Advanced R: Visualization and Programming

Computational Economics Practice
Winter Term 2015/16
Stefan Feuerriegel

Today's Lecture

Objectives

1. Visualizing data in R graphically as points, lines, contours or areas
2. Understanding the programming concepts of if-conditions and loops
3. Implementing simple functions in R
4. Measuring execution time

Outline

1. Visualization
2. Control Flow
3. Timing
4. Wrap-Up

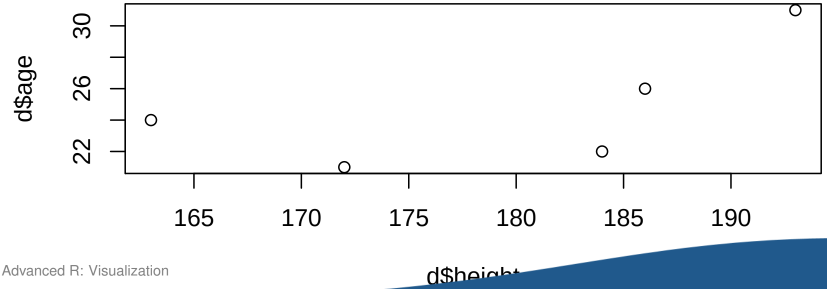
Outline

1. Visualization
2. Control Flow
3. Timing
4. Wrap-Up

Point Plot

- Creating simple point plots (also named scatter plots) via `plot(...)`
- Relies upon vectors denoting the x-axis and y-axis locations
- Various options can be added to change appearance

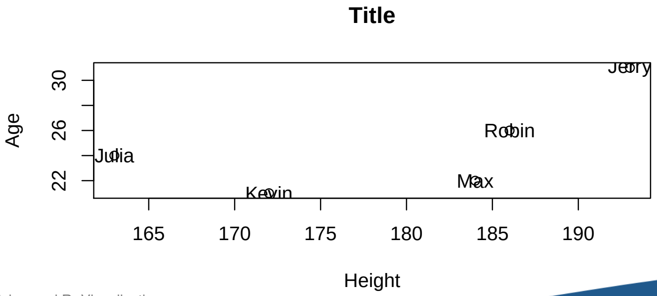
```
d <- read.csv("persons.csv", header=TRUE, sep=",",
              stringsAsFactors=FALSE)
plot(d$height, d$age)
```



Adding Title, Labels and Annotations

- Title is added through additional parameter `main`
- Axis labels are set via `xlab` and `ylab`
- Annotations next to points with `text(...)`

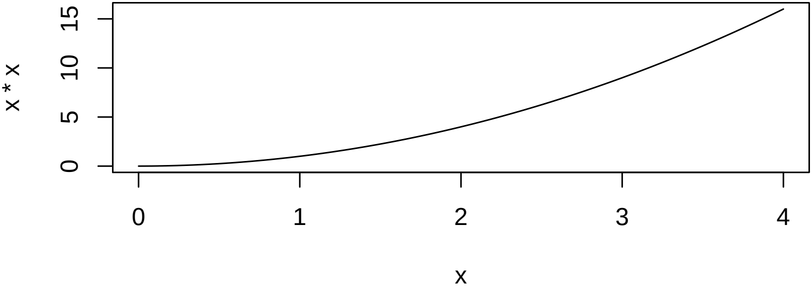
```
plot(d$height, d$age,
     main="Title",                      # title for the plot
     xlab="Height", ylab="Age")  # labels for x and y axis
text(d$height, d$age, d$name) # d$name are annotations
```



Line Plot

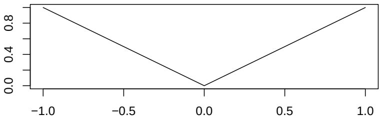
Generate line plot using the additional option `type="l"`

```
x <- seq(0, 4, 0.01)
plot(x, x*x, type="l")
```



Exercise: Plotting

```r
x <- seq(-1, +1, 0.01)
```



Question

- How would you reproduce the above plot?
  - `plot(x, kink(x), type="l", main="")`
  - `plot(x, kink(x), type="l", lab="")`
  - `plot(x, abs(x), type="l", ylab="", xlab="")`
- Visit `http://pingo.upb.de` with code 1523

3D Plots

- Consider the function $f(x,y) = x^3 + 3y - y^3 - 3x$

```r
f <- function(x, y) x^3+3*y-y^3-3*x
```

- Create axis ranges for plotting

```r
x <- seq(-5, 5, 0.1)
y <- seq(-5, 5, 0.1)
```

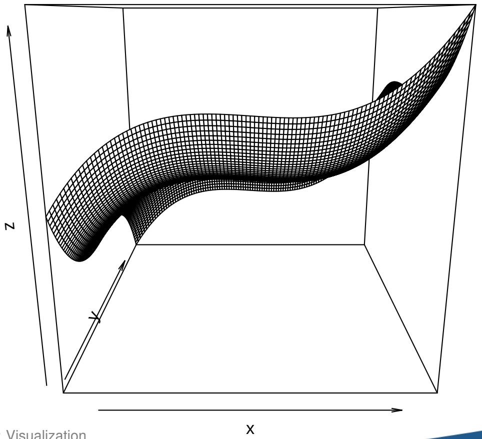
- Function `outer(x,y,f)` evaluates `f` all combinations of `x` and `y`

```r
z <- outer(x, y, f)
```

3D Plots

Function `persp(...)` plots the plane through `x`, `y` and `z` in 3D

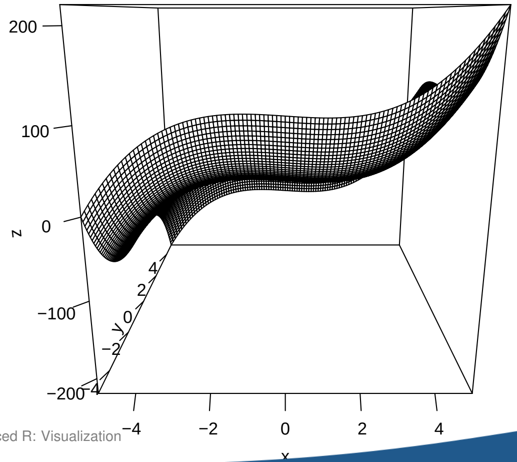
```
persp(x, y, z)
```



3D Plots

Turn on ticks on axes via `ticktype="detailed"`

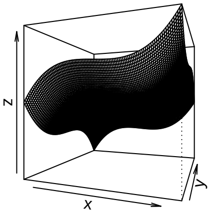
```
persp(x, y, z, ticktype="detailed")
```



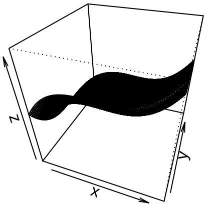
3D Plots

Parameters `theta` (left/right) and `phi` (up/down) control viewing angle

```r
persp(x, y, z, theta=20, phi=0)
```



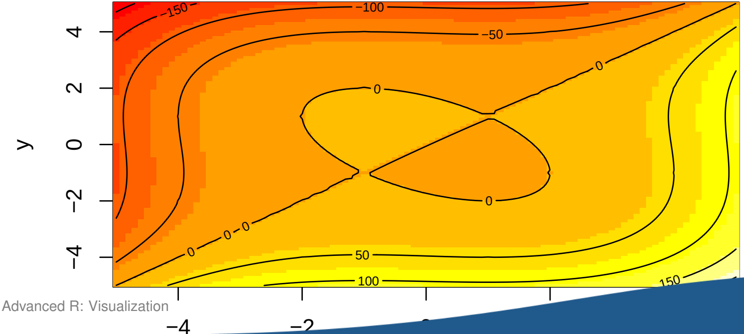

```r
persp(x, y, z, theta=20, phi=35)
```



Contour Plots

- A contour line is a curve along which the function has the same value
- `image(...)` plots a grid of pixels colored corresponding to `z`-value
- `contour(..., add=TRUE)` adds contour lines to an existing plot

```r
image(x, y, z) # Plot colors
contour(x, y, z, add=TRUE) # Add contour lines
```



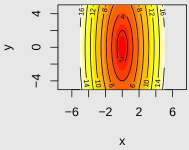
Contour Plots

```r
f <- function(x, y) sqrt(x^2+y^2)
z <- outer(x, y, f)
image(x, y, z, asp=1) # set aspect ratio, i.e. same scale for x and y
contour(x, y, z, add=TRUE)
```

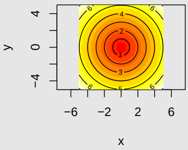
Question

- What would the above plot look like?

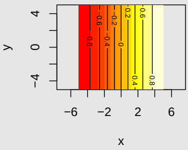
Answer A



Answer B



Answer C



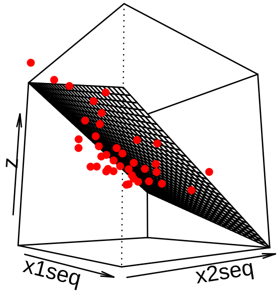
- Visit `http://pingo.upb.de` with code 1523

Plotting Regression Plane

```
library(car) # for dataset Highway1

## Warning:  no function found corresponding to methods exports from 'SparseM' for:
'coerce'

model <- lm(rate ~ len + slim, data=Highway1)
model

##
## Call:
## lm(formula = rate ~ len + slim, data = Highway1)
##
## Coefficients:
## (Intercept)          len         slim
##    16.61050     -0.09151     -0.20906

x1r <- range(Highway1$len)
x1seq <- seq(x1r[1], x1r[2], length=30)
x2r <- range(Highway1$slim)
x2seq <- seq(x2r[1],x2r[2], length=30)
z <- outer(x1seq, x2seq,
           function(a,b) predict(model,
                                 newdata=data.frame(len=a,slim=b)))
```

Plotting a Regression Plane

```
res <- persp(x=x1seq, y=x2seq, z=z,
             theta=50, phi=-10)
dp <- trans3d(Highway1$len, Highway1$slim,
              Highway1$rate, pmat=res)
points(dp, pch=20, col="red")
```



Outline

Managing Code Execution

- Control flow specifies order in which statements are executed
- Previous concepts can only execute R code in a linear fashion
- Control flow constructs can choose which execution path to follow

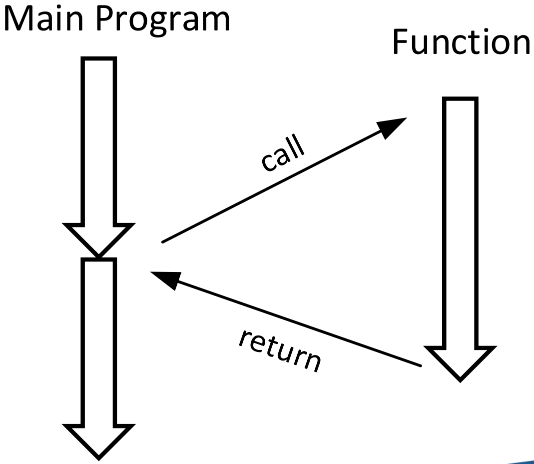
Functions: Combines sequence of statements into a self-contained task

Conditional expressions: Different computations according to a specific condition

Loops: Sequence of statements which may be executed more than once

Functions

- Functions avoid repeating the same code more than once
- Leave the current evaluation context to execute pre-defined commands



Functions

- Extend set of built-in functions with opportunity for customization
- Functions **can** consist of the following:
  1. Name to refer to (avoid existing function names in R)
  2. Function body is a sequence of statements
  3. Arguments define additional parameters passed to the function body
  4. Return value which can be used after executing the function
- Simple example

```
f <- function(x,y) {
  return(2*x + y^2)
}
f(-3, 5)

## [1] 19
```

Functions

- General syntax

```
functionname <- function(argument1, argument2, ...) {
  function_body
  return(value)
}
```

- Return value is the last evaluated expression
  → Alternative: set explicitly with `return(...)`
- Curly brackets can be omitted if the function contains only one statement (not recommended)
- Be cautious since the order of the arguments matters
- Values in functions are not printed in console
  → Remedy is `print(...)`

Examples of Functions

```
square <- function(x) x*x # last value is return value
square(10)

## [1] 100
```

```
cubic <- function(x) {
  # Print value to screen from inside the function
  print(c("Value: ", x, " Cubic: ", x*x*x))
  # no return value
}
cubic(10)

## [1] "Value: "   "10"         " Cubic: " "1000"
```

Examples of Functions

```r
hello <- function() { # no arguments
  print("world")
}
hello()

## [1] "world"
```

```r
my.mean <- function(x) {
  return (sum(x)/length(x))
}
my.mean(1:100)

## [1] 50.5
```

Scope in Functions

- Variables created inside a function only exists within it $\rightarrow$ local
- They are thus inaccessible from outside of the function
- Scope denotes when the name binding of variable is valid

```
x <- "A"
g <- function(x) {
  x <- "B"
  return(x)
}
x <- "C"
```

- What are the values?

```
g(x) # Return value of function x
x    # Value of x after function execution
```

- Solution

```
## [1] "B"
## [1] "C"
```

Scope in Functions

- Variables created inside a function only exists within it $\rightarrow$ local
- They are thus inaccessible from outside of the function
- Scope denotes when the name binding of variable is valid

```
x <- "A"
g <- function(x) {
  x <- "B"
  return(x)
}
x <- "C"
```

- What are the values?

```
g(x) # Return value of function x
x    # Value of x after function execution
```

- Solution

```
## [1] "B"
## [1] "C"
```

Unevaluated Expressions

- Expressions can store symbolic mathematical statements for later modifications (e. g. symbolic derivatives)
- Let's define an example via `expression(...)`

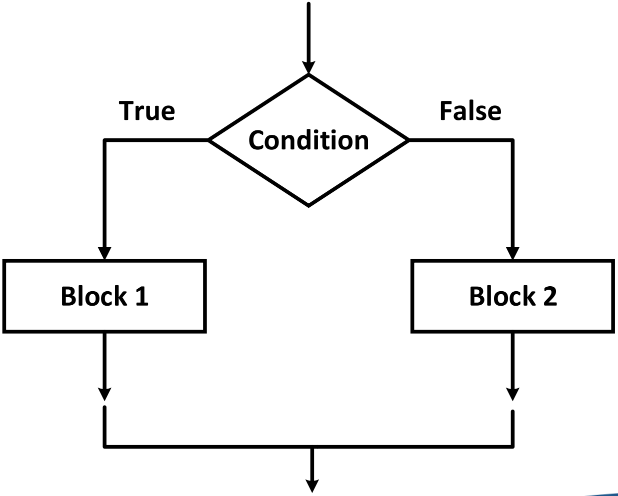
```
f <- expression(x^3+3*y-y^3-3*x)
f

## expression(x^3 + 3 * y - y^3 - 3 * x)
```

- If evaluation of certain parameters becomes necessary, one can use `eval(...)`

```
x <- 2
y <- 3
eval(f)

## [1] -16
```

If-Else Conditions

- Conditional execution requires a condition to be met



If-Else Conditions

- Keyword `if` with optional `else` clause
- General syntax:

if condition

```
if (condition) {
  statement1
}
```

If `condition` is true, then `statement` is executed

if-else condition

```
if (condition) {
  statement1
} else {
  statement2
}
```

If `condition` is true, then `statement1` is executed, otherwise `statement2`

If-Else Conditions

- Example

```r
grade <- 2
if (grade <= 4) {
  print("Passed")
} else {
  print("Failed")
}

## [1] "Passed"
```

```r
grade <- 5
if (grade <= 4) {
  print("Passed")
} else {
  print("Failed")
}

## [1] "Failed"
```

- Condition must be of length 1 and evaluate as either `TRUE` or `FALSE`

```r
if (c(TRUE, FALSE)) { # don't do this!
  print("something")
}

## Warning in if (c(TRUE, FALSE)) {:  Bedingung hat Länge
> 1 und nur das erste Element wird benutzt

## [1] "something"
```

Else-If Clauses

- Multiple conditions can be checked with `else if` clauses
- The last `else` clause applies when no other conditions are fulfilled
- The same behavior can also be achieved with nested if-clauses

else-if clause

```
if (grade == 1) {
  print("very good")
} else if (grade == 2) {
  print("good")
} else {
  print("not a good grade")
}
```

Nested if-condition

```
if (grade == 1) {
  print("very good")
} else {
  if (grade == 2) {
    print("good")
  } else {
    print("not a good grade")
  }
}
```

If-Else Function

- As an alternative, one can also reach the same control flow via the function `ifelse(...)`

```
ifelse(condition, statement1, statement2)
# executes statement1 if condition is true,
# otherwise statement2
```

```
grade <- 2
ifelse(grade <= 4, "Passed", "Failed")

## [1] "Passed"
```

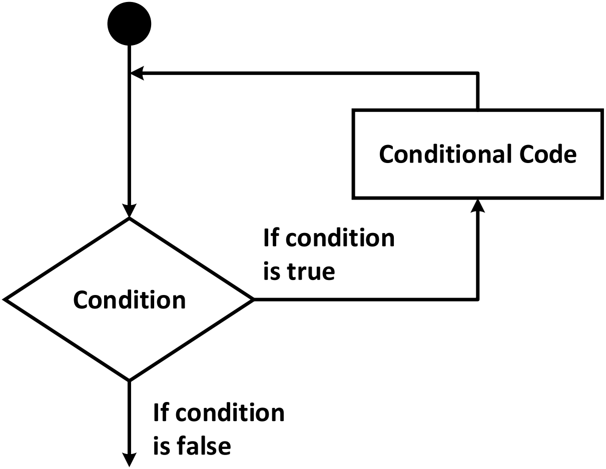
- `ifelse(...)` can also work with vectors as if it was applied to each element separately

```
grades <- c(1, 2, 3, 4, 5)
ifelse(grades <= 4, "Passed", "Failed")

## [1] "Passed" "Passed" "Passed" "Passed" "Failed"
```

- This allows for the efficient comparison of vectors

For Loop

- `for` loops execute statements for a fixed number of repetitions



For Loop

- General syntax

```
for (counter in looping_vector){
  # code to be executed for each element in the sequence
}
```

- In every iteration of the loop, one value in the looping vector is assigned to the `counter` variable that can be used in the statements of the body of the loop.
- Examples

```
for (i in 4:7) {
  print(i)
}

## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

```
a <- c()
for (i in 1:3){
  a[i] <- sqrt(i)
}
a

## [1] 1.000000 1.414214 1.732051
```

While Loop

- Loop where the number of iterations is controlled by a condition
- The condition is checked in every iteration
- When the condition is met, the loop body in curly brackets is executed
- General syntax

```
while (condition) {
  # code to be executed
}
```

- Examples

```
z <- 1
# same behavior as for loop
while (z <= 4) {
    print(z)
    z <- z + 1
}

## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

```
z <- 1
# iterates all odd numbers
while (z <= 5) {
    z <- z + 2
    print(z)
}

## [1] 3
## [1] 5
## [1] 7
```

Outline

Measuring Timings via Stopwatch

- Efficiency is a major issue with larger datasets and complex codes
- Timings can help in understanding scalability and bottlenecks
- Use a stopwatch approach measuring the duration between two `proc.time()` calls

```
start.time <- proc.time() # Start the clock

g <- rnorm(100000)
h <- rep(NA, 100000)
for (i in 1:100000) { # Loop over vector, always add +1
  h[i] <- g[i] + 1
}

# Stop clock and measure duration
duration <- proc.time() - start.time
```

Measuring Timings via Stopwatch

- Results of `duration` have the following format

```
##    user  system elapsed
##    0.71    0.02    0.72
```

- Timings are generally grouped into 3 categories
  - User time measures the understanding of the R instructions
  - System time measures the underlying execution time
  - Elapsed is the difference since starting the stopwatch ($=$ user $+$ system)
- Alternative approach avoiding loop

```
start.time <- proc.time() # Start clock
g <- rnorm(100000)
h <- g + 1
proc.time() - start.time # Stop clock

##    user  system elapsed
##    0.08    0.00    0.08
```

- Rule: vector operations are faster than loops

Measuring Timings of Function Calls

Function `system.time(...)` can directly time function calls

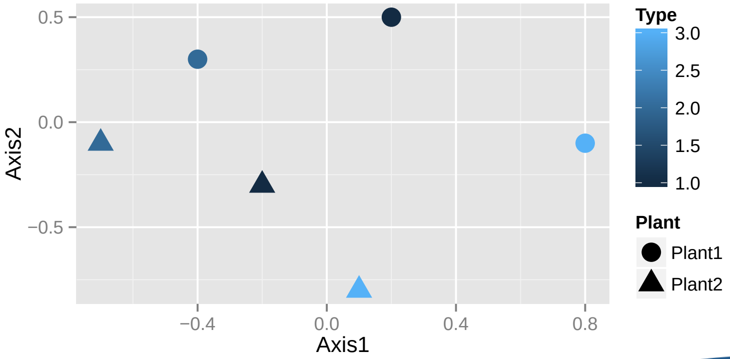
```
slowloop <- function(v){
        for (i in v) {
          tmp <- sqrt(i)
        }
}

system.time(slowloop(1:1000000))

##    user  system elapsed
##    2.06    0.05    2.13
```

Outline

Fancy Diagrams with ggplot2

```
library(ggplot2)
```

```
df <- data.frame(Plant=c("Plant1", "Plant1", "Plant1", "Plant2", "Plant2", "Plant2"),
                 Type=c(1, 2, 3, 1, 2, 3),
                 Axis1=c(0.2, -0.4, 0.8, -0.2, -0.7, 0.1),
                 Axis2=c(0.5, 0.3, -0.1, -0.3, -0.1, -0.8))
ggplot(df, aes(x=Axis1, y=Axis2, shape=Plant,
               color=Type)) + geom_point(size=5)
```



Summary: Visualization and Timing

`plot()`	Simple plot function
`text()`	Add text to an existing plot
`outer()`	Apply a function to two arrays
`persp()`	Plot a surface in 3D
`image()`	Plot a colored image
`contour()`	Add contour lines to a plot
`trans3d()`	Add point to an existing 3D plot
`points()`	Add points to a plot
`proc.time()`	Stopwatch for measuring execution time
`system.time(expr)`	Measures execution time of an expression

Summary: Programming

`function(){}`	Self-defined function
`expression()`	Function with arguments not evaluated
`eval()`	Evaluate an expression
`if, else`	Conditional statement
`for(){}`	Loops over a fixed vector
`while`	Loops while a condition is fulfilled

Outlook

Additional Material

- Further exercises as homework
- Advanced materials beyond our scope
  - Advanced R (CRC Press, 2014, by Wickham)
    `http://adv-r.had.co.nz/`

Future Exercises

R will be used to implement optimization algorithms