# Graphics and Plotting

## FRE6871 & FRE7241, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

*NYU Tandon School of Engineering*

May 15, 2024

NYU | TANDON SCHOOL OF ENGINEERING

# Plotting in R

## Functions in package *graphics* for creating static plots:

`plot()` plots a line or scatter plot,
`lines()` adds lines to a plot,
`curve()` plots a function given by its name,
`title()` adds a title to a plot,
`legend()` adds a legend to a plot,
`x11()` opens a *Windows* graphics device,
`par()` sets graphical plot parameters,

## Functions in package *ggplot2* for creating static plots:

`ggplot()` creates a new ggplot object,
`aes()` specifies *aesthetics*, i.e. mappings between data elements (rows, columns) and plot elements (points, lines),
`geom_point()` adds plot points,
`geom_line()` adds plot lines,
`theme()` customizes plot objects,

## Functions in package *plotly* for creating dynamic plots:

`plot_ly()` creates a new plotly object,
`add_trace()` adds elements to a *plotly* plot,
`layout()` modifies the layout of a *plotly* plot,

## Functions in package *dygraphs* for creating dynamic time series plots:

`dygraph()` creates an interactive time series plot,
`dyOptions()` adds options (like colors, etc.) to a *dygraph* plot,
`dyRangeSelector()` adds a date range selector to the bottom of a *dygraphs* plot,

# Package *graphics* Help and Documentation

## General

CRAN Graphics task view: http://cran.r-project.org/web/views/Graphics.html

## Tutorials on Charts and Plotting

http://www.statmethods.net/graphs/index.html
http://www.statmethods.net/advgraphs/index.html
http://en.wikibooks.org/wiki/R_Programming/Graphics#Standard_R_graphs
http://ww2.coastal.edu/kingw/statistics/R-tutorials/graphs.html
http://www.harding.edu/fmccown/r/

## Graphical Parameters

http://www.statmethods.net/advgraphs/parameters.html
http://research.stowers-institute.org/efg/R/Graphics/Basics/mar-oma/index.htm
http://www.programmingr.com/content/controlling-margins-and-axes-oma-and-mgp/

## Galleries of Charts

Vistat reproducible gallery of statistical graphics: http://vis.supstat.com/
Gallery of data charts: http://zoonek2.free.fr/UNIX/48_R/03.html
Gallery of data charts and R code: http://rgm3.lab.nig.ac.jp/RGM/R_image_list

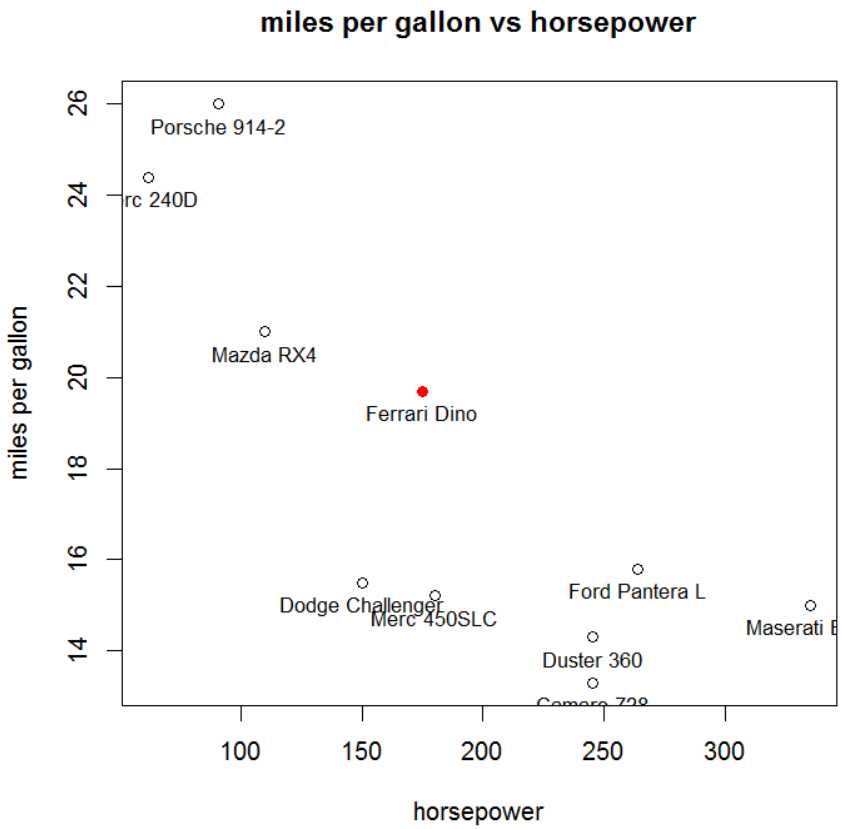
# Plotting Scatter Plots With Labels

The package *graphics* is one of the base packages in R, and offers a number of plotting capabilities.

The function `plot()` by default plots a scatter plot, but can also plot lines using the argument `type="l"`.

The function `text()` draws text on a plot, and can be used to draw plot labels.

```
> cardf <- mtcars[sample(NROW(mtcars), 10), ]
> # Plot scatter plot horsepower vs miles per gallon
> plot(cardf[, "hp"], cardf[, "mpg"],
+      xlab="horsepower", ylab="miles per gallon",
+      main="miles per gallon vs horsepower")
> # Add a solid red point (pch=16) for the last car
> points(x=cardf[NROW(cardf), "hp"],
+  y=cardf[NROW(cardf), "mpg"],
+  col="red", pch=16)
> # Add labels with the car names
> text(x=cardf[, "hp"], y=cardf[, "mpg"],
+      labels=rownames(cardf[, ]),
+      pos=1, cex=0.8)
> # Labels using wordcloud, to prevent overlaps
> library(wordcloud)
> textplot(x=cardf[, "hp"], y=cardf[, "mpg"],
+    words=rownames(cardf))
```



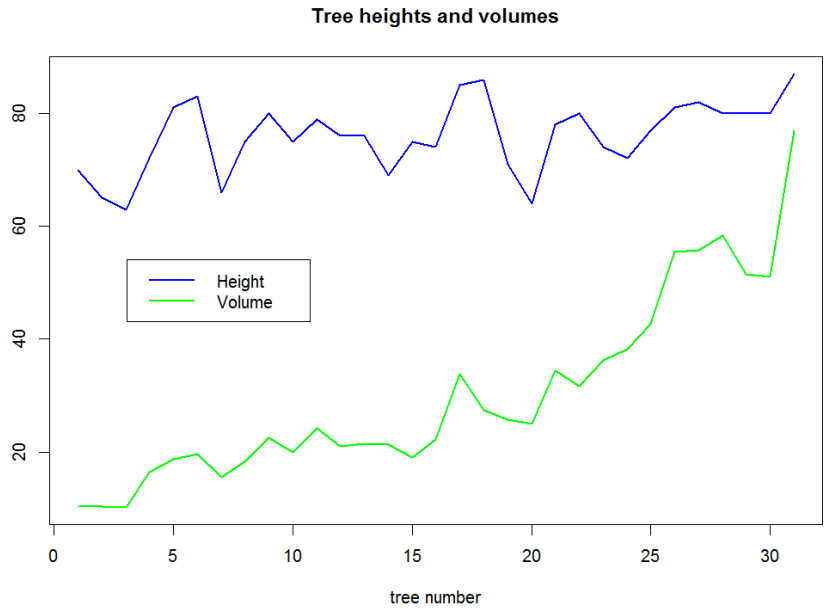

Rule of Thumb:
Always plot in a large x11() window, not in the default RStudio plot panel.
First open a large x11() window, then run your plot code.

# Plotting Line Plots Using the Base *graphics* Package

The function `plot()` accepts many different graphical parameters, including:

- `type`: type of plot,
- `lwd`: line width,
- `col`: plotting object color,
- `xlab`, `ylab`: axis titles,
- `xlim`, `ylim`: axis range,
- `main`: plot title,

```
> plot(trees[, "Height"],
+      type="l",
+      lwd=2,
+      col="blue",
+      main="Tree heights and volumes",
+      xlab="tree number", ylab="",
+      ylim=c(min(trees[, c("Height", "Volume")]),
+       max(trees[, c("Height", "Volume")])))
> # Plot the tree Volume
> lines(trees[, "Volume"], lwd=2, col="green")
> # Add legend
> legend(x="left", legend=c("Height", "Volume"),
+  inset=0.1, cex=1.0, bg="white", bty="n", y.intersp=0.4,
+  lwd=2, lty=1, col=c("blue", "green"))
```




The function `lines()` adds lines to a plot.

The function `legend()` adds a legend to a plot.

# Plotting Mathematical Functions

Plotting functions in package *graphics*:

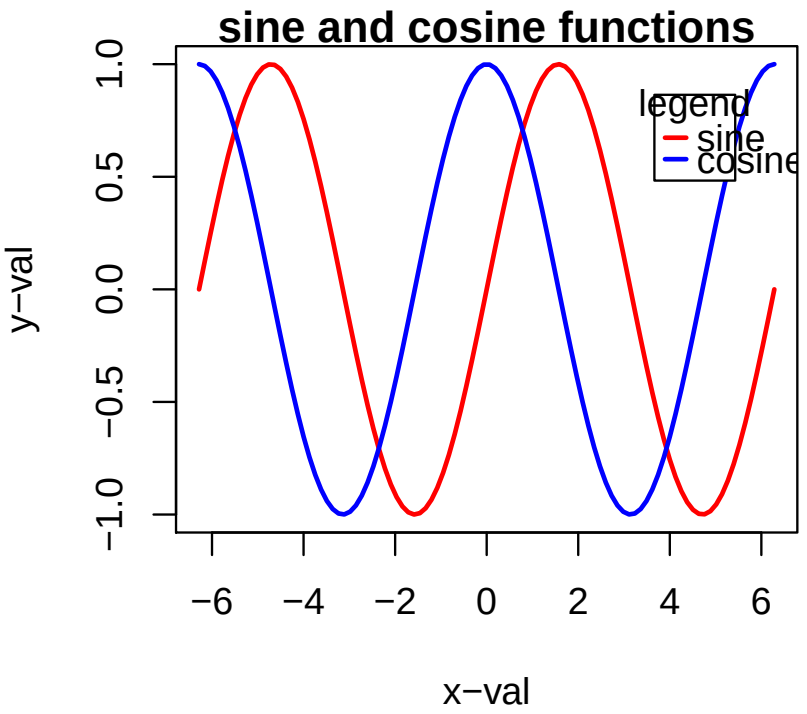
`x11()` opens a *Windows* graphics device.

`plot()` creates a scatter plot or line plot.

`lines()` plots a line on an existing plot.

`title()` adds a title to a plot.

```
> xvar <- seq(-2*pi, 2*pi, len=100)  # x values
>
> # open Windows graphics device
> x11(width=11, height=7, title="simple plot")
>
> # Plot a sine function using basic line plot
> plot(x=xvar, y=sin(xvar), xlab="x-val",
+      ylab="y-val", type="l", lwd=2, col="red")
> # Add a cosine function
> lines(x=xvar, y=cos(xvar), lwd=2, col="blue")
> # Add title
> title(main="sine and cosine functions", line=0.1)
> # Add legend
> legend(x="topright", legend=c("sine", "cosine"),
+  title="legend", inset=0.1, cex=1.0, bg="white", y.intersp=0.4,
+  lwd=2, lty=1, bty="n", col=c("red", "blue"))
> graphics.off()  # Close all graphics devices
```

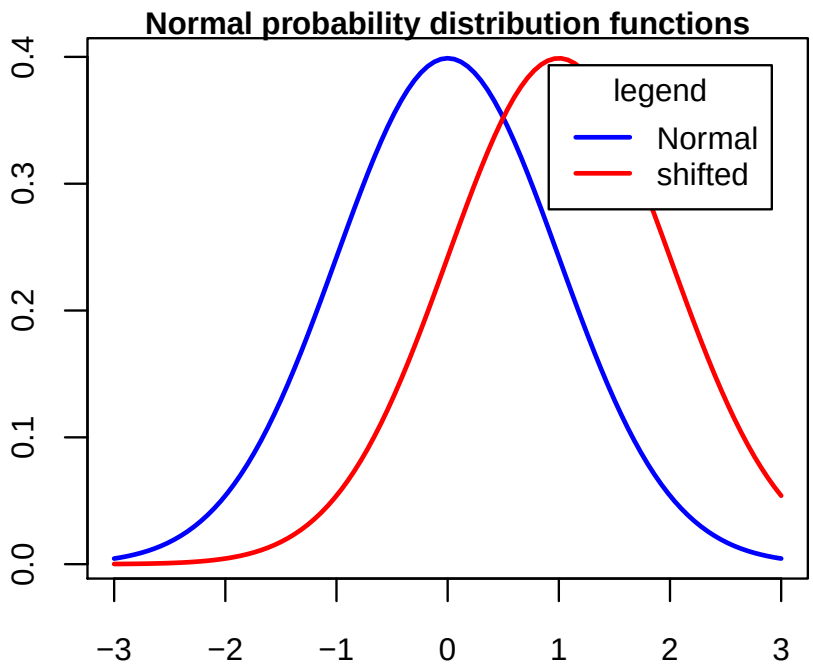


# Plotting Mathematical Functions Using curve()

R has dedicated functions for plotting mathematical functions.

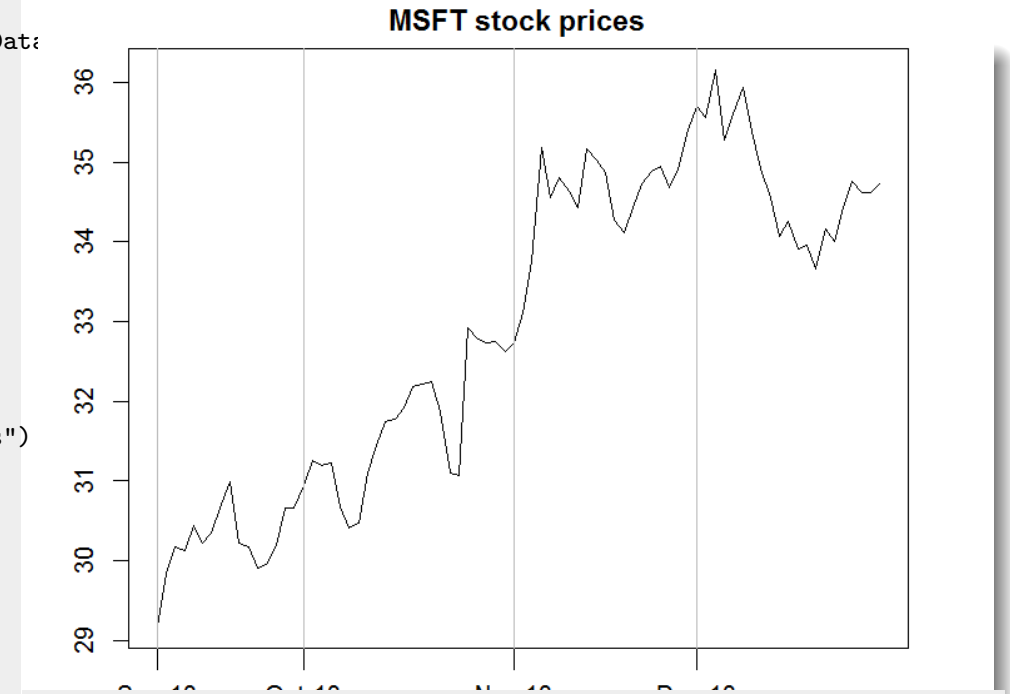
The function curve() plots a function defined by its name.

```
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-3, 3),
+ xlab="", ylab="", lwd=2, col="blue")
> # Add shifted Normal probability distribution
> curve(expr=dnorm(x, mean=1), add=TRUE,lwd=2, col="red")
>
> # Add title
> title(main="Normal probability distribution functions",
+ line=0.1)
> # Add legend
> legend(x="topright", legend=c("Normal", "shifted"),
+  title="legend", inset=0.05, cex=0.8, bg="white", y.intersp=0.4,
+  lwd=2, lty=1, bty="n", col=c("blue", "red"))
```



# Plotting zoo Time Series With Custom x-axis

```
> library(zoo)  # Load zoo
> load(file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RDat
> zoots <- window(zoo_stx[, "AdjClose"],
+    start=as.Date("2013-01-01"),
+    end=as.Date("2013-12-31"))
> # Extract time index and monthly dates
> datev <- zoo::index(zoots)
> # Coerce index to monthly dates
> monthv <- as.yearmon(datev)
> # tick locations at beginning of month
> tickv <- datev[match(unique(monthv), monthv)]
> # tickv <- as.Date(tapply(X=datev, INDEX=monthv, FUN=min))
> # first plot zoo without "x" axis
> plot(zoots, xaxt="n", xlab=NA, ylab=NA, main="MSFT stock prices")
> # Add "x" axis with monthly ticks
> axis(side=1, at=tickv, labels=format(tickv, "%b-%y"), tcl=-0.7)
> # Add vertical lines
> abline(v=tickv, col="grey", lwd=0.5)
> # Plot zoo using base plotting functions
> plot(as.vector(zoots), xaxt="n",
+  xlab=NA, ylab=NA, t="l", main="MSFT stock prices")
> tickd <- match(tickv, datev)
> # tickd <- seq_along(datev)[datev %in% tickv]
> # Add "x" axis with monthly ticks
> axis(side=1, at=tickd, labels=format(tickv, "%b-%y"), tcl=-0.7)
> abline(v=tickd, col="grey", lwd=0.5)
```




```
> library(microbenchmark)
> summary(microbenchmark(
+ match=datev[match(unique(monthv), monthv)],
+ tapply=as.Date(tapply(X=datev,
+     INDEX=monthv, FUN=min)),
+ times=10)
+   )[, c(1, 4, 5)]
```

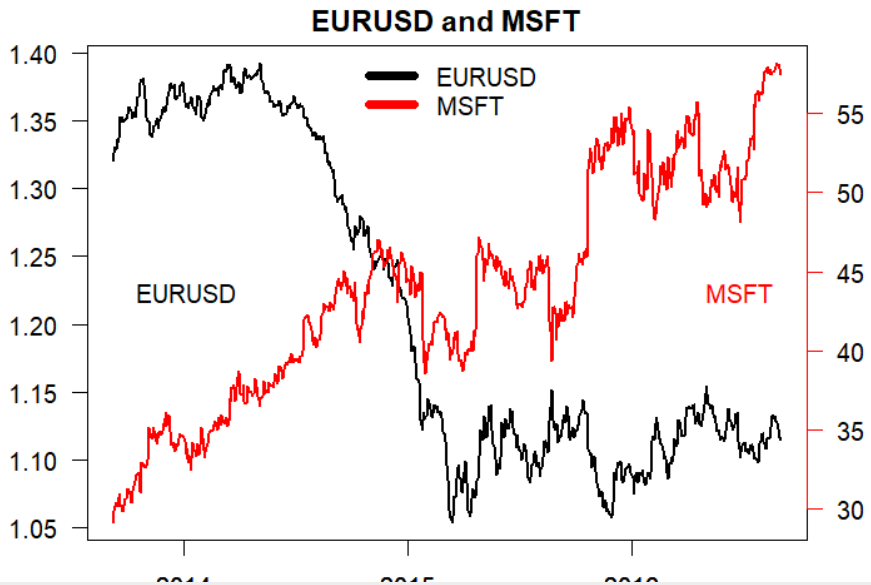
# Plotting Two Time Series With Two "y" Axes

The function `par()` sets graphical parameters used for plotting, and invisibly returns existing parameters as a named list.

The function `axis()` plots an axis on the current plot.

The function `lines()` plots a line on the current plot.

```
> par(las=1)  # Set text printing to horizontal
> ## Plot with two y-axes - plot first time series
> zoo::plot.zoo(zoo_stxeur[, 1], lwd=2, xlab=NA, ylab=NA)
> par(new=TRUE)  # Allow new plot on same chart
> # Plot second time series without y-axis
> zoo::plot.zoo(zoo_stxeur[, 2], xlab=NA, ylab=NA,
+       lwd=2, yaxt="n", col="red")
> # Plot second y-axis on right
> axis(side=4, col="red")
> # Add axis labels
> colnamev <- colnames(zoo_stxeur)
> mtext(colnamev[1], side=2, adj=-0.5)
> mtext(colnamev[2], side=4, adj=1.5, col="red")
> # Add title and legend
> title(main=paste0(colnamev, collapse=" and "),
+ line=0.5)
> legend("top", legend=colnamev,
+   bg="white", lty=1, lwd=6, y.intersp=0.4,
+   col=c("black", "red"), bty="n")
```

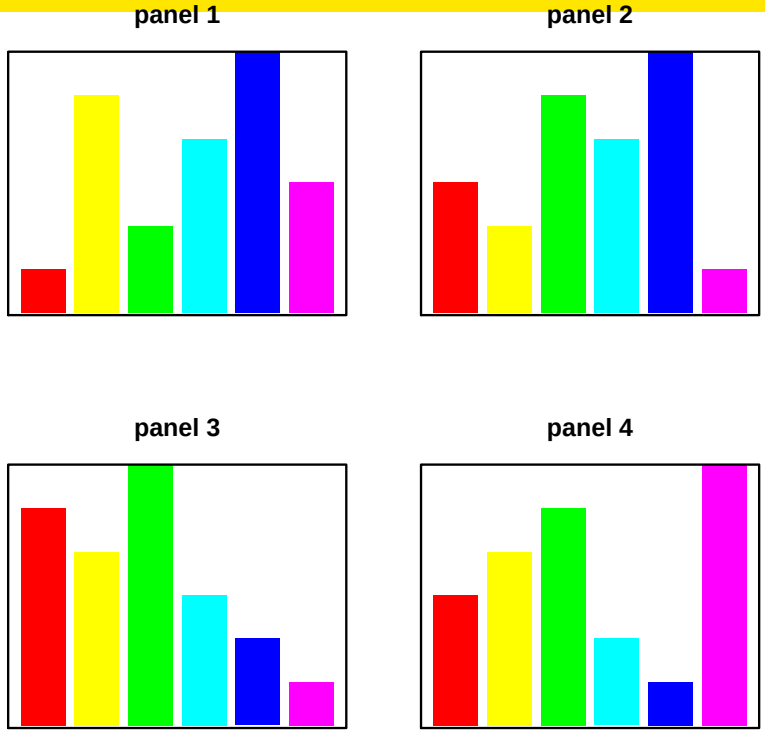



```
> ## Slightly different method using par("usr")
> par(las=1)  # Set text printing to horizontal
> zoo::plot.zoo(zoo_stxeur[, 1], xlab=NA, ylab=NA, lwd=2)
> # Set range of "y" coordinates for second axis
> par(usr=c(par("usr")[1:2], range(zoo_stxeur[,2])))
> lines(zoo_stxeur[, 2], col="red", lwd=2)  # Second plot
> axis(side=4, col="red")  # Second y-axis on right
> # Add axis labels
> mtext(colnamev[1], side=2, adj=-0.5)
> mtext(colnamev[2], side=4, adj=1.5, col="red")
> # Add title and legend
> title(main=paste0(colnamev, collapse=" and "),
+ line=0.5)
> legend("top", legend=colnamev,
+   bg="white", lty=1, lwd=6, y.intersp=0.4,
+   col=c("black", "red"), bty="n")
```

# Graphical Parameters

The function `par()` sets graphical parameters used for plotting, and invisibly returns existing parameters as a named list.

```
> graph_params <- par()  # get existing parameters
> par("mar")  # get plot margins
> par(mar=c(2, 1, 2, 1))  # Set plot margins
> par(oma=c(1, 1, 1, 1))  # Set outer margins
> par(mgp=c(2, 1, 0))  # Set title and label margins
> par(cex.lab=0.8,  # Set font scales
+     cex.axis=0.8, cex.main=0.8, cex.sub=0.5)
> par(las=1)  # Set axis labels to horizontal
> par(ask=TRUE)  # Pause, ask before plotting
> par(mfrow=c(2, 2))  # Plot on 2x2 grid by rows
> for (i in 1:4) {  # Plot 4 panels
+   barplot(sample(1:6), main=paste("panel", i),
+     col=rainbow(6), border=NA, axes=FALSE)
+   box()
+ }  # end for
> par(ask=FALSE)  # Restore automatic plotting
> par(new=TRUE)  # Allow new plot on same chart
> par(graph_params)  # Restore original parameters
```




- `cex` set graphic scales,
- `mar` & `oma` set plot margins,
- `mgp` set title and label margins,
- `las` set orientation of axis labels,
- `mfcol` & `mfrow` set number of plot panels,

# *Normal (Gaussian)* Probability Distribution

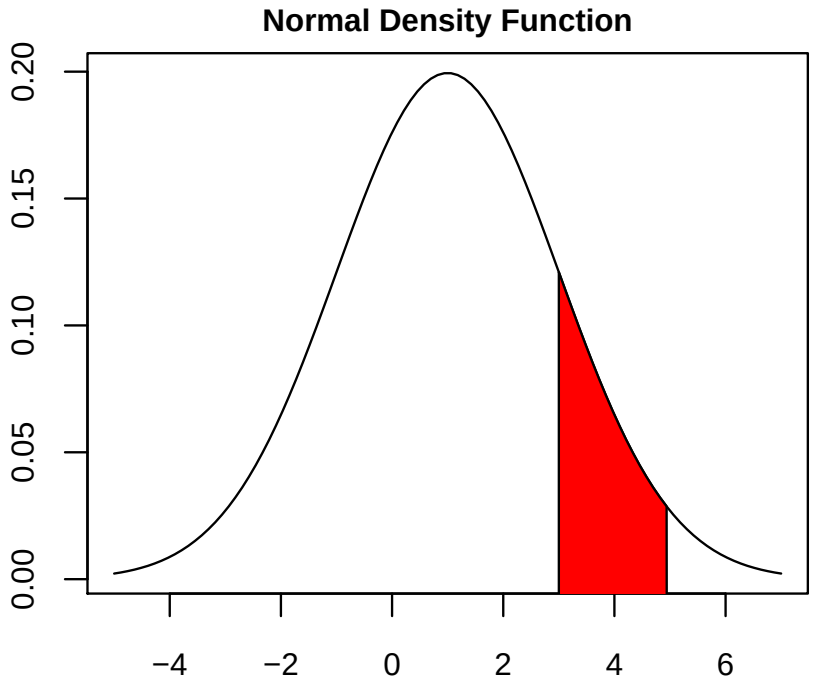
The *Normal (Gaussian)* probability density function is given by:

$$\phi(x, \mu, \sigma) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

The *Standard Normal* distribution $\phi(0, 1)$ is a special case of the *Normal* $\phi(\mu, \sigma)$ with $\mu = 0$ and $\sigma = 1$.

The function `dnorm()` calculates the *Normal* probability density.

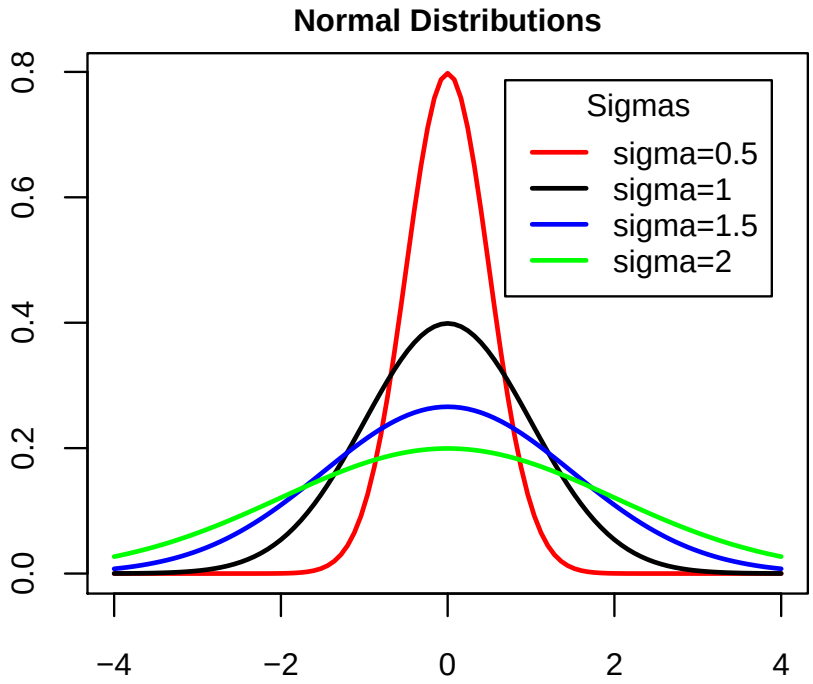
```
> xvar <- seq(-5, 7, length=100)
> yvar <- dnorm(xvar, mean=1.0, sd=2.0)
> plot(xvar, yvar, type="l", lty="solid", xlab="", ylab="")
> title(main="Normal Density Function", line=0.5)
> startp <- 3; endd <- 5  # Set lower and upper bounds
> # Set polygon base
> subv <- ((xvar >= startp) & (xvar <= endd))
> polygon(c(startp, xvar[subv], endd),  # Draw polygon
+   c(-1, yvar[subv], -1), col="red")
```



# *Normal (Gaussian)* Probability Distributions

Plots of several *Normal* distributions with different values of $\sigma$, using the function `curve()` for plotting functions given by their name.

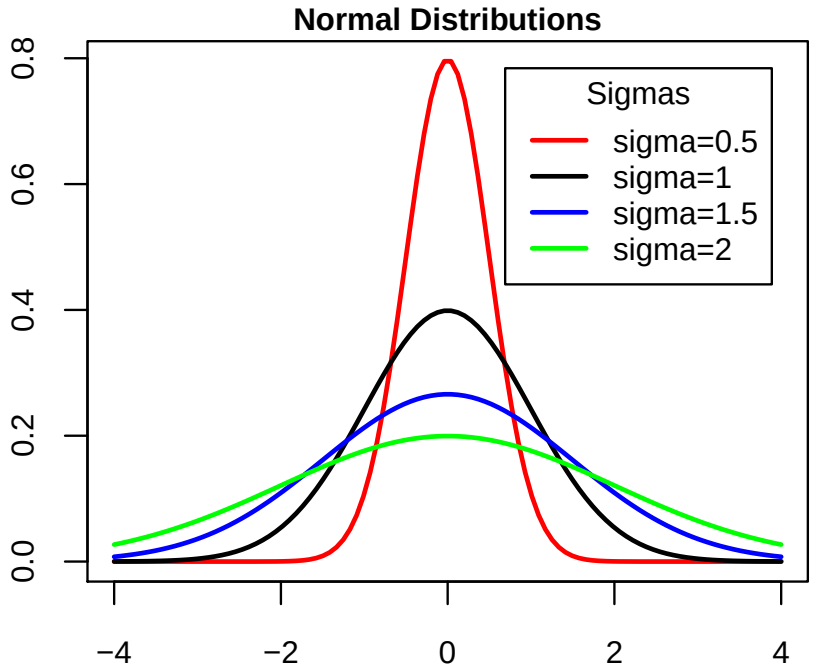
```
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colorv <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> for (indeks in 1:4) {  # Plot four curves
+   curve(expr=dnorm(x, sd=sigmavs[indeks]),
+   xlim=c(-4, 4), xlab="", ylab="", lwd=2,
+   col=colorv[indeks], add=as.logical(indeks-1))
+ }  # end for
> # Add title
> title(main="Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, title="Sigmas", y.intersp=0.4,
+  labelv, cex=0.8, lwd=2, lty=1, bty="n", col=colorv)
```



# *Normal* Probability Distributions Plotted as Lines

Plots of several *Normal* distributions with different values of $\sigma$.

```
> xvar <- seq(-4, 4, length=100)
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colorv <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> # Plot the first chart
> plot(xvar, dnorm(xvar, sd=sigmavs[1]),
+      type="n", xlab="", ylab="", main="Normal Distributions")
> # Add lines to plot
> for (indeks in 1:4) {
+   lines(xvar, dnorm(xvar, sd=sigmavs[indeks]),
+   lwd=2, col=colorv[indeks])
+ }  # end for
> # Add legend
> legend("topright", inset=0.05, title="Sigmas", y.intersp=0.4,
+  labelv, cex=0.8, lwd=2, lty=1, bty="n", col=colorv)
```



# The *Log-normal* Probability Distribution

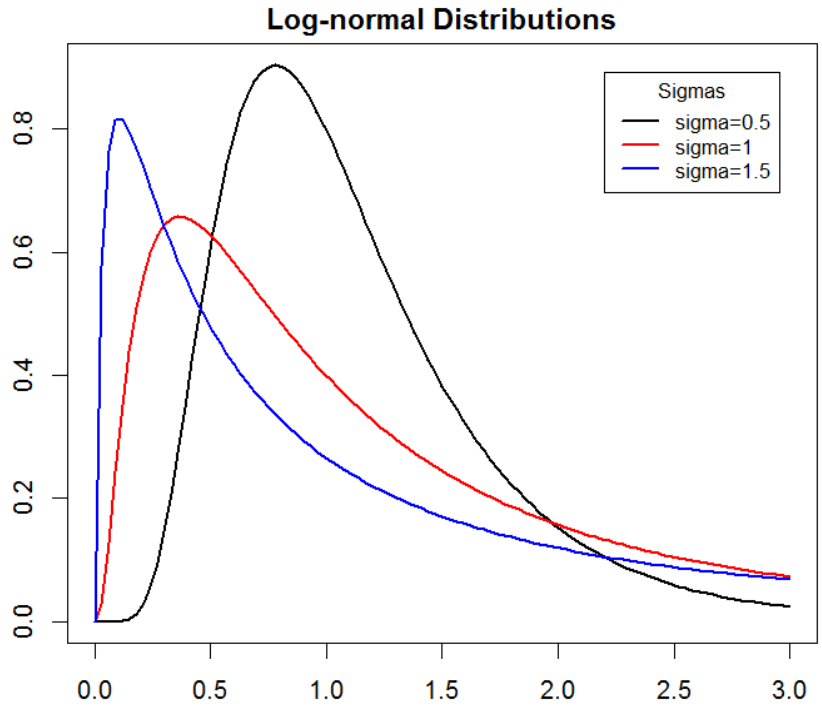
If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

$$\log \phi(y, \mu, \sigma) = \frac{\exp(-(\log y - \mu)^2/2\sigma^2)}{y\sigma\sqrt{2\pi}}$$

With mean equal to: $\bar{y} = \mathbb{E}[y] = e^{(\mu+\sigma^2/2)}$, and median equal to: $\tilde{y} = e^{\mu}$

With variance equal to: $\sigma_y^2 = (e^{\sigma^2} - 1)e^{(2\mu+\sigma^2)}$, and skewness (third moment) equal to:
$\varsigma = \mathbb{E}[(y - \mathbb{E}[y])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$

```
> # Standard deviations of log-normal distribution
> sigmavs <- c(0.5, 1, 1.5)
> # Create plot colors
> colorv <- c("black", "red", "blue")
> # Plot all curves
> for (indeks in 1:NROW(sigmavs)) {
+   curve(expr=dlnorm(x, sdlog=sigmavs[indeks]),
+   type="l", xlim=c(0, 3), lwd=2,
+   xlab="", ylab="", col=colorv[indeks],
+   add=as.logical(indeks-1))
+ }  # end for
```




```
> # Add title and legend
> title(main="Log-normal Distributions", line=0.5)
> legend("topright", inset=0.05, title="Sigmas",
+  paste("sigma", sigmavs, sep="="), y.intersp=0.4,
+  cex=0.8, lwd=2, lty=rep(1, NROW(sigmavs)), col=colorv)
```

# *Chi-squared* Distribution

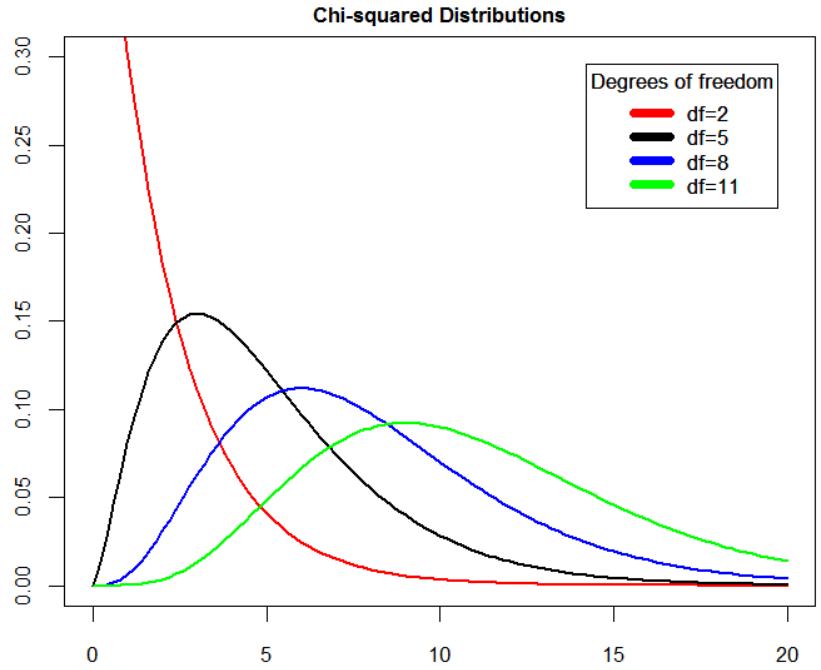
Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with $k$ degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1}\, e^{-x/2}}{2^{k/2}\, \Gamma(k/2)}$$

The *Chi-squared* distribution with $k$ degrees of freedom has mean equal to $k$ and variance equal to $2k$.

```
> # Degrees of freedom
> degf <- c(2, 5, 8, 11)
> # Plot four curves in loop
> colorv <- c("red", "black", "blue", "green")
> for (indeks in 1:4) {
+   curve(expr=dchisq(x, df=degf[indeks]),
+   xlim=c(0, 20), ylim=c(0, 0.3),
+   xlab="", ylab="", col=colorv[indeks],
+   lwd=2, add=as.logical(indeks-1))
+ }  # end for
```




```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

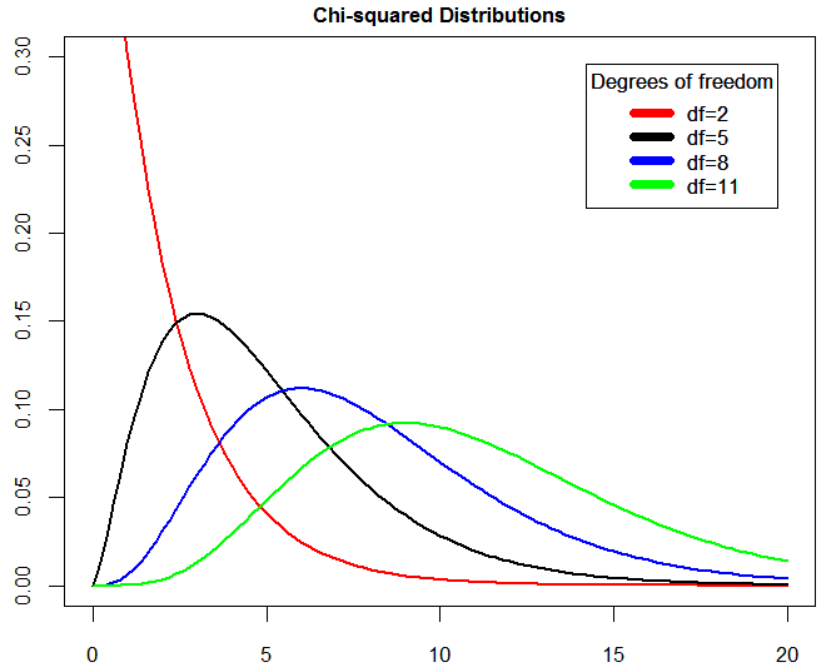
# *Chi-squared* Distribution Plotted as Line

Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with k degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1}\, e^{-x/2}}{2^{k/2}\, \Gamma(k/2)}$$

```
> degf <- c(2, 5, 8, 11)  # df values
> # Create plot colors
> colorv <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("df", degf, sep="=")
> # Plot an empty chart
> xvar <- seq(0, 20, length=100)
> plot(xvar, dchisq(xvar, df=degf[1]),
+      type="n", xlab="", ylab="", ylim=c(0, 0.3))
> # Add lines to plot
> for (indeks in 1:4) {
+   lines(xvar, dchisq(xvar, df=degf[indeks]),
+ lwd=2, col=colorv[indeks])
+ }  # end for
```




```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, y.intersp=0.4,
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, bty="n", col=colorv)
```

# Fisher's *F-distribution*

Let $\chi^2_m$ and $\chi^2_n$ be independent random variables following *chi-squared* distributions with $m$ and $n$ degrees of freedom.

Then the random variable:

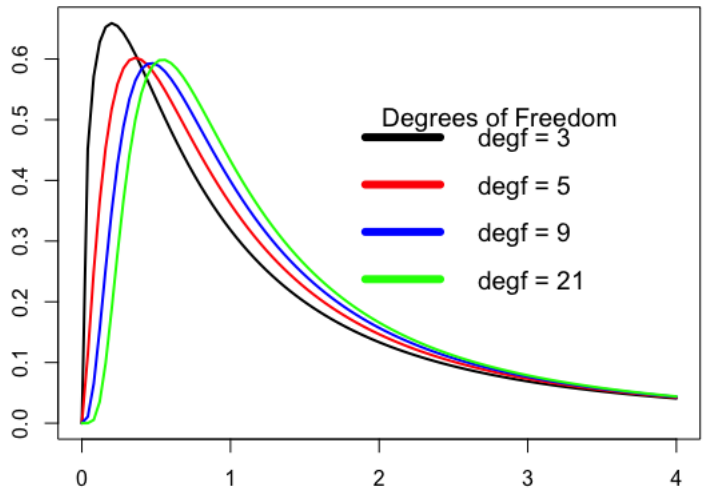
$$F = \frac{\chi^2_m / m}{\chi^2_n / n}$$

Follows the *F-distribution* with $m$ and $n$ degrees of freedom, with the probability density function:

$$f(F) = \frac{\Gamma((m+n)/2) m^{m/2} n^{n/2}}{\Gamma(m/2)\Gamma(n/2)} \frac{F^{m/2-1}}{(n+mF)^{(m+n)/2}}$$

The *F-distribution* depends on the ratio $F$ and also on the degrees of freedom, $m$ and $n$.

The function `df()` calculates the probability density of the *F-distribution*.




```
> # Plot four curves in loop
> degf <- c(3, 5, 9, 21)  # Degrees of freedom
> colorv <- c("black", "red", "blue", "green")
> for (indeks in 1:NROW(degf)) {
+   curve(expr=df(x, df1=degf[indeks], df2=3),
+     xlim=c(0, 4), xlab="", ylab="", lwd=2,
+     col=colorv[indeks], add=as.logical(indeks-1))
+ }  # end for
```

```
> # Add title
> title(main="F-Distributions", line=0.5)
> # Add legend
> labelv <- paste("degf", degf, sep=" = ")
> legend("topright", title="Degrees of Freedom", inset=0.0, bty="n",
+        y.intersp=0.4, labelv, cex=1.2, lwd=6, lty=1, col=colorv)
```

# Student's *t-distribution*

Let $z_1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu} \sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu - 1} \sum_{i=1}^{\nu} (z_i - \bar{z})^2$

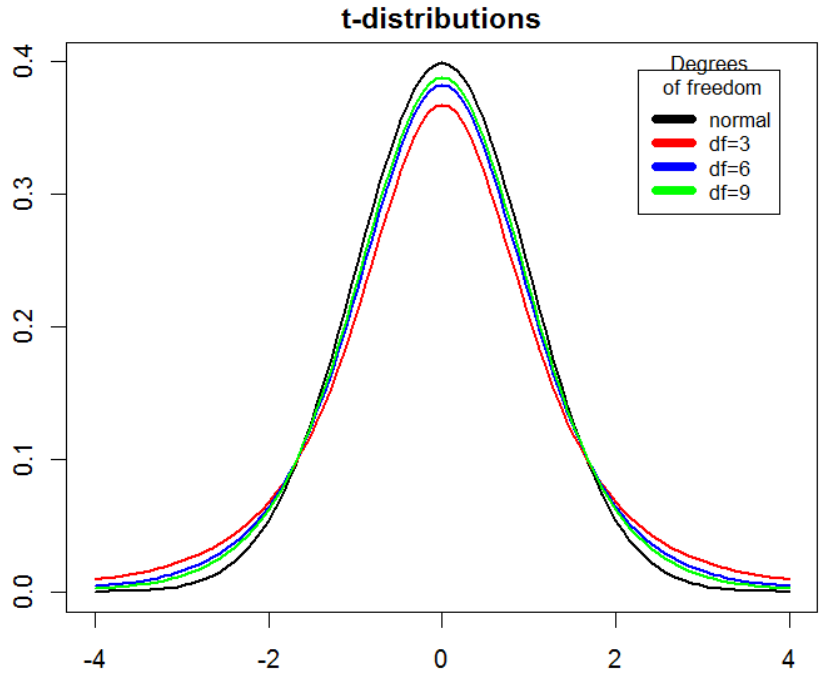
Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma} / \sqrt{\nu}}$$

Follows the *t-distribution* with $\nu$ degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)} \left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

```
> degf <- c(3, 6, 9)  # df values
> colorv <- c("black", "red", "blue", "green")
> labelv <- c("normal", paste("df", degf, sep="="))
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-4, 4), xlab="", ylab="", lwd=2)
> for (indeks in 1:3) {  # Plot three t-distributions
+   curve(expr=dt(x, df=degf[indeks]),
+ lwd=2, col=colorv[indeks+1], add=TRUE)
+ }  # end for
```




```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", labelv,
+        y.intersp=0.4, cex=0.8, lwd=6, lty=1, col=colorv)
```

# Student's *t-distribution* Plotted as Line

Let $z_1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu}\sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu-1}\sum_{i=1}^{\nu}(z_i - \bar{z})^2$

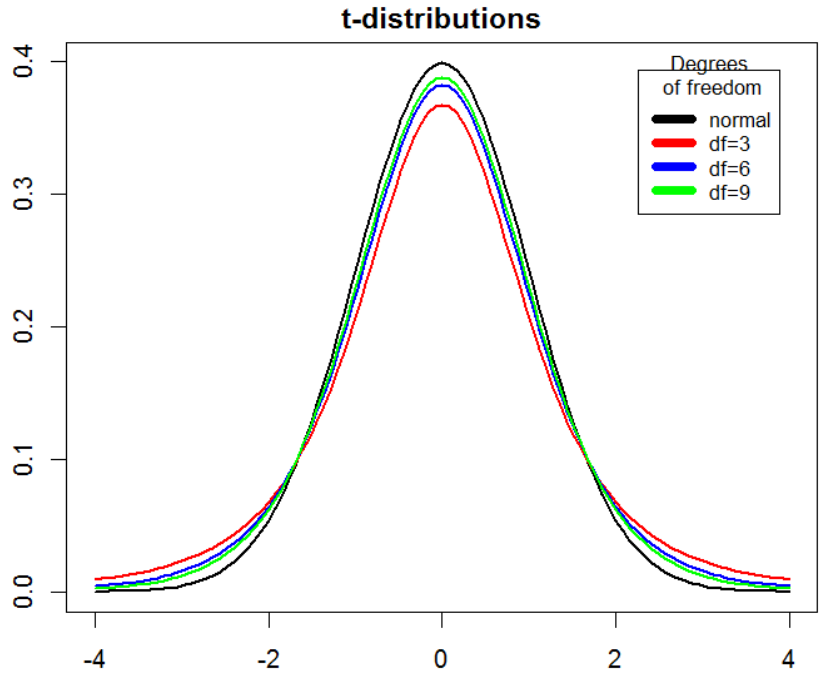
Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma}/\sqrt{\nu}}$$

Follows the *t-distribution* with $\nu$ degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}\left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

```
> xvar <- seq(-4, 4, length=100)
> degf <- c(3, 6, 9)  # df values
> colorv <- c("black", "red", "blue", "green")
> labelv <- c("normal", paste("df", degf, sep="="))
> # Plot chart of normal distribution
> plot(xvar, dnorm(xvar), type="l", lwd=2, xlab="", ylab="")
> for (indeks in 1:3) {  # Add lines for t-distributions
+   lines(xvar, dt(xvar, df=degf[indeks]),
+ lwd=2, col=colorv[indeks+1])
+ }  # end for
```




```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", labelv,
+        y.intersp=0.4, cex=0.8, lwd=6, lty=1, col=colorv)
```

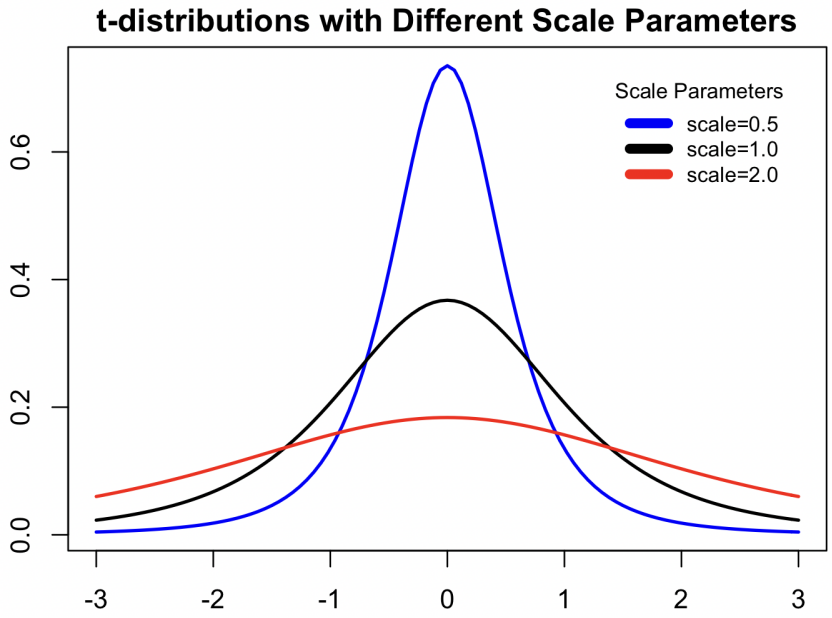
# Non-standard Student's *t-distribution*

The non-standard Student's *t-distribution* has the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\sigma\,\Gamma(\nu/2)}\,(1+(\frac{t-\mu}{\sigma})^2/\nu)^{-(\nu+1)/2}$$

It has non-zero mean equal to the location parameter $\mu$, and a standard deviation proportional to the scale parameter $\sigma$.

```
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # x11(width=6, height=5)
> # Define density of non-standard t-distribution
> tdistr <- function(x, dfree, loc=0, scalev=1) {
+   dt((x-loc)/scalev, df=dfree)/scalev
+ }  # end tdistr
> # Or
> tdistr <- function(x, dfree, loc=0, scalev=1) {
+   gamma((dfree+1)/2)/(sqrt(pi*dfree)*gamma(dfree/2)*scalev)*
+     (1+((x-loc)/scalev)^2/dfree)^(-(dfree+1)/2)
+ }  # end tdistr
> # Calculate vector of scale values
> scalev <- c(0.5, 1.0, 2.0)
> colorv <- c("blue", "black", "red")
> labelv <- paste("scale", format(scalev, digits=2), sep="=")
> # Plot three t-distributions
> for (indeks in 1:3) {
+   curve(expr=tdistr(x, dfree=3, scalev=scalev[indeks]), xlim=c(-3, 3),
+ xlab="", ylab="", lwd=2, col=colorv[indeks], add=(indeks>1))
+ }  # end for
```




```
> # Add title
> title(main="t-distributions with Different Scale Parameters", line
> # Add legend
> legend("topright", inset=0.05, bty="n", title="Scale Parameters",
+        y.intersp=0.4, cex=0.8, lwd=6, lty=1, col=colorv)
```

# *Cauchy* Distribution

The *Cauchy* distribution is Student's *t-distribution* with one degree of freedom $\nu = 1$, with the probability density function:

$$f(x) = \frac{1}{\pi\sigma} \frac{1}{((x-\mu)/\sigma)^2 + 1}$$

Where $\mu$ is the location parameter (equal to the mean) and $\sigma$ is the scale parameter.

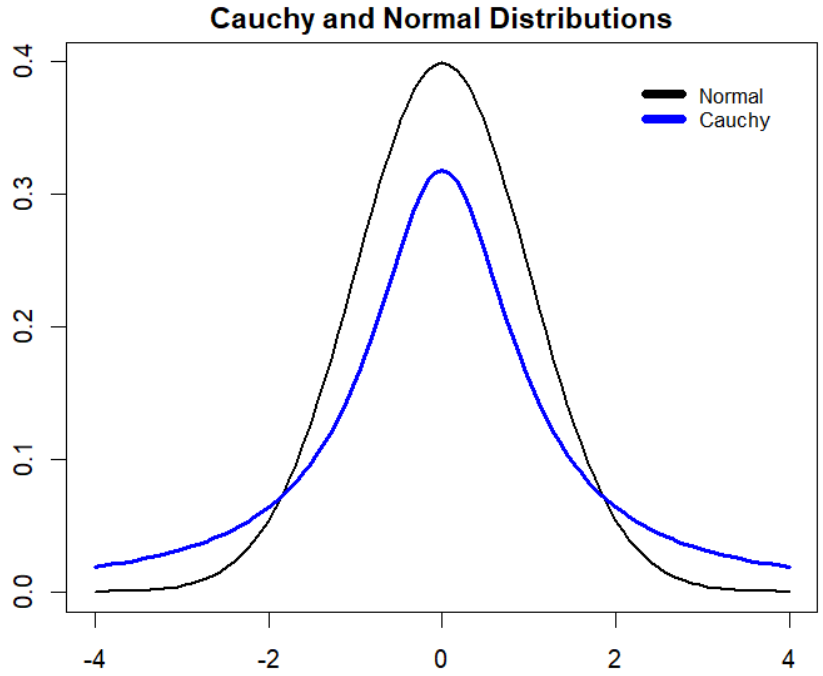
Since the *Cauchy* distribution has an infinite standard deviation, its measure of dispersion is the *interquartile range* (IQR), which is equal to $\sigma$.

The *interquartile range* is a *robust* measure of dispersion (scale), defined as the difference between the `75th` minus the `25th` percentiles.

The function `dcauchy()` calculates the *Cauchy* probability density.

The probability density of the *Cauchy* distribution decreases as the second power for large values of $x$:

$$f(x) \propto 1/x^2$$




```
> # Plot the Normal and Cauchy probability distributions
> curve(expr=dnorm, xlim=c(-4, 4), xlab="", ylab="", lwd=2)
> curve(expr=dcauchy, lwd=3, col="blue", add=TRUE)
> # Add title
> title(main="Cauchy and Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        y.intersp=0.4, title=NULL,leg=c("Normal", "Cauchy"),
+        cex=0.8, lwd=6, lty=1, col=c("black", "blue"))
```

# *Pareto* Distribution and *Zipf's* Law

The probability density of Student's *t-distribution* decreases as a power for large values of $x$:

$$f(x) \propto |x|^{-(\nu+1)}$$

The probability density of the *Pareto* distribution decreases as a power of the random variable $x$:

$$f(x) = \alpha x^{-(\alpha+1)}$$

For $x > 1$ and decay parameter $\alpha > 1$.

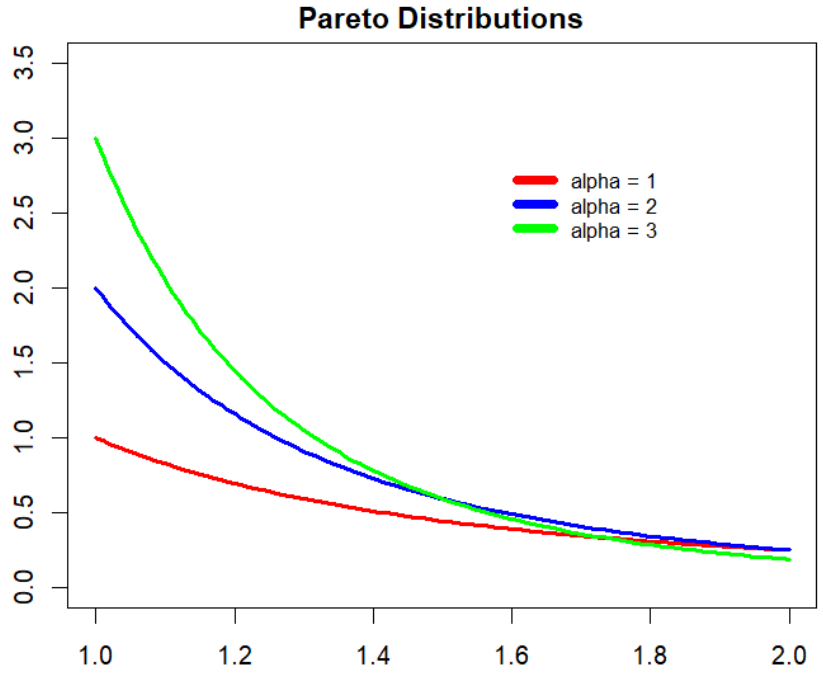
The mean $\mu$ and variance $\sigma^2$ of the *Pareto* distribution are equal to:

$$\mu = \frac{\alpha}{\alpha - 1} \qquad \sigma^2 = \frac{\alpha}{(\alpha - 1)^2(\alpha - 2)}$$

*Zipf's law* is analogous to the *Pareto* distribution, and applies to discrete variables.

*Zipf's law* states that the frequency $f$ of a given value is inversely proportional to its rank $n$ in the frequency table: $f(n) \propto n^{-s}$.

For example, *Zipf's law* applies to the frequency of words in a natural language.




```
> # Define Pareto function
> paretofun <- function(x, alpha) alpha*x^(-alpha-1)
> colorv <- c("red", "blue", "green")
> alphas <- c(1.0, 2.0, 3.0)
> for (indeks in 1:3) {  # Plot three curves
+   curve(expr=paretofun(x, alphas[indeks]),
+   xlim=c(1, 2), ylim=c(0.0, 3.5), xlab="", ylab="",
+   lwd=3, col=colorv[indeks], add=as.logical(indeks-1))
+ }  # end for
> # Add title and legend
> title(main="Pareto Distributions", line=0.5)
> labelv <- paste("alpha", 1:3, sep=" = ")
> legend("topright", inset=0.2, bty="n", y.intersp=0.4,
+  title=NULL, labelv, cex=0.8, lwd=6, lty=1, col=colorv)
```

# *Poisson* Probability Distribution

The *Poisson* distribution gives the probability of the number of events observed in an interval of space or time.

The *Poisson* probability function is given by:

$$f(n;\lambda) = \frac{\lambda^n \cdot e^{-\lambda}}{n!}$$

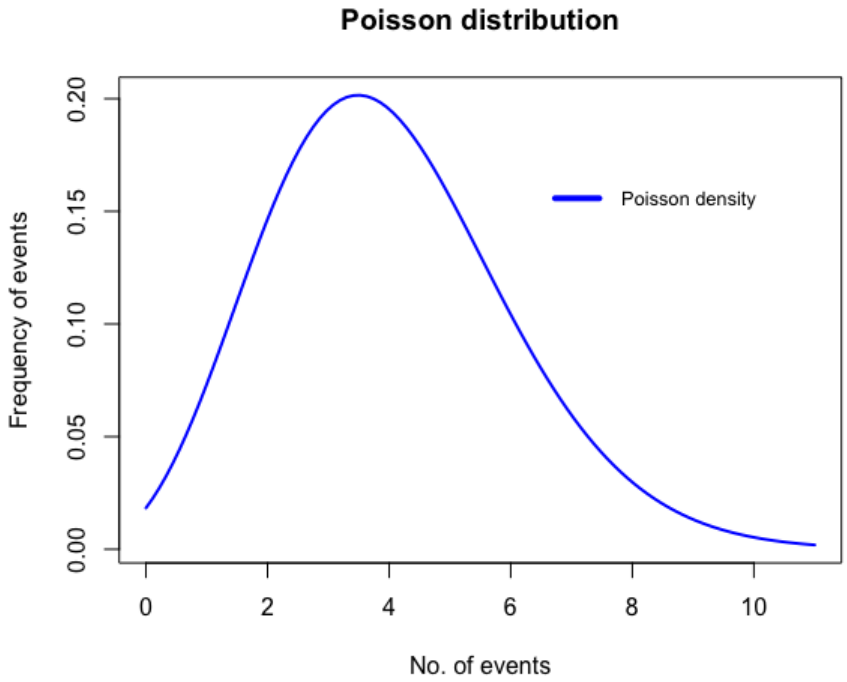
The *Poisson* random variable `n` is the number of events observed in the interval.

The parameter $\lambda$ is the average number of events that are observed in the interval.

An example of a *Poisson* distribution is the number of mail items received each day.

The function `dpois()` returns the probability density of the *Poisson* distribution.

The function `rpois()` returns random numbers following the *Poisson* distribution.




```
> # Poisson frequency
> eventv <- 0:11  # Poisson events
> poissonf <- dpois(eventv, lambda=4)
> names(poissonf) <- as.character(eventv)
> # Poisson function
> poissonfun <- function(x, lambdaf) {exp(-lambdaf)*lambdaf^x/factorial(x)}
> curve(expr=poissonfun(x, lambda=4), xlim=c(0, 11), main="Poisson distribution",
+ xlab="No. of events", ylab="Frequency of events", lwd=2, col="blue")
> legend(x="topright", legend="Poisson density", title="", bty="n",
+  inset=0.05, cex=0.8, bg="white", lwd=6, lty=1, col="blue")
```

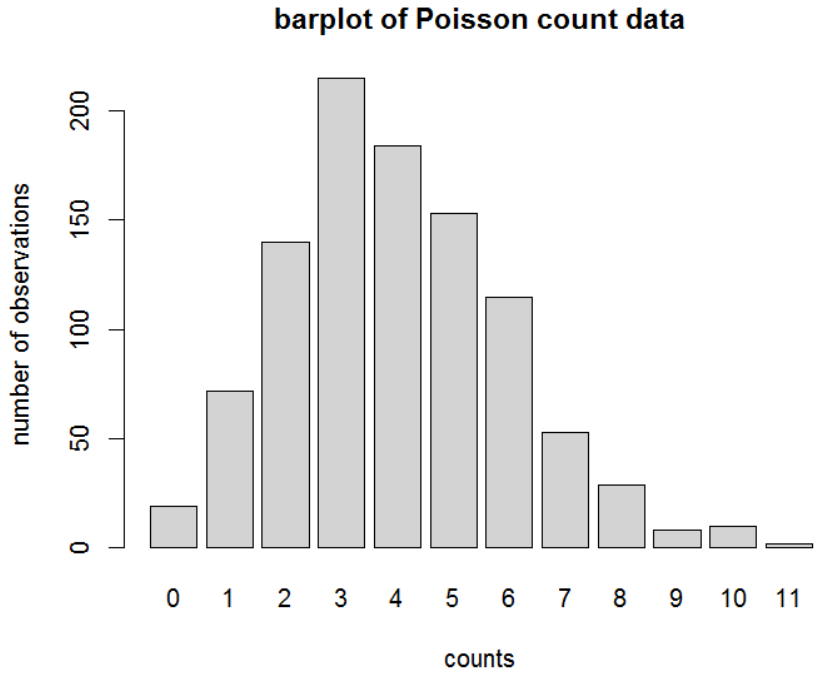
# Plotting Bar Charts of Table Data

The function `barplot()` plots a bar chart for a table of data.

The function `rpois()` produces random numbers from the *Poisson* distribution.

The function `table()` calculates the frequency distribution of categorical data.

```
> # Simulate Poisson variables
> poissonv <- rpois(1000, lambda=4)
> head(poissonv)
[1] 5 5 2 4 4 5
> # Calculate contingency table
> poissonf <- table(poissonv)
> poissonf
poissonv
  0   1   2   3   4   5   6   7   8   9  10  11
 20  74 136 206 202 156 101  58  26  17   3   1
```




```
> # Create barplot of table data
> barplot(poissonf, col="lightgrey",
+   xlab="counts", ylab="number of observations",
+   main="Barplot of Poisson Count Data")
```

# Plotting Histograms of Frequency Data

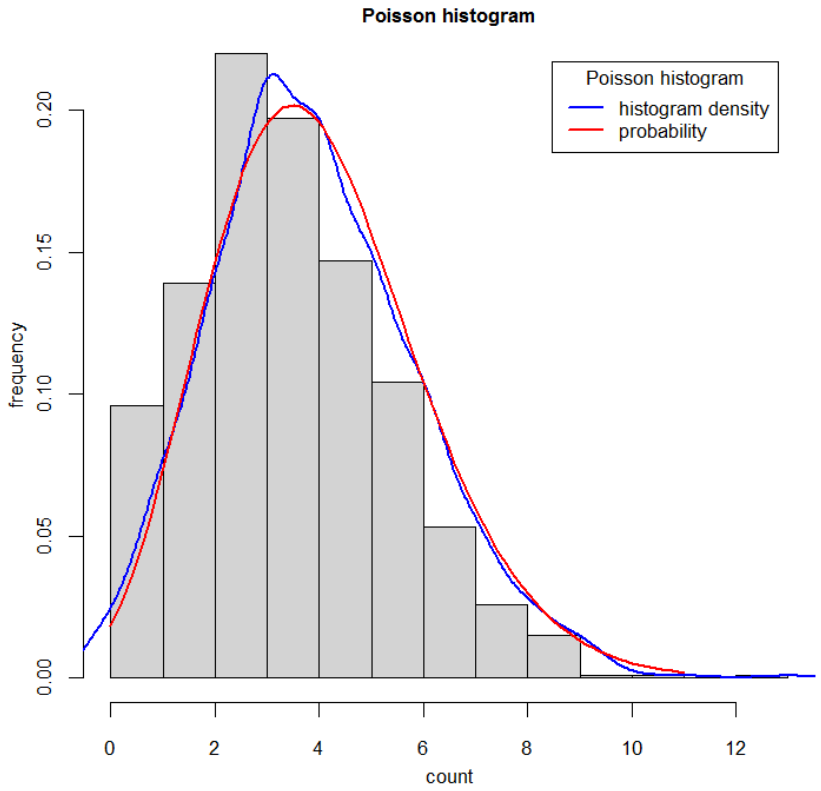
The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

If the argument `freq` is `TRUE` then the frequencies (counts) are plotted, and if it's `FALSE` then the probability density is plotted (with total area equal to 1).

The function `density()` calculates a kernel estimate of the probability density for a sample of data.

The function `lines()` draws a line through specified points.

```
> # Create histogram of Poisson variables
> histp <- hist(poissonv, col="lightgrey", xlab="count",
+       ylab="frequency", freq=FALSE, main="Poisson histogram")
> lines(density(poissonv, adjust=1.5), lwd=2, col="blue")
> # Poisson probability distribution function
> poissonfun <- function(x, lambdaf)
+   {exp(-lambdaf)*lambdaf^x/factorial(x)}
> curve(expr=poissonfun(x, lambda=4), xlim=c(0, 11), add=TRUE, lwd=2
> # Add legend
> legend("topright", inset=0.01, title="Poisson histogram",
+  c("histogram density", "probability"), cex=1.1, lwd=6,
+  y.intersp=0.4, lty=1, bty="n", col=c("blue", "red"))
> # total area under histogram
> diff(histp$breaks) %*% histp$density
```



# Plotting Boxplots of Distributions of Values

Box-and-whisker plots (*boxplots*) are graphical representations of a distribution of values.

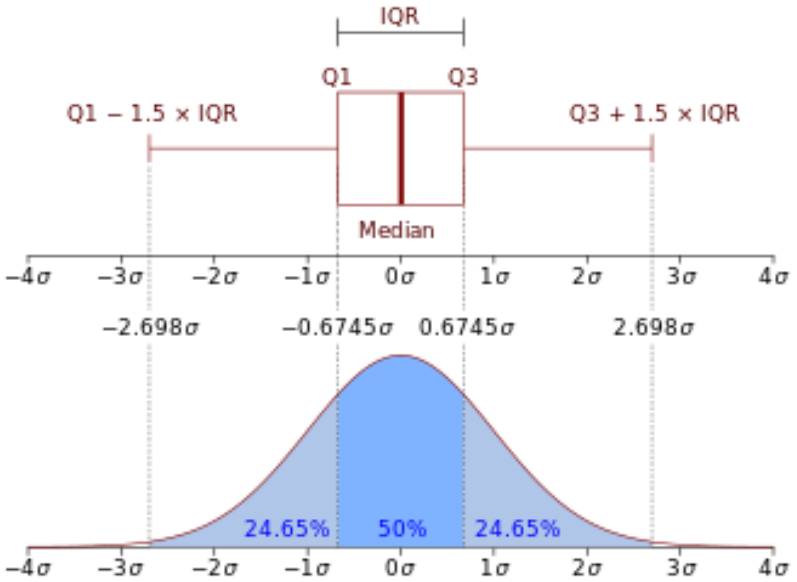
The bottom and top box edges (*hinges*) are equal to the first and third quartiles, and the *box* width is equal to the interquartile range (*IQR*).

The nominal range is equal to 1.5 times the *IQR* above and below the box *hinges*.

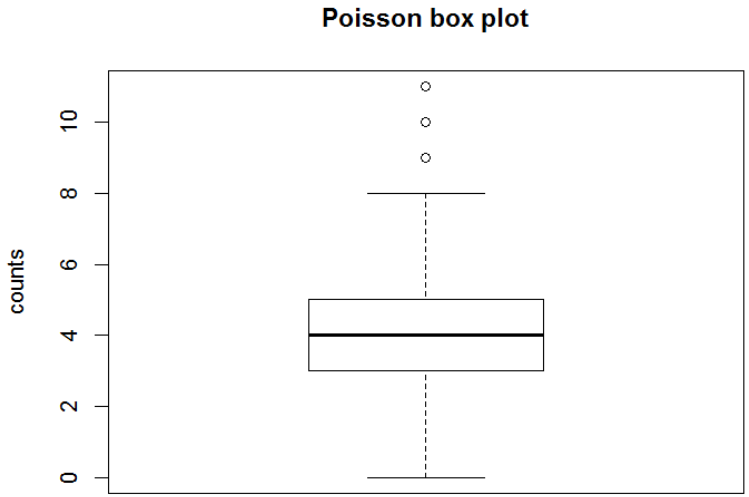
The *whiskers* are dashed vertical lines representing values beyond the first and third quartiles, but within the nominal range.

The *whiskers* end at the last values within the nominal range, while the open circles represent outlier values beyond the nominal range.

The function `boxplot()` has two `methods`: one for `vectors` and `data frames`, and another for `formula` objects (for categorical variables).




```
> # boxplot of Poisson count data
> boxplot(x=poissonv, ylab="counts",
+   main="Poisson box plot")
> # boxplot method for formula
> boxplot(formula=mpg ~ cyl, data=mtcars,
+   main="Mileage by number of cylinders",
+   xlab="Cylinders", ylab="Miles per gallon")
```



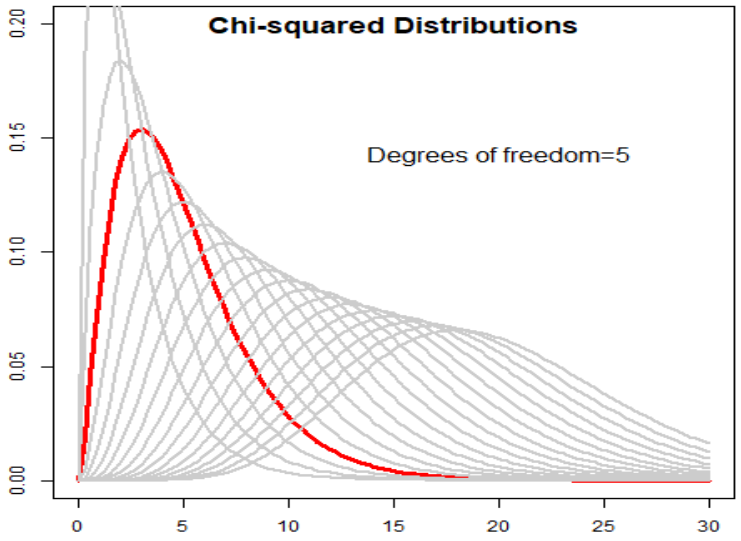
# Plotting Using Expression Objects

It's sometimes convenient to create an *expression* object containing plotting commands, to be able to later create plots using it.

The function `quote()` produces an *expression* object without evaluating it.

The function `eval()` evaluates an *expression* in a specified *environment*.

```
> # Create a plotting expression
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   indeks <- 4
+   # Plot a curve
+   curve(expr=dchisq(x, df=degf[indeks]),
+ xlim=c(0, 30), ylim=c(0, 0.2),
+ xlab="", ylab="", lwd=3, col="red")
+   # Add grey lines to plot
+   for (it in rangev[-indeks]) {
+     curve(expr=dchisq(x, df=degf[it]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+   }  # end for
+   # Add title
+   title(main="Chi-squared Distributions", line=-1.5, cex.main=1.5)
+   # Add legend
+   text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+ })  # end quote
```




```
> # View the plotting expression
> expv
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
```

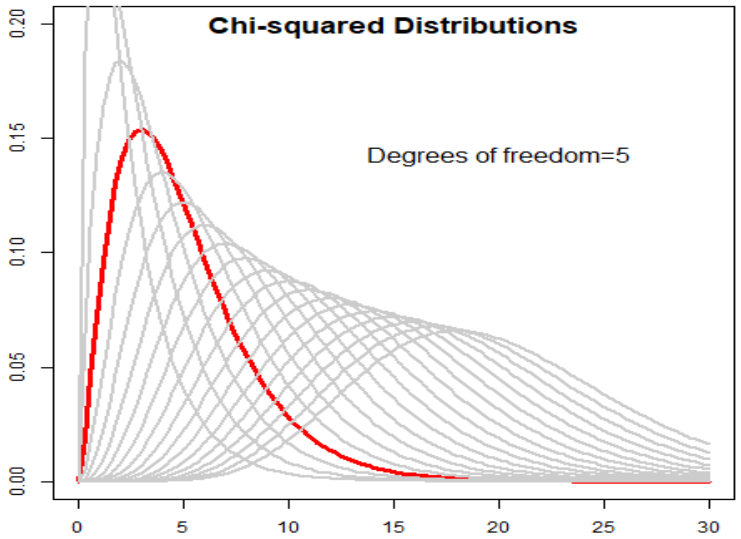
# Animated Plots Using Package *animation*

The package *animation* allows creating animated plots in the form of *gif* and *html* documents.

The function `saveGIF()` produces a *gif* image with an animated plot.

The function `saveHTML()` produces an *html* document with an animated plot.

```
> library(animation)
> # Create an expression for creating multiple plots
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   # Set image refesh interval
+   animation::ani.options(interval=0.5)
+   # Create multiple plots with curves
+   for (indeks in rangev) {
+     curve(expr=dchisq(x, df=degf[indeks]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=3, col="red")
+     # Add grey lines to plot
+     for (it in rangev[-indeks]) {
+       curve(expr=dchisq(x, df=degf[it]),
+     xlim=c(0, 30), ylim=c(0, 0.2),
+     xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+     }  # end for
+     # Add title
+     title(main="Chi-squared Distributions", line=-1.5, cex.main=
+     # Add legend
+     text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+   }  # end for
+ })  # end quote
```




```
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
> # Create gif with animated plot
> animation::saveGIF(expr=eval(expv),
+   movie.name="chi_squared.gif",
+   img.name="chi_squared")
> # Create html with animated plot
> animation::saveHTML(expr=eval(expv),
+   img.name="chi_squared",
+   htmlfile="chi_squared.html",
+   description="Chi-squared Distributions")  # end saveHTML
```

# Dynamic Documents Using *R markdown*

*markdown* is a simple markup language designed for creating documents in different formats, including *pdf* and *html*.

*R Markdown* is a modified version of *markdown*, which allows creating documents containing *math formulas* and `R` code embedded in them.

An *R Markdown* document (with extension `.Rmd`) contains:

- A *YAML* header,
- Text in *R Markdown* code format,
- Math formulas (equations), delimited using either single "$" symbols (for inline formulas), or double "$$" symbols (for display formulas),
- `R` code chunks, delimited using either single "`" backtick symbols (for inline code), or triple "```" backtick symbols (for display code).

The packages *rmarkdown* and *knitr* compile `R` documents into either *pdf*, *html*, or *MS Word* documents.

````
---
title: "My First R Markdown Document"
author: Jerzy Pawlowski
date: ''`r format(Sys.time(), "%m/%d/%Y")`''
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)

# install package quantmod if it can't be loaded success
if (!require("quantmod"))
  install.packages("quantmod")
```

### R Markdown
This is an *R Markdown* document. Markdown is a simple f

One of the advantages of writing documents *R Markdown*

You can read more about publishing documents using *R* h
https://algoquant.github.io/r,/markdown/2016/07/02/Publi

You can read more about using *R* to create *HTML* docum
https://algoquant.github.io/2016/07/05/Interactive-Plots

Clicking the **Knit** button in *RStudio*, compiles the

Example of an *R* code chunk:
```{r cars}
summary(cars)
```

### Plots in *R Markdown* documents

Plots can also be embedded, for example:
```{r pressure, echo=FALSE}
plot(pressure)
````
````

# draft: Online *markdown* Tutorials

## Datacamp Interactive Courses

Datacamp introduction to R: https://www.datacamp.com/courses/introduction-to-r/

Datacamp list of free courses: https://www.datacamp.com/community/open-courses

Datacamp basic statistics in R: https://www.datacamp.com/community/open-courses/basic-statistics

Datacamp computational finance in R:
https://www.datacamp.com/community/open-courses/computational-finance-and-financial-econometrics-with-r

Datacamp machine learning in R:
https://www.datacamp.com/community/open-courses/kaggle-r-tutorial-on-machine-learning

## Try R

Interactive R tutorial, but rather basic: http://tryr.codeschool.com/

# Package *shiny* for Creating Interactive Applications

The package *shiny* creates interactive applications running in R, with their outputs presented as live visualizations.

*Shiny* allows changing the model parameters, recalculating the model, and displaying the resulting outputs as plots and charts.

A *shiny app* is a file with *shiny* commands and R code.

The *shiny* code consists of a *shiny interface* and a *shiny server*.

The *shiny interface* contains widgets for data input and an area for plotting.

The *shiny server* contains the R model code and the plotting code.

The function `shiny::fluidPage()` creates a GUI layout for the user inputs of model parameters and an area for plots and charts.

The function `shiny::renderPlot()` renders a plot from the outputs of a live model.

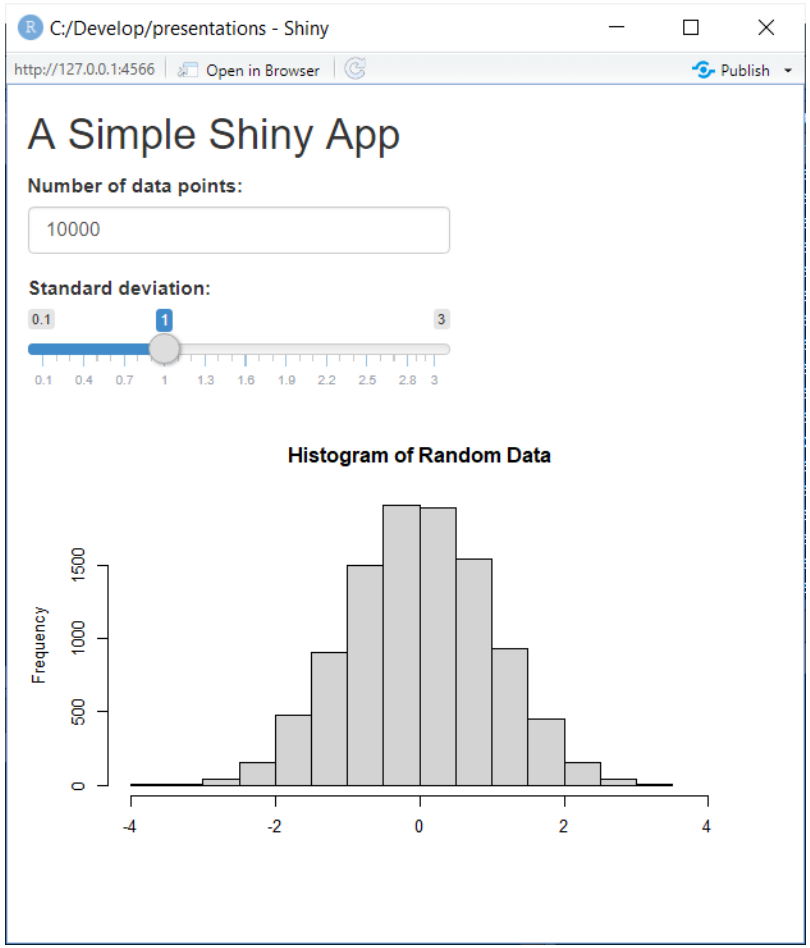
The function `shiny::shinyApp()` creates a `shiny` app from a *shiny interface* and a *shiny server*.

```
> ## App setup code that runs only once at startup.
> ndata <- 1e4
> stdev <- 1.0
>
> ## Define the user interface
> uiface <- shiny::fluidPage(
+   # Create numeric input for the number of data points.
+   numericInput("ndata", "Number of data points:", value=ndata),
+   # Create slider input for the standard deviation parameter.
+   sliderInput("stdev", label="Standard deviation:",
+         min=0.1, max=3.0, value=stdev, step=0.1),
+   # Render plot in a panel.
+   plotOutput("plotobj", height=300, width=500)
+ )  # end user interface
>
> ## Define the server function
> servfun <- function(input, output) {
+   output$plotobj <- shiny::renderPlot({
+     # Simulate the data
+     datav <- rnorm(input$ndata, sd=input$stdev)
+     # Plot the data
+     par(mar=c(2, 4, 4, 0), oma=c(0, 0, 0, 0))
+     hist(datav, xlim=c(-4, 4), main="Histogram of Random Data")
+   })  # end renderPlot
+ }  # end servfun
>
> # Return a Shiny app object
> shiny::shinyApp(ui=uiface, server=servfun)
```

# Running Shiny Apps in *RStudio*

A *shiny app* can be run by pressing the "Run App" button in *RStudio*.

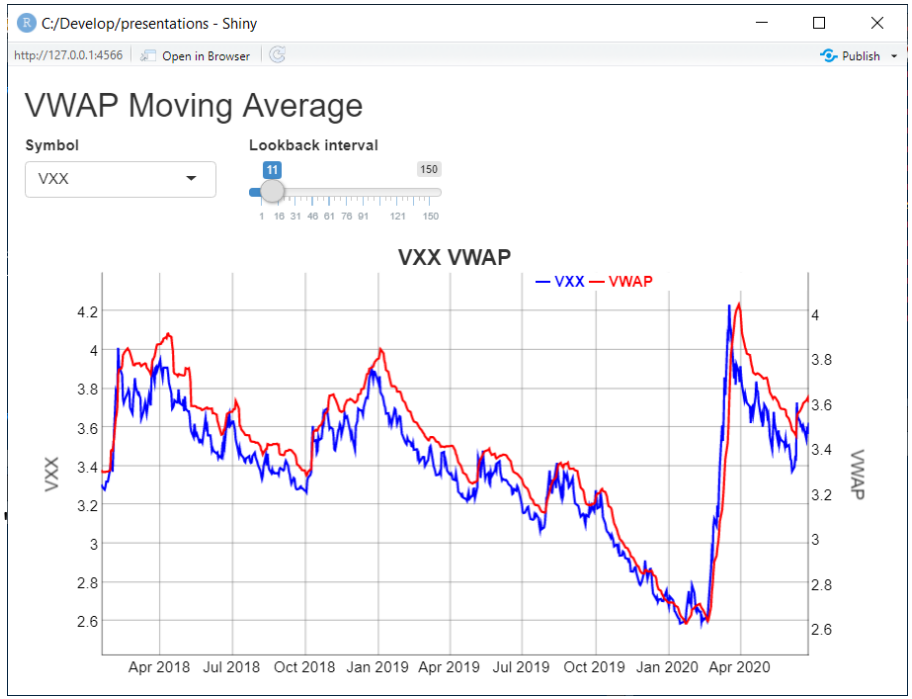
When the *shiny app* is run, the *shiny* commands are translated into *JavaScript* code, which creates a graphical user interface (GUI) with buttons, sliders, and boxes for data input, and also with the output plots and charts.



# Positioning and Sizing Widgets Within the Shiny GUI

The functions `shiny::fluidRow()` and `shiny::column()` allow positioning and sizing widgets within the *shiny* GUI.

```
> ## Create elements of the user interface
> uiface <- shiny::fluidPage(
+   titlePanel("VWAP Moving Average"),
+   # Create single row of widgets with two slider inputs
+   fluidRow(
+     # Input stock symbol
+     column(width=3, selectInput("symbol", label="Symbol",
+                           choices=symbolv, selected=symbol)),
+     # Input look-back interval
+     column(width=3, sliderInput("lookb", label="Lookback interval"
+                           min=1, max=150, value=11, step=1))
+   ),  # end fluidRow
+   # Create output plot panel
+   mainPanel(dygraphs::dygraphOutput("dyplot"), width=12)
+ )  # end fluidPage interface
```



# Shiny Apps With Reactive Expressions

The package *shiny* allows specifying reactive expressions which are evaluated only when their input data is updated.

Reactive expressions avoid performing unnecessary calculations.

If the reactive expression is invalidated (recalculated), then other expressions that depend on its output are also recalculated.

This way calculations cascade through the expressions that depend on each other.

The function `shiny::reactive()` transforms an expression into a reactive expression.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+   # Get the close and volume data in a reactive environment
+   closep <- shiny::reactive({
+     # Get the data
+     ohlc <- get(input$symbol, data_env)
+     closep <- log(quantmod::Cl(ohlc))
+     volum <- quantmod::Vo(ohlc)
+     # Return the data
+     cbind(closep, volum)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   vwapv <- shiny::reactive({
+     # Get model parameters from input argument
+     lookb <- input$lookb
+     # Calculate the VWAP indicator
+     closep <- closep()[, 1]
+     volum <- closep()[, 2]
+     vwapv <- HighFreq::roll_sum(tseries=closep*volum, lookb=lookb)
+     volumroll <- HighFreq::roll_sum(tseries=volum, lookb=lookb)
+     vwapv <- vwapv/volumroll
+     vwapv[is.na(vwapv)] <- 0
+     # Return the plot data
+     datav <- cbind(closep, vwapv)
+     colnames(datav) <- c(input$symbol, "VWAP")
+     datav
+   })  # end reactive code
+
+   # Return the dygraph plot to output argument
+   output$dyplot <- dygraphs::renderDygraph({
+     colnamev <- colnames(vwapv())
+     dygraphs::dygraph(vwapv(), main=paste(colnamev[1], "VWAP")) %>
+ dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+ dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWidt
+ dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeWi
+   })  # end output plot
+ })  # end server code
```

# Reactive Event Handlers

Event handlers are functions which evaluate expressions when an event occurs (like a button press).

The functions `shiny::observeEvent()` and `shiny::eventReactive()` are event handlers.

The function `shiny::eventReactive()` returns a value, while `shiny::observeEvent()` produces a side-effect, without returning a value.

The function `shiny::reactiveValues()` creates a list for storing reactive values, which can be updated by event handlers.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+
+   # Create an empty list of reactive values.
+   value_s <- reactiveValues()
+
+   # Get input parameters from the user interface.
+   nrows <- reactive({
+     # Add nrows to list of reactive values.
+     value_s*nrows <- input$nrows
+     input$nrows
+   })  # end reactive code
+
+   # Broadcast a message to the console when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     cat("Input button pressed\n")
+   })  # end observeEvent
+
+   # Send the data when the button is pressed.
+   datav <- eventReactive(eventExpr=input$button, valueExpr={
+     # eventReactive() executes on input$button, but not on nrows()
+     cat("Sending", nrows(), "rows of data\n")
+     datav <- head(mtcars, input$nrows)
+     value_s$mpg <- mean(datav$mpg)
+     datav
+   })  # end eventReactive
+   #   datav
+
+   # Draw table of the data when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     datav <- datav()
+     cat("Received", value_s*nrows, "rows of data\n")
+     cat("Average mpg = ", value_s$mpg, "\n")
+     cat("Drawing table\n")
+     output$tablev <- renderTable(datav)
+   })  # end observeEvent
+
+ })  # end server code
>
```

# draft: Interactive Plots Using Markdown and *shiny*

The package *shiny* creates interactive plots that display the outputs of live models running in R.

The function `inputPanel()` creates a panel for user input of model parameters.

The function `renderPlot()` renders a plot from the outputs of a live model running in R.

To create a shiny chart, you can first create an `.Rmd` file, embed the *shiny* code in an R chunk, and then compile the `.Rmd` file into an *html* document, using the *knitr* package.


```
> # R startup chunk
> # ```{r setup, include=FALSE}
> library(shiny)
> library(quantmod)
> interval <- 31
> closep  <- quantmod::Cl(rutils::etfenv$VTI)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> # ```
> ### end R startup chunk
> inputPanel(
+   sliderInput("lambdaf", label="lambdaf:",
+     min=0.01, max=0.2, value=0.1, step=0.01)
+ )  # end inputPanel
```

```
> renderPlot({
+   # Calculate EMA prices
+   lambdaf <- input$lambdaf
+   weightv <- exp(-lambdaf*1:interval)
+   weightv <- weightv/sum(weightv)
+   emacpp <- .Call(stats:::C_cfilter, closep, filter=weightv, sides
+   emacpp[1:(interval-1)] <- emacpp[interval]
+   emacpp <- xts(cbind(closep , emacpp), order.by=zoo::index(closep
+   colnames(emacpp) <- c("VTI", "VTI EMA")
+   # Plot EMA prices
+   chobj <- chart_Series(emacpp, theme=plot_theme, name="EMA prices
+   plot(chobj)
+   legend("top", legend=colnames(emacpp),
+    y.intersp=0.4, inset=0.1, bg="white", lty=1, lwd=2,
+    col=plot_theme$col$line.col, bty="n")
+ })  # end renderPlot
```

# *ggplot2* Package for Plotting

The package *ggplot2* allows plotting using the *grammar of graphics* framework, where the plot is specified by combining different geometric objects (lines, rectangles, etc.).

In the *ggplot2* framework a plot object is created by combining the outputs of several plotting functions using the "+" operator.

The function `ggplot()` defines the plot data and creates a plot template.

The function `aes()` specifies the plot "*aesthetics*", which are mappings between data elements (rows, columns) and plot elements (points, lines).

`geom_point()` adds plot points.

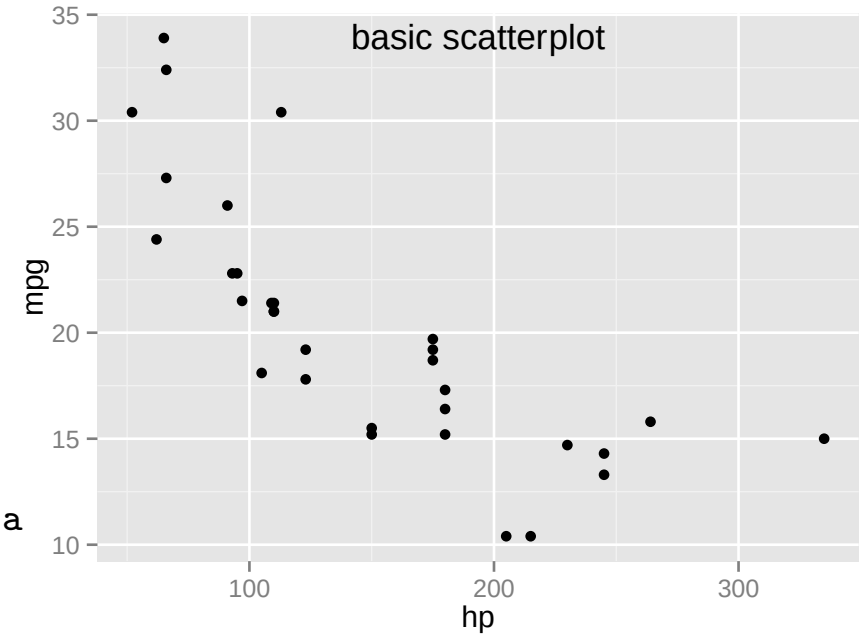
`geom_line()` adds plot lines.

The function `theme()` customizes plot objects.

Simply calling the plot object renders the plot.

*ggplot2* is designed for plotting data formatted as `data frames`.

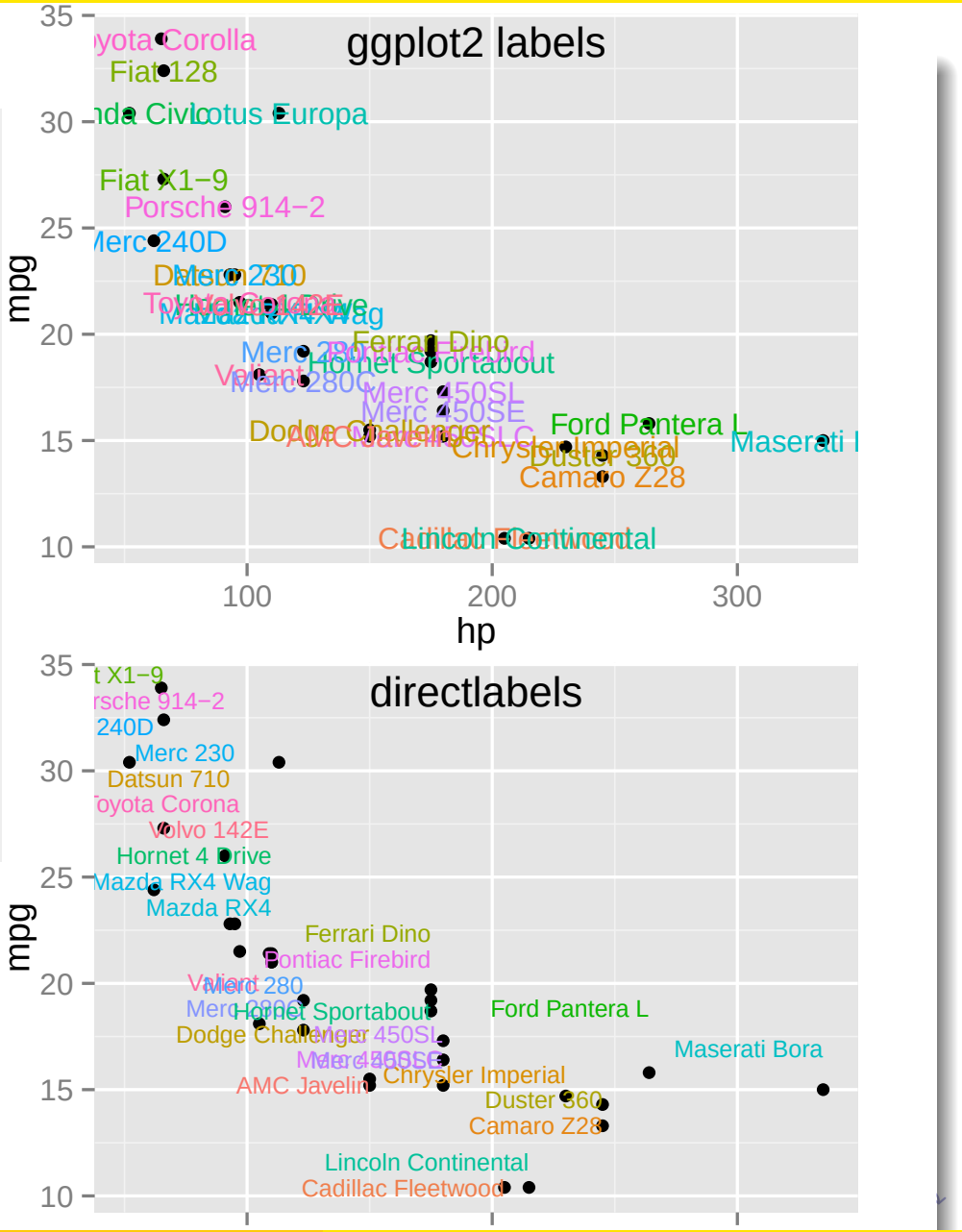
```
> my_ggplot <- ggplot(  # Specify data and aesthetics
+   data=mtcars, mapping=aes(x=hp, y=mpg)) +
+   geom_point() +  # Plot points
+   ggtitle("basic scatter plot") +  # Add title
+   theme(  # Customize plot object
+   plot.title=element_text(vjust=-2.0),
+   plot.background=element_blank()
+   )  # end theme
> my_ggplot  # Render the plot
```



# *ggplot2* Scatter Plots with Labels

The package *directlabels* allows adding flexible lables to *ggplot2* scatter plots.

```
> library(directlabels)  # Load directlabels
> my_ggplot <- ggplot(  # Data and aesthetics
+   data=mtcars, mapping=aes(x=hp, y=mpg)) +
+   geom_point() +  # Plot points
+   theme(  # Customize plot object
+   legend.position="none",
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.0,0.0,-0.5,0.0),"cm"),
+   plot.background=element_blank()
+   ) +
+   scale_colour_discrete(guide="none")  # no label guide
> car_names <- rownames(mtcars)
> gg_labels <- geom_text(aes(  # ggplot2 labels
+   label=car_names, color=car_names, size=5))
> d_labels <- geom_dl(mapping=aes(  # Directlabels
+   label=car_names, color=car_names),
+   method=list("last.bumpup", cex=0.7,
+         hjust=1))
> # Render plots in single column
> grid.arrange(my_ggplot +
+   ggtitle("ggplot2 labels") + gg_labels,
+   my_ggplot + ggtitle("directlabels") +
+     d_labels, ncol=1)  # end grid.arrange
```

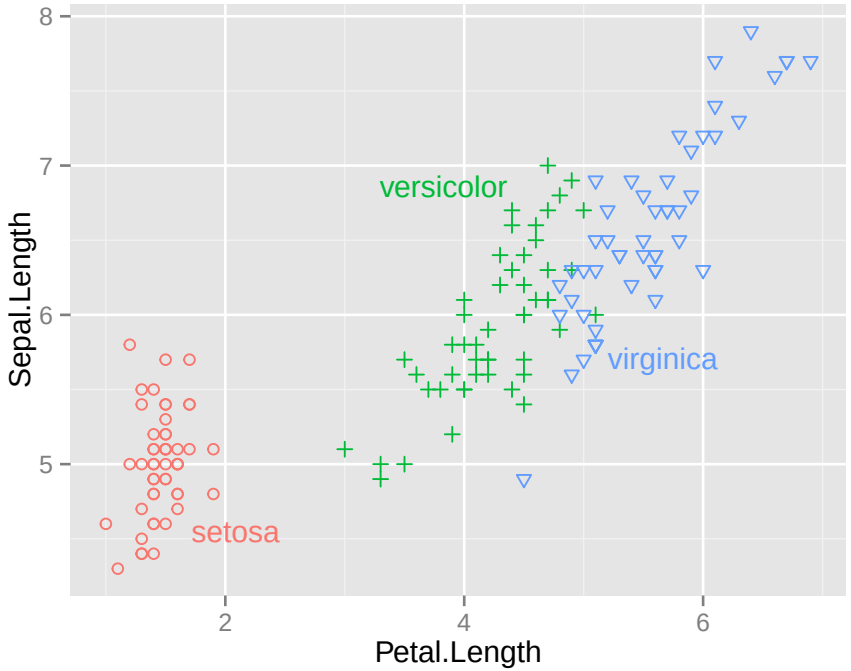


# *ggplot2* Scatter Plots with Group Labels

The *Poisson* distribution gives the probability of the number of events in intervals of space or time.

An example of a *Poisson* distribution is the number of mail items received each day.

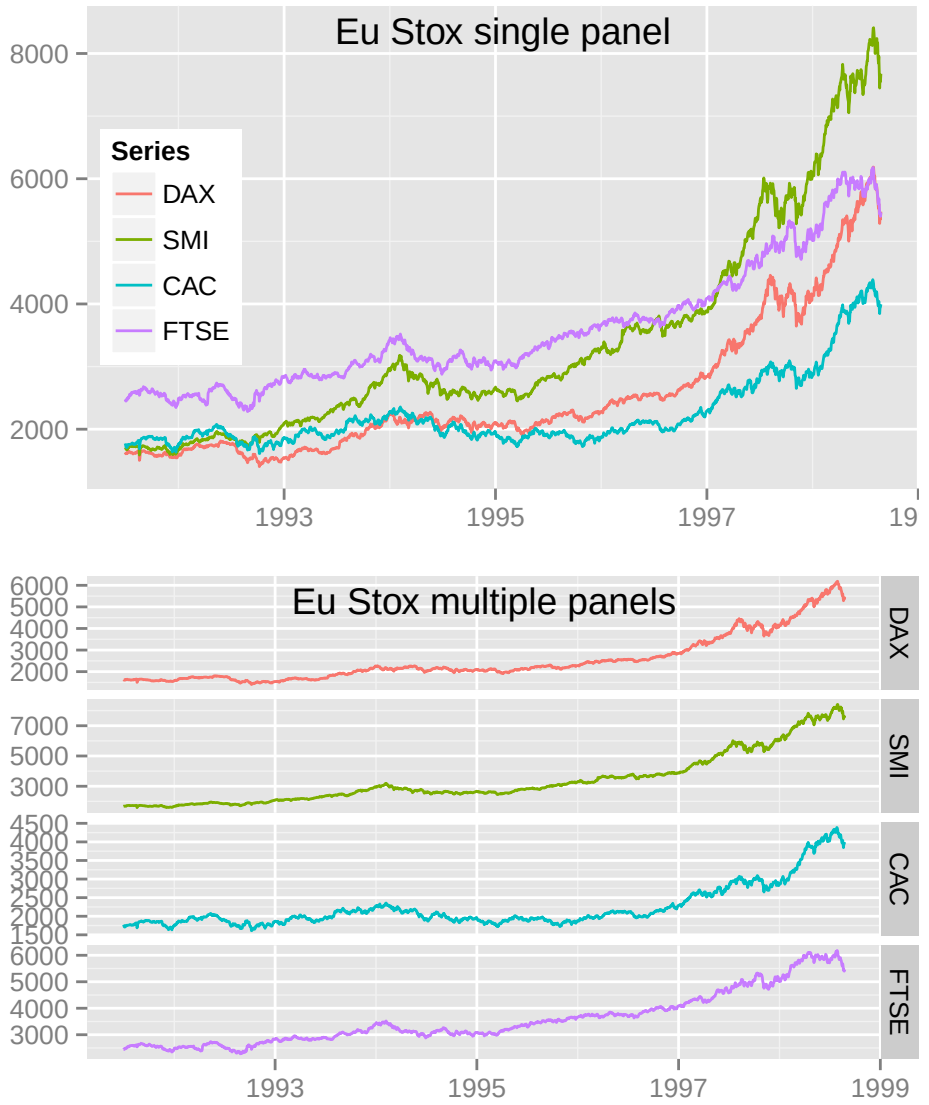
```
> my_ggplot <- ggplot(data=iris,
+       mapping=aes(Petal.Length, Sepal.Length)) +
+   geom_point(aes(shape=Species, color=Species)) +
+   geom_dl(aes(label=Species, color=Species),
+     method="smart.grid") +
+   scale_shape_manual(values=c(setosa=1,
+     virginica=6, versicolor=3), guide="none") +
+   scale_colour_discrete(guide="none")  # no label guide
> my_ggplot  # Render the plot
```



# Plotting zoo Time Series Using autoplot()

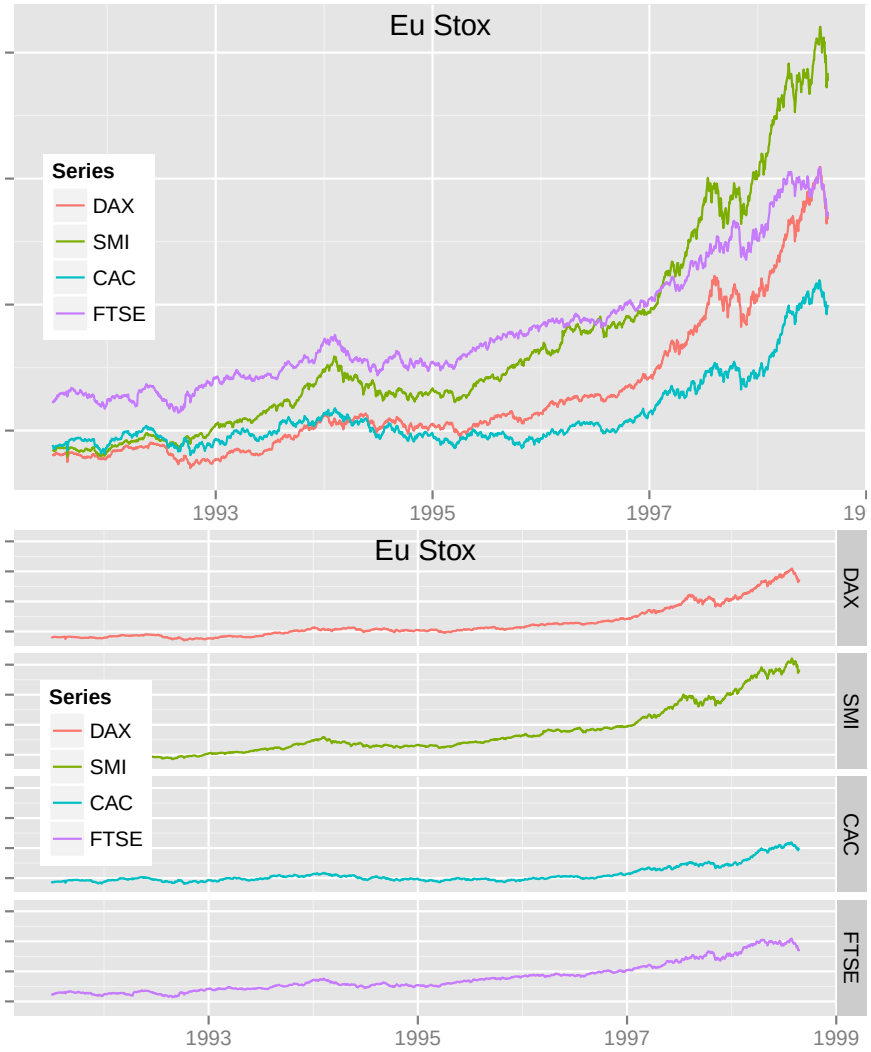
autoplot() is a convenience wrapper for *ggplot2* functions, designed to plot zoo time series.

```
> zoots <- as.zoo(EuStockMarkets)
> # Create ggplot2 theme object
> auto_theme <- theme(
+   legend.position="none",
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.0,0.0,-0.5,0.0),"cm"),
+ #  axis.text.y=element_blank(),
+   plot.background=element_blank()
+   )  # end theme
> # ggplot2 object for plotting in single panel
> ggp_zoo_single <- autoplot(zoots,
+             main="Eu Stox single panel",
+             facets=NULL) + xlab("") +
+             auto_theme
> # ggplot2 object for plotting in multiple panels
> ggp_zoo_multiple <- autoplot(zoots,
+             main="Eu Stox multiple panels",
+             facets="Series ~ .") + xlab("") +
+             facet_grid("Series ~ .",
+             scales="free_y") + auto_theme
> # Render plots in single column
> grid.arrange(ggp_zoo_single +
+          theme(legend.position=c(0.1, 0.5)),
+        ggp_zoo_multiple, ncol=1)
```



# legacy - Plotting zoo Time Series Using *ggplot2*

```
> #
> auto_theme <- theme(legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+   plot.background=element_blank(),
+   axis.text.y=element_blank()
+   )
>
> # Plot ggplot2 in single pane
> ggp.zoo1 <- autoplot(zoots, main="Eu Stox",
+    facets=NULL) + xlab("") +
+   theme(legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+   plot.background=element_blank(),
+   axis.text.y=element_blank()
+   )
> # Plot ggplot2 in multiple panes
> ggp.zoo2 <- autoplot(zoots, main="Eu Stox",
+    facets=Series ~ .) + xlab("") +
+   theme(legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+   plot.background=element_blank(),
+   axis.text.y=element_blank()
+   )
> # Create plot ggplot2 in multiple panes
> grid.arrange(ggp.zoo1, ggp.zoo2, ncol=1)
```



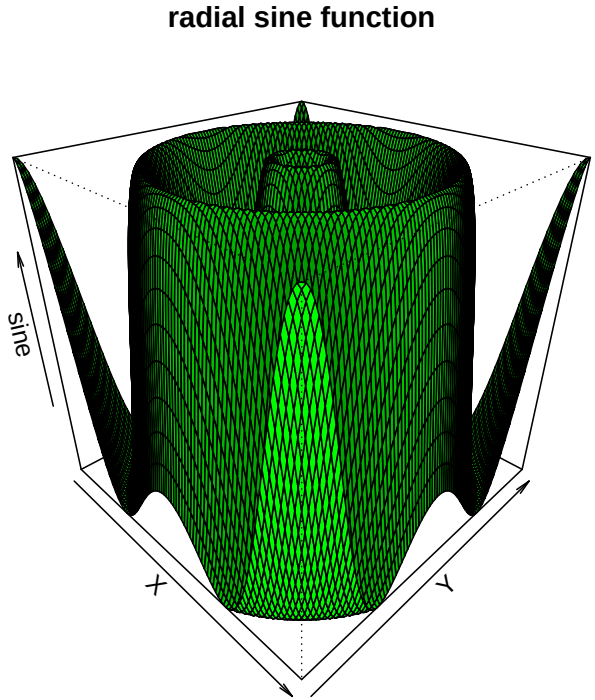
# Plotting 3d Perspective Surface Plots

The function `persp()` plots a 3d perspective surface plot of a function specified over a grid of argument values.

The argument *z* accepts a matrix containing the function values.

`persp()` belongs to the base *graphics* package, and doesn't create interactive plots.

```
> # Define function of two variables
> fun2d <- function(x, y) sin(sqrt(x^2+y^2))
> # Calculate function over matrix grid
> xlim <- seq(from=-10, to=10, by=0.2)
> ylim <- seq(from=-10, to=10, by=0.2)
> # Draw 3d surface plot of function
> persp(z=outer(xlim, ylim, FUN=fun2d),
+ theta=45, phi=30, zlab="sine",
+ shade=0.1, col="green",
+ main="radial sine function")
```



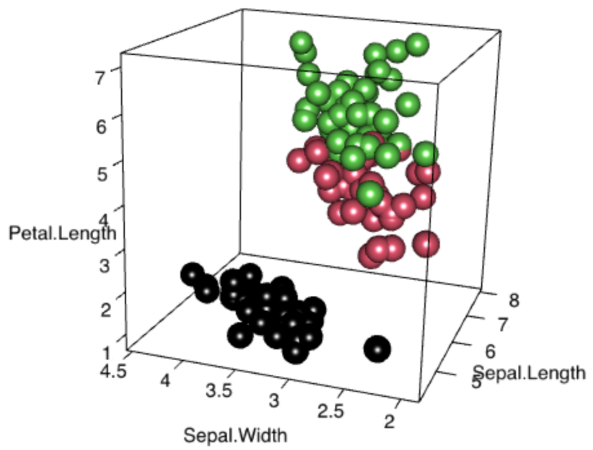
# Interactive 3d Scatter Plots Using Package *rgl*

The package *rgl* creates *interactive* 3d scatter plots and surface plots by calling the *WebGL JavaScript* library.

*WebGL* is a *JavaScript* library which creates plots by calling graphics code written in the *OpenGL* language.

The function `rgl::plot3d()` plots an *interactive* 3d scatter plot from three vectors of data.

```
> # Set rgl options
> options(rgl.useNULL=TRUE)
> # Load package rgl
> library(rgl)
> # Create 3d scatter plot of function
> with(iris, rgl::plot3d(Sepal.Length, Sepal.Width, Petal.Length,
+             type="s", col=as.numeric(Species)))
> # Render the 3d scatter plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
```



# Interactive 3d Surface Plots Using Package *rgl*

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a *vectorized* function or a matrix.

*rgl* is an `R` package for 3d and perspective plotting, based on the *OpenGL* framework.

```
> library(rgl)  # Load rgl
> # Define function of two variables
> fun2d <- function(x, y) y*sin(x)
> # Create 3d surface plot of function
> rgl::persp3d(x=fun2d, xlim=c(-5, 5), ylim=c(-5, 5),
+   col="green", axes=FALSE)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="surfacergl", width=500, height=500)
> # Draw 3d surface plot of matrix
> xlim <- seq(from=-5, to=5, by=0.1)
> ylim <- seq(from=-5, to=5, by=0.1)
> rgl::persp3d(z=outer(xlim, ylim, FUN=fun2d),
+   xlab="x", ylab="y", zlab="fun2d", col="green")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
> # Save current view to png file
> rgl::rgl.snapshot("surface_plot.png")
> # Define function of two variables and two parameters
> fun2d <- function(x, y, lambdaf1=1, lambdaf2=1)
+   sin(lambdaf1*x)*sin(lambdaf2*y)
> # Draw 3d surface plot of function
> rgl::persp3d(x=fun2d, xlim=c(-5, 5), ylim=c(-5, 5),
+   col="green", axes=FALSE, lambdaf1=1, lambdaf2=2)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
```