# Blockpraktikum
# Zeitreihenanalyse mit R

16. Februar 2010

Sebastian Mentemeier und Matti Schneider

Institut für Mathematische Statistik
Universität Münster
WS 2009/2010

# Gliederung

1. Hinweise zum Praktikum
2. Einführung in R
   - Programmstart
   - R Umgebung und elementare Funktionen
3. Datenstrukturen
   - Vektoren
   - Datentabellen
   - Matrizen
   - Zeitreihen
   - Arrays, Listen, ...
4. Programmierung
   - Funktionen in R schreiben
   - Schleifen und Abfragen

# Gliederung

# Ablauf

- 10h - 12h: Theorie im **SR2** (bzw. jetzt: Einführung in R im SRA)
- 13h30 - 15h: Besprechung der Übungsaufgaben vom Vortag (bzw. heute: Theorie) im **SRA**
- 15h - 18h: betreute Bearbeitung der Übungsaufgaben im **SRA**
- Es darf/soll in 2er Gruppen gearbeitet werden, um die Aufgaben zu lösen
- Materialen (Übungszettel, Folien, ...) befinden sich auf der Praktikums-Homepage: `http://wwwmath.uni-muenster.de/statistik/lehre/WS0910/BlockprakZeit/`

# Literatur


Silke Ahlers
*Einführung in die Statistik mit R*
`http://wwwmath.uni-muenster.de/statistik/lehre/WS0910/BlockprakZeit/Skript.pdf`

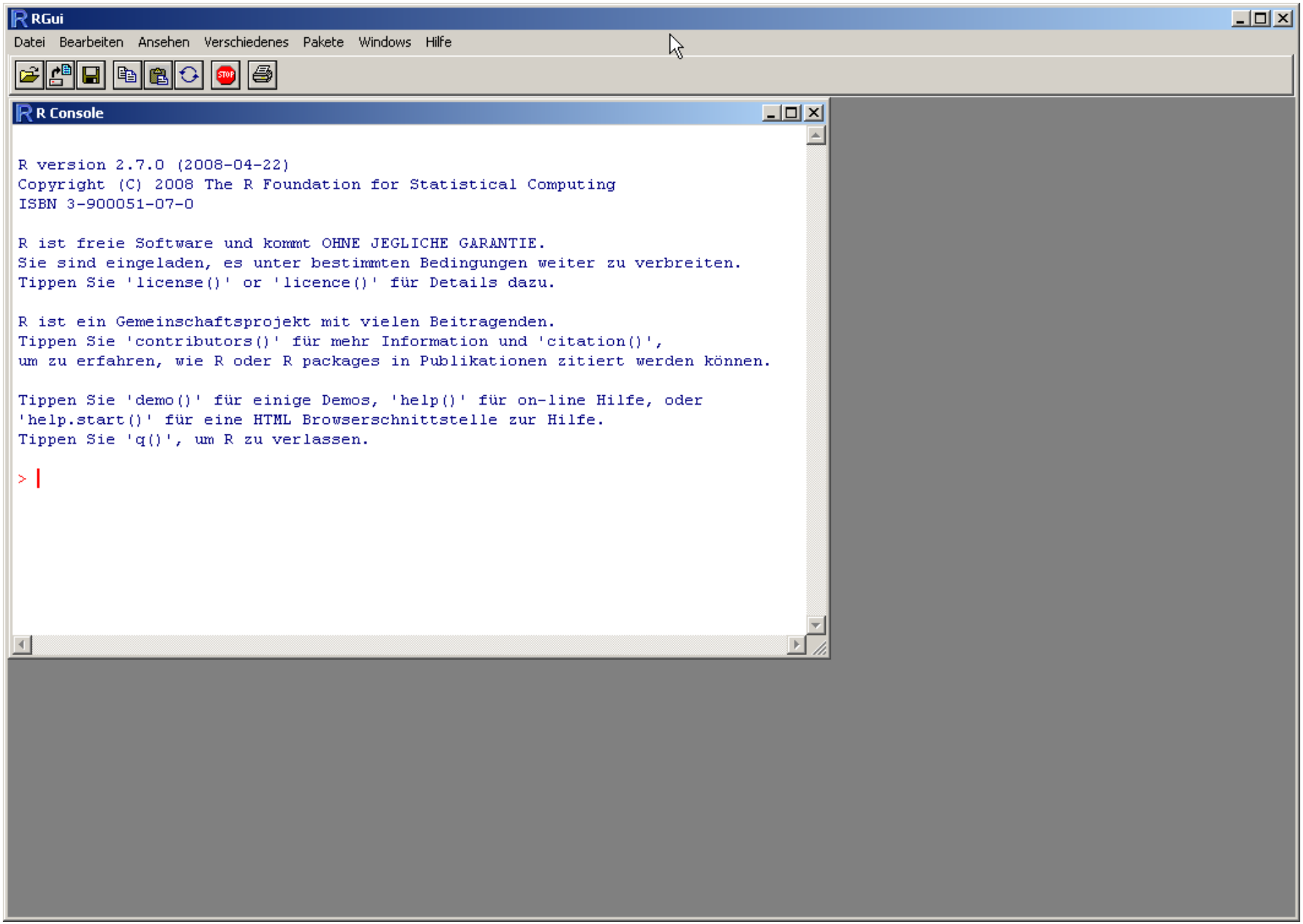
Paul Cowpertwait und Andrew Metcalfe
*Introductory Time Series with R*
Springer

Peter Brockwell und Richard Davis
*Time Series: Theory and Methods*
Springer

# Gliederung

# Programmstart

- Im **SR A** loggen Sie sich zunächst unter CentOS ein, Öffnen Sie eine Konsole und führen Sie `rdesktop -f zivtserv.uni-muenster.de` aus. Loggen Sie sich unter Windows ein und starten Sie R (Start → Programme → R)
- In den **CIP-Pools** loggen Sie sich unter Windows wie gewohnt ein, und starten Sie dann die Remotedesktopverbindung. Diese finden Sie im Startmenü → (Alle) Programme → Zubehör. Geben Sie als Computer `zivtserv.uni-muenster.de` ein, und klicken Sie auf Verbinden. Es erscheint ein neues Anmeldefenster, in dem Sie sich wiederum mit Ihrem üblichen Benutzernamen und Kennwort einloggen. Wählen Sie als Domäne (Anmelden an) UNI-MUENSTER. Nun können Sie R direkt vom Desktop aus starten.


RGui
Datei Bearbeiten Ansehen Verschiedenes Pakete Windows Hilfe
R Console
R version 2.7.0 (2008-04-22)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.
R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert werden können.
Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.
>

# Befehlsmodus

- R bietet eine interaktive Umgebung, den *Befehlsmodus*, in dem man Daten direkt eingeben und analysieren kann
- Der Befehlsmodus dient auch als *Taschenrechner*, z. B. können die Grundrechenarten `+,-,*,/` direkt eingegeben werden
- Zum Potenzieren muss `^` benutzt werden

**Beispiel**

```
> 4+5*5.7
> 5/6-2
> 2^3
```

# Mathematische Funktionen

- Die Eingabe `2^0.5` liefert das Ergebnis $2^{0.5} = \sqrt{2} \cong 1.414214$
- Einfacher: Eingabe von `sqrt(2)` (sqrt = square root)
- Auch andere mathematische Funktionen sind bereits in R implementiert, etwa Logarithmus, Sinus, Cosinus, ...

**Beispiel**

```
> tan(7/8)
> exp(3.72)
> abs(sin(5))
```

- Eine Übersicht über wichtige Funktionen gibt es auf der Praktikums-Homepage: `http://wwwmath.uni-muenster.de/statistik/lehre/WS0910/BlockprakZeit/R-Befehle.pdf`

# obligatorische und optionale Argumente

- R rundet Zahlen (falls nötig) Standardmäßig auf sechs Dezimalstellen
- Runden auf weniger Stellen mit `round(x,digits)`
- `x`, die Zahl, die gerundet werden soll, ist hierbei ein *obligatorisches Argument*
- Das *optionale Argument* `digits` gibt die Anzahl der Dezimalstellen an, die gerundet werden sollen. Wird dieses Argument nicht angeben, so rundet R auf eine ganze Zahl

**Beispiel**

```
> round(x=sqrt(2))
> round(x=sqrt(2), digits=3)
> round(sqrt(2),3)
```

# Hilfefunktion

- Kennt man die Reihenfolge der Argumente im sogenannten Kopf der Funktion, so kann man Werte direkt eingeben
- Ist dies nicht der Fall, gibt man sie mit „Name = ·“ ein
- Welche Argumente eine Funktion besitzt, lässt sich mit den Hilfeseiten herausfinden, durch die Eingabe `help(round)` oder kürzer `?round`
- Auch mit `args` lässt sich herausfinden, welche Argumente eine Funktion erhalten kann

**Beispiel**

```
> help(cos)
> ?choose
```

# Gliederung

# Vektorenerstellung

- Vektoren erzeugt man in R mit der Funktion c
- Die Erzeugung eines Vektors mit den Daten $x_1, \ldots, x_n$ erhält man durch Eingabe von c($x_1, \ldots, x_n$)

**Beispiel**

```
> c(3,0,-4,16)
> c(sin(-5),-5.66, -5*6)
```

# Variablenzuweisung I

- Um nicht jedes mal einen Datensatz neu eingeben zu müssen, kann man einem Vektor (oder einer anderen Datenstruktur) einen Namen zuweisen
- Dies geschieht mit dem *Zuweisungsoperator* `<-`
- Links von diesem Operator steht der Name der Variablen, dem die Daten zugeordet werden sollen,
  z. B. `Daten <- c(3,0,-4,16)`
- Durch Eingabe des Variablennames in den Befehlsmodus ruft man den Inhalt der Variablen auf

**Beispiel**

```
> Messung <- c(6,7,5,5,12)
> Messung
> x <- 1:10
> x
```

# Variablenzuweisung II

- R achtet auf Groß- und Kleinschreibung: Die Eingabe von `messung` liefert einen Fehler
- Variablennamen dürfen aus Buchstaben, Zahlen und dem Punkt „.“ bestehen
- Sie beginnen aber immer mit einem Buchstaben
- Will man einen Wert an eine Variable nach rechts übergeben, so geschiet dies mit Hilfe von `->`

**Beispiel**

```
> 15:1 -> abst.15
> abst.15
```

# Komponentenzugriff I

- Die $i$-te Komponente eines Vektors $x$ erhält man durch Eingabe von `x[i]`
- Die 2., 4. und 7. Komponente erhält man durch `x[c(2,4,7)]`

**Beispiel**

```
> Messung[4]
> Messung[2:5]
```

# Logische Operatoren

Will man Komponenten nach Bedingungen auswählen, so geschiet dies unter Benutzung von *logischen Operatoren*

| | |
|---|---|
| == | gleich |
| != | ungleich |
| < | kleiner |
| > | größer |
| <= | kleiner gleich |
| >= | größer gleich |

**Beispiel**

```
> 4<2
> 2*6 == 12
> Messung >= 7
```

# Komponentenzugriff II

- Durch `Messung[Messung >= 7]` erhält man alle Messungen, die einen Wert größer (oder gleich) 7 besitzen
- Die Positionen dieser Werte innerhalb des Vektors kann man mit der Funktion `which` herausfinden
- Mit den Funktionen `any` bzw. `all` lässt sich überprüfen, ob eine bzw. alle Komponenten eines Vektors eine bestimmte Bedingung erfüllen

**Beispiel**

```
> which(Messung>=7)
> any(Messung<6)
> all(Messung<6)
```

# Vektorenfunktionen I

- Es gibt eine Fülle nützlicher Funktionen, die man auf einen Vektor anwenden kann
- Mit `sum(x)` erhält man etwa die Summe alle Komponenten von $x$, mit `min(x)` das Minimum der Einträge, mit `max(x)` das Maximum der Einträge, mit `length(x)` die Anzahl der Komponenten von $x$, ...
- Eine Sortierung der Vektoreinträge geschieht mit `sort(x)`

**Beispiel**

```
> prod(Messung)
> sort(Messung)
> sort(Messung, decreasing = TRUE)
```

# Vektorenfunktionen II

- Auch elementare Funktionen lassen sich auf Vektoren anwenden (Auswertung komponentenweise)
- Ebenso nützlich ist die *Vektorarithmetik* `v+w`, `v*w`, `v^w`, ... für Vektoren $v = (v_1, \ldots, v_n)$ und $w = (w_1, \ldots, w_m)$
- Für m = n geschieht die Auswertung komponentenweise
- Ist n ein Vielfaches von m, so geschieht die Auswertung zyklisch, d. h. $\mathtt{v} + \mathtt{w} = (\mathtt{v[1]} + \mathtt{w[1]}, \ldots, \mathtt{v[m]} + \mathtt{w[m]}, \mathtt{v[m+1]} + \mathtt{w[1]}, \ldots, \mathtt{v[n]} + \mathtt{w[m]})$

**Beispiel**

```
> log(Messung)
> Messung * 4
> round(exp(Messung), 2)
> Messung - 1:5
> (1:6)^(2:3)
```

# qualitative Merkmale

- Für die Eingabe von Daten bei einem qualitativem Merkmal müssen die Komponenten aus Zeichenketten bestehen, die in Hochkommata eingschlossen sind, z. B. `c("blau","grün","gelb")`
- Zur Beseitigung der Hochkommata dient die Funktion `factor`
- Diese macht aus einem Vektor ein *Faktor*, d. h. ein qualitatives Merkmal

**Beispiel**

```
> Geschlecht <- c("m", "w", "w", "m", "w")
> Geschlecht <- factor(Geschlecht)
> Geschlecht
> Messung[Geschlecht=="w"]
```

# Datentabellen

- In *Datentabellen* können Werte von Merkmalen unterschiedlichen Typs gespeichert werden
- Jedem Merkmal muss dabei die gleiche Anzahl von Beobachtungen zugrundeliegen
- Der R Befehl zur Erstellung von Datentabellen lautet `data.frame`

**Beispiel**

```
> Tabelle <- data.frame(Geschlecht = c("m", "w", "w"),
  Alter = c(24,32,20))
> Tabelle
```

# Zugriff auf Datentabellen

- `Tabelle[3,2]` liefert den 2. Zeileneintrag der 3. Spalte der Datentabelle
- Mit `Tabelle[,2]` erhält man die komplette 2. Spalte der Tabelle, mit `Tabelle[3,]` die komplette 3. Zeile
- Die „Alter“-Spalte erhält man auch durch `Tabelle$Alter`
- Benutzt man den Befehl `attach(Tabelle)`, so kann man direkt auf die Variable `Alter` zugreifen
- Der Befehl `detach(Tabelle)` hebt diese Zugriffsmöglichkeit wieder auf

**Beispiel**

```
> Tabelle[,1]
> attach(Tabelle)
```

# Variablen im Workspace

- Da wir den Variablennamen „Geschlecht“ bereits in Benutzung haben (siehe oben), gibt R einen entsprechenden Fehlerhinweis aus
- Eine Auflistung aller im Workspace benutzten Variablen gibt es mit `ls()`
- Will man eine Variable löschen, so geschieht dies mit `rm()`

**Beispiel**

```
> geschlecht<-Geschlecht
> rm(Geschlecht)
> Geschlecht
> detach(Tabelle)
```

# Teiltabellen

- Die Funktion `subset(x, condition, select)` bietet die Möglichkeit, aus einer Tabelle $x$ kleinere Datensätze gemäß der Bedingung `condition` auszuwählen
- Mit dem optionalen Argument `select` ist es möglich, nur gewisse Spalten auszuwählen

**Beispiel**

```
> subset(Tabelle, Geschlecht == "w")
> subset(Tabelle, Geschlecht == "w", select = Alter)
```

# Einlesen aus externen Dateien

Damit man große Datensätze nicht mit der Hand abtippen muss, gibt es einige Funktionen, mit denen es möglich ist, externe Dateien einzulesen:

| | |
|---|---|
| `read.table("Pfad")` | liest externen Datensatz ein |
| `read.csv("Pfad")` | liest durch Kommata getrennte Spalten |
| `read.csv2("Pfad")` | liest durch Semikola getrennte Spalten, Komma als Dezimalkomma |
| `read.delim("Pfad")` | liest Tab-getrennte Spalten |
| `read.delim2("Pfad")` | liest Tab-getrennte Spalten, Komma als Dezimalkomma |

# Matrizen

- Mit dem Befehl `matrix(data,nrow,ncol,byrow(=FALSE))` lässt sich eine Matrix in R erzeugen.
- `data` ist der Vektor, mit dem die Matrix gefüllt werden soll
- `nrow` die Anzahl der Zeilen, `ncol` die Anzahl der Spalten (es genügt, wenn man eines angibt)
- Optional: `byrow=TRUE` gibt an, dass `data` Zeilenweise in die Matrix eingefüllt wird

**Beispiel**

```
> matrix(1:9,3)
> matrix(1:12,3,byrow=TRUE)
> A<-matrix(c(3,4,0,1),2)
> B<-matrix(9:6,2)
```

# Matrizenzugriff und -Funktionen

- Der Zugriff auf Matrizeneinträge, Zeilen und Spalten geschieht wie bei Datentabellen
- Auch für Matrizen gibt es in R eingebaute Funktionen, etwa für die Matrizenmultipliktaion `A %*% B`, die Determinantenberechnung `det(A)` oder das Lösen linearer Gleichungssysteme $Ax = b$ mit `solve(A,b)`
- Unterschied zwischen Matrizen und Datentabellen: Es können nur die Werte von Merkmalen *eines* Typs in einer Matrix gespeichert werden

**Beispiel**

```
> A %*% B
> det(A)
> solve(A,c(5,2))
```

# Zeitreihen

- Eine *Zeitreihe* ist eine (oder mehrere) aufeinanderfolgende Beobachtung(en) eines Merkmals
- In R erstellt man sie mit dem Befehl `ts(data,start,frequency)`
- `data` kann dabei ein Vektor, eine Matrix oder eine Datentabelle sein, jenachdem, wieviele Zeitreihen man in einer Variablen zusammenfassen will
- `start` muss ein Vektor mit 2 Komponenten sein: Die erste gibt das Startjahr an, die zweite den Startmonat
- `frequency` gibt die Häufigkeit der Beobachtungen pro Zeiteinheit an, etwa 12, wenn man 12 Beobachtungen pro Jahr hat

**Beispiel**

```
> ZR<-ts(rep(1:10,10:1),start=c(1996,1),frequency=4)
> ZR.2<-ts(matrix(c(1:12,60:49),ncol=2), start=c(1980,1),
  frequency=12)
```

# Ausschnitte aus Zeitreihen

- Mit dem Befehl `window(x, start, end, frequency)` kann man einen Ausschnitt aus einer Zeitreihe herausnehmen
- `x` ist dabei die Zeitreihe, `start`, `end` und `frequency` haben die offensichtlichen Bedeutungen
- Bsp: Liegen für `x` Monatsdaten vor, so erhält man mit
  `window(x,start=c(start(x)[1],4),freq=1)`
  alle Werte des Aprils (4. Monat)
- `start(x)[1]` gibt dabei das Startjahr von `x` aus

**Beispiel**

```
> end(ZR)
> ZR.2[,2]
> window(ZR.2, start=c(1980,3), end=c(1980,6))
```

# weitere Datenstrukturen

Es gibt noch einige weitere Datenstrukturen, etwa

- *Array* = $d$-dimensionalen Datensatz (R Befehl: `array`)
- *Liste* = Liste, deren Komponenten unterschiedliche Struktur haben dürfen (R Befehl: `list`)
- ...
- Welche Struktur eine Variable hat, lässt sich mit dem Befehl `str` herausfinden

**Beispiel**

```
> str(Tabelle)
> str(A)
> str(ZR.2)
```

# Gliederung

# Funktionen

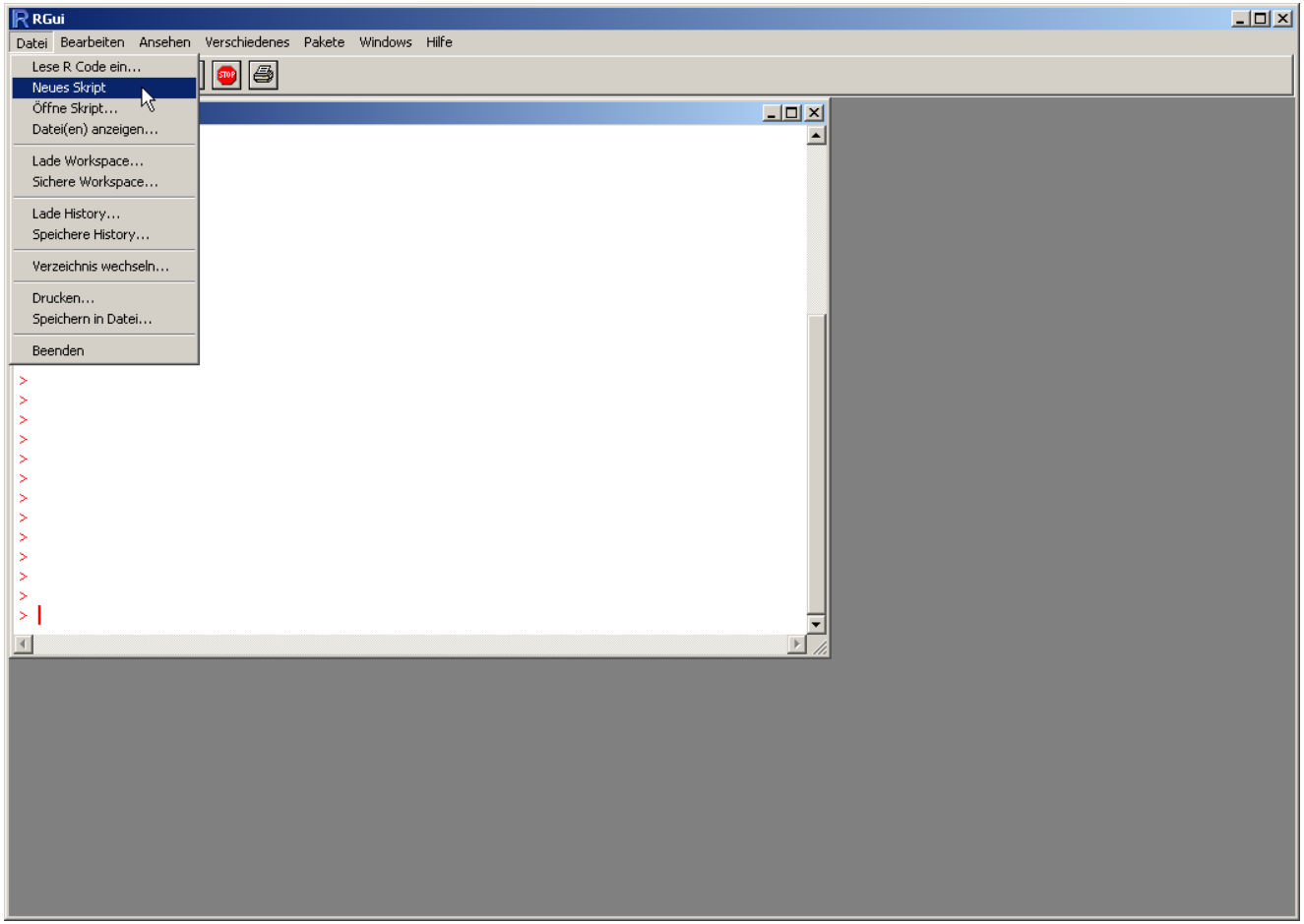
- Eigene Funktionen erstellt man mit dem Befehl
  `function(Argumente) {`
  `Körper der Funktion`
  `return(Ergebnis)}`
- Optionale Argumente gibt man durch `arg=Wert` an
- Alle im Körper definiertern Variablen sind lokal, d. h. man kann sie außerhalb der Funktion nicht nutzen

**Beispiel**

Wir wollen eine Funktion `f` schreiben, die $f(x, y) = x^y - x$ berechnet. $y$ soll dabei ein optionales Argument mit Standardwert 2 sein.

```
> f<-function(x,y=2) {erg<-x^y-x; return(erg)}
> f(3)
> f(3,4)
> f(1:9,4)
```

Eine längere Funktion, oder eine Abfolge von vielen Befehlen, sollten nicht direkt im Befehlsmodus definiert werden, sondern in einem *Skript*




Die in einem Skript geschriebenen Befehle werden erst im Befehlsmodus ausgeführt, wenn man sie markiert und `Strg + R` drückt

# allgemeine Programmierhinweise

- Benutzen Sie Variablen, um ein Skript flexibel ändern zu können: Wenn man etwa mit 100 Simulationen arbeiten muss, setzt man `n<-100` und arbeitet mit `n`, um später leicht $n = 1000$ Simulationen durchführen zu können
- Verwenden Sie aussagekräftige Variablennamen um den Code lesbarer zu machen: Lieber `sim.anz` statt `n` als Namen für die Simulationsanzahl wählen
- Kommentieren Sie Ihren Code so, dass Sie auch später noch verstehen, was er bewirkt (Kommentare schreibt man mit #, alle Zeichen dahinter werden ignoriert)
- Benutzen Sie die Vektorarithmetik: Mit ihr lassen sich viele `for` Schleifen umgehen und ein Skript braucht i. d. R. weniger Laufzeit
- Speichern Sie ein Skript regelmäßig (mit `Strg + S`) um bei einem Programmabsturz nicht alles neu schreiben zu müssen

# for Schleife

- Mit den Befehlen `for` und `while` kann man Schleifenprozesse programmieren, also solche Vorgänge, die vom Programm immer wiederholt werden, bis eine gewisse Bedingung erfüllt ist
- Die `for`-Schleife unterliegt folgender Syntax:
  `for(Name in Vektor) { Befehle }`
- Dadurch wird eine Variable, die Name heißt, schrittweise gleich den Elementen des Vektors gesetzt
- In jedem Schritt wird für den entsprechenden Wert des Vektors der zugehörige Befehl aus den geschweiften Klammern ausgeführt

**Beispiel**

Wir wollen die ersten 12 Fibonacci-Zahlen erzeugen.

```
> Fibo <- numeric(12)
> Fibo[1] <- Fibo[2] <- 1
> for (i in 3:12) Fibo[i] <- Fibo[i-2]+Fibo[i-1]
```

# while Schleife

- Will man einen Vorgang wiederholen, aber weiß im Vorweld nicht, wann die Schleife abbrechen soll, so hilft einem die `while` Schleife
- Sie unterliegt folgender Syntax: `while(Bedingung) { Befehle }`
- Vor jedem Schritt wird die Bedingung überprüft: Solange diese `TRUE` ist, werden die Befehle ausgeführt, tritt `FALSE` ein, wird die Schleife beendet

**Beispiel**

Wir wollen alle Fibonacci-Zahlen auflisten, die kleiner als 300 sind.

```
> Fib1 <- Fib2 <- Fibonacci <- 1
> while(Fib2<300) {
  Fibonacci <- c(Fibonacci,Fib2)
  oldFib2 <- Fib2
  Fib2 <- Fib1+Fib2
  Fib1 <- oldFib2 }
```

# if-else Abfragen

- Häufig müssen in die Definition einer Funktion Fallunterscheidungen bzgl. des Outputs in Abhängigkeit von den eingesetzten Werten vorgenommen werden.
- Dazu verwendet man Anweisungen der Art `if(Bedingung) { Befehlsfolge } else { Befehlsfolge }`

**Beispiel**

Es soll die Indikatorfunktion auf dem abgeschlossenen Einheitsintervall programmiert werden.

```
> indikator<-function(x) {
  if(x<0) return(0)
  else { if(x>1) return(0) else return(1) }
  }
```

# Logische Operatoren II

Da in den Fällen $x < 0$ und $x > 1$ der gleiche Wert zurückgegeben wird, könnte man auch beide Bedingungen mit einem logischen ODER miteinander verknüpfen

| | |
|---|---|
| & | logisches UND |
| \| | logisches ODER |
| ! | logisches NICHT |

**Beispiel**

```
> indikator2<-function(x) {
  if(x<0 | x>1) return(0)
  else return(1)
  }
```