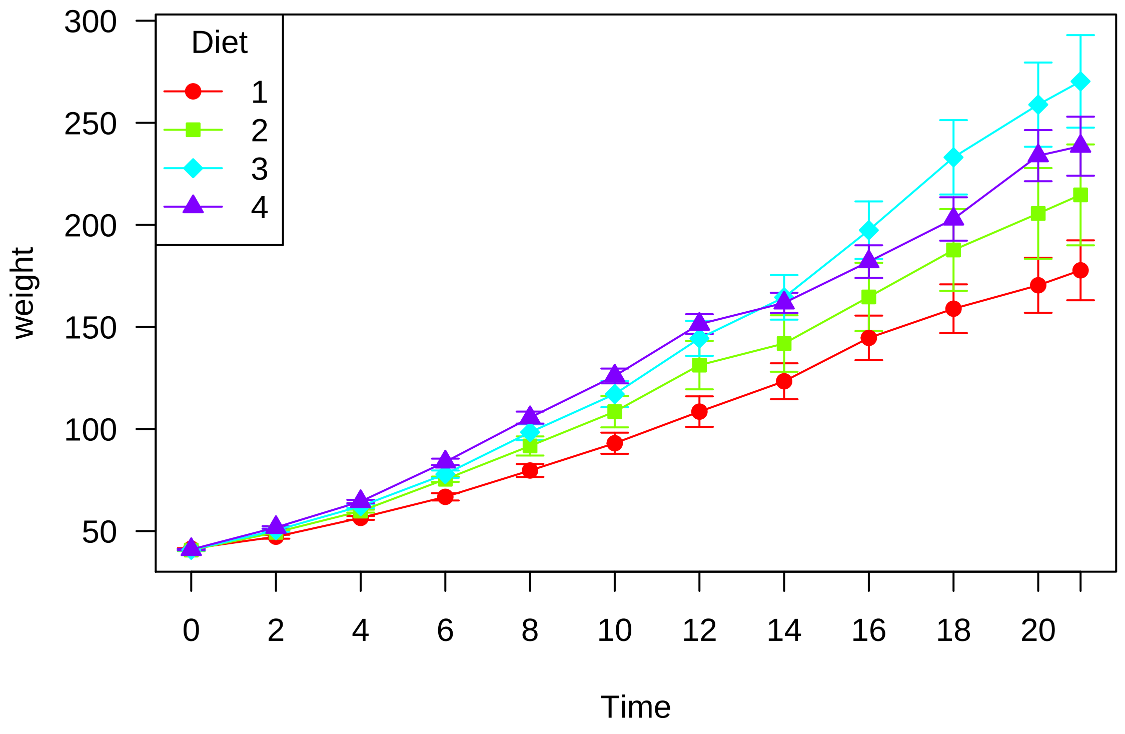
LineChart Package Introduction

The LineChart package can be used for simple line graphs that have a continuous dependent variable, a continuous independent variable, and a categorical independent variable. Line graphs are very common in a lot of areas, such as poultry farming, so we will use the ChickWeight data set in our examples. The ChickWeight data set has weight measurements at 12 time points for each of 50 chicks which were assigned to 1 of 4 diet conditions. We want to plot weight as a function of time and diet, collapsing across chicks (we don't care about individual differences).

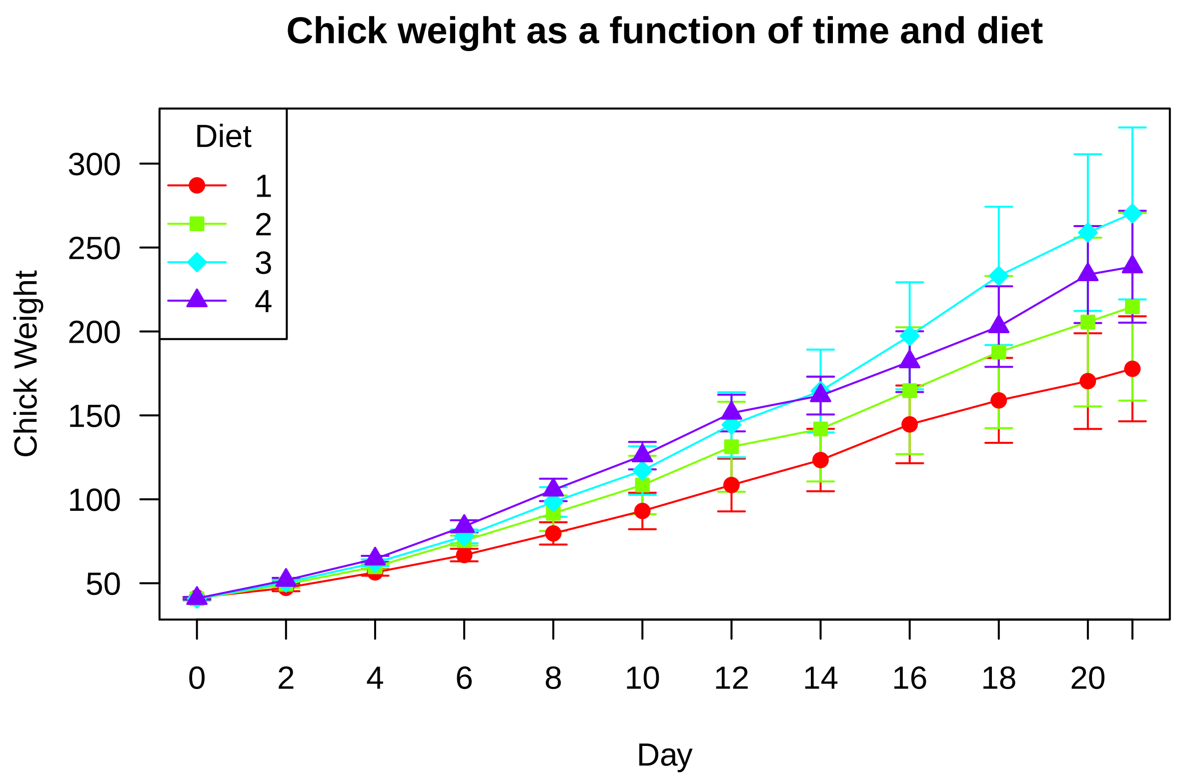
```r
library(LineChart)

data(ChickWeight)

lineChart(weight ~ Time * Diet, ChickWeight)
```



As you can see, the main interface is very simple. Your provided data are aggregated to find the means, error bars are generated, the grouping variables are given a default appearance, the legend and axes are titled based on the names of the data sources, and everything is plotted.

There are a variety of things we can do to customize how the line graph is plotted. We can change what type of error bar is used (standard error, standard deviation, and 95% confidence or credible intervals can be calculated for you). We'll use 95% confidence intervals. We can also add a title and change the axis labels.

```r
lineChart(weight ~ Time * Diet, ChickWeight, errBarType="CI95",
    title="Chick weight as a function of time and diet",
    xlab="Day", ylab="Chick Weight")
```



Plot Appearance Settings

For more control over the appearance of the different groups, you can modify the appearance settings. The diet conditions are numbered 1 through 4, so our groups to apply settings to are the numbers 1 through 4. Instead of using the diet numbers for symbols, we'll use the fillable plotting characters. We'll make all of the lines black, but fill the symbols with a rainbow of color.

```
settings = buildGroupSettings(group=1:4, symbol=21:24,
    color="black", fillColor=rainbow(4),
    lty=1:4)
```

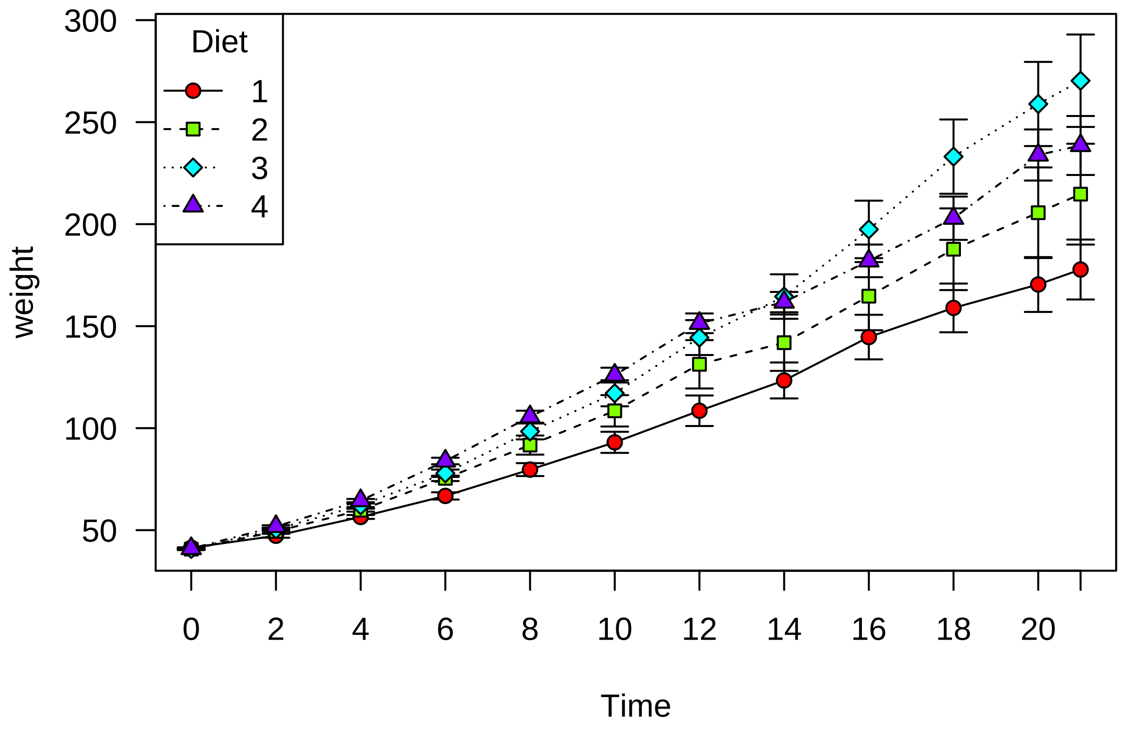
```
## Note: Plotting defaults used for: cex.symbol, width.errBar, lwd, include
```

`buildGroupSettings` complains about things that are left as defaults, but that can be stopped if the `suppressWarnings` argument is set to `TRUE`. The appearance settings are stored in a simple data frame with a number of columns with special names:

```
##   group groupLabel color fillColor symbol cex.symbol lty lwd width.errBar
## 1     1          1 black   #FF0000     21          1   1   1         0.07
## 2     2          2 black   #80FF00     22          1   2   1         0.07
## 3     3          3 black   #00FFFF     23          1   3   1         0.07
## 4     4          4 black   #8000FF     24          1   4   1         0.07
##   include
## 1    TRUE
## 2    TRUE
## 3     TRUE
## 4     TRUE
```

The `include` column specifies whether that group should be included in plots. The other settings control the appearance in straightforward ways. `color` sets the color for lines and symbols. If the symbol is fillable, `fillColor` sets the fill color. `symbol` and `cex.symbol` control the plotting character and the size of plotting characters, respectively. `lwd` and `lty` set the line width and type in the standard R fashion. `width.errBar` controls the width of the whiskers at the ends of the error bars. We can use these settings when calling `lineChart` by passing them to the `settings` argument.

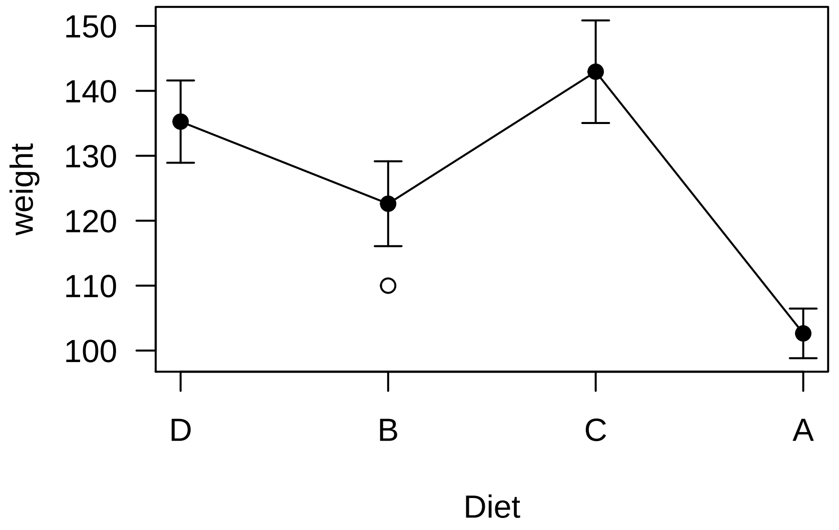
```
lineChart(weight ~ Time * Diet, ChickWeight, settings=settings, legendPosition="topleft")
```



Non-Numeric X-Variables

You are free to use non-numeric x-variables, as shown in the following example where `Diet` is converted to letters.

```
dat = ChickWeight

dat$Diet = LETTERS[dat$Diet] # Make Diet a string

lineChart(weight ~ Diet, dat, xOrder = c("D", "B", "C", "A"))

# Add a point to the plot
points(2, 110)
```



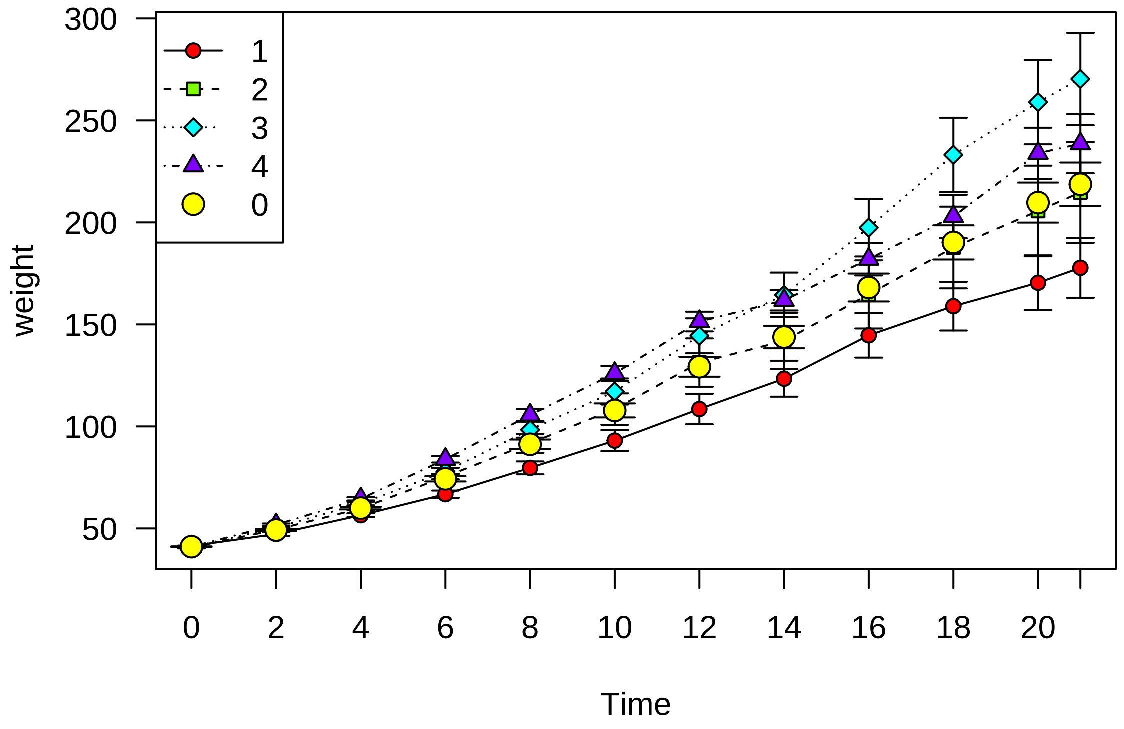
When working with non-numeric x-variables, you may want to specify their order, which you can do with the `xOrder` argument. It takes a vector of the values of the x-variable in an order which is used to determine the order along the x-axis that the variables are plotted.

Note that a point was added a (2, 110). When using non-numeric x-variables, the x-location of the values starts at 1 and increases by 1 for each value.

Overplotting additional data

You might be interested in seeing the grand mean overplotted on the data. This can be done by calling `lineChart` repeatedly, but with the `add` argument set to `TRUE`, so that the subsequent plot adds to the existing plot rather than starting a new plot. Because the plot for the mean does not use a group, just the x variable, there is no obvious name for the mean group, so it is given the value `0` by the LineChart package. Thus, when settings are made for the mean group, the group name should be `0`. Finally, because the two plots use different groups, neither would make a legend including all plotted groups. The solution is to suppress legend plotting for the first two calls, then manually create a legend with `legendFromSettings`, `rbind`-ing together the settings data frames.

```r
#The first plot, that we are so used to
lineChart(weight ~ Time * Diet, ChickWeight, settings=settings, legendPosition=NULL)

#Create settings for the grand mean group and plot it
meanSettings = buildGroupSettings(0, color="black", fillColor="yellow",
    symbol=21, cex.symbol=1.5,
    lty=0, suppressWarnings=TRUE)

lineChart(weight ~ Time, ChickWeight, settings=meanSettings, legendPosition=NULL, add=TRUE)

#Make a legend from the settings
legendFromSettings("topleft", rbind(settings, meanSettings))
```



Working with the plotting data frame directly

`lineChart` uses a plotting data frame as part of its process, which is returned invisibly and can also be gotten from `createPlottingDf`. Using the plotting data frame directly allows for some advanced uses of the LineChart package. We start by getting a plotting data frame.

```
plotDf = createPlottingDf(weight ~ Time * Diet, ChickWeight, settings=settings)
plotDf[ 1:5, ]
```

```
##    x     y group NObs    ebLower   ebUpper xLabel groupLabel color fillColor
## 1  0 41.40     1   20 -0.2224268 0.2224268      0          1 black   #FF0000
## 13 0 40.70     2   10 -0.4725816 0.4725816      0          2 black   #80FF00
## 25 0 40.80     3   10 -0.3265986 0.3265986      0          3 black   #00FFFF
## 37 0 41.00     4   10 -0.3333333 0.3333333      0          4 black   #8000FF
## 2  2 47.25     1   20 -0.9566251 0.9566251      2          1 black   #FF0000
##    symbol cex.symbol lty lwd width.errBar include
## 1      21          1   1   1         0.07    TRUE
## 13     22          1   2   1         0.07    TRUE
## 25     23          1   3   1         0.07    TRUE
## 37     24          1   4   1         0.07    TRUE
## 2      21          1   1   1         0.07    TRUE
```

The plotting data frame has `x`, `y`, and `group` columns that specify the data to plot. The `y` column contains the average value of the dependent variable for the given combination of `x` and `group`. The `errBar` and `errBarLower` columns specify offsets from the `y` values at which error bars will be drawn. We can see that appearance settings have been applied to the plotting data frame. Those settings can later be extracted from a plotting data frame with `extractGroupSettings`. By working directly with the plotting data frame, we can do more than is possible just by using `lineChart`.

One thing we can do is maually specify different types of error bars than the standard build-in types, including asymmetrical and single-sided error bars. The two columns in the plotting data frame which control error bars are `errBar` and `errBarLower`. If both are `NULL`, `NA`, or 0, no error bars are drawn. If `errBarLower` is `NULL` or `NA`, `errBar` is used for both the upper and lower error bars. If `errBarLower` is used, it should be a negative value. If single-sided error bars are desired, simply set one of `errBar` or `errBarLower` to 0. This can, naturally, be done separately for individual groups.

For this example, we will just use a subset of the data, diets 1 and 3. We'll make single-sided standard deviation error bars by zeroing out one side of the error bars for two different groups.

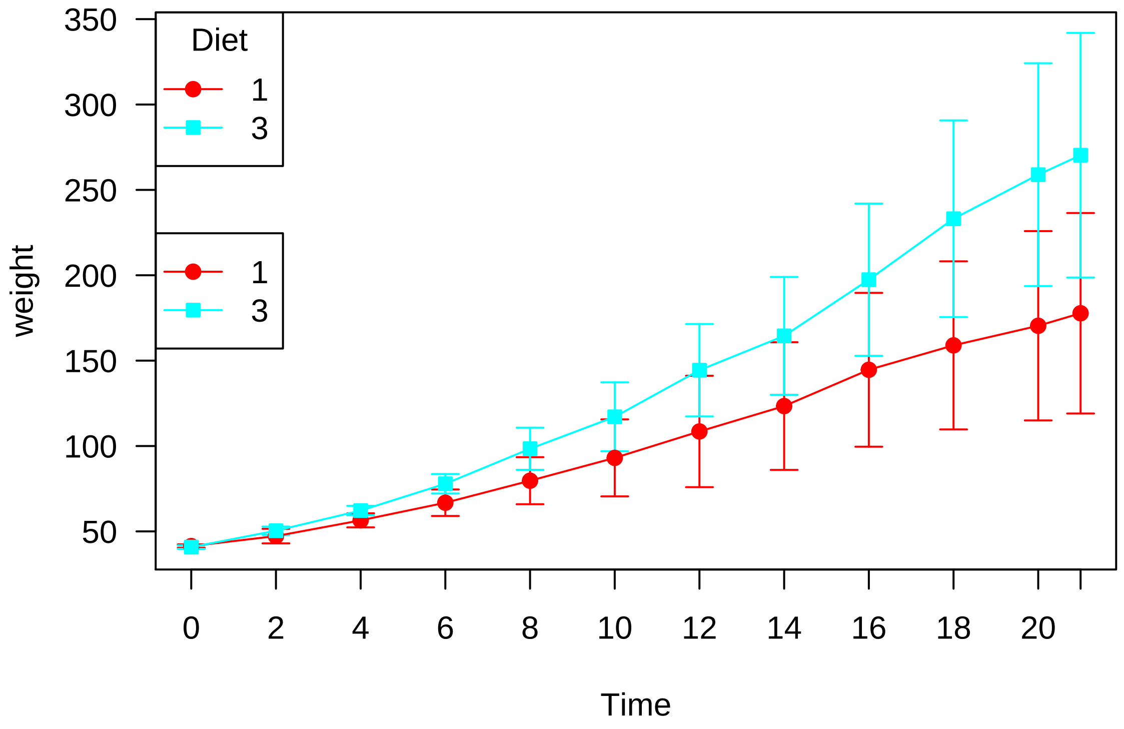
```
cw13 = ChickWeight[ ChickWeight$Diet %in% c(1, 3), ]

plotDf = createPlottingDf(weight ~ Time * Diet, cw13, errBarType="SD")

plotDf[plotDf$group == 1, ]$errBar = 0
```

```
## Warning in `[<-.data.frame`(`*tmp*`, plotDf$group == 1, , value =
## structure(list(: provided 17 variables to replace 16 variables
```

```
plotDf[plotDf$group == 3, ]$errBarLower = 0
```

```
## Warning in `[<-.data.frame`(`*tmp*`, plotDf$group == 3, , value =
## structure(list(: provided 17 variables to replace 16 variables
```

```
lineChartDf(plotDf)
legendFromPlottingDf("left", plotDf)
```



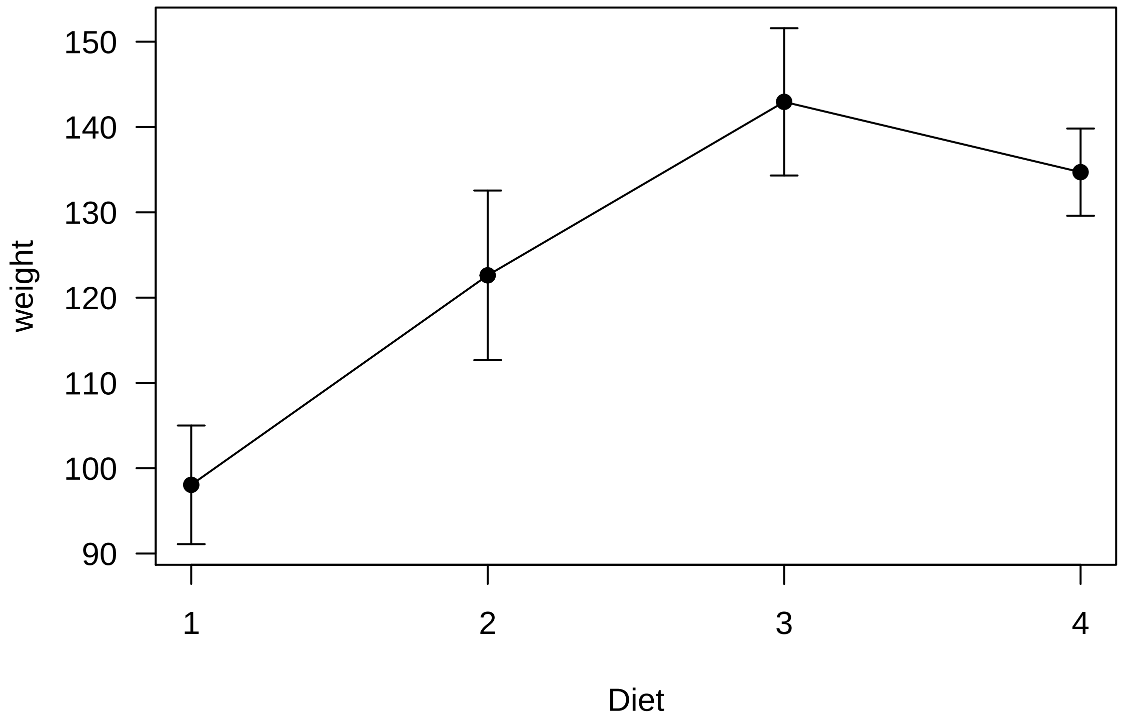
Plots made by `lineChartDf` do not have a legend, so we added one with `legendFromPlottingDf`. If you do not have the plotting data frame, but just the appearance settings data frame, you can use `legendFromSettings` instead. If you want to get the settings from a plotting data frame, use `extractGroupSettings`.

Replicates

LineChart can help deal with replicates in the data (e.g. in repeated-measures designs). In the ChickWeight data, the Chicks are replicates within the Diets.

To treat a column in a data frame as a replicate column, pass that column name to the `replicate` argument of `lineChart` (or `createPlottingDf`).

```
lineChart(weight ~ Diet, ChickWeight, replicate = "Chick")
```



Note that Chick is not included in the formula, only in the `replicate` argument.

Comparison with no replicates

The effect of using replicates is demonstrated below.

Make two plotting DFs that differ in use of the Chick replicate.

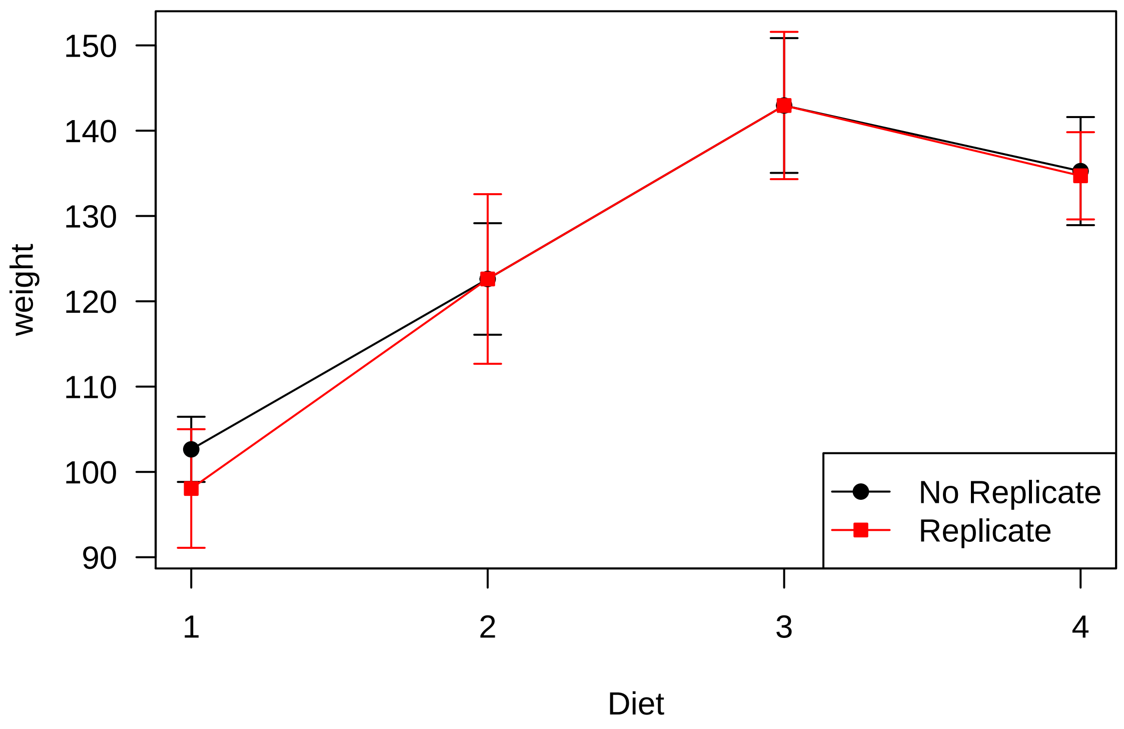
```
noRepDf = createPlottingDf(weight ~ Diet, ChickWeight)

repDf = createPlottingDf(weight ~ Diet, ChickWeight, replicate = "Chick", repFun=mean)
```

The `repFun` argument specifies what function will be used to aggregate each replicate's values, in this case the mean.

Then do some work to plot both in 1 plot.

```
# Set plotting group to 1 for replicate df
repDf$group = 1

# combine into 1 df
bothDf = rbind(noRepDf, repDf)
```

```
# Create group settings
settings = buildGroupSettings(group=c(0,1),
                              color=c("black", "red"),
                              groupLabel = c("No Replicate", "Replicate"),
                              suppressWarnings = TRUE)
# apply settings to bothDf
bothDf = applySettingsToPlottingDf(settings, bothDf)
# and plot
lineChartDf(bothDf)
```



The means and error bars differ depending on whether replicates are used. Notice in particular the `y` (central tendency), `NObs` (number of observations), and `ebLower`/`ebUpper` for error bar length.

```
print(bothDf)
```

```
##   x         y group NObs   ebLower  ebUpper xLabel   groupLabel color fillColor
## 1 1 102.64545     0  220 -3.819784 3.819784      1 No Replicate black     black
## 2 2 122.61667     0  120 -6.536840 6.536840      2 No Replicate black     black
## 3 3 142.95000     0  120 -7.900146 7.900146      3 No Replicate black     black
## 4 4 135.26271     0  118 -6.336197 6.336197      4 No Replicate black     black
## 5 1  98.05446     1   20 -6.951285 6.951285      1    Replicate   red       red
## 6 2 122.61667     1   10 -9.941800 9.941800      2    Replicate   red       red
## 7 3 142.95000     1   10 -8.631581 8.631581      3    Replicate   red       red
## 8 4 134.71000     1   10 -5.113737 5.113737      4    Replicate   red       red
##   symbol cex.symbol   lty lwd width.errBar include
## 1     21          1 solid   1         0.07    TRUE
## 2     21          1 solid   1         0.07    TRUE
## 3     21          1 solid   1         0.07    TRUE
## 4     21          1 solid   1         0.07    TRUE
## 5     22          1 solid   1         0.07    TRUE
## 6      22          1 solid   1         0.07     TRUE
## 7      22          1 solid   1         0.07     TRUE
## 8      22          1 solid   1         0.07     TRUE
```

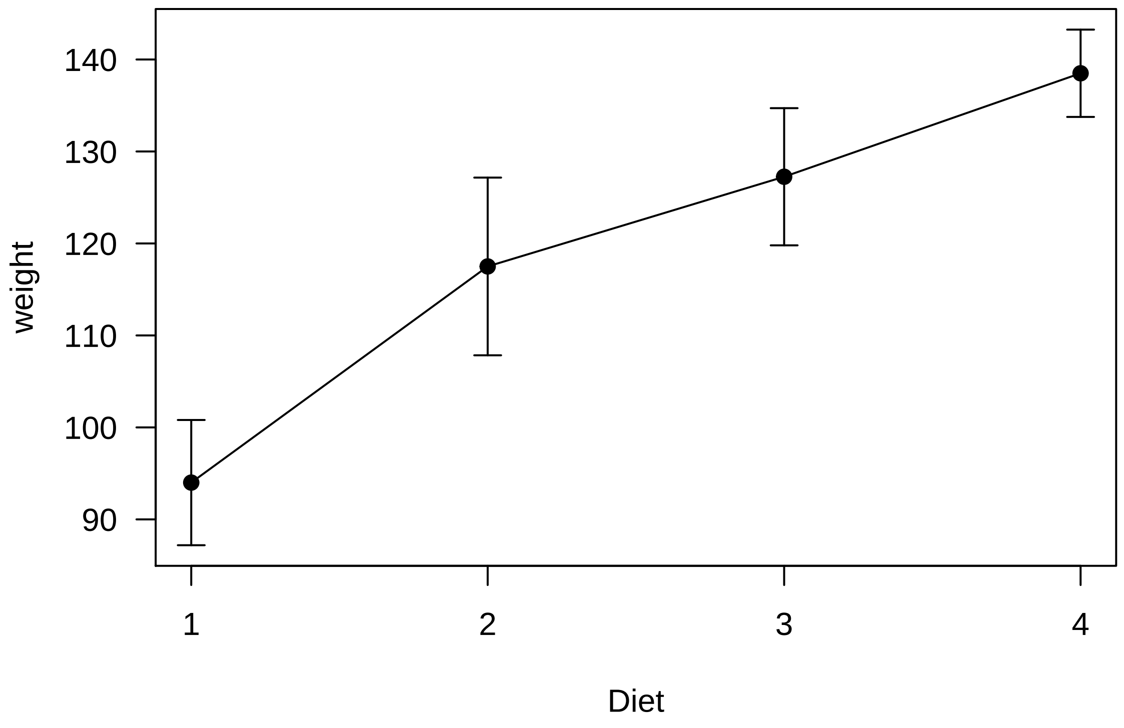
It is typical in psychology research to use participants as replicates with conditions, so the conventionally correct choice would be to use the `replicate` feature when plotting.

Details on Replicates

With `replicate = "Chick"`, `lineChart` first calculates the central tendency of each replicate within the other factors that are being used. In this case, `Chick` is within `Diet`.

For this example, the function used to aggregate replicates is set with `repFun` to the `mean` function while the central tendency is set to the median.

```
lineChart(weight ~ Diet, ChickWeight, replicate = "Chick",
          repFun = mean,
          centralTendencyType = median)
```



For the call, the following two steps are used to calculate the central tendency. Note that the following is pseudocode that is close to actual code.

```
# From the call:
repFun = mean
ctFun = median
```

1) Aggregate ChickWeight, keeping the Chick replicate. Use repFun.

```
temp = aggregate(weight ~ Diet * Chick, ChickWeight, repFun)
```

2) Aggregate temp, dropping the Chick replicate to get the Diet means. Use ctFun.

```
aggregate(weight ~ Diet, temp, ctFun)
```

```
##   Diet    weight
## 1    1  95.91667
## 2    2 120.66667
## 3    3 138.62500
## 4    4 135.91667
```

The error bars are also calculated at step 2(b) using temp with the error bar function (see `errBarType`).

When `replicate` is not supplied or is `NULL`, `lineChart` only does the second aggregate, getting a somewhat different result:

```
aggregate(weight ~ Diet, ChickWeight, ctFun)
```

```
##   Diet weight
## 1    1   88.0
## 2    2  104.5
## 3    3  125.5
## 4    4  129.5
```

Central Tendency and Error Bar Functions

You can provide custom functions to calculate the measure of central tendency (which is where the points are placed) and the error bars. Instead of plotting the mean and SE, let's imagine that you wanted to plot trimmed mean and standard error.

The first function defined here, `getTrimmedX`, is a helper function that returns values of `x` within `sds` standard deviations from the mean.

```
getTrimmedX = function(x, sds = 2) {
    mx = mean(x)
    sdx = sd(x)

    keep = x > (mx - sds * sdx) & x < (mx + sds * sdx)

    x[ keep ]
}
```

The next function, `trimmedMean`, will be passed to `lineChart`, so it has specific requirements. It must take one vector-valued argument and return a scalar.

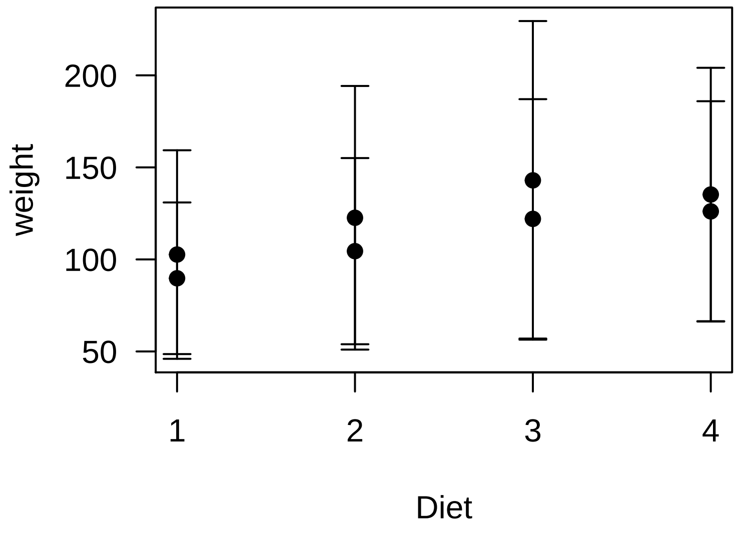
```
trimmedMean = function(x) {
    xt = getTrimmedX(x, 1.5)
    mean( xt )
}
```

The final function, `trimmedSD`, will also be passed to `lineChart`. It must take one vector-valued argument and return information about the error bars that should be drawn. See the documentation for `createPlottingDf` for an exact description of what it must be. In short, in this example, it is a list with two elements: `eb`, The length of the error bars and `includesCenter`, which is `FALSE` to indicate that `eb` gives lengths rather than endpoints of the error bars.

```
trimmedSD = function(x) {
    xt = getTrimmedX(x, 1.5)
    sdtx = sd( xt )
    list(eb = c(-sdtx, sdtx), includesCenter = FALSE)
}
```

I will first plot the data using mean and standard deviation, then overplot the trimmed mean and standard deviation.

```
settings = buildGroupSettings(0, lty="blank", suppressWarnings = TRUE)

lineChart(weight ~ Diet, ChickWeight, settings=settings,
    centralTendencyType = "mean", errBarType="SD")

# Change colors so the overplotted bars are visible
settings$color = "black"
settings$fillColor = "black"

lineChart(weight ~ Diet, ChickWeight, settings=settings,
    centralTendencyType = trimmedMean, errBarType = trimmedSD,
    add=TRUE)
```



Conclusion

This document gives the gist of what is available in the package. The point of the package is to do basic line graphs well and with the minimum of effort. You can play around with plotting data frames and measures of central tendency or variability to get fairly specific results, or you can just use `lineChart` and see your data immediately.