

# The R language and why you should be using it

Roger Barlow  
Huddersfield University

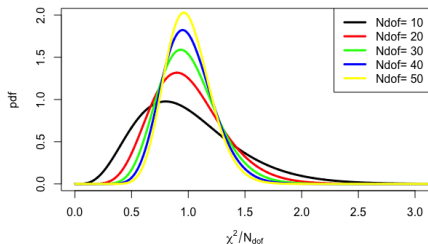
Terascale Statistics School 2020  
virtually at DESY, Hamburg

6<sup>th</sup> July 2020



# Reasons for learning R

- 1 It's the definitive language for serious statistics
- 2 It's very easy to learn. Gentle learning curve: you can do useful stuff straight away
- 3 It provides really neat and easy plots for your talks and publications



```
1 N_dof <- seq(10,50,10)
2 colours <- c("black","red","green","blue","yellow")
3 plot(0,0,type='n',xlim=c(0,3),ylim=c(0,2),ylab="pdf",xlab=expression(chi^2/N[dof])
4 xx <- seq(0,300,.01)
5 for(n in N_dof) lines(xx/n,n*dchisq(xx,n),lwd=3,col=colours[n/10])
6 legend('topright',col=colours,legend=paste("N_dof=",N_dof),lwd=3)
```

- 4 It's a beautiful language enabling you to write really elegant code

Download in seconds

From [r-project.org](http://r-project.org) for R or [rstudio.com](http://rstudio.com) for the Rstudio IDE.

# R - key facts

- R is the freeware version of the statistics language S
- It's an interpreted language - hence user-friendly
- Incorporates nice features of earlier languages: Algol, APL, LISP etc
- Only four types: `numeric`, `logical`, `character`, `complex` .  
(`complex` is rare.) All numbers are stored as doubles.
- Very weakly typed. No need to declare variables before use.
- R is a rich language: the same thing can often be done in several ways. Sometimes they are equivalent, sometimes not.
- Run programs (also known as scripts) through `source("filename")` or use R-studio
- Statements terminated by newline (unless incomplete) or semicolon
- **Everything is a vector**
- Easy to use basic functions are generally extended through optional arguments
- Comprehensive online help through `help(topic)` and user manual

# A first R session

What you type is shown in blue

It has the usual operators

including exponentiation,  
which can be `**` or `^`

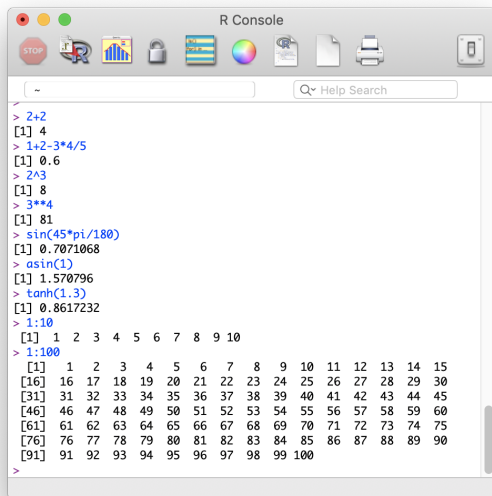
It has the usual functions

and some less usual ones

The `:` operator generates a  
vector

The meaning of the square  
brackets is revealed.

**Everything is a vector**



```
R Console
~ Help Search

> 2+2
[1] 4
> 1+2-3*4/5
[1] 0.6
> 2^3
[1] 8
> 3**4
[1] 81
> sin(45*pi/180)
[1] 0.7071068
> asin(1)
[1] 1.570796
> tanh(1.3)
[1] 0.8617232
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 1:100
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
[46] 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
[61] 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
[76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
[91] 91 92 93 94 95 96 97 98 99 100
>
```

# Doing things with vectors

The `seq` (sequence) function generalises the `:` operator

You can specify the increment or the length

The `rep` (repeat) function is also handy

The `c` function - for combine. concatenate or column-vector - is used all the time

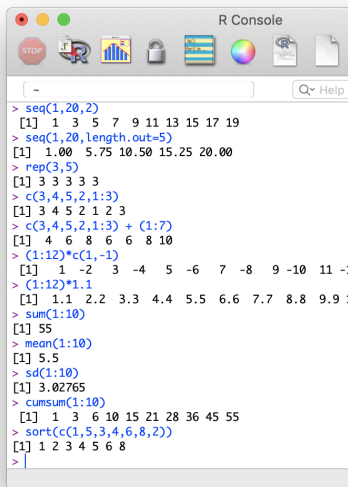
Operations between two vectors take place element by element

If the vectors are not the same length, the shorter one is it recycled

Operations with a single number are a special case of this

The functions `sum`, `mean` and `sd` operate on vectors to give a number

Other useful functions like `cumsum` and `sort`. Also `max`, `min`, `range`, `length`...



```
> seq(1,20,2)
[1] 1 3 5 7 9 11 13 15 17 19
> seq(1,20,length.out=5)
[1] 1.00 5.75 10.50 15.25 20.00
> rep(3,5)
[1] 3 3 3 3 3
> c(3,4,5,2,1:3)
[1] 3 4 5 2 1 2 3
> c(3,4,5,2,1:3) + (1:7)
[1] 4 6 8 6 6 8 10
> (1:12)*c(1,-1)
[1] 1 -2 3 -4 5 -6 7 -8 9 -10 11 -12
> (1:12)*1.1
[1] 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 11.0 12.1
> sum(1:10)
[1] 55
> mean(1:10)
[1] 5.5
> sd(1:10)
[1] 3.02765
> cumsum(1:10)
[1] 1 3 6 10 15 21 28 36 45 55
> sort(c(1,5,3,4,6,8,2))
[1] 1 2 3 4 5 6 8
>
```

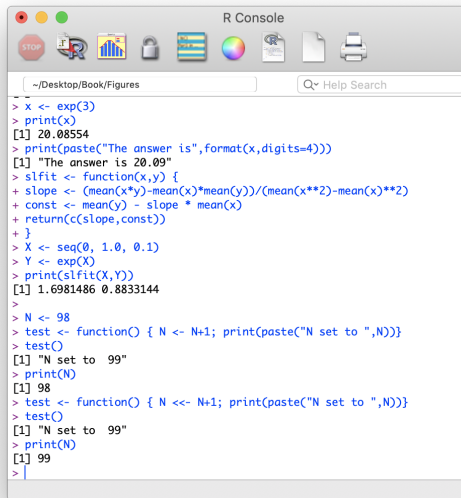
# Variables and functions

from a calculator to a computer

The assignment operator is `<-`  
(`=` works but `<-` is preferable)

For printing results, `paste` is vital. It inserts spaces - use `paste0` if you don't want them. Use `format` to tidy the output.

Functions are easy to define. You can only return one thing - but that can be a vector. They have read access to higher scope variables but write only to local copies except with the 'super-assign' double-headed arrow `<<-`.



```
R Console
~/Desktop/Book/Figures
> x <- exp(3)
> print(x)
[1] 20.08554
> print(paste("The answer is",format(x,digits=4)))
[1] "The answer is 20.09"
> slfit <- function(x,y) {
+ slope <- (mean(x*y)-mean(x)*mean(y))/(mean(x**2)-mean(x)**2)
+ const <- mean(y) - slope * mean(x)
+ return(c(slope,const))
+ }
> X <- seq(0, 1.0, 0.1)
> Y <- exp(X)
> print(slfit(X,Y))
[1] 1.6981486 0.8833144
>
> N <- 98
> test <- function() { N <- N+1; print(paste("N set to ",N))}
> test()
[1] "N set to 99"
> print(N)
[1] 98
> test <- function() { N <<- N+1; print(paste("N set to ",N))}
> test()
[1] "N set to 99"
> print(N)
[1] 99
> |
```

# Plots

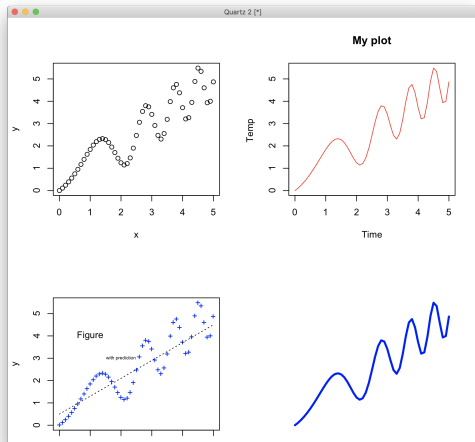
`plot(x,y)` . gets you started  
Then you can add options

```
R Console
~/Desktop/Book/Figures
> par(mfrow=c(2,2))
> x <- seq(0,5,.1)
> y <- x*sin(x**2)
> plot(x,y)
> plot(x,y,col='red',type='l',main='My plot',xlab='Time',ylab='Temp ')
> plot(x,y,col='blue',pch='+')
> lines(c(0,5),c(-5,4.5),lty=3)
> text(1,4,"Figure")
> text(2,3,"with prediction",cex=.5)
> plot(x,y,col='blue',type='l',lwd=3,xaxt='n',yaxt='n',ann=FALSE,bty='n')
```

Use `lines` `points` `text`  
`legend` and `polygon` to add  
to an existing plot

Options apply only to that  
function call. To make them  
stick use `par`

`par` also used for multiframes



# Indexing vectors

More to this than you expect!

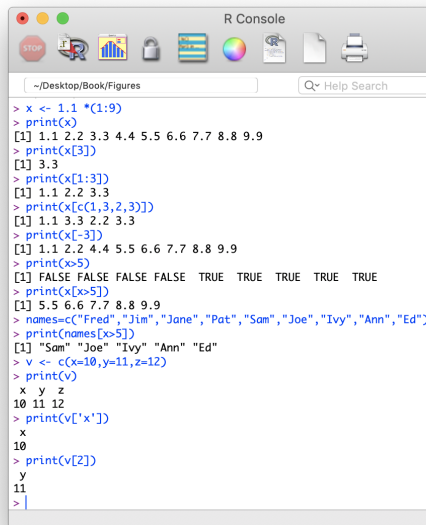
Use square brackets. The index is a number (and starts at 1)

But the index can also be a vector (as everything is a vector)

If the index is negative, it means 'everything except'

Or the index can be logical.

Or the index can be character



```
R Console
~/Desktop/Book/Figures
> x <- 1.1 *(1:9)
> print(x)
[1] 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
> print(x[3])
[1] 3.3
> print(x[1:3])
[1] 1.1 2.2 3.3
> print(x[c(1,3,2,3)])
[1] 1.1 3.3 2.2 3.3
> print(x[-3])
[1] 1.1 2.2 4.4 5.5 6.6 7.7 8.8 9.9
> print(x>5)
[1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
> print(x[x>5])
[1] 5.5 6.6 7.7 8.8 9.9
> names=c("Fred", "Jim", "Jane", "Pat", "Sam", "Joe", "Ivy", "Ann", "Ed")
> print(names[x>5])
[1] "Sam" "Joe" "Ivy" "Ann" "Ed"
> v <- c(x=10,y=11,z=12)
> print(v)
  x y z
10 11 12
> print(v['x'])
  x
10
> print(v[2])
  y
11
> |
```



# Program Control

## Part 1: How to use them

The usual looping and branching commands are provided.

{ parentheses } define blocks

```
for (i in 1:10) {  
  stuff  
}
```

```
error=100 ; count=0  
while (error>1E-5 & count<100) {  
  error <- iteratefit(data)  
  count <- count+1  
}  
if (x>99) {  
  stuff  
} else {  
  more stuff  
}
```

# Program Control

## part 2: How to avoid them

You have 100 values in `x` and you need to find the averages for the positive and negative values. A C++ program could be

```
int np=nm=0;
double sp=sm=0;
for(int i=0;i<100;i++){
    if (x[i]>0) {sp+= x[i]; np++;}
    if (x[i]<0) {sm+= x[i]; nm++;}
}
cout<<" average of positive values "<<sp/np<<" and of
negative values "<<sm/nm<<endl;
```

The R version is just

```
print (paste( " average of positive values", mean(x[x>0]),
"and of negative values ",mean(x[x<0])))
```

If you find yourself using an `if` statement in R, maybe think again.

If you find yourself using a `for` loop in R, definitely think again.

# From vectors to matrices

Vector: a block of elements, all of the same type.

It has a length `length(v)`

An array is a vector which *also* has a dimension `dim(v)`.

Can be read or written.

(Product of dimensions must equal length)

Omitting index gives all values (slicing)

Function `matrix` also available

```
> x <- 1:12
> print(x)
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> length(x)
[1] 12
> dim(x)
NULL
> dim(x)=c(3,4)
> print(x)
      [,1] [,2] [,3] [,4]
[1,]  1   4   7  10
[2,]  2   5   8  11
[3,]  3   6   9  12
> print(x[3,4])
[1] 12
> print(x[3,])
[1] 3 6 9 12
> dim(x)=c(2,6)
> print(x)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  1   3   5   7   9  11
[2,]  2   4   6   8  10  12
> dim(x)=c(2,3,2)
> print(x)
, , 1
      [,1] [,2] [,3]
[1,]  1   3   5
[2,]  2   4   6
, , 2
      [,1] [,2] [,3]
[1,]  7   9  11
[2,]  8  10  12
> y <- matrix(1:12,3,4)
> print(y)
      [,1] [,2] [,3] [,4]
[1,]  1   4   7  10
[2,]  2   5   8  11
[3,]  3   6   9  12
```

# Matrix calculations

Functions `t` for transpose, `det` for determinant

Operator `%*%` does matrix product

Operator `%o%` does outer (Cartesian) product

`solve(A,b)` for linear equations

**$Ax=b$**

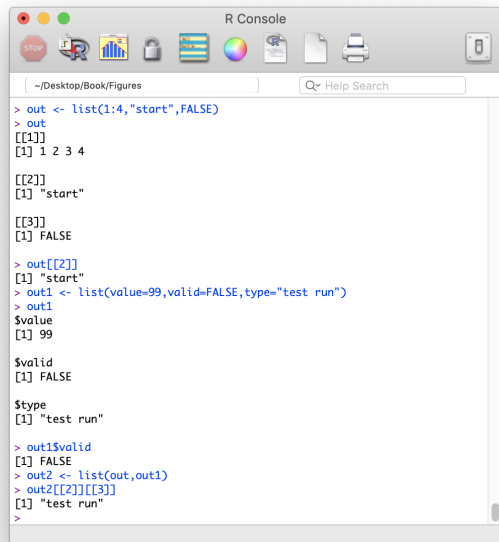
`solve(A)` on its own gives inverse

Eigenvector/value analysis comes as standard

```
R Console
~/Desktop/Book/Figures  Help Search
> M <- matrix(1:4,2,2,byrow=TRUE)
> print(M)
      [,1] [,2]
[1,]    1    2
[2,]    3    4
> t(M)
      [,1] [,2]
[1,]    1    3
[2,]    2    4
> det(M)
[1] -2
> v <- c(3,2)
> M %*% v
      [,1]
[1,]    7
[2,]   17
> v %o% v
      [,1] [,2]
[1,]    9    6
[2,]    6    4
> solve(M,v)
[1] -4.0  3.5
> solve(M)
      [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
> eigen(M)
eigen() decomposition
$values
[1]  5.3722813 -0.3722813

$vectors
      [,1] [,2]
[1,] -0.4159736 -0.8245648
[2,] -0.9093767  0.5657675
```

# Lists



```
> out <- list(1:4,"start",FALSE)
> out
[[1]]
[1] 1 2 3 4

[[2]]
[1] "start"

[[3]]
[1] FALSE

> out[[2]]
[1] "start"
> out1 <- list(value=99,valid=FALSE,type="test run")
> out1
$value
[1] 99

$valid
[1] FALSE

$type
[1] "test run"

> out1$valid
[1] FALSE
> out2 <- list(out,out1)
> out2[[2]][[3]]
[1] "test run"
>
```

A vector is a block of elements, all of the same type

A list is a block of elements which may be of different types

List indexing by double square brackets

or by name - very useful for function returns

List elements can be lists

# Statistics functions and random numbers

Note: this *is* a statistics package so Gaussians are called 'Normal'

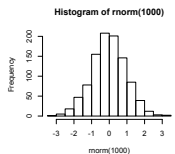
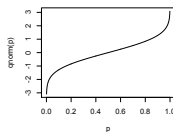
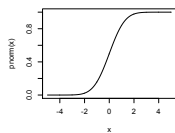
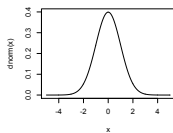
`dnorm(x)` gives unit Gaussian  $\frac{1}{\sqrt{2\pi}} e^{-x^2/2}$

`dnorm(x,mu,sigma)` gives general Gaussian

`pnorm(x)` gives integrated Gaussian.

`qnorm(p)` gives the inverse. ("How many sigma is 95%?" etc)

`rnorm(n)` generates n random numbers



Same pattern for other functions:

Poisson `dpois(r,mu)` `ppois(r,mu)` `qpois(p,mu)` `rpois(n,mu)`

Uniform. `dunif(x)` or `dunif(x,a,b)` `punif(x)` `qunif(p)` `runif(n)`

$\chi^2$  `dchisq(x,N)` `pchisq(x,N)` `qchisq(p,N)` `rchisq(n,N)`.

Binomial `dbinom(r,N,P)` `pbinom(r,N,P)` `qbinom(p,N,P)` `rbinom(n,N,P)`

... and so on

# Histograms

Just so easy using `hist` !

Override default binning with `breaks`.  
Histogram contents available through self-describing list

```
R Console
~/Desktop/Book/Figures
> par(mfrow=c(2,2))
> x <- rchisq(1000,5)
> hist(x)
> hist(x,breaks=100)
> hist(x[x<25],breaks=seq(0,25,.5))
> h <- hist(x,plot=FALSE)
> print(h)
$hbreaks
 [1] 0 2 4 6 8 10 12 14 16 18 20 22 24

$countss
 [1] 164 300 220 154 82 37 24 13 2 2 1 1

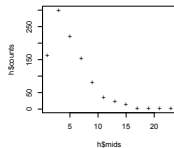
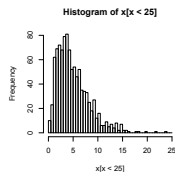
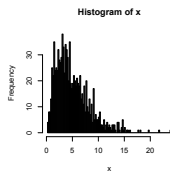
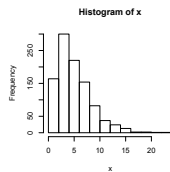
$density
 [1] 0.0820 0.1500 0.1100 0.0770 0.0410 0.0185 0.0120 0.0065 0.0010
 [10] 0.0010 0.0005 0.0005

$mids
 [1] 1 3 5 7 9 11 13 15 17 19 21 23

$xname
 [1] "x"

$equidist
 [1] TRUE

attr(,"class")
 [1] "histogram"
> plot(h$mids,h$counts,pch="+")
```



Compare and contrast:

A histogram in R  
`h <- hist(x)`

A histogram in ROOT

```
#include "TH1F.h"
TH1F* h = new TH1F("h","My title",100,0,1);
for(int_t i=0;i<N;i++) {h->Fill(x[i]);}
h->Draw();
c1->Update();
delete h;
```



# Character Strings

Character strings are demarcated by single or double quotes.

"Hello, world" or 'Hello, world!'

Length given by `nchar(string)`

Strings can be tested for equality (unlike C)

Merge by `paste` . Coerces all its arguments into being character strings.

Separate by `strsplit("Hello, world", " ")`

Substrings `substr(string, from, to)` may be read or assigned-to

Replacement by `sub(find, replace, string)` and `gsub`

Location using `grep(pattern, string)` - lots of variants

If you're not into `grep` and regular expressions, use the `stringr` package.

`library(stringr)`

# Fitting I

## Solving an equation

```
F <- function(x){  
  return(1+x-x**2+x**3-x**4)}
```

```
uniroot(F,c(0,2))
```

```
$root
```

```
[1] 1.290647
```

```
$f.root
```

```
[1] 8.489406e-06
```

```
$iter
```

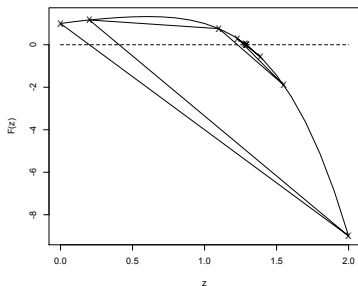
```
[1] 9
```

```
$init.it
```

```
[1] NA
```

```
$estim.prec
```

```
[1] 6.103516e-05
```



# Fitting II

## Maximising a 1-D function

Take same function  $F(x)$

```
optimize(F,c(0,2),maximum=TRUE)
```

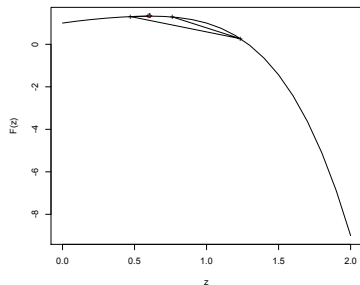
```
$maximum
```

```
[1] 0.6058243
```

```
$objective
```

```
[1] 1.326447
```

Takes 10 iterations



# Fitting III

## Maximising in higher dimensions

Test with famous Rosenbrock function

```
Rosenbrock <- function(x) {  
  return((x[1]-1)**2 +  
  100*(x[1]**2-x[2])**2)}
```

Minimum at (1,1) but valley sides steep

```
optim(c(-1,-1),Rosenbrock)
```

– taking (-1,-1) as the starting value

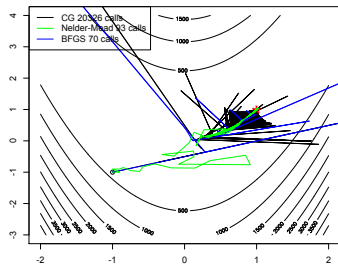
Default method is Nelder-Mead (like 'SIMPLEX')

Can also try conjugate-gradient

`method='CG'` - does worse

and quasi-Newton `method='BFGS'` does

better



# Other stuff

I haven't had time to talk about

- Data Frames: An array that looks like a lab notebook. Or a spreadsheet. Used a lot, for input and output and packages
- Contour plots: and other 2D graphics
- ggplot: an entirely separate plotting / visualisation package
- Models: used for fitting. You can say things like  $z \sim 1 + x + x*y$
- Objects and classes: You can use R for writing OO code, though not quite like Stroustrup
- Contributed software: there is lots out there that's not in the core, and it's easy to download and include
- Other features that I haven't learned about yet

# Web scraping

+++ Reduced online version +++ Terascale Statistics School 2020

6-10 July 2020  
DESY Hamburg  
Europe/Berlin timezone

- Overview
- Timetable
- Registration
  - Registration Form
- List of registrants

Support  
✉ anacen@desy.de

### List of registrants

Number of participants: 105

<a href="#">Name</a>	<a href="#">institution</a>	<a href="#">city</a>	<a href="#">country/region</a>
ABELING, Kira	Uni Göttingen	Göttingen	Germany
Ms. AKOLKAR, Nilima	University of Bonn	Bonn	Germany
Ms. ALBRECHT, Anna	Universität Hamburg	Hamburg	Germany
Mr. ALHAWITI , Fawaz	Physics	Dresden	Germany
BALTES, Lisa Marie	Ruprecht-Karls-Universität	Heidelberg	Germany
Prof. BARLOW, Roger	The University of Huddersfield	Huddersfield	United Kingdom
Mr. BARTELS, Falk	Kirchhoff-Institut für Physik	Heidelberg	Germany
Mr. BECHERER, Fabian	University of Freiburg	Freiburg	Germany
Dr. BEHNKE, Olaf	DESY	Hamburg	Germany
Mr. BEYER, Jakob	DESY	Hamburg	Germany
Mr. BHALLA, Naman Kumar	Georg-August-Universität	Göttingen	Germany
Mr. BHARGAVA , Parth	RWTH Aachen University	Aachen	Germany
Mr. BIRK, Joschka	University of Freiburg	Freiburg	Germany
Mr. BRUNNER, Martin	Julius Maximilian University of Würzburg	Würzburg	Germany

# Web scraping

The screenshot shows the RStudio interface with a script editor on the left and a viewer on the right. The script editor contains R code that scrapes data from a website and creates a horizontal bar chart. The console shows the execution of the code, including a warning message about an incomplete final line in the scraped data.

```
1 # Read and plot data from a web page
2 # There are better ways of doing this! Like rvest
3 a <- readLines("https://indico.desy.de/indico/event/25594/registration/registrants")
4 b <- grep('nowrap',a) # find lines in the table
5 b <- b[-1] # omit header line
6 N <- length(a)
7 towns=character()
8 for (i in b){ # for each line in the table (i.e. student)
9   c <- grep('<td',a[i:N]) # find the data columns
10  d <- a[i-1+c[3]] # and extract the third
11  d <- sub('<.*?>',"",d) # remove <td> and </td> formatting
12  d <- sub('<.*?>',"",d)
13  d <- trimws(d) # trim white space
14  towns <- append(towns,d) # what's left is the town, or city
15 }
16
17 print(paste("There are ",length(towns)," entries"))
18 f <- factor(towns) # tells you how many different towns and how many of each
19
20 plot(f,col='yellow',horiz=TRUE,yaxt='n')
21
22 for(i in 1:length(levels(f))){ text(7,i*1.2-.4,levels(f)[i],cex=0.5)}
```

Console output:

```
+ d <- a[i-1+c[3]] ... [TRUNCATED]
> print(paste("There are ",length(towns)," entries"))
[1] "There are 105 entries"
> f <- factor(towns) # tells you how many different towns and how many of each
> plot(f,col='yellow',horiz=TRUE,yaxt='n')
> for(i in 1:length(levels(f))){ text(7,i*1.2-.4,levels(f)[i],cex=0.5)}
Warning message:
In readLines("https://indico.desy.de/indico/event/25594/registration/registrants") :
incomplete final line found on 'https://indico.desy.de/indico/event/25594/registration/registrants'
```

The horizontal bar chart displays the frequency of towns. The x-axis represents the count of registrants, ranging from 0 to 25. The y-axis lists the towns. The most frequent town is 'Kornwestheim', with approximately 26 registrants. Other towns with significant counts include 'Wurzburg' (approx. 10), 'Wuppertal' (approx. 8), 'Worms' (approx. 7), 'Tübingen' (approx. 6), 'Darmstadt' (approx. 5), 'Münster' (approx. 4), 'Lindlar' (approx. 3), 'Johanneshaus' (approx. 2), 'Aachen' (approx. 2), and 'Zürich' (approx. 2).

# Conclusions

R is

- Powerful
- Elegant
- Useful
- User-friendly
- Stuffed with great packages
- Easy-to learn

But don't take my word for it – try it for yourself!