ggsave()`:

```r
ggsave(file="average_activity.svg", device="svg", width=10, height=6, dpi=300)
```

3.4 Smoothing the data

Next, our goal is to apply a filter on the fly counts that removes the high frequencies in the spectrum, i.e. a low-pass filter that smooths out the data. For this purpose, we use a **Butterworth** filter, which is a recursive filter of the form:

$$v_j = \sum_{p=0}^{P} a_p u_{j-p} + \sum_{q=1}^{Q} b_q v_{j-q} \tag{1}$$

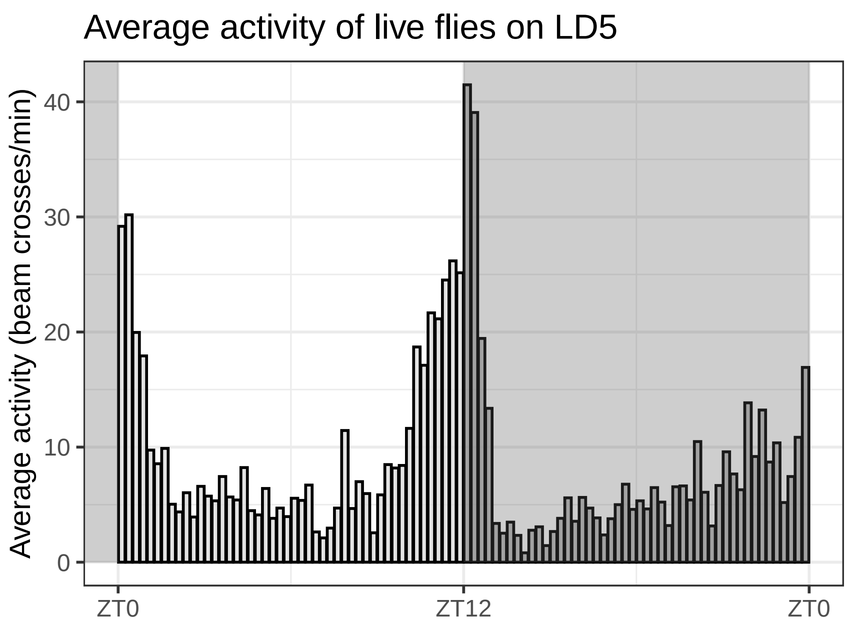

Figure 2: The average activity of all live flies on LD5.

where $\{u_j\}_{j=1}^N$ is the original signal, and $\{v_j\}_{j=1}^N$ is the filtered signal. Note that this filter depends not only on the current and the previous P original data points, but also on the previously filtered Q points, relative to the current point. Thus, the current filtered value depends not only on a **weighted average** of the unfiltered values (first sum), but also **recursively** on the previously computed Q filtered values (second sum). The magic of filter design is to compute the values of the $P+1$ parameters $\{a_p\}_{p=0}^P$ (it is really P since they sum to 1), and the Q parameters $\{b_q\}_{q=1}^Q$ so that the frequency requirements are met. Using more weights (bigger P and Q), one can design a filter with a sharper frequency cutoff at the expense of more boundary issues. For our purpose, a sharp frequency cutoff is not really needed and a lower-order filter should work just fine. We use a 2nd order low-pass Butterworth filter with $P = Q = 2$.

This filter is **causal** since it uses only past data and this leads to a **phase delay** in the filtered data. The phase delay can be removed by additional filtering backward in time. Note that Butterworth filters with order bigger than 1 can overshoot and yield negative values even if all original data points are non-negative.

In order to compute the Butterworth filter weights, we use the function `butter()` from the **signal** package. We also use the function `filtfilt()`, which applies a linear digital filter twice, once forward and once backward in time. The combined filter has zero phase delay and a filter order twice that of the original. *Having a zero phase delay is essential for the phase computations in the context of fly activity.*

The description of this function in the **signal** R package suggests that it is still work in progress. If one needs a more mature version of the functions in the **signal** R package, we suggest using the Python **signal** library in **scipy**.

We give the following example to illustrate the phase delay that occurs if only a single forward filter is applied to the signal. The signal here is generated by a 2.3 Hz shifted sinusoid $y = \sin(2\pi\omega t) + 2$ and then adding some white noise $y = \sin(2\pi\omega t) + 0.4N(0,1) + 2$, where $\omega = 2.3$, $N(0,1)$ is the standard Normal random variable, and we can generate a random sample from it using `rnorm()`. Figure 3 shows the results of applying single and double filter to noisy data generated by a pure sinusoid. The double filter traces very closely the original pure sinusoidal signal used to generate the noisy data, except for the boundary issues at the end points. The phase delay of the single filter is also quite clear.

```
set.seed(777)
# Example of Butterworth filter applied to noisy data
```

```
bf <- butter(n = 2, W = 0.1, type = "low", plane = "z") # order 2, 10 Hz low-pass filter
t <- seq(0, 1, len = 96)  # 1 second sample of time values
x <- sin(2*pi*t*2.3) + 2  # 2.3 Hz pure sinusoid
y <- sin(2*pi*t*2.3) + 2 + 0.4*rnorm(length(t))  # 2.3 Hz sinusoid + white noise
z <- filtfilt(bf, x) # forward and backward filter
w <- filter(bf, x) # forward filter only
```

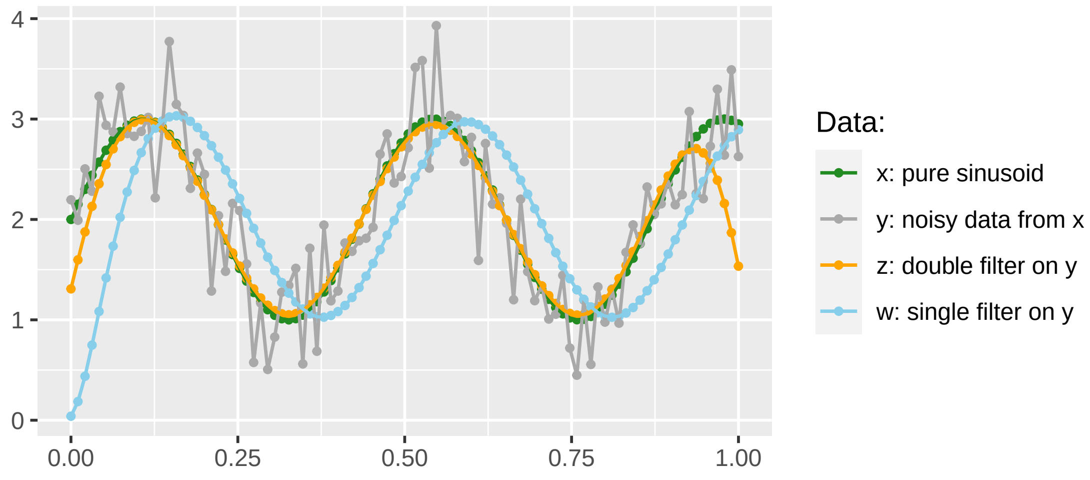

Figure 3: Applying single and double Butterworth filter to noisy data.

3.5 Obtaining the evening phase of a single fly

We apply a Butterworth filter to smooth out the fly data and compute the evening phase for a single fly using a window centered at 10:00 PM on LD5. For the 15 min bin data, we use the **49th bin**, which is the first bin in `dark` state, to represent the evening phase bin relative to which we compute the phase shift. Of course, if the data come in different format, say 30 min bins then one must use the appropriate bin for the evening phase computations.

As we mentioned earlier, to generate the Butterworth filter parameters, we use the `butter()` function from the **signal** R package. The function has the following signature:

```
butter(n, W, type = 'low', plane = 'z')
```

The arguments are:

1. `n` is the filter order. We first explore the data with `n=1,2` for a first and second order filter.
2. `W` is the critical frequency of the filter and it must be a scalar for a low-pass filter. For digital filters, `W` must be between 0 and 1, where 1 is the Nyquist frequency. We begin our explorations with either `W=0.1` (10Hz) or `W=0.2` (5Hz).

3. `type` is set to `'low'` for a low-pass filter.
4. `plane` is set to `'z'` for a digital filter.

The function `butter()` returns an `ARMA` object with list elements:

1. `a`: autoregressive (AR) polynomial coefficients for the recursive part.
2. `b`: moving average (MA) polynomial coefficients for the weighted sum part.

Next, we show how to compute the evening phase for the first fly in the data, column `fly1`. We add to the original data the smoothed data for the single fly, using the `mutate()` function and applying the double filter `filtfilt()` to the fly data using the Butterworth filter, described above. Then, we use the `findpeaks()`

function from the **pracma** R package to find a single peak (the maximum) in the time series of fly activity. We use the following function signature:

```
peak <- findpeaks(x, npeaks = 1, sortstr = TRUE) # finding the single highest peak
```

where `x` is the time series of fly activity, `npeaks=1` specifies returning the highest peak after all local peaks are sorted in decreasing order of their maximum value using `sortstr = TRUE`. In this case, the function returns a matrix with one row that represents the maximum peak found. The first column gives the height, the second column gives the position (index) where the maximum is reached, the third and forth columns give the indices of where the peak begins and ends, which can be useful to detect phase peak onsets and offsets. Knowing the index of the peak, we can find how far it is from the 49th bin, and we can scale the difference into hours to find the evening phase shift as positive or negative hours relative to the lights-off time point of 10:00 PM. More precisely, the phase shift in hours is given by:

```
phase <- (peak[1,2] - C)/4 # phase shift in hours for 15 min bin data
```

where `peak[1,2]` is the index of the peak (1st row, 2nd column), and `C=49` is the index of the bin that represents the lights-off time point (10:00 PM). Finally, if the fly is dead, we want to return `NA`. Here is the complete code that computes the evening phase shift for the first fly `fly1` on LD5:

```
C <- 49 # index for lights-off (first "dark" bin for the evening phase)
binsize <- 15 # for data with 15 min bins
bins_hour <- 60/binsize # number of bins in one hour
# create the Butterworth filter
bf <- butter(n=2, W=0.1, type='low', plane='z')
# add to data the smoothed data for flyN -> change N to change the fly
data <- data %>% mutate(flysmooth = filtfilt(bf,fly1), .after = fly1)
# find peaks in smoothed data, sort the peaks and extract the single max peak
peak <- findpeaks(data$flysmooth,sortstr=TRUE,npeaks = 1)
# peak[1,2] is the index of the single peak
phase <- (peak[1,2] - C)/bins_hour  # peak index relative to lights-off (ZT12) in hours
# dead flies have zero data, so return NA in this case
phase <- ifelse(is_empty(phase),NA,phase)
print(phase)
```

```
## [1] -0.5
```

Thus, the evening phase of the first fly in the data is -0.5 hours.

3.6 Plotting the raw, filtered data and the evening phase

For additional insight, let us visualize the raw data for the first fly (`fly1`), along with the filtered data and the evening phase, all on one plot. In Figure 4, we plot the raw data for the first fly (`fly1`) using bars shaded according to the `light` or `dark` state of the bin. The filtered data `flysmooth` for this particular fly is plotted using a line plot in blue color. We label the evening phase that corresponds to the single, **highest** peak in the filtered data.

```
# INPUT ################
C <- 49 # index for lights-off (first "dark" bin for the evening phase)
binsize <- 15 # for data with 15 min bins
bins_hour <- 60/binsize # number of bins in one hour
# create a Butterworth filter
bf <- butter(n=2, W=0.1, type='low', plane='z')
flyname <- as.name("fly1") # to change the fly, change N in flyN (N=1,2,...,32)
# END OF INPUT #########
# add to data the filtered data for flyN
```

```
data <- data %>% mutate(flysmooth=filtfilt(bf,eval(flyname)), .after=all_of(flyname))
# find peaks in smoothed data, sort the peaks and extract the single max peak
peak <- findpeaks(data$flysmooth, sortstr=TRUE, npeaks = 1)
# peak[1,2] is the index of the single peak
phase <- (peak[1,2] - C)/bins_hour # peak index relative to lights-off (ZT12) in hours
# dead flies have zero data, so return NA in this case
phase <- ifelse(is_empty(phase), NA, phase)
# ggplot breaks
breaks <- c(0.5, C - 0.5, 2*(C-1) + 0.5)
data %>%
  ggplot(aes(x=seq_along(flysmooth))) +
  geom_col(aes(y=eval(flyname), fill=state),col="black") +
  scale_fill_manual(values = c("#b2b2b2", "#dedede"), guide = FALSE) +
  annotate("rect", xmin = breaks[2], xmax = breaks[3], ymin = 0, ymax = Inf,alpha=0.3) +
  annotate("rect", xmin = -Inf, xmax = breaks[1], ymin = 0, ymax = Inf, alpha = 0.3) +
  geom_label(aes(x = peak[1,2], y = peak[1,1],label=paste("phase = ",phase,"h",sep="")),
           hjust = "center", vjust = 0, size = 2.5, col = "blue") +
  geom_line(aes(y = flysmooth), col = "mediumblue", size = 0.9, alpha = 0.7) +
  scale_x_continuous(breaks = breaks, labels = c("ZT0", "ZT12", "ZT0")) +
  ggtitle(paste("Evening Phase for",flyname,"on LD5")) +
  labs(x = "", y = "Activity (beam crosses/min)") +
  theme_bw()
```

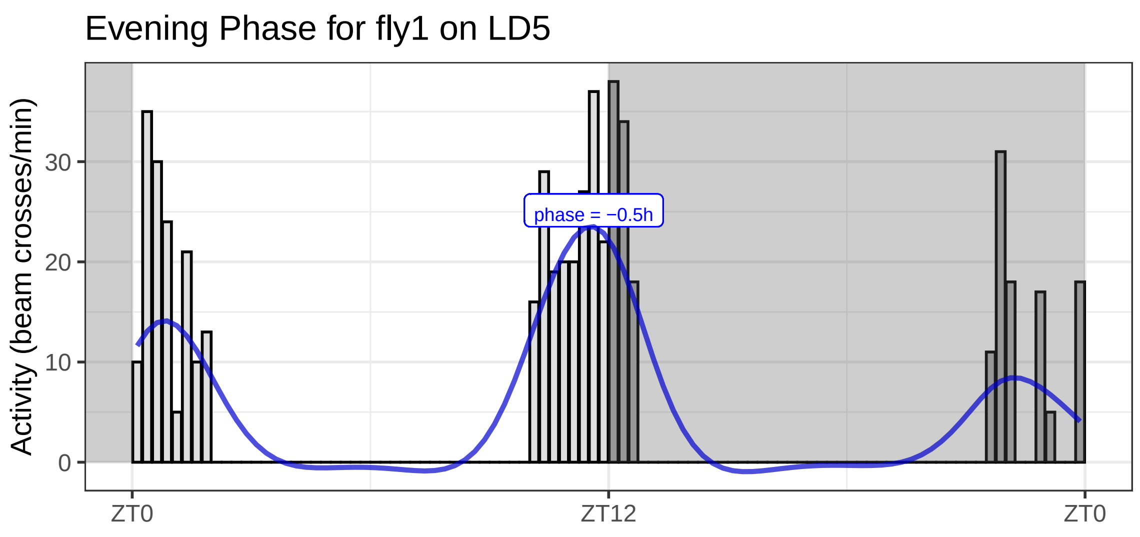

Figure 4: Raw and filtered data for a single fly and its evening phase on LD5.

Unfortunately, using the highest peak in the filtered data for a given fly does not always work. The reason is that there are could be cases where the highest peak is not the appropriate peak to use for computing the evening phase. For example, this is the case with `fly18` in our data. In Figure 5, we show this case where the highest peak is clearly not the right peak to use for the evening phase, but we should rather use the second highest peak. This problem can be addressed in a couple of different ways. One way is to find the peak **closest** to the `ZT12` (lights-off) time point, even if it is not the highest among all peaks in the filtered data. Another way is to use a window centered at `ZT12` that could eliminate peaks that are too far, but perhaps some could still be left, unless the window is small enough, but then there could be an issue with the much smaller number of data points left to produce an accurate filtering, thus the evening phase could get distorted.

We implement the first approach based on finding the peak closest to `ZT12`, in terms of the time difference.

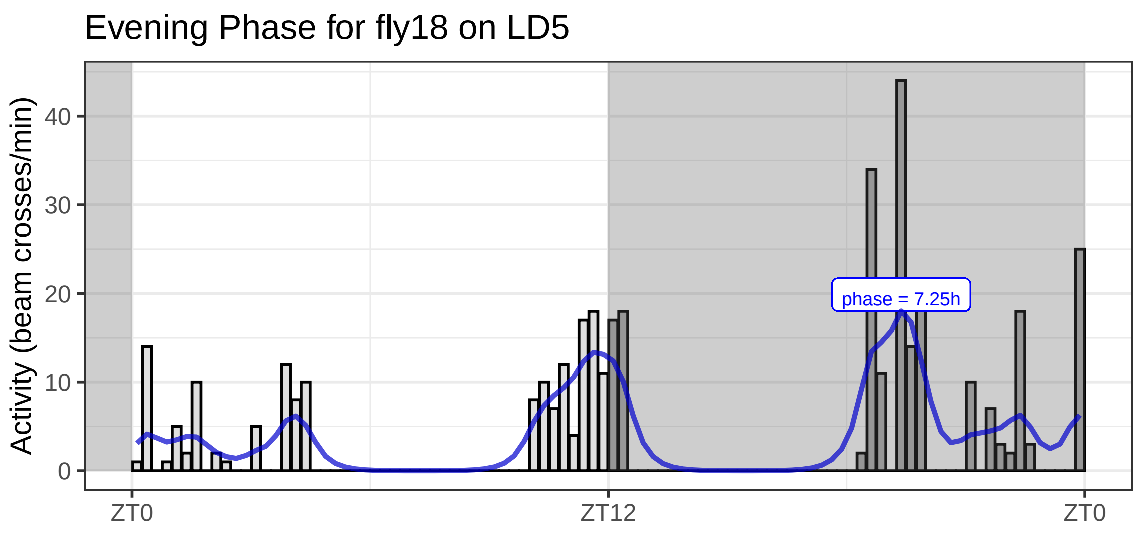

Figure 5: Raw and filtered data for a single fly and its evening phase on LD5.

For this purpose, we find all peaks in the filtered data, using `findpeaks()`, and then we find the peak whose time index **minimizes the distance** from the time index of `ZT12`. Note that distance is given as the absolute value of the difference.

More precisely, the function `findpeaks()` returns the matrix `peaks` (see the code below), where each row represents one peak found: the first column holds the height, the second column holds the position (index) where the maximum is reached, the third and fourth columns give the indices of where the peak begins and ends, which can be useful to detect phase peak onsets and offsets. We still sort the heights in decreasing order, but we use the second column `peaks[,2]` to find the index that corresponds to the minimum distance between the peaks and the index for ZT12 (given by `C`), implemented by the code `which.min(abs(peaks[,2]-C))`.

The code chunk below gives the full implementation of this approach applied again to `fly18`, which gives a new phase.

```r
flyname <- "fly18"
filter_order <- 1
filter_freq <- 0.2
C <- 49 # index for ZT12 lights-off (first "dark" bin for the evening phase)
binsize <- 15 # for data with 15 min bins
bins_hour <- 60/binsize # number of bins in one hour
## End of Input #########
# Create a Butterworth filter
bf <- butter(n=filter_order, W=filter_freq, type='low', plane='z')
flyname <- as.name(flyname) # flyname is converted from string to name
# add to pre-processed data the filtered data for the fly
data <- data %>% mutate(flysmooth=filtfilt(bf,eval(flyname)), .after=all_of(flyname))
# UPDATED CODE
# find all peaks in smoothed data, sort the peaks
peaks <- findpeaks(data$flysmooth, sortstr=TRUE)
row.index <- which.min(abs(peaks[,2]-C)) # index of min distance from C (ZT12)
peak.index <- peaks[row.index,2] # time index for the corresponding peak to be used
phase <- (peak.index - C)/bins_hour # peak index relative to lights-off (ZT12) in hours
# dead flies have zero data, so return NA in this case
phase <- ifelse(is_empty(phase), NA, phase)
print(phase)
```

```
## [1] -0.5
```

This way we obtain the evening phase value of -0.5 hours for `fly18`.

We can bundle the code above into the function `phase()` that computes the evening phase of a single fly.

```
# compute the phase for a given fly and pre-processed data
phase <- function(data, flyname, filter_order, filter_freq){
  C <- 49 # index for ZT12 lights-off (first "dark" bin for the evening phase)
  binsize <- 15 # for data with 15 min bins
  bins_hour <- 60/binsize # number of bins in one hour
  # create a Butterworth filter
  bf <- butter(n=filter_order, W=filter_freq, type='low', plane='z')
  flyname <- as.name(flyname) # flyname is converted from string to name
  # add to pre-processed data the filtered data for the fly
  data <- data %>% mutate(flysmooth=filtfilt(bf,eval(flyname)), .after=all_of(flyname))
  # find all peaks in smoothed data, sort the peaks
  peaks <- findpeaks(data$flysmooth, sortstr=TRUE)
  row.index <- which.min(abs(peaks[,2]-C)) # index of min distance from C (ZT12)
  peak.index <- peaks[row.index,2] # time index for the corresponding peak to be used
  phase <- (peak.index - C)/bins_hour # peak index relative to lights-off (ZT12) in hours
  # dead flies have zero data, so return NA in this case
  phase <- ifelse(is_empty(phase), NA, phase)
  return(phase)
}
```

As a check, we apply the `phase()` function to `fly18` to make sure we get the same phase.

```
phase(data, "fly18", 1, 0.2)
```

```
## [1] -0.5
```

We can also bundle into the function `phasePlot()` the updated version of the code that we used for visualization, based on the updated algorithm for finding the peak closest to the lights-off time point. We still need to pre-process the data before we use it in both the phase and the visualization functions.

```
phasePlot <- function(data, flyname, filter_order, filter_freq ){
  # INPUT ###########
  C <- 49 # index for lights-off (first "dark" bin for the evening phase)
  binsize <- 15 # for data with 15 min bins
  bins_hour <- 60/binsize # number of bins in one hour
  # END OF INPUT ####
  # ggplot breaks
  breaks <- c(0.5, C - 0.5, 2*(C-1) + 0.5)
  flyname <- as.name(flyname) # flyname is converted from string to name
  # create a Butterworth filter
  bf <- butter(n=filter_order, W=filter_freq, type='low', plane='z')
  # add to pre-processed data the filtered data for the fly
  data <- data %>% mutate(flysmooth=filtfilt(bf,eval(flyname)), .after=all_of(flyname))
  # find all peaks in smoothed data
  peaks <- findpeaks(data$flysmooth, sortstr=TRUE) # all peaks, sorted
  row.index <- which.min(abs(peaks[,2]-C)) # index of min distance from C (ZT12)
  peak.index <- peaks[row.index,2] # time index for the corresponding peak to be used
  phase <- (peak.index - C)/bins_hour # peak index relative to lights-off (ZT12) in hours
  # dead flies have zero data, so return NA in this case
  phase <- ifelse(is_empty(phase), NA, phase)
  if (is.na(phase)) {
    return(NA)
  }
  # ggplot begins
  p <- data %>%
    ggplot(aes(x=seq_along(flysmooth))) +
    geom_col(aes(y=eval(flyname), fill=state),col="black") +
    scale_fill_manual(values = c("#b2b2b2", "#dedede"), guide = FALSE) +
    annotate("rect", xmin = breaks[2], xmax = breaks[3], ymin = 0, ymax=Inf,alpha=0.3) +
    annotate("rect", xmin = -Inf, xmax = breaks[1], ymin = 0, ymax = Inf, alpha = 0.3) +
    geom_label(aes(x = peak.index, y = peaks[row.index,1],
                   label = paste("phase=", phase, "h", sep = "")),
               hjust = "center", vjust = 0, size = 2.5, col = "blue") +
    geom_line(aes(y = flysmooth), col = "mediumblue", size = 0.9, alpha = 0.7) +
    scale_x_continuous(breaks = breaks, labels = c("ZT0", "ZT12", "ZT0")) +
    ggtitle(paste("Evening Phase for",flyname)) +
    labs(x = "", y = "Activity (beam crosses/min)") +
    theme_bw()
  return(p)
}
```

In particular, using `phasePlot()`, we obtain the plot for `fly18` in Figure 6 based on the updated algorithm for finding the peak closest to `ZT12`.

```
phasePlot(data, "fly18", 1, 0.2)
```

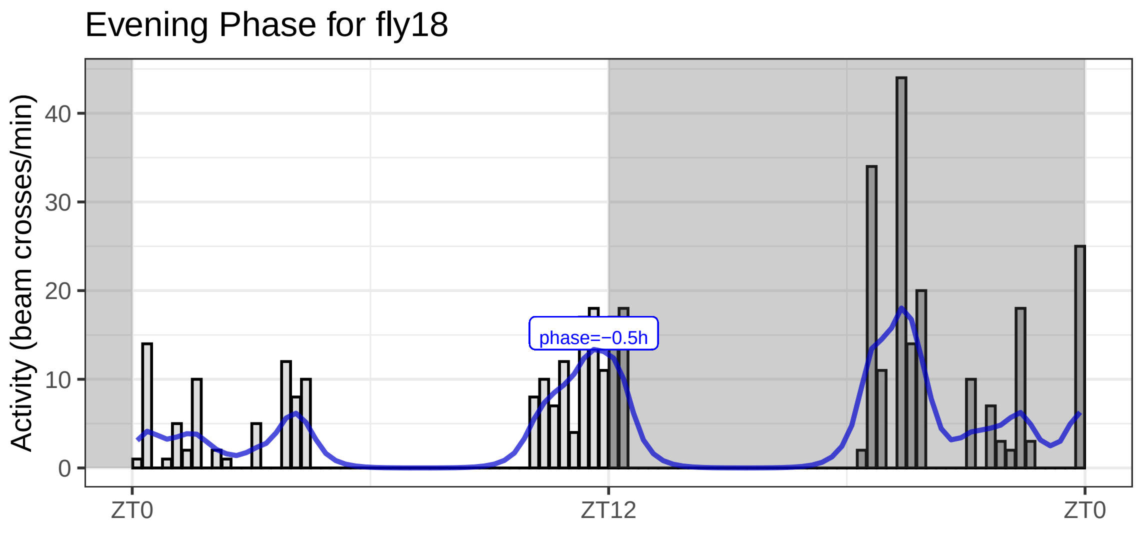

Figure 6: Raw and filtered data for a single fly and its evening phase on LD5.

3.7 The full phase sample on LD5

Next, we want to compute the full phase sample for all 32 flies on LD5. The statistics of the phase sample on LD5 could be used for calibrating the Butterworth filter parameters: order and critical frequency. So far, we have been smoothing the data using a first order low-pass Butterworth filter with critical frequency of 0.1 (10Hz). For example, for `fly2`, the evening phase on LD5 is computed using the `phase()` function and visualized in Figure 7.

```
phase(data, 'fly2', 1, 0.2)
```

```
## [1] -0.25
```

```
phasePlot(data, "fly2", 1, 0.2)
```

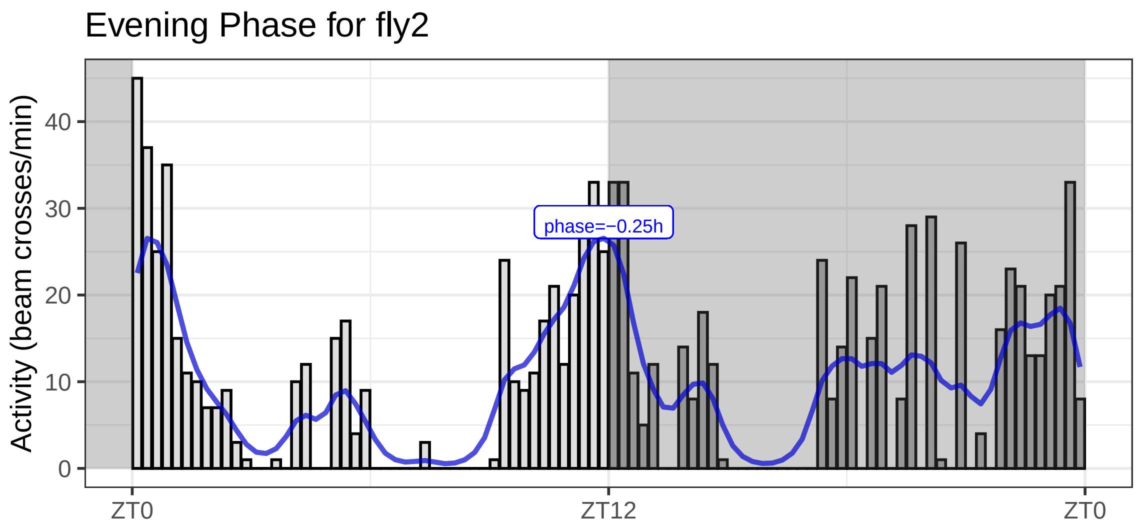

Figure 7: Raw and filtered activity for fly2 and its evening phase on LD5.

Since the `phase()` function takes the name of the fly, in order to be able to compute all 32 phases, we need a vector of all fly names, which we can get from the column names of `data`, but we need to remove the first two names `time` and `state`. This is done with the code `tail(names(data),-2)`.

We use `map_dbl(.x, .f, ...)` from the **purrr** package in **tidyverse**, which transforms the input vector `.x` by applying the function `.f` to each element of the vector and returning a double vector of the same length as the input. In our case, the function `.f` will be the `phase` function, the input vector `.x` will be the vector with the fly names and `...` is used to bind the other arguments of the function `phase()`, which are `data`, `filter_order` and `filter_freq`. The result will be the vector with all 32 phases. The code below computes the full phase sample in the vector `phase_sample`. Note that there are 5 dead flies, which return `NA` for the phase.

```
# Butterworth filter parameters
filter_order <- 1
filter_freq <- 0.2
flynames <- tail(names(data),-2) # all 32 fly names "fly1", "fly2", ..., "fly32"
phase_sample <- map_dbl(flynames, phase, data=data,
                        filter_order = filter_order, filter_freq = filter_freq)
```

The summary statistics of the `phase_sample` are as follows:

```
summary(phase_sample)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's 
## -1.00000 -0.25000  0.00000 -0.05172  0.00000  0.50000        5
```

We can visualize the distribution of the phase sample by plotting its histogram, shown in Figure 8.

```
tibble(phase_sample) %>%
  ggplot(aes(x=phase_sample)) +
  geom_histogram(col="black",fill="gray") +
```

```r
  labs(title=paste("Histogram of phase sample (N=32) with n=",
                   filter_order,"and W=",filter_freq), x="phase")
```

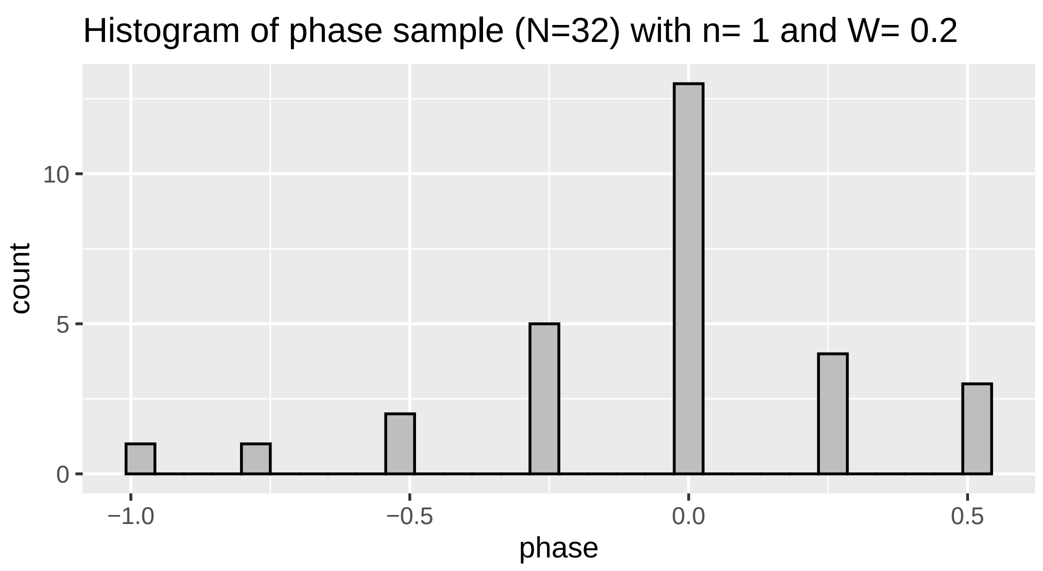

Figure 8: The frequency histogram of the phase sample on LD5.

For these specific filter parameters, the phase sample on LD5 appears to have a symmetric distribution centered at zero. However, the sample phase distribution depends on the filter parameters and we need to make some effort to choose them properly.

The filter should be calibrated to some specific conditions using LD fly activity data. Once the filter is calibrated on LD data, it should capture in a more consistent way the phases for the DD data, relative to LD.

Using some arbitrary parameters for the filter may introduce some bias in the LD phases, but if this bias propagates in a consistent way so that the relative difference between the LD and DD phases captures the real effect, then there is no need to worry too much about the filter parameters. However, it may be the case that the bias does not propagate properly if the filter is not calibrated properly, and this could lead to potential issues.

One possible approach to calibrating the filter could be based on plotting the average fly activity on the last day of LD (or the last several days of LD), for the entire monitor of 32 flies. Then, we could calibrate the filter so that we get an average for the phase sample that matches the observed phase from the average activity plot, which we can determine manually (subjectively) from the plot shown in Figure 2.

Once we have determined manually the observed phase from the average activity plot, we can find the optimal filter parameters `n` and `W` that **minimize the distance between the observed average phase and the computed mean of the phase sample** on LD5 (the last day of LD), by grid optimization, using a grid of values for the two parameters. More precisely, we consider only two values for the filter order `n=1` and `n=2` (since we want a low order filter):

```r
n<-1:2 # filter order
W<-seq(0.05, 0.15, by=0.001) # filter frequency vector
comb<-expand_grid(n,W) # all combinations of n and W
```

and a vector of 101 values for `W` between 0.05 and 0.15 with step 0.001. We then form all possible combinations between the values of `n` and `W`, which gives a 202 by 2 matrix, with the help of the `expand_grid(n,W)` function from the **tidyr** package in the **tidyverse**.

Then, we write a function `matchPhase(data,n,W,observedPhase)`, which returns the distance between the observed phase from the average activity plot (held in `observedPhase`), and the computed mean of the phase

Table 3: The optimal filter parameters.

n	W
2	0.077

sample for the given pair of `n` and `W` values. This part is based on our previous computations using the `phase()` function, which computes the phase of a single fly, given by name, for the given pair of parameter values.

```r
# we use the phase() function inside
matchPhase<-function(data, n, W, observedPhase){
  flynames <- tail(names(data),-2) # all 32 fly names "fly1", "fly2", ..., "fly32"
  phase_sample <- map_dbl(flynames, phase, data=data, filter_order = n, filter_freq = W)
  avg <- mean(phase_sample, na.rm = TRUE) # mean of phase sample, remove NAs (dead flies)
  return(abs(observedPhase-avg)) # distance between observedPhase and computed sample mean
}
```

Next, we map the `matchPhase()` function to all pairs of `n` and `W` values from the combinations matrix `comb` and obtain each element in the vector `distances` for each pair `(n,W)`. We use the handy function `which.min()` to find the index that corresponds to the minimum distance and use it to find the values of the corresponding filter parameters `n` and `W`. Here, we have to provide the observed phase from the average activity plot, shown in Figure 2, which we determined manually to be -0.5h.

```r
# Grid Optimization:
distances <- map2_dbl(comb$n, comb$W, matchPhase, data=data, observedPhase=-0.5)
opt_index <-which.min(distances) # the index for the minimum value
knitr::kable(comb[opt_index, ], align = "c", caption="The optimal filter parameters.")
```

Thus, minimizing the desired distances over all combinations of `(n,W)` pairs gives the optimal filter parameters `n`=2 and `W`=0.077. The function `meanPhase()` computes the mean of the phase sample.

```r
meanPhase<-function(data,n,W){
  flynames <- tail(names(data),-2) # all 32 fly names "fly1", "fly2", ..., "fly32"
  phase_sample <- map_dbl(flynames, phase, data=data, filter_order = n, filter_freq = W)
  avg <- mean(phase_sample, na.rm = TRUE) # mean of phase sample, remove NAs (dead flies)
  return(avg)
}
```

Using the optimal filter parameters, we can check that the computed mean of the phase sample matches the observed average phase.

```r
meanPhase(data, n=2, W=0.077)
```

```
## [1] -0.5
```

Let us recompute the phase sample using the optimal filter parameters.

```r
# optimal filter parameters from matching the observed average phase
filter_order <- 2
filter_freq <- 0.077
flynames <- tail(names(data),-2) # all 32 fly names "fly1", "fly2", ..., "fly32"
phase_sample <- map_dbl(flynames, phase, data=data,
                        filter_order = filter_order, filter_freq = filter_freq)
```

Now, the summary statistics of the new `phase_sample` are as follows:

```r
summary(phase_sample)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   -1.25   -0.75   -0.50   -0.50   -0.50    0.25       5
```

We can save the full phase sample into a csv file using the `write_csv()` function from the **readr** package inside the **tidyverse**.

```
write_csv(phase_sample,file="phases.csv")
```

The distribution of the phase sample obtained with the optimal filter parameters is shown in Figure 9.

```
tibble(phase_sample) %>%
  ggplot(aes(x=phase_sample)) +
  geom_histogram(col="black",fill="gray") +
  labs(title=paste("Histogram of phase sample (N=32) with n=",
                   filter_order,"and W=",filter_freq), x="phase")
```

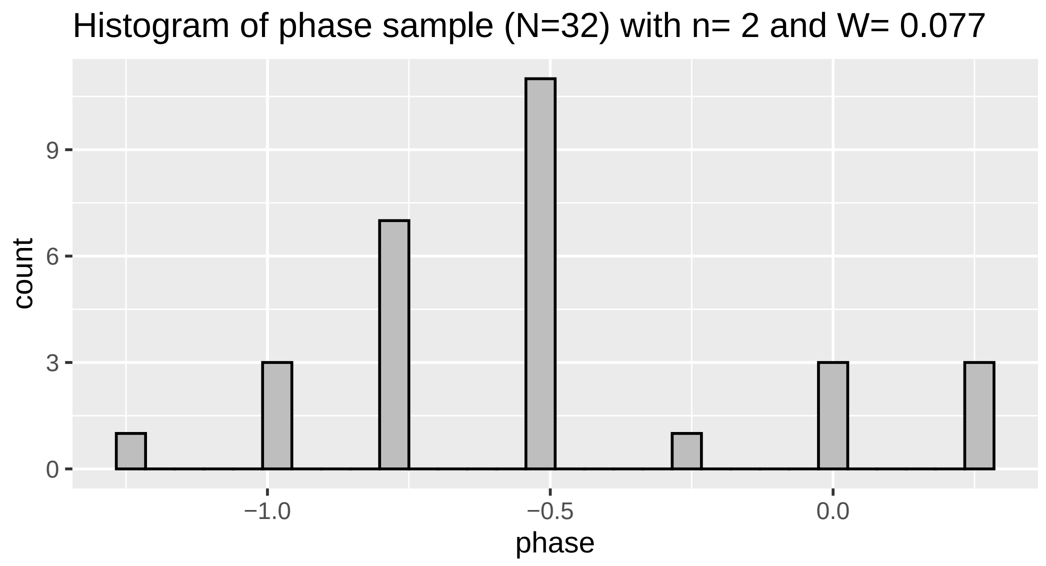

Figure 9: The frequency histogram of the phase sample on LD5 with optimal filter parameters.

For the sake of completeness, let us recompute the evening phase for `fly18` on LD5 using the optimal filter parameters and create the phase plot in Figure 10.

```
phase(data, 'fly18', 2, 0.077)
```

```
## [1] -0.75
```

```
phasePlot(data, "fly18", 2, 0.077)
```

4 Phase Analysis on DD1

4.1 Loading and processing the DD1 data

First, we load the DD1 data file `DD1APm15mCtM016.txt` obtained from the **DAMFileScan111X** software. Note that the DD1 data are centered at 10:00PM on DD1.

```
# load the text data file
dd1 <- read_delim("DD1APm15mCtM016.txt",delim="\t",col_names = FALSE)
```

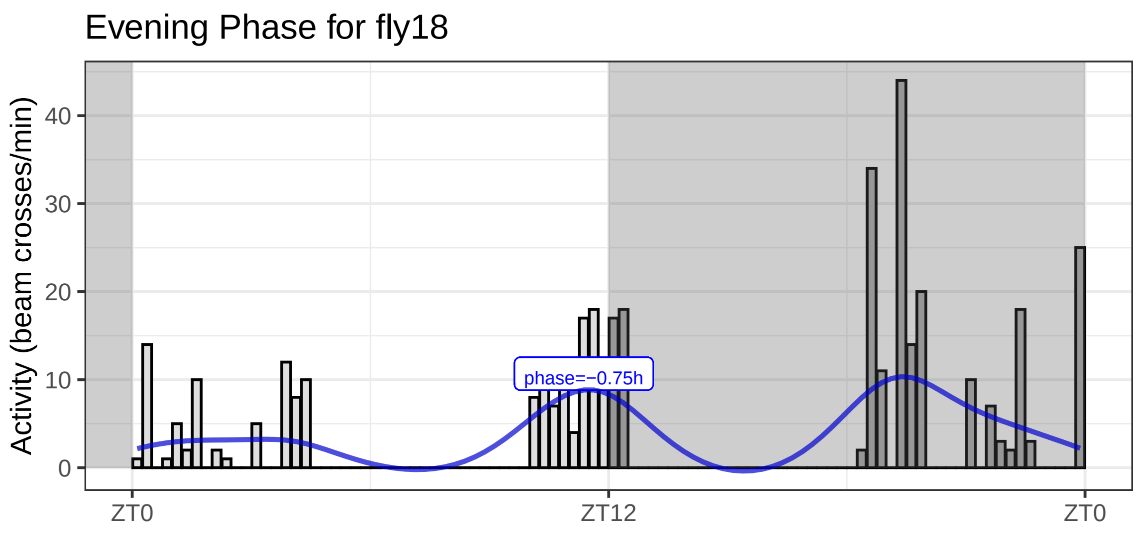

Figure 10: Raw and filtered activity for fly18 and its evening phase on LD5 with optimal filter parameters.

Table 4: The head of the DD1 data.

X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12
402780	7 Jul 20	10:15:00	1	14	0	0	0	0	0	27	15
402795	7 Jul 20	10:30:00	1	14	0	0	0	0	0	25	7
402810	7 Jul 20	10:45:00	1	14	0	0	0	0	0	10	0
402825	7 Jul 20	11:00:00	1	14	0	0	0	0	0	23	25

The dataframe `dd1` has 96 observations for the 96 bins covering the 24 hours that we have selected, divided into 15 min bins. It has 42 variables, where the last 32 variables (columns) are representing the 32 flies. In Table 4, we show the first 12 columns and the first 4 rows of the data. Column 11 (`X11`) is the first fly.

```r
# from dd1 select the first 12 columns then print the first 4 rows
dd1 %>%
  select(1:12) %>%
  head(4) %>%
  knitr::kable(caption = "The head of the DD1 data.")
```

Next, we perform the same data processing that we did for the LD5 data. Eventually, we will embed this data processing into the main functions for computing phases and creating the phase plots.

The fly activity for the 32 flies is given in the last 32 columns, so we need to remove the first 10 columns, but it is useful to keep for now column 3, which is the column of times that specify the bins. We also rename the resulting first column from its default name `X3` (it was the 3rd column of times in the original data) to `time`.

```r
# keep only columns 3 and then 11:42
dd1 <- dd1 %>% select(c(3,11:42)) %>% rename(time = X3)
```

Now, `dd1` has 33 columns for the `time` and the 32 flies. It is also useful to rename the fly columns `2:33` as `fly1`, `fly2`, etc. For this purpose, we define a function to rename the fly columns. The function `flyname(x)` takes as an argument `x` the name of a fly column, say `X11` (for the first fly), and it returns the name `fly1`. We use the `str_sub()` function from the **stringr** package in the **tidyverse**, which extracts a substring.

```r
# to rename the fly columns
flyname <- function(x){
  flyindex <- as.numeric(str_sub(x,2,3)) - 10
```

Table 5: The DD1 data around 10:00PM, showing rows 47:50.

time	state	fly1	fly2	fly3	fly4	fly5	fly6	fly7	fly8	fly9	fly10
21:45:00	light	20	14	22	18	12	34	13	31	20	17
22:00:00	light	30	11	38	10	16	27	25	35	27	21
22:15:00	dark	28	12	20	20	26	37	27	38	20	10
22:30:00	dark	14	21	0	35	25	21	20	45	25	18

```
  return(paste("fly",flyindex,sep=""))
}
```

Using the `flyname` function, we can rename the fly columns `2:33` as `fly1`, `fly2`, etc.

```
# rename fly columns
dd1 <- rename_with(dd1, flyname, .cols = 2:33)
```

For plotting, it is useful to add to the data a new variable `state` for the 96 bins with 48 `light` and 48 `dark` values, depending on which part of the light/dark cycle the bin belongs.

```
# add a state variable after time
dd1 <- dd1 %>% mutate(state = c(rep("light",48), rep("dark",48)), .after=time)
```

Table 5 shows the first 12 columns and a slice of rows `47:50` around 10:00 PM of the resulting data, created with the code below:

```
# from dd1 select the first 12 columns and then take a slice of rows 47:50
dd1 %>%
  select(1:12) %>%
  slice(47:50) %>%
  knitr::kable(caption="The DD1 data around 10:00PM, showing rows 47:50.")
```

Note that the state changes from light to dark at bin 49 that corresponds to time `22:15`, which captures the fly activity between 10:00PM and 10:15PM. This is an important observation for computing the evening phase.

4.2 A single evenning phase on DD1

Let us start by computing the evening phase again for `fly2` on DD1, using the optimal filter parameters obtained by calibrating the filter to the LD5 data. We also create the phase plot in Figure 11.

```
phase(dd1, 'fly2', 2, 0.077) # evening phase of fly2 on DD1
```

```
## [1] 0
```

```
phasePlot(dd1, "fly2", 2, 0.077)
```

Next, we compute the entire phase sample on DD1 using the optimal filter parameters.

```
# optimal filter parameters from matching the observed average phase
filter_order <- 2
filter_freq <- 0.077
flynames <- tail(names(dd1),-2) # all 32 fly names "fly1", "fly2", ..., "fly32"
phase_sample <- map_dbl(flynames, phase, data=dd1,
                        filter_order = filter_order, filter_freq = filter_freq)
```

Now, the summary statistics of the `phase_sample` on DD1 are as follows:

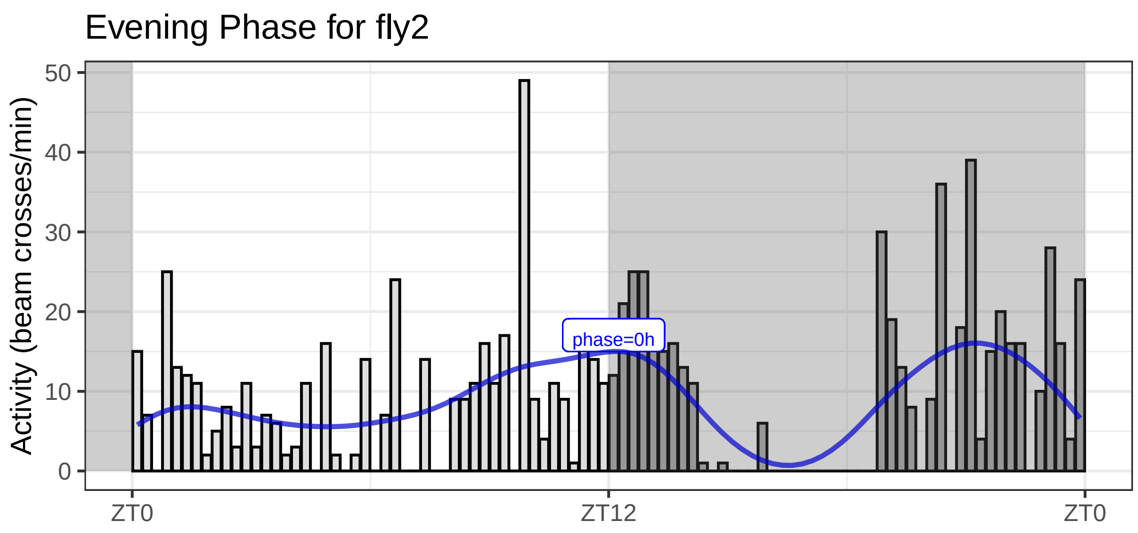

Figure 11: Raw and filtered activity for fly2 and its evening phase on DD1 with optimal filter parameters.

```
summary(phase_sample)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
## -3.7500 -1.2500 -0.5000 -0.8981 -0.2500  0.5000       5
```

In Figure 12, we show the distribution of the phases on DD1 obtained with the optimal filter parameters.

```
tibble(phase_sample) %>%
  ggplot(aes(x=phase_sample)) +
  geom_histogram(col="black",fill="gray") +
  labs(title=paste("Histogram of phase sample (N=32) with n=",
                   filter_order,"and W=",filter_freq), x="phase")
```

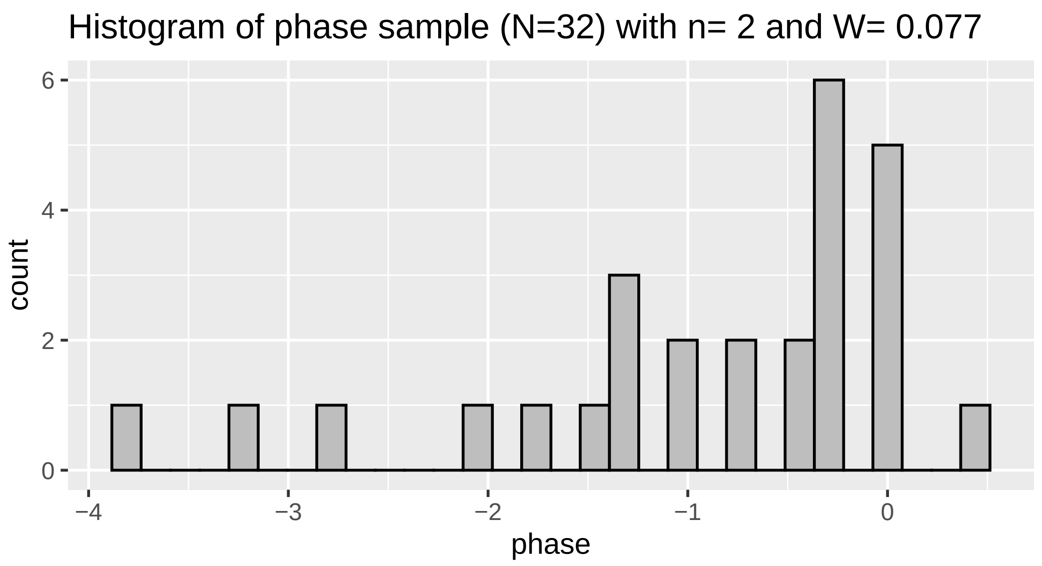

Figure 12: The frequency histogram of the phase sample on DD1.

Table 6: The head of the phases dataframe.

PdfGal4	uasUNC5	PdfUNC5	uasDBTL	uasDBTLuasUNC5	PdfDBTL	PdfDBTLUNC5
-0.5	-1.5	-1.5	0	-1.5	3.5	2
-0.5	-1.0	0.5	-1	-1.0	2.0	2
0.5	0.0	NA	0	-1.0	1.5	2
1.0	-1.5	0.0	0	-0.5	1.0	3

5 Circular Phase Analysis and Visualizations

In this section, we illustrate the circular data analysis and visualization functionality of the R package **circular**, which can be installed in RStudio from CRAN. You can make the **circular** package available in your current R session by loading it with the command `library(circular)`.

For illustration purposes, we use the actual phase data from Fernandez et al. [3]. A good introduction to the analytical aspects of circular statistics can be found in Batschelet [1], and the R package **circular** is illustrated in Pewsey, Neuhäuser, and Ruxton [4].

Data were obtained from a system entrained to a 24h environmental cycle (lights-on = `ZT00`). Negative phases represent anticipation relative to lights-on or lights-off transitions, thus if a transition was anticipated there would be a negative phase value. On the other hand, the positive phases represent a delay in response relative to the `ZT00` time set at "midnight" on our clock. For example, if a phase that happened very late in the night would occur at, say Zeitgeber time 23.2h, anticipating lights-on at `ZT00`, it would therefore be given a phase value of -0.8h.

Before we analyze and visualize the phase data, we transform the negative and positive phases into proper hours on the 00-24h time scale by taking all phases modulo 24, and then converting the proper hours into radians. The zero hour `ZT00` is set at 24h, or 2π radians. For example, the negative phase of -0.8h, when taken modulo 24, would be converted to 23.2h and then into 6.0737 radians.

We also illustrate the application of the Watson-Wheeler Test for Homogeneity of Angles and the Watson Two-Sample Test of Homogeneity to determine whether the phases for control and experimental lines are significantly different. Watson's non-parametric two sample U_2 statistic provides a criterion to test whether two samples differ significantly from each other. For both tests, the null hypothesis is that the two samples of angles come from the same underlying population.

5.1 Importing the Phase Data

The raw phase data are contained in the file `phases.csv`, which we import in **RStudio** as `phases`.

```
phases <- read_csv("phases.csv") # import into RStudio
```

The `phases` dataframe has 7 variables and 64 observations, including `NA`s. The variables represent the phases for different genotypes. Here is the head of the data:

```
phases %>% head(4) %>%
  knitr::kable(label="phases",caption = "The head of the phases dataframe.")
```

5.2 Transforming the Phase Data

We transform the phases into proper hours by taking the original phases modulo 24, and then converting the hours into radians. The zero hour `ZT00` is set at 24h, or 2π radians. The conversion formula is given by:

$$\phi_1 = \frac{(\phi_0 \bmod 24)2\pi}{24}, \tag{2}$$

where ϕ_0 is the raw phase (positive or negative), $\phi_0 \bmod 24$ is taking the raw phase modulo 24, which turns it into a proper hour on the 24h clock, and ϕ_1 is the converted angular phase, given in radians. The R code below transforms all raw phases at once into angular measurements in radians, following (2), contained in the vector `radphases`:

```r
# phases modulo 24 are transformed into hours and converted to radians
radphases <- (phases %% 24)*pi/12 # phases in radians
```

An alternative approach to transforming the phases is based on using the R package **circular**. First, we take the raw phases (positive and negative) modulo 24, which transforms them into proper hours, and we name the resulting dataframe by `mod24phases`. Note that `x modulo y` in R is implemented by `x %% y`. Then, we use the `circular()` function from the **circular** package to create a *circular data* object. Now, we can transform `mod24phases` into another circular data object with units of radians, modulo 2π, named `test`, using the `conversion.circular()` function. Finally, we save the previously computed `radphases` as a circular data object with units of radians, so that we can compare the phases we computed in radians from scratch with the phases computed by the `conversion.circular()` function.

```r
# library(circular) # circular must be installed first
mod24phases <- (phases %% 24) # take raw phases mod 24 to transform into hours
mod24phases <- circular(mod24phases, units ="hours", modulo = "asis",
                        template = "clock24",zero=24)
test <- conversion.circular(mod24phases, units="radians", template = "clock24",
                            modulo="2pi",zero=pi/2)
cirradphases <- circular(radphases, units = "radians", template = "clock24",
                         modulo="2pi", zero=pi/2)
```

The result of this alternative approach is that the transformed phases are indeed the same. This is confirmed by applying the function `all.equal(x,y)`, which compares R objects `x` and `y` by testing for *near equality*, within the default numerical tolerance.

```r
all.equal(cirradphases, test) # test if the two objects are equal
```

```
## [1] TRUE
```

5.3 Circular Data Plots

From now on, we are going to use the circular data object `cirradphases`, which contains phases transformed into angles measured in radians on a 24h clock. Applying the generic `plot()` function to a circular data object, creates a circular data plot.

In Figure 13, we plot the first variable `PdfGal4` in `cirradphases`. Inside `plot()`, the argument `stack = TRUE` allows for different observations corresponding to the same (or close enough) angle to be stacked on the top of each other. The argument `cex=1` sets the size of the plotting symbol used to represent each data point in the plot. The argument `pch = 16` sets the type of the plotting symbol used to represent each data point in the plot, in this case a solid dot. The argument `sep = 0.05` sets the separation between the stacked dots on the plot. The argument `shrink = 1.7` reduces the radius of the circle to ensure that all data points are visible within the plotting window. We also use a helper function `alpha()` from the **scales** package, which controls the color transparency.

```r
# library(scales) # scales must be installed first
plot(cirradphases[,1], stack = TRUE, col=alpha("blue",0.8), pch = 16, cex=0.8,
     sep = 0.08, shrink = 1.9)
```

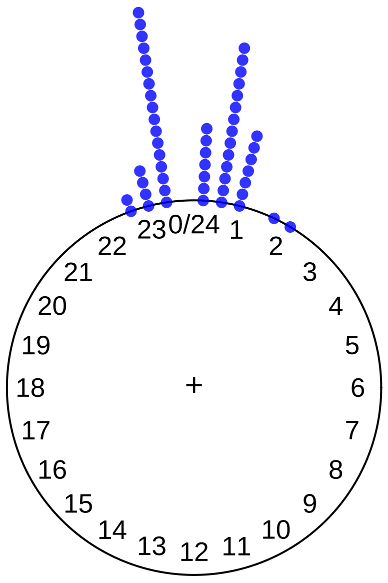

Figure 13: PdfGal4 phases in radians.

The `points()` function is helpful for displaying more than one circular data set within the same circular data plot. For example, the code chunk below generated Figure 14, where we plot the first variable `PdfGal4` in `cirradphases`, as before, but then we also add to this plot the observations contained in the second variable `uasUNC5`.

```r
plot(cirradphases[,1], stack = TRUE, col=alpha("blue",0.8), pch = 16, cex=0.8,
     sep = 0.08, shrink = 1.9)
points(cirradphases[,2], stack = TRUE, col=alpha("red",0.8), pch = 16, cex=0.8,
       sep = 0.08)
```

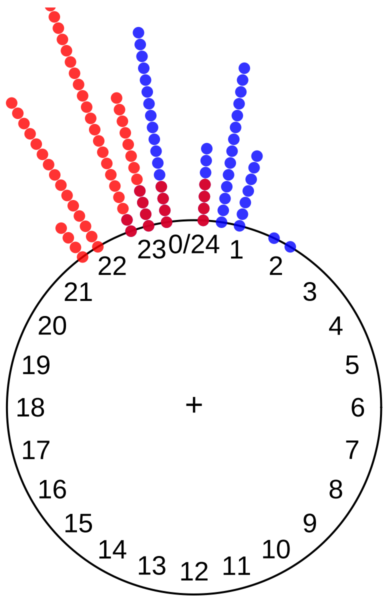

Figure 14: PdfGal4 and uasUNC5 phases on the same plot.

5.4 The Circular Mean

The **circular mean** computes the mean direction of a circular variable. Each observation in the circular variable is treated as a unit vector, or a point on the unit circle. The resultant vector of all observations is computed and the direction of the resultant vector is returned as the circular mean, which is implemented in the **circular** package as a method for the generic function `mean()`, applied to a circular data object. Since the phase data have `NA` values, we have to use the `na.rm = TRUE` argument to remove the `NA`s.

In the code chunk below, we compute the circular mean of the `PdfGal4` variable in the data.

```r
# calculate the circular mean
data <- cirradphases[,1] # first variable PdfGal4
circmean <- mean(data,na.rm = TRUE)
show(circmean)
```

```
## Circular Data:
## Type = angles
## Units = radians
## Template = clock24
## Modulo = 2pi
## Zero = 1.570796
## Rotation = clock
## [1] 0.009657737
```

We can visualize the circular mean as the mean direction of the circular variable by plotting an arrow that represents the resultant mean direction vector by using `arrows.circular()`. The call `arrows.circular(data)` displays the vectors on the unit circle corresponding to the angles (in radians) in the circular data, relative to the zero on the 24h clock. Note that `data` has 11 `NA` values, and arrows are displayed only for the remaining 53 numerical values, but with overlaps captured by the color density of the arrows. For displaying the mean resultant vector, we use the **mean resultant length** of the circular data computed with `rho.circular()`. This is shown in Figure 15.

```r
R <- rho.circular(data, na.rm = TRUE)
plot(data, stack=TRUE, cex=0.8, shrink = 1.9, sep=0.08, col=alpha("blue",0.8), pch = 16)
arrows.circular(data, col=alpha("black",0.3))
arrows.circular(circmean, y=R, lwd=3, col=alpha("blue",0.5))
```

5.5 The Rose Diagram

The rose diagram represents frequencies in the circular data by areas of sectors on the unit circle. The rose diagram can be added to an existing circular plot using the `rose.diag()` function. For more details on the many arguments this function has, please refer to the **circular** package documentation.

There are two conventions in the literature regarding rose diagrams. Here we use the default convention of the `rose.diag()` function, where the radius of a segment is taken to be the square root of the relative frequency. With this convention, when we compare segments in a rose diagram, **the ratio of the areas of two segments is the same as the ratio of the relative frequencies**, since the segment areas are proportional to the radius squared.

We can choose how many segments to divide the circular data into in order to create the rose diagram. Keep in mind that how people read the rose diagram is sensitive to this choice and it is advisable to try a range of values for the number of segments and make a choice informed by the resulting plots. A good first guess is to take the square root of the sample size, but multiples of 4 are popular choices for circular data.

In Figure 16, we add a rose diagram to a circular plot as well as an arrow pointing in the mean circular direction of the circular data. In the `rose.diag()` function, the argument `bins=16` controls the number of

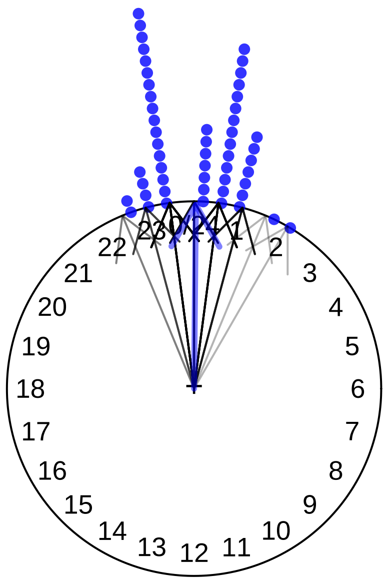

Figure 15: Mean direction, shown by the blue arrow.

segments, and the argument `prop=1.1` scales the size of all segments relative to the outer circle, with larger values increasing the size of the segments, having a default value of one.

```
plot(data, stack=TRUE, cex=0.8, shrink = 1.9, sep=0.08, col=alpha("blue",0.8), pch = 16)
arrows.circular(circmean, y=R, lwd=3, col=alpha("blue",0.7))
rose.diag(data, bins=16, col=alpha("blue",0.3), prop=1.1, axes=FALSE, add=TRUE)
```

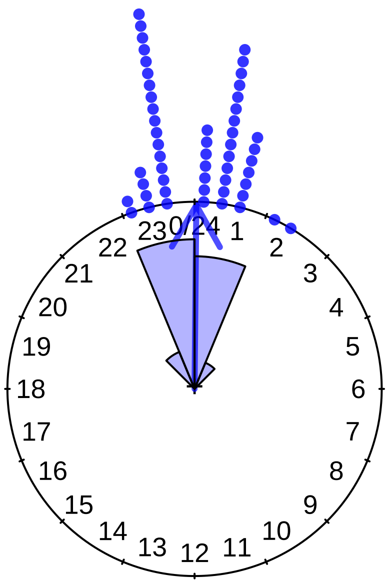

Figure 16: Mean direction vector (in blue), and a rose diagram of PdfGal4.

5.6 The Watson Two-Sample Test

We apply the Watson two-sample test to determine whether the phases for control and experimental lines are significantly different. Watson's non-parametric two sample U^2 statistic provides a criteria to test whether two samples differ significantly from each other. The difference is not specified, it may be in the mean, the angular variance, or something else.

The circular data should consist of two independent random samples of circular observations drawn from populations with a continuous distribution. The null hypothesis H_0 states that **the two samples belong to the same parent population**.

The **decision rule** states that:

1. If $U^2 <$ `critical value`: the null hypothesis cannot be dismissed.
2. If $U^2 >$ `critical value`: reject the null hypothesis, and conclude that the two samples differ significantly.

The larger the value of U^2, the more likely the two samples belong to different populations. The smaller the value of U^2, the more likely the two samples belong to the same population.

In the code chunk below, we run the Watson Two-Sample Test using the function `watson.two.test()` from the **circular** package. The two circular variables correspond to **uasDBTL** and **PdfDBTL** genotypes, in our phase data.

```
data1 <- cirradphases[,4] # uasDBTL phases
data2 <- cirradphases[,6] # PdfDBTL phases
watson.two.test(data1,data2, alpha=0.05)
```

```
##
##       Watson's Two-Sample Test of Homogeneity
##
## Test Statistic: 1.6978
## Level 0.05 Critical Value: 0.187
## Reject Null Hypothesis
```

The `alpha=0.05` argument specifies the *significance level* of the test. Valid levels are 0.001, 0.01, 0.05, 0.1. This argument can be omitted, in which case, a range for the p-value will be returned. The p-value is estimated by assuming that the test statistic follows a chi-squared distribution. For this approximation to be valid, each dataset must have at least 10 elements.

One can also apply the Watson-Wheeler test for homogeneity on two or more samples of circular data, using the function `watson.wheeler.test(list(data1,data2))` from the **circular** package. The difference between the samples can be in either the mean or the variance.

5.7 A Circular Plot and Rose Diagrams for PdfDBTL vs. uasDBTL

In Figure 17, we show the circular plot of `PdfDBTL` and `uasDBTL`, along with arrows pointing to the circular means of the two datasets, and their rose diagrams. The code chunk below contains the complete code that generates Figure 17.

```
# circular data plot
plot(cirradphases[,4], stack = TRUE, col=alpha("gray45",0.8), pch = 16, cex=0.8,
              sep = 0.08, shrink = 2.2, main="") # for uasDBTL
# adding points for PdfDBTL
points(cirradphases[,6],stack=TRUE,col=alpha("gold3",0.8),pch=16,cex=0.8,sep=0.08)
# circular means
circ.mean4<-mean(cirradphases[,4],na.rm = TRUE) # for uasDBTL
circ.mean6<-mean(cirradphases[,6],na.rm = TRUE) # for PdfDBTL
# adding arrows to circular means
arrows.circular(circ.mean4, col = alpha("gray45",0.6),lwd=3)
arrows.circular(circ.mean6, col = alpha("gold3",0.6),lwd=3)
# adding a legend
```

```
legend(-0.3,-0.2,legend=c("uasDBTL","Pdf>DBTL"),
       col=c(alpha("gray45",0.8),alpha("gold3",0.8)),pch=16,
       cex=0.8,box.lty=0)
# adding rose diagrams
rose.diag(cirradphases[,4],bins=12,col=alpha("gray45",0.4),cex=0.8,prop=0.85,add=TRUE)
rose.diag(cirradphases[,6],bins=12,col=alpha("gold3",0.4),cex=0.8,prop=0.85,add=TRUE)
```

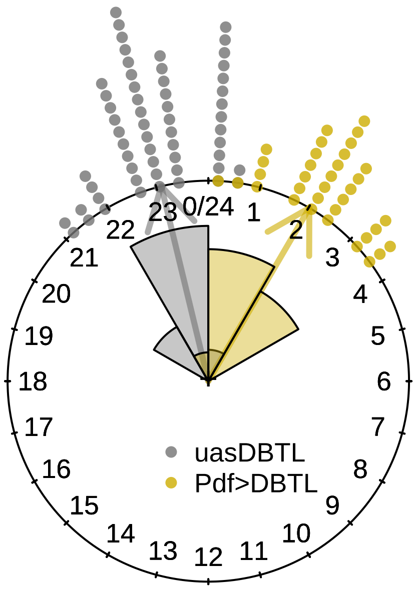

Figure 17: The circular plot and rose diagrams of PdfDBTL and uasDBTL.

References

[1] E. Batschelet. *Circular Statistics in Biology*. Mathematics in biology. Academic Press, 1981. ISBN: 9780120810505.

[2] Maria P. Fernandez et al. *GitHub Repository*. Oct. 2020. URL: https://github.com/bkostadi/STARProtocolsExtendedMethods.

[3] Maria P. Fernandez et al. "Sites of Circadian Clock Neuron Plasticity Mediate Sensory Integration and Entrainment". In: *Current Biology* 30.12 (2020), 2225–2237.e5. ISSN: 0960-9822. DOI: https://doi.org/10.1016/j.cub.2020.04.025. URL: http://www.sciencedirect.com/science/article/pii/S0960982220305054.

[4] A. Pewsey, M. Neuhäuser, and G.D. Ruxton. *Circular Statistics in R*. OUP Oxford, 2013. ISBN: 9780191650772.

[5] Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st. O'Reilly Media, 2017. ISBN: 1491910399.